Linux Random Number Generator – A New Approach

Stephan Müller <smueller@chronox.de>

November 13, 2019

Abstract

The venerable Linux /dev/random has served users of cryptographic mechanisms well for a long time. The random number generator is well understood how entropic data is delivered. 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 reperform 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 random data to cryptographic and 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 [6], the behavior of the legacy /dev/random in virtual environments [8] and presentations on /dev/random such as [7] given at the ICMC 2015. All the studies show that the random data out of /dev/random are highly entropic and offer 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[2] 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[3].

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:

- 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 time stamp for those events are considered to have relatively little entropy which implies that 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.

Due to the correlation effect between the HID and block device events on one side and the associated interrupts on the other hand, the legacy /dev/random implementation always credits interrupts very little entropy to prevent any potential overestimation of entropy.

What are the challenges for those aforementioned three noise sources?[4]

[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] If two or more disks are present in the system that are deemed to provide entropy, the order of event arrivals for the different disks may provide some small entropy.

[3] Such entropy naturally relies on the assumption that the time variances of events are hard to predict with sufficient precision relative to the resolution of the timer. In addition, any attacker is assumed to not have access to the kernel memory holding the entropy as otherwise it is eliminated.

[4] 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.

At the time when block devices were chosen as a noise source for the legacy /dev/random, computer were commonly equipped with spinning hard disks. For those disk devices, 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. The attacker's inability to predict or resolve the exact 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[5]. 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[6]. Thus, for example, on a system with an SSD, no entropy is collected when accessing that disk.

The timing of HID events using a high-resolution timer is commonly a great source of entropy as it delivers much entropy for any random number generator due to the fact that large numbers of events occur. In addition, assuming the precise timing of an event must be assumed to be unknown to any attacker. 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.

With the findings above the following conclusion can be drawn: 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

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

[6] 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.

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 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. Conversely, the entire assessment in this document, specifically chapter 3 and following show that interrupt events on a system with high-resolution time stamps provide large amounts of entropy. However, due to the correlation issue, the legacy /dev/random implementation's entropy heuristics cannot be changed to award interrupt events a higher entropy rate.

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. Although the naming refers only to the interface, the entirety of the legacy random number generation in the Linux kernel available through different interfaces like /dev/random, /dev/urandom, getrandom(2) or the in-kernel function of get_random_bytes is covered by the name.

- The newly proposed approach for entropy collection is called Linux Random Number Generator (LRNG) throughout this document. It provides an API and ABI compatible replacement implementation of the legacy /dev/random implementation providing the same interfaces and identical user-visible behavior but with a completely different collection and management of entropy as well as generation of random numbers.

The new approach provides the following significant differences compared to the legacy /dev/random implementation:

1. The LRNG considers the timing of interrupts as the source of entropy. Therefore, the entropy heuristic applied by the LRNG only rests on the timing of interrupts. Other event information like the HID event data (e.g. which key stroke was received, which mouse coordinate was recorded) or

block device numbers are picked up and stirred into the entropy pool but without awarding them any heuristic entropy. Thus, the LRNG is not affected by the aforementioned correlation issue.

2. The LRNG introduces the concept of slow and fast noise sources. Fast noise sources provide entropy at the time of request. A slow noise source collects data over time into an entropy pool – the interrupt events are considered such a slow noise source. The LRNG combines both types of noise sources that when the entropy pool is queried for entropy, all fast noise sources are also queried for additional entropy and the concatenated data is handled by the post-processing to generate random numbers. With this, the LRNG can use both types of noise sources without allowing one noise source to dominate another.

3. The seeding mechanism of the LRNG during boot ensures that entropy data is forwarded to the deterministic random number generators in well-defined chunks of 32 bits, 128 bits and 256 bits where these chunks denominate the of initial, minimal and full seed levels of the LRNG. Thus, the LRNG cannot be tricked into repeatedly releasing small entropy levels to the deterministic random number generators and thus to callers. Such attack approach would diminish the collected entropy significantly. 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.

4. The LRNG supports a runtime-switchable deterministic random number generator that generates data for a calling user that can be enabled during compile time. With a well defined API developers can implement their own deterministic random number generator if the provided ones are not considered appropriate. Per default, a ChaCha20-based deterministic random number generator is used. In addition, an SP800-90A [1] DRBG offering all three DRBG types is provided as well. The SP800-90A DRBG and its cryptographic primitives are taken from the kernel crypto API which implies that these implementations are covered by the power-on health tests offered by the kernel crypto API.

5. The LRNG provides a health test interface that monitor the received entropy for the slow noise source can can be enabled during compile time. Using this test interface, SP800-90B [11] or AIS31 [5] compliant health tests are implemented. With these health tests and additional logic, the LRNG is considered SP800-90B compliant. When using an SP800-90A DRBG at the same time the LRNG operates compliant to SP800-90B, the output of an LRNG can be used directly for purposes requiring data from a FIPS 140-2 approved noise source and random number generator.

6. To analyze the slow noise source, the LRNG provides a development interface allowing to extract the raw unconditioned noise data[7].

[7] This interface is solely intended for development. It is intended to be disabled at compile time for production systems.

7. Tests are developed for various aspects of the LRNG allowing a user-space simulation of those LRNG functions. Such simulations allow developers to further analyze and assess the implementation and the resulting behavior of the LRNG. In addition, tests for all types of entropy assessments required by SP800-90B are provided. Finally, this document provides a full SP800-90B entropy analysis.

The idea for the LRNG design occurred during a study that was 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.

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

1. During boot time, the LRNG is designed to 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 conducted very much later, if at all which implies that until such re-seeding happens, the DRNG may provide weak random numbers. The LRNG is intended to ensure that for such use cases, sufficient entropy is available during early user space boot. 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 chapter 4 the DRNG is seeded with full security strength of 256 bits during the first steps of the initramfs time after about 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` or the ChaCha20 DRNG after 30 seconds or more after boot with 128 bits of entropy. In addition, the LRNG maintains the information by when the DRNG is "minimally" seeded with 128 bits of entropy to trigger in-kernel callers requesting random numbers with sufficient quality. 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. All user-visible behavior implemented by the legacy /dev/random – such as the per-NUMA-node DRNG instances are provided by the LRNG as well.

4. 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.

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

6. 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.

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

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

9. 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.

10. 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 statistical testing including the SP800-90B compliance assessment is provided in chapter 3.

- The comparison of the LRNG with the legacy /dev/random is covered in chapter 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.

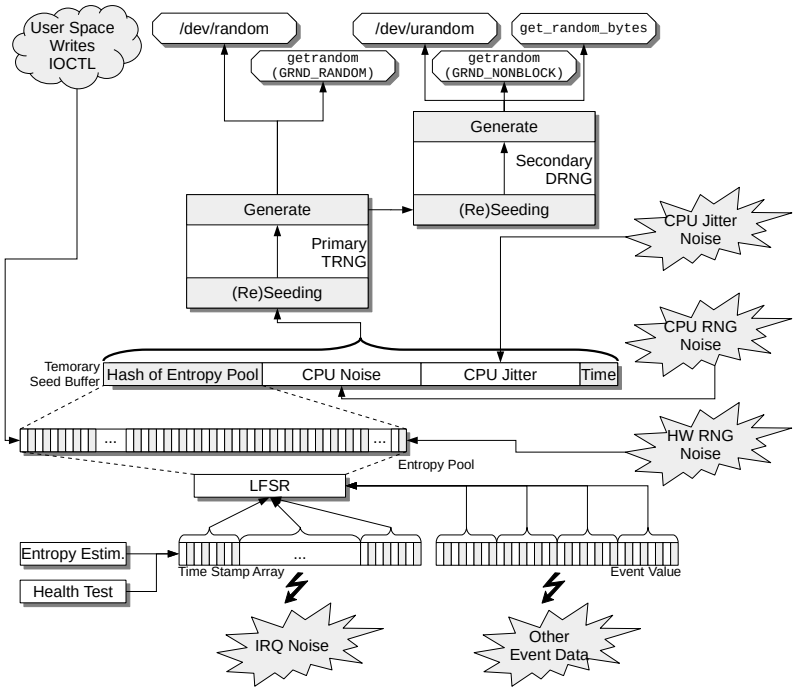

Figure 2.1: LRNG Big Picture

During compile time the TRNG may be deconfigured and thus would not be present. In this case, the big picture would look as follows.

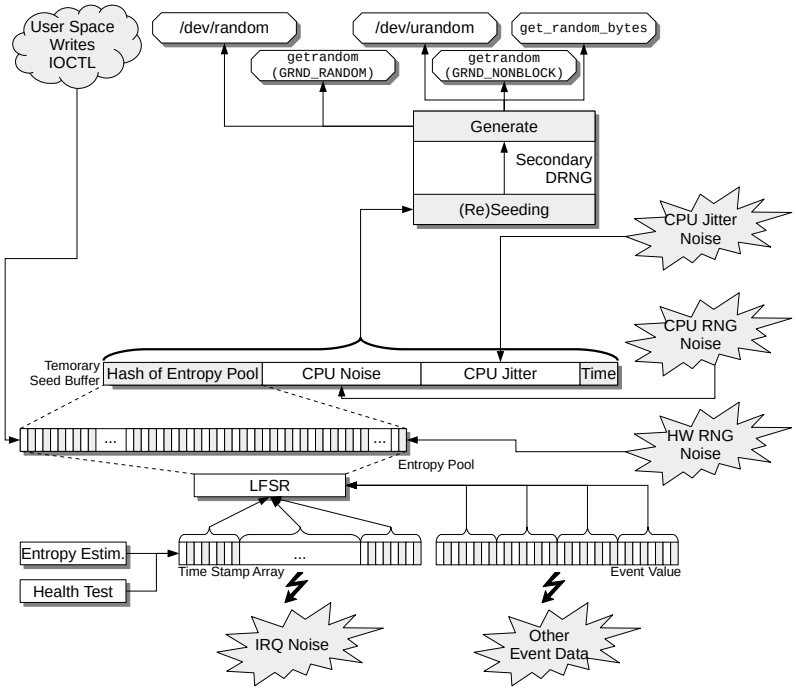

Figure 2.2: LRNG Big Picture without TRNG

2.1 LRNG Components

The LRNG consists of the following components shown in figure 2.1:

1. The LRNG may contain a True Random Number Generator (TRNG). The TRNG is a deterministic random number generator that is operated as a true random number generator. Using SP800-90A terminology, the TRNG is a DRBG with prediction resistance. The TRNG has a behavior similar to SP800-90A's concept of prediction resistance by only generating output when it is re-seeded with an equal amount of entropy. Every time a caller requests random numbers, the TRNG must be re-seeded with at least that amount of entropy from its noise sources. During boot time, the TRNG is not yet operated with prediction resistance. As the external interfaces to the TRNG to obtain random numbers start to be accessible after boot time completes, random numbers generated via these interfaces always access the TRNG that is operated with prediction resistance.

   During compile time, the TRNG may be deselected and not compiled. In this case, the LRNG links the temporary seed buffer as well as the output interfaces of /dev/random and getrandom(2) with the secondary DRNG instances. The remainder of the LRNG remains unchanged. If the TRNG is not compiled, all discussions regarding the seeding of the TRNG apply to the secondary DRNGs.

2. The LRNG implements a secondary DRNG. The secondary DRNG always generates the requested amount of output. When using the SP800-90A terminology it operates without prediction resistance. The secondary DRNG

maintains a counter of how many bytes were generated since last re-seed and a timer of the elapsed time since last re-seed. If either the counter or the timer reaches a threshold, the secondary DRNG is seeded from the TRNG.

In case the Linux kernel detects a NUMA system, one secondary DRNG instance per NUMA node is maintained.

3. The TRNG is seeded by concatenating the data from the following sources:

   (a) the output of the entropy pool,

   (b) the Jitter RNG if available, and

   (c) the CPU-based noise source such as Intel RDRAND if available.

   The entropy estimate of the data of all noise sources are added to form the entropy estimate of the data used to seed the TRNG with. The LRNG ensures, however, that the TRNG after seeding is at maximum the security strength of the TRNG.

   The LRNG is designed such that none of these noise sources can dominate the other noise sources to provide seed data to the TRNG during boot time due to the following:

   (a) During boot time, the amount of received interrupts are the trigger points to (re)seed the TRNG following the explanation in the next section.

   (b) At runtime, the caller requesting random numbers from the TRNG drives the reseeding where always as much entropy as available is used to reseed the TRNG.

4. The entropy pool collects noise data from slow noise sources. Any data received by the LRNG from the slow noise sources is inserted into the entropy pool using an LFSR with a primitive and irreducible polynomial. The following sources of entropy are used:

   (a) When an interrupt occurs, the high-resolution time stamp is mixed into the LFSR. This time stamp is credited with heuristically implied entropy.

   (b) HID event data like the key stroke or the mouse coordinates are mixed into the LFSR. This data is not credited with entropy by the LRNG.

   (c) Device drivers may provide data that is mixed into the LFSR. This data is not credited with entropy by the LRNG.

   (d) After the entropy pool is "read" by the TRNG, the data used to seed the TRNG is mixed back into the entropy pool to stir the pool. This data is not credited with entropy by the LRNG.

   Any data provided from user space by either writing to /dev/random, /dev/urandom or the IOCTL of `RNDADDENTROPY` on both device files are always injected into the entropy pool.

   In addition, when a hardware random number generator covered by the Linux kernel HW generator framework wants to deliver random numbers,

it is injected into the entropy pool as well. HW generator noise source is handled separately from the other noise source due to the fact that the HW generator framework may decide by itself when to deliver data whereas the other noise sources always requested for data driven by the LRNG operation. Similarly any user space provided data is inserted into the entropy pool.

When the TRNG requires data from the entropy pool, the entire entropy pool is processed with an SP800-90A section 10.3.1 compliant hash_df function to generate random numbers.

To speed up the interrupt handling code of the LRNG, the time stamp collected for an interrupt event is truncated to the 8 least significant bits. 64 truncated time stamps are concatenated and then jointly inserted into the LFSR. During boot time, until the fully seeded stage is reached, each time stamp with its 32 least significant bits is inserted into the LFSR at the time of arrival. More details are given in section 2.4.1.

The LRNG allows the TRNG and secondary DRNG mechanism to be changed at runtime. Per default, a ChaCha20-based DRNG is used[8]. The LRNG also offers an SP800-90A DRBG based on the Linux kernel crypto API DRBG implementation.

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 TRNG'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 TRNG generate request is limited to the amount of entropy the of the DRNG was seeded with, and
  2. each TRNG generate request is preceded by a reseeding of the DRNG to implement a TRNG / a DRNG with prediction resistance.

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

1. LFSR: The 8 least significant bits of the time stamp data received from the interrupts are processed with an LFSR. That LFSR is implemented identically 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-approach is used in the OpenBSD /dev/random equivalent.

[8]The ChaCha20-DRNG implemented for the LRNG is also provided as a stand-alone user space deterministic random number generator.

2. Concatenation: The temporary seed buffer used to seed the TRNG 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.

The TRNG may not be compiled. In this case, the aforementioned statements covering the TRNG are not applicable. The secondary DRNG is seeded directly from the entropy pool just like the TRNG would have been seeded.

2.2 LRNG Architecture

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 sources is collected and accumulated in the entropy pool.

At the time the TRNG shall be seeded, the entire entropy pool is hashed with a hash defined at runtime[9] . The entire entropy pool is processed using the derivation function hash_df as defined in [1] with a cryptographic hash function which can be chosen at runtime. The hash_df is a function which can produce output of a desired length defined by the security strength of the TRNG. The reason to use the hash_df function is to support the SP800-90B assessment.

The output of the hash_df function is mixed back into the entropy pool using the LFSR for backtracking resistance. The hash_df output is concatenated with data from the fast noise sources.

The TRNG always tries to seed itself with 256 bits of entropy, except during boot. In any case, if the noise sources cannot deliver that amount, the available entropy is used and the TRNG keeps track on how much entropy it was seeded with. The entropy implied by the LRNG available in the entropy pool may be too conservative. To ensure that during boot time all available entropy from the entropy pool is transferred to the TRNG, the hash_df function always generates 256 data bits during boot to seed the TRNG. Yet, the TRNG entropy estimate is only increased by the amount of entropy the LRNG assumes to be present in that data. During boot, the TRNG is seeded as follows:

1. The DRNG is reseeded from the entropy pool and potentially the fast noise sources if the entropy pool has collected at least 32 bits of entropy from the interrupt noise source. The goal of this step is to ensure that the primary and secondary DRNG receive some initial entropy as early as possible. In addition it receives the entropy available from the fast noise sources.

2. The DRNG is reseeded from the entropy pool and potentially the fast noise sources if all noise sources collectively can provide at least 128 bits of entropy.

[9]In case of the CTR DRBG, the CMAC-AES is used to hash the entropy pool. 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 and the key is not considered to provide any cryptographic strength. 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.

3. The DRNG is reseeded from the entropy pool and potentially the fast noise sources if all noise sources collectivel can provide at least 256 bits[10] of entropy.

At the time of the reseeding steps, the DRNG requests as much entropy as is available in order to skip certain steps and reach the seeding level of 256 bits. This may imply that one or more of the aforementioned steps are skipped.

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

Before the TRNG is seeded with 256 bits of entropy in step 3, requests of random data from /dev/random are not processed.

At runtime, the TRNG delivers only random bytes equal to the entropy amount it was seeded with. E.g. if the TRNG 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 TRNG.

The TRNG will always require that all entropy sources collectively can deliver at least as many entropy bits as configured with /proc/sys/kernel/random/read_wakeup_threshold, i.e. per default 129 bits (128 bits of entropy for seeding plus one bit of entropy that is lost with the post processing as defined in SP800-90B – more details about the entropy assessment is given in section 3.2).

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. When longer random numbers are requested, multiple DRNG generate operations are performed. The ChaCha20 DRNG as well as the SP800-90A DRBGs implement an update of their state after completing a generate request for backtracking resistance.

- The secondary DRNG is reseeded with whatever entropy is available – in the worst case where no additional entropy can be provided by the noise sources, the DRNG is not re-seeded and continues its operation to try to reseed again after again the expiry of one of these thresholds:

  - 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

[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 128 bit is generated by the TRNG 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 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.

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 TRNG and the TRNG 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.

2.2.1 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 defines that that the secondary DRNG is considered "minimally" seeded whereas reaching the 256 bits level is defined as 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.

Note, the initial seeding level with 32 bits is implemented to ensure that early boot requests are served with random numbers having some entropy, i.e. the DRNG has some meaningful level of entropy for non-cryptographic use cases as soon as possible.

2.2.2 Seeding Examples

The following tables provide examples how the seeding is performed by the LRNG. The tables contain various seeding stages, how much data is injected

[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.

into the TRNG, how much entropy is injected into the TRNG, how much data will be produced by the TRNG and finally actions performed by the LRNG at the respective seeding level.

The following table shows the seeding during boot time with the default entropy levels for the fast noise sources as outlined in sections 2.4.2 and 2.4.3.

Seed Stage	Noise source data bits	Noise source entropy bits	TRNG output bits	LRNG behavior
Receipt of 33 fresh IRQs	IRQ: 256 Jitter: 256 CPU: 256	IRQ: 32 Jitter: 16 CPU: 8	56	/dev/random blocked /dev/urandom operational `getrandom`(2) blocked `wait_for_random_bytes` blocked `add_random_ready_callback` blocked `get_random_bytes` operational
Receipt of 105 fresh IRQs	IRQ: 256 Jitter: 256 CPU: 256	IRQ: 104 Jitter: 16 CPU: 8	128	/dev/random blocked /dev/urandom operational `getrandom`(2) blocked `wait_for_random_bytes` released `add_random_ready_callback` released `get_random_bytes` operational
Receipt of 233 fresh IRQs	IRQ: 256 Jitter: 256 CPU: 256	IRQ: 232 Jitter: 16 CPU: 8	256	/dev/random operational /dev/urandom operational `getrandom`(2) operational `wait_for_random_bytes` released `add_random_ready_callback` released `get_random_bytes` operational

The next table outlines the runtime reseeding behavior with again assuming the fast noise sources have the default entropy levels.

Seed Stage	Noise source data bits	Noise source entropy bits	TRNG output bits	LRNG behavior
2000 unused IRQs in entropy pool	IRQ: 257 Jitter: 256 CPU: 256	IRQ: 256 Jitter: 16 CPU: 8	256	/dev/random operational /dev/urandom operational `getrandom`(2) operational `wait_for_random_bytes` released `add_random_ready_callback` released `get_random_bytes` operational
105 unused IRQs in entropy pool	IRQ: 105 Jitter: 256 CPU: 256	IRQ: 104 Jitter: 16 CPU: 8	128	/dev/random operational /dev/urandom operational `getrandom`(2) operational `wait_for_random_bytes` released `add_random_ready_callback` released `get_random_bytes` operational
104 unused IRQs in entropy pool	IRQ: 104 Jitter: 256 CPU: 256	IRQ: 103 Jitter: 16 CPU: 8	0	/dev/random blocked /dev/urandom operational `getrandom`(2) operational `wait_for_random_bytes` released `add_random_ready_callback` released `get_random_bytes` operational
0 unused IRQs in entropy pool	IRQ: 0 Jitter: 256 CPU: 256	IRQ: 0 Jitter: 16 CPU: 8	0	/dev/random blocked /dev/urandom operational `getrandom`(2) operational `wait_for_random_bytes` released `add_random_ready_callback` released `get_random_bytes` operational

The following table outlines the runtime reseeding behavior assuming the fast noise sources are configured to deliver zero bits of entropy.

Seed Stage	Noise source data bits	Noise source entropy bits	TRNG output bits	LRNG behavior
2000 unused IRQs in entropy pool	IRQ: 257 Jitter: 256 CPU: 256	IRQ: 256 Jitter: 0 CPU: 0	256	/dev/random operational /dev/urandom operational `getrandom`(2) operational `wait_for_random_bytes` released `add_random_ready_callback` released `get_random_bytes` operational
0 unused IRQs in entropy pool	IRQ: 0 Jitter: 256 CPU: 256	IRQ: 0 Jitter: 0 CPU: 0	0	/dev/random blocked /dev/urandom operational `getrandom`(2) operational `wait_for_random_bytes` released `add_random_ready_callback` released `get_random_bytes` operational

2.2.3 NUMA Systems

To prevent bottlenecks in large systems, the secondary DRNG will be instantiated once for each NUMA node. The instantiations 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 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.

2.2.4 Flexible Design

Albeit the preceding sections look like the DRNG and the management logic are highly interrelated, the LRNG code allows for 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.

2.2.5 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 with insufficient entropy, callers may read random data from it. Since the internal state of the `nonblocking_pool` after initialization must be assumed to be predictable, an attack becomes possible which can be explained by the following simplified scenario. Suppose the internal state of an DNRG instance is known to an observer. It then receives 10 bits unknown to the observer and generates output. The observer may now simply try out all 2^10 possible values the DRNG may have received and compare the resulting output to the actual output. This reveals the unknown bits to the observer and current internal state also becomes known to him. Even if the DRNG instance continues to receive a large amount of entropy in this way, say 200 bits, it will always be predictable and thus insecure, because the observer can guess 20 contributions of 10 bits separately instead of having to brute-force 2^200 values at once. The conclusion from this example is that entropy must be added in quantities equal to the amount of the required computational security level.

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 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.

2.3 LRNG Data Structures

The LRNG uses three main data structures:

- The data from 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

[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.

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. Per default, 128 atomic_t words equaling to 4096 bits are used.

- The deterministic random number generator data structure for the TRNG 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.

2.4 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 8 bits, i.e. the fast moving bits, are used for further processing. To ensure fast processing, these 8 bits are concatenated. After the receipt of 64 time stamps, the time stamp array with all concatenated time stamps is inserted into the LFSR. During boot time until the LRNG reaches the fully seeded level, the 32 least significant bits of the time stamp are directly inserted into the LFSR. Entropy is contained in the variations of the time of events and its time delta variations. Figure 2.1 depicts the time stamp array holding the 8-bit time stamp values.

2. The health tests discussed in section 2.4.4 are performed on each received time stamp where the truncated time stamp value is forwarded to the health test. Unless 64 time stamps have been received, the processing of an interrupt stops now.

3. The time stamp array is processed with an LFSR to mix the data into the entropy pool. The LFSR uses a primitive and irreducible polynomials derived from [10, 12] with a size equal to the number of atomic_t words of 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. The word-wise is applied for data that is word-aligned such as the interrupt time stamp array to ensure a fast operation in a hot code path. The byte-wise processing is applied to non-aligned data.

   (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 with the LFSR polynomial. 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. 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 pointer for the first data word points to the first word of the entropy pool. 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. The LRNG increases the counter of the received interrupt events by the number of healthy interrupts stored in the time stamp array. This counter is translated into an entropy statement when the LRNG wants to know how much entropy is present in the entropy pool. This counter is also adjusted when reading data from the entropy.

5. 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 smallest amount of random numbers generated from the entropy pool 32 bits anyway, i.e. the initially seeded level.

6. If the TRNG 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 TRNG.

7. 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 2.4.1 explains this default value. Note, during boot time, this value is set to 128 bits of entropy.

8. Otherwise, the LRNG triggers a kernel work queue to perform a seeding operation discussed in section 2.6.

The entropy collection mechanism is available right from the start 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.

2.4.1 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.

The entropy of high-resolution time stamps is provided with the fast-moving least significant bits of a time stamp which is supported by the quantitative measurement shown in section 3.2. Although only one bit of entropy is assumed to be delivered with a given time stamp the LRNG uses the 8 least significant bits (LSB) of the time stamp to provide a cushion for ensuring that at any given time stamp there is always at least one bit of entropy collected on all types of environments.

However, the question may be raised of why not use more data of the time stamp, i.e. why not using 32 bits or the full 64 bits of the time stamp to increase that cushion? There main answer is performance. The collection of a time stamp and its injection into the LFSR is performed as part of an interrupt handler. Therefore, the performance of the LRNG in this code section is highly performance-critical. To limit the impact on the interrupt handler, the LRNG concatenates the 8 LSB of 64 time stamps received by the current CPU before those 64 bytes are injected into the LFSR. The performance of this approach is demonstrated with the measurements shown in section 4.2. The second aspect is that the higher bits of the time stamp must always be considered to have zero bits of entropy when considering the worst case of a skilled attacker. As the LRNG cannot identify whether it is under attack by a skilled attacker, it always assumes it is under attack.

The Linux kernel allows unprivileged user space processes to monitor the arrival of interrupts by reading the file /proc/interrupts. Also, assuming a remote attacker connected to the victim system running the LRNG via a low-latency network link, the attacker is able to trigger an interrupt via a network packet and predict the processing of the interrupt and thus the time stamp generation by the LRNG with a certain degree of accuracy. The LRNG uses a high-resolution

[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.

time stamp that executes with nanosecond precision on 1 GHz systems. Local attackers via /proc/interrupts as well as remote attackers via low-latency networks are expected to be measure the occurrence of an interrupt with a microsecond precision. The distance between a microsecond and a nanosecond is 2^{10}. Thus, when the attacker is assumed to predict the interrupt occurrence with a microsecond precision and the time stamp operates with nanosecond precision, 10 bits of uncertainty remains that cannot be predicted by that attacker. Hence, only these 10 bits can deliver entropy.

To ensure the LRNG interrupt handling code has the maximum performance, it processes time stamp values with a number of bits equal to a power of two. Thus, the LRNG implementation uses 8 LSB of the time stamp.

During boot time, the presence of attackers is considered to be very limited as no remote access is yet possible and no local attack applications are assumed to execute. On the other hand, the performance of the interrupt handler is not considered to be very critical during the boot process. Thus, the LRNG uses the 32 LSB of the time stamp that is injected immediately into the LFSR when the time stamp is collected – the LRNG still awards this time stamp one bit of entropy. Once the LRNG is considered to be fully seeded – see section 2.2.1 – the aforementioned runtime behavior of concatenating the 8 LSB of 64 time stamps before mixing them into the LFSR is enabled.

2.4.2 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_archrandom/parameters/archrandom or by setting the kernel command line option of `lrng_archrandom.archrandom`.

2.4.3 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_jent/parameters/jitterrng or by setting the kernel command line option of `lrng_jent.jitterrng`.

2.4.4 Health Tests

The LRNG implements the following health tests:

- Stuck Test
- Repetition Count Test (RCT)
- Adaptive Proportion Test (APT)

Those tests are detailed in the following sections.

Please note that these health tests are only performed for the interrupt noise source. Other noise sources like the Jitter RNG or the CPU-based noise sources

are not covered by these tests as they are fully self-contained noise sources where the LRNG does not have access to the raw noise data and does not include a model of the noise source to implement appropriate health tests. The LRNG considers both as external noise source. Thus, the user must ensure that either those other noise sources implement all health tests as needed or the kernel must be started such that these noise sources are credited with zero bits of entropy. Not crediting any entropy to these other noise sources can be achieved with the following kernel command line options:

- CPU-based noise source: `lrng_base.archrandom=0`
- Jitter RNG: `lrng_base.jitterrng=0`

These options ensure that random data from the noise sources are pulled, but are not credited with any entropy.

The RCT, and the APT health test are only performed when the kernel is booted with `fips=1` and the kernel detects a high-resolution time stamp generator during boot.

In addition, the health tests are only enabled if a high-resolution time stamp is found. Systems with a low-resolution time stamp will not deliver sufficient entropy for the interrupt noise source which implies that also the health tests are not applicable.

Stuck Test The stuck test calculates the first, second and third discrete derivative of the time stamp to be processed by the LFSR. Only if all three values are zero, the received time delta is considered to be non-stuck. The first derivative calculated by the stuck test verifies that two successive time stamps are not equal, i.e. are "stuck". The second derivative calculates that there is no linear repetitive signal.

The third derivative of the time stamp is considered relevant based on the following: The entropy is delivered with the variations of the occurrence of interrupt events, i.e. it is mathematically present in the time differences of successive events. The time difference, however, is already the first discrete derivative of time. Now, if the time difference delivers the actual entropy, the stuck test shall catch that the time differences are not stuck, i.e. the first derivative of the time difference (or the second derivative of the absolute time stamp) shall not be zero. In addition, the stuck test shall ensure that the time differences do not show a linear repetitive signal – i.e. the second discrete derivative of the time difference (or the third discrete derivative of the absolute time stamp) shall not be zero.

Repetition Count Test The LRNG uses an enhanced version of the Repetition Count Test (RCT) specified in SP800-90B [11] section 4.4.1. Instead of counting identical back-to-back values, the input to the RCT is the counting of the stuck values during the processing of received interrupt events. The data that is mixed into the entropy pool is the time stamp. As the stuck result includes the comparison of two back-to-back time stamps by computing the first discrete derivative of the time stamp, the RCT simply checks whether the first discrete derivative of the time stamp is zero. If it is zero, the RCT counter is increased. Otherwise, the RCT counter is reset to zero.

The RCT is applied with $\alpha = 2^{-30}$ compliant to the recommendation of FIPS 140-2 IG 9.8.

During the counting operation, the LRNG always calculates the RCT cut-off value of C. If that value exceeds the allowed cut-off value, the LRNG will trigger the health test failure discussed below. An error is logged to the kernel log that such RCT failure occurred.

This test is only applied and enforced in FIPS mode, i.e. when the kernel compiled with `CONFIG_CONFIG_FIPS` is started with fips=1.

Adaptive Proportion Test Compliant to SP800-90B [11] section 4.4.2 the LRNG implements the Adaptive Proportion Test (APT). Considering that the entropy is present in the least significant bits of the time stamp, the APT is applied only to those least significant bits. The APT is applied to the four least significant bits.

The APT is calculated over a window size of 512 time deltas that are to be mixed into the entropy pool. By assuming that each time stamp has (at least) one bit of entropy and the APT-input data is non-binary, the cut-off value $C = 325$ as defined in SP800-90B section 4.4.2.

This test is only applied and enforced in FIPS mode, i.e. when the kernel compiled with `CONFIG_CONFIG_FIPS` is started with fips=1.

Runtime Health Test Failures If either the RCT, or the APT health test fails irrespective whether during initialization or runtime, the following actions occur:

1. The entropy of the entire entropy pool is invalidated.
2. The TRNG and all secondary DRNGs are reset which imply that they are treated as being not seeded and require a reseed during next invocation.
3. The SP800-90B startup health test are initiated with all implications discussed in section 2.4.4. That implies that from that point on, new events must be observed and its entropy must be inserted into the entropy pool before random numbers are calculated from the entropy pool.

SP800-90B Startup Tests The aforementioned health tests are applied to the first 1,024 time stamps obtained from interrupt events. In case one error is identified for either the RCT, or the APT, the collected entropy is invalidated and the SP800-90B startup health test is restarted.

As long as the SP800-90B startup health test is not completed, all LRNG random number output interfaces that may block will block and not generate any data. This implies that only those potentially blocking interfaces are defined to provide random numbers that are seeded with the interrupt noise source being SP800-90B compliant. All other output interfaces will not be affected by the SP800-90B startup test and thus are not considered SP800-90B compliant.

To summarize, the following rules apply:

- SP800-90B compliant output interfaces
  - `/dev/random`
  - `getrandom(2)` system call

  - `get_random_bytes` kernel-internal interface when being triggered by the callback registered with `add_random_ready_callback`

- SP800-90B non-compliant output interfaces
  - `/dev/urandom`
  - `get_random_bytes` kernel-internal interface called directly
  - `randomize_page` kernel-internal interface
  - `get_random_u32` and `get_random_u64` kernel-internal interfaces
  - `get_random_u32_wait`, `get_random_u64_wait`, `get_random_int_wait`, and `get_random_long_wait` kernel-internal interfaces

2.5 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.

2.6 TRNG 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 TRNG 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.

To support the hash_df operation, the entropy pool is lead by an 8 bit counter variable and a 32 bit requested size variable as defined by the hash_df operation in section 10.3.1 of SP800-90A.

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 processed with the hash_df function. The result of the hash_df function is injected back into the entropy pool using the LFSR described in section 2.4. 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 requested data size, the hash_df output returned to the caller for the TRNG 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_df operation.

Finally, also a 32 bit time stamp indicating the time of the request is mixed into the TRNG. 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 ([3]) and complies with the minimum entropy requirement from BSI TR-02102 ([4]) 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.

2.7 Secondary DRNG Seeding Operation

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

The secondary DRNG seeding operation may trigger a reseeding operation of the TRNG. In this case, the reseeding operation of the TRNG will always leave an emergency level of entropy in the entropy pool to be used exclusively for the TRNG. In addition, the seeding operation of the TRNG 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 2.2.5.

In the worst case, the TRNG 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.

In case the TRNG is not available, the secondary DRNG seeds from the entropy pool in the same manner as the TRNG would do.

2.8 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 TRNG by invoking the DRNG's update function.

2.8.1 Kernel Hardware Random Number Generator Drivers

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

2.8.2 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 added to the entropy pool 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 entropy pool and define the amount of entropy associated with that data.

2.9 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.

2.9.1 /dev/urandom and `get_random_bytes_full`

Users that want to obtain data via the /dev/urandom user space interface or the `get_random_bytes_full` 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 2.6 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.

2.9.2 /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 TRNG.

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.

2.10 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 [9], section 2.3.

2.10.1 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 [9] chapter 4.

2.10.2 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.

2.10.3 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.

2.11 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.

2.12 `get_random_bytes` in Atomic Contexts

The in-kernel API call of `get_random_bytes` 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. When the caller uses `get_random_bytes` the still present ChaCha20 DRNG is used to serve that request instead of the current secondary DRNG. When using the in-kernel API of `get_random_bytes_full`, the caller gets access to the selected DRNG type. However, the caller must be able to handle the fact that this API call can sleep.

The seeding operation of the "atomic DRNG", however, cannot be triggered while `get_random_bytes` is invoked, because the TRNG 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.

Thus to summarize, the kernel function `get_random_bytes` always accesses the "atomic DRNG" whereas the function `get_random_bytes_full` accesses the secondary DRNG instances that are allocated by the switchable DRNG support. This implies that `get_random_bytes_full` must be expected to sleep.

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() is needed for keys that need to stay secret after they are erased from the kernel. For example, any key that will be wrapped and stored encrypted. And session encryption keys: we'd like to know that after the session is closed and the keys erased, the plaintext is unrecoverable to someone who recorded the ciphertext. This function is appropriate for all in-kernel use cases. However, it will always use the ChaCha20 DRNG.

`get_random_bytes_full` This function purpose is identical to `get_random_bytes`. The difference is that this function provides access to all features of the

LRNG including to switchable DRNGs. Yet, this function may sleep and thus is inappropriate for atomic contexts.

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 LRNG is fully seeded.

del_random_ready_callback Delete the registered callback.

get_random_[u32|u64|int|long] These are produced by a cryptographic RNG seeded from get_random_bytes, and so do not deplete the entropy pool as much. These are recommended for most in-kernel operations if the result is going to be stored in the kernel[17].

Specifically, the get_random_int() family do not attempt to do "anti-backtracking". If you capture the state of the kernel (e.g. * by snapshotting the VM), you can figure out previous get_random_int() return values. But if the value is stored in the kernel anyway, this is not a problem.

It is safe to expose get_random_int() output to attackers (e.g. as * network cookies); given outputs 1..n, it's not feasible to predict outputs 0 or n+1. The only concern is an attacker who breaks into the kernel later; the get_random_int() engine is not reseeded as often as the get_random_bytes() one.

For network ports/cookies, stack canaries, PRNG seeds, address space layout randomization, session *authentication* keys, or other applications where the sensitive data is stored in the kernel in plaintext for as long as it's sensitive, the get_random_int() family is just fine.

Consider ASLR. We want to keep the address space secret from an outside attacker while the process is running, but once the address space is torn down, it's of no use to an attacker any more. And it's stored in kernel data structures as long as it's alive, so worrying about an attacker's ability to extrapolate it from the get_random_int() DRNG is silly.

Even some cryptographic keys are safe to generate with get_random_int(). In particular, keys for SipHash are generally fine. Here, knowledge of the key authorizes you to do something to a kernel object (inject packets to a network connection, or flood a hash table), and the key is stored with the object being protected. Once it goes away, we no longer care if anyone knows the key.

wait_for_random_bytes With this function, a synchronous wait until the DRNG is minimally seeded is implemented inside the kernel. This function is used to implement the wait_get_random_[u32|u64|int|long] functions which

[17] This functionality discussion is taken from a patch set sent to the Linux kernel mailing list.

turn the aforementioned get_random_[u32|u64|int|long] functions into potentially sleeping functions.

`prandom_u32` For even weaker applications, see the pseudorandom generator prandom_u32(), prandom_max(), and prandom_bytes(). If the random numbers aren't security-critical at all, these are far cheaper. Useful for self-tests, random error simulation, randomized backoffs, and any other application where you trust that nobody is trying to maliciously mess with you by guessing the "random" numbers.

/dev/random User space interface to provide random data with full entropy – read function may block if insufficient entropy is available. It provides access to a DRNG with prediction resistance as defined in SP800-90A. If the TRNG support is not compiled, /dev/random behaves list the getrandom(2) system call.

/dev/urandom User space interface to provide random data from a constantly reseeded DRNG – the read function will generate random data on demand. It provides access to a DRNG executing without prediction resistance as defined in SP800-90A but is subject to regular re-seeding. 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 1 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 TRNG 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/lrng_type String referencing the DRNG type and the security strength of the DRNG. It also contains a hint whether the LRNG operates SP800-90B compliant, a boolean indicating whether the DRNG is fully seeded with entropy equal to the DRNG security strength, a boolean indicating whether the DRNG is seeded the minimum entropy of 128 bits.

IOCTLs are implemented as documented in random(4).

3 Standards Compliance

3.1 FIPS 140-2 Compliance

FIPS 140-2 specifies entropy source compliance in FIPS 140-2 IG 7.18. This section analyzes each requirement for compliance. The general requirement to comply with SP800-90B [11] is analyzed in section 3.2.

3.1.1 FIPS 140-2 IG 7.18 Requirement For Statistical Testing

The LRNG is provided with the following testing tools:

- Raw Entropy Tests: The tests obtain the raw unconditioned and unprocessed noise information and records it for analysis with the SP800-90B non-IID statistical test tool. The test tool includes the gathering of raw entropy for one execution run as well as for the restart tests required in SP800-90B section 3.1.4. The tool adjusts the data to be processed by the SP800-90B statistical test tool. The test tool provides the SP800-90B minimum entropy values.

- LSFR Tests: A test is available that feeds a monotonic counter to the LFSR to verify that the output of the LFSR does not exhibit statistical weaknesses. The output is processed with the statistical tool dieharder as well as the SP800-90B IID statistical tests.

In particular the first test covers the test requirement of FIPS 140-2 IG 7.18.

3.1.2 FIPS 140-2 IG 7.18 Heuristic Analysis

FIPS 140-2 IG 7.18 requires a heuristic analysis compliant to SP800-90B section 3.2.2. The discussion of this SP800-90B requirement list is given in section 3.2.

3.1.3 FIPS 140-2 IG 7.18 Additional Comment 1

The first test referenced in section 3.1.1 covers this requirement.

The test collects the time stamps of interrupts as they are received by the LRNG. Instead of having these interrupts processed by the LRNG to add them to the entropy pool, they are sent to a user space application for storing them to disk.

The collection of the interrupt data for the raw entropy testing is invoked from the same code path that would otherwise add it to the LRNG entropy pool. Thus, the test collects the exact same data that would otherwise have

been used by the LRNG as noise data. Thus, the testing does not alter the LRNG processing.

However, the tester performing the test should observe the following caveat: the raw entropy data obtained with the user space tool should be stored on "disk space" that will not generate interrupts as otherwise the testing would itself generate new interrupts and thus alter the measurement. For example, a ramdisk can be used to store the raw entropy data while the test is ongoing. On common Linux environments, the path `/dev/shm` is usually a ramdisk that can readily be used as a target for storing the raw entropy data. If that partition is non-existent, the tester should mount a ramdisk or use different backing store that is guaranteed to not generate any interrupts when writing data to it.

3.1.4 FIPS 140-2 IG 7.18 Additional Comment 2

The lowest entropy yield is analyzed by gathering raw entropy data received from interrupts that come in high frequency. In this case, the time stamps would be close together where the variations and thus the entropy provided with these time stamps would be limited.

The extreme case would be to send a flood of ICMP echo request messages with a size of only one byte to the system under test from a neighboring system that has a close proximity with very little network latency. Each ICMP request would trigger an interrupt as it is processed by the network card. The most extreme case can be achieved when executing the LRNG in a virtual machine where the VMM host sends a ping flood to the virtual machine. In this case, network latency would be reduced to a minimum. In the subsequent sections, test results are shown which are generated with an LRNG executing in a virtual machine where the host sends a flood of ICMP echo request messages to trigger a worst case measurement.

The entropy is not considered to degrade when using the hardware within the environmental constraints documented for the used CPU. The online health tests are intended to detect entropy source degradation. In case of online health test failures, section 2.4.4 explains the applied actions.

3.1.5 FIPS 140-2 IG 7.18 Additional Comment 3

The LRNG uses the following conditioning components:

- For collecting of entropy data from noise sources, an LFSR with a primitive and irreducible polynomial is used. This LFSR feeds the received raw entropy data into the entropy pool.

- For reading the entropy pool and compressing the data to the security strength of the used DRNG, the hash_df function specified in SP800-90A [1] section 10.3.1 is used. The ciphers for the hash_df operation are all subject to ACVP testing.

3.1.6 FIPS 140-2 IG 7.18 Additional Comment 4

The restart test is covered by the first test documented in section 3.1.1.

3.1.7 FIPS 140-2 IG 7.18 Additional Comment 6

The entropy assessment usually shows this conclusion – tests performed on Intel x86-based systems show the following conclusions:

The entropy rate for all devices validated with the raw entropy tests outlined in section 3.1.1 show that the minimum entropy values are always above one bit of entropy per four data bits. The data bits are the least significant bits of the time stamp generated by the raw noise.

Assuming the worst case that all other bits in the time delta have no entropy, that entropy value above one bit of entropy applies to one time stamp.

The LRNG continuously gathers time stamps to be combined with an LFSR with a primitive and irreducible polynomial which is entropy preserving. The read function of the LFSR to provide data to the DRNG gathers only as much bits as time stamps were received. For example, if the LRNG only received 16 time stamps, the LRNG will only deliver 2 bytes of data to the DRNG. This effectively implies that the LRNG assumes that one bit of entropy is received per time stamp.

As the LRNG maintains an entropy pool, its entropy content cannot be larger than the pool itself. Thus, the entropy content in the pool after collecting as many time stamps as the entropy pool's size in bits is the maximum amount of entropy that can be held. Yet, as new time stamps are received, they are mixed into the entropy pool. In case the entropy pool is considered to have fully entropy, existing entropy is overwritten with new entropy.

This implies that the LRNG data generated from the entropy pool has (close to) 1 bit of entropy per data bit.

3.1.8 FIPS 140-2 IG 7.18 Additional Comment 9

N/A as the raw entropy is a non-IID source and processed with the non-IID SP800-90B statistical tests as documented in section 3.1.1.

3.2 SP800-90B Compliance

This chapter analyzes the compliance of the LRNG to the SP800-90B [11] standard considering the FIPS 140-2 implementation guidance 7.18 which alters some of the requirements mandated by SP800-90B.

3.2.1 SP800-90B Section 3.1.1

The collection of raw data for the SP800-90B entropy testing documented in section 3.1.1 uses 1,000,000 consecutive time stamps obtained in one execution round.

The gathering of post-LFSR output data using the test documented in section 3.1.1 includes 1,000,000 consecutive 4096 bit blocks, i.e. 1,000,000 snapshots of the LFSR pool. The individual LFSR blocks are concatenated to form a bit stream.

The restart tests documented in section 3.1.1 perform 1,000 restarts collecting 1,000 consecutive time stamps.

3.2.2 SP800-90B Section 3.1.2

The entropy assessment of the raw entropy data including the restart tests follows the non-IID track.

The entropy assessment of the LFSR output data follows the IID track.

3.2.3 SP800-90B Section 3.1.3

Please see section 3.1.7: The entropy of the raw noise source is believed to have more than one bit of entropy per time stamp to allow to conclude that one output block of the LRNG has (close to) one bit of entropy per data bit.

The first test referenced in section 3.1.1 performs the following operations to provide the SP800-90B minimum entropy estimate:

1. Gathering of the raw entropy data of the time stamps.

2. Obtaining the four least significant bits of each time stamp and concatenate them to form a bit stream.

3. The bit stream is processed with the SP800-90B entropy testing tool to gather the minimum entropy.

For example, on an Intel Core i7 Skylake system executing the LRNG in a KVM guest, the SP800-90B tool shows the following minimum entropy values when multiplying the SP800-90B tool bit-wise minimum entropy by four since four bits are processed: 3.543896.

3.2.4 SP800-90B Section 3.1.4

For the restart tests, the raw entropy data is collected for the first 1,000 interrupt events received by the LRNG after a reboot of the operating system. That means, for one collection of raw entropy the test system is rebooted. This implies that for gathering the 1,000 restart samples, the test system is rebooted 1,000 times.

Each restart test round stores its time stamps in an individual file.

After all raw entropy data is gathered, a matrix is generated where each line in the matrix lists the time stamp of one restart test round. The first column of the matrix, for example, therefore contains the first time stamp for each boot cycle of the Linux kernel with the LRNG.

The SP800-90B minimum entropy values column and row-wise is calculated the same way as outlined above:

1. Gathering of the raw restart entropy data of the time stamps.

2. Obtaining the four least significant bits of each time stamp either row-wise or column-wise and concatenate them to form a bit stream. There are 1,000 bit streams row-wise, and 1,000 bit streams column-wise boundary generated.

3. The bit streams are processed with the SP800-90B entropy testing tool to gather the minimum entropy.

In a following step, the sanity check outlined in SP800-90B section 3.1.4.3 is applied to the restart test results. The steps given in 3.1.4.3 are applied.

For example, on an Intel Core i7 Skylake system executing the LRNG in a KVM guest, the SP800-90B tool shows the following minimum entropy values when multiplying the SP800-90B tool bit-wise minimum entropy by four since four bits are processed:

- Using the 4 least significant bits of the time stamps in column-wise assessment – lowest entropy value of all 1,000 column entries: 2.053240
- Using the 4 least significant bits of the time stamps in row-wise assessment – lowest entropy value of all 1,000 column entries: 2.014472
- Sanity check of the 1,000 x 1,000 matrix passes with value of one

With the shown values, the restart test validation passes according to SP800-90B section 3.1.4.

3.2.5 SP800-90B Section 3.1.5

The conditioning component applied to the interrupt noise source contains two parts: The time stamps holding the raw entropy obtained when a new interrupt event arrives are processed by an LFSR. When random bits are to be generated by the noise source, the entire LFSR content is processed with a hash_df function to generate the requested number of bits which in turn are used to (re)seed the TRNG. At the same time, the hash_df output is re-injected into the LFSR to ensure backtracking resistance.

The two parts of the conditioning operation are assessed independently in the following.

LFSR The LSFR operation of the LRNG may be considered as a conditioning component as defined in SP800-90B. Thus, the section 3.1.5 of SP800-90B is relevant.

The input of the LFSR n_{in} is not fixed as follows: The LFSR is invoked for an arbitrary amount of time stamps from interrupt events to collect and compress entropy. Only when a DRNG is required to be seeded, the LFSR is read. The LFSR therefore acts as a mechanism to decouple the raw noise collection from the DRNG seeding. Yet, when reading data from the LFSR with the hash_df function, the hash_df operation generates either 256 bits of data or the amount of entropy considered to be present in the LRNG (i.e. the number of yet unused time stamps held in the entropy pool), whatever is lower. Thus, the basic requirement of SP800-90B behind the requirement for a fixed $n_{in}$is met: the LSFR conditioning component in conjunction with the hash_df operation will return only as much data as entropy was received. Mathematically speaking the following process of data transmission is applied:

$$n_{in_{LFSR}} \Rightarrow n_{out_{LFSR}} = n_{in_{hash\_df}} \Rightarrow n_{out_{hash\_df}}$$

where it is always ensured that:

$$n_{in_{LFSR}} \geq n_{out_{hash\_df}}$$

This implies that the combination of the LFSR and hash_df operation complies with the requirement in section 3.1.5 that "the entropy of the output is at most h_{in}".

The output of the LFSR n_{out} is fixed as follows: The LFSR always generates 4,096 bits of output data.

hash_df The output of the LFSR is processed by the hash_df function to generate random bits to be used as a seed data for the subsequent DRNG.

The input of the hash_df n_{in} is fixed as it processes the LFSR pool. Although it may process the LSFR pool multiple times depending on the amount of output data and the used message digest size, the LFSR pool size does not change.

The output of the hash_df n_{out} is not fixed as it is the minimum of either 256 bits or the number of "unprocessed" time stamps held in the LFSR.

The following hashes are used for the hash_df function depending on the loaded DRNG:

- ChaCha20: SHA-1
- SP800-90A Hash DRBG: SHA-512
- SP800-90A HMAC DRBG: SHA-512
- SP800-90A CTR DRBG: CMAC-AES256 – The author notes that SP800-90B section 3.1.5.1.1 requires the use of an approved FIPS 180 or FIPS 202 hash function, the use of CMAC-AES256 is considered to still meet the requirement as follows: The CMAC-AES is initialized with a static key which will not change. Thus, the CMAC-AES effectively operates like a hash. The LRNG uses this hash-like operation of CMAC-AES to apply it as a replacement for SHA-1 or SHA-2 as used for the hash_df function. The reason is to limit the number of dependencies on the underlying ciphers: When using the CTR DRBG, the LRNG only requires the presence of AES. If for some reason the use of CMAC-AES is not considered as a valid use for the hash_df function, the CMAC-AES can be trivially replaced with a hash by updating the respective entry in `struct lrng_drbg_types[]`.

The narrowest internal width nw of the hash_df conditioning function is equal to n_{out}.

Approach for Calculating Entropy Although the aforementioned sections explain that the input and output sizes may not be fixed, in regular operation they are quasi-fixed. In order to reseed a DRNG, 256 bits of entropy are to be generated from the noise source. Although the LFSR receives interrupt time stamps continuously, only the entropy from 257 time stamps are required as illustrated below. Only when the LFSR has received too little interrupt time stamps to satisfy the 256 bit entropy request, less output data is generated. This commonly happens during boot or at runtime when too much entropy is requested. Though, during boot time, the DRNG will receive a (re)seed with 256 bits of entropy before the LRNG is considered fully operational. Therefore,

the prior boot-time (re)seed events with less entropy may even be disregarded for the entropy assessment.

With the given combination of the LFSR and the hash_df as outlined above, the following approach for the entropy calculation is taken:

- $n_{in_{LFSR}}$ equals to 257 * 8 bits – i.e. 257 time stamps with a size of 8 bits each are received. The reason is that at $n_{out_{hash\_df}}$ is most 256 bits due to the aforementioned fact that $n_{in_{LFSR}} \geq n_{out_{hash\_df}}$. So in case $n_{in_{LFSR}}$ would be lower than 256 bits, the noise source with generate a proportionally lower amount of output data. In case the LFSR receives more than 256 time stamps, these additional time stamps are not considered and their entropy is left in the entropy pool. The LRNG ensures that the LFSR receives 257 bits of entropy from 257 time stamps to account for the loss of entropy in the hash_df.
- $n_{out_{LFSR}}$ is 4096 bits
- nw_{LFSR} is 4096 bits which equals to the internal size of the LFSR entropy pool as well as the period of the primitive and irreducible polynomial.
- $n_{in_{hash\_df}}$ equals to $n_{out_{LFSR}}$
- $n_{out_{hash\_df}}$ is 256 bits in the common case as outlined above.
- nw_{hash_df} equals to $n_{out_{hash\_df}}$

3.2.6 SP800-90B Section 3.1.5.1

The hash_df is considered to be a vetted conditioning component. Thus the entropy rate of the hash_df output is calculated as follows using the aforementioned variables for the hash_df function. In addition, the following consideration applies:

- The entropy content of the input h_{in}: The input entropy of the hash_df is equal to output entropy of the LFSR considering that the entire LSFR entropy pool is processed by the hash_df.

When using these values to calculate the Output_Entropy using the minimum estimated entropy, the following lower boundary of the Output_Entropy value is calculated:

1. $P_{high} = 2^{-257}$ and $P_{low} = \frac{1-2^{-257}}{2^{4096}-1}$
2. $n = min(256, 256) = 256$
3. $\psi = 2^{4096-256} \cdot \frac{1-2^{-257}}{2^{4096}-1} + 2^{-257} \approx \frac{1-2^{-257}}{2^{256}} + 2^{-257} \approx 2^{-256}$
4. $U = 2^{4096-256} + \sqrt{2 \cdot 256 \cdot 2^{4096-256} \cdot ln(2)} = \sqrt{2^9 \cdot 2^{3840} \cdot ln(2)} = 2^{\frac{3849}{2}} \cdot \sqrt{ln(2)} \approx 2^{1926}$
5. $\omega = 2^{1926} \times \frac{1-2^{-256}}{2^{4096}-1} \approx \frac{1-2^{-256}}{2^{2170}}$
6. $\omega < \psi \Rightarrow h_{out} = -log_2(\psi) = 256$

Thus, the output entropy of the hash_df is 256 bits when processing 257 bits of input entropy from the LFSR. This is the output entropy from the combined consideration of the LSFR and the hash_df.

The calculation allows the following conclusion to be drawn considering that the output size of the hash_df may be variable depending on the amount of entropy present: when the LFSR is considered to have h_{LFSR} bits of entropy, the entropy of the output of the hash_df can be characterized as:

$$h_{hash\_df} = min(h_{LFSR} - 1, n_{out_{hash\_df}})$$

3.2.7 SP800-90B Section 3.1.5.2

As the LFSR is considered to be a non-vetted conditioning component, the entropy rate of the LFSR output is calculated as follows using the aforementioned variables for the LFSR. In addition, the following consideration applies:

- The entropy content of the input h_{in}: The non-IID SP800-90B entropy assessment of the raw input data discussed in section 3.2.3 is (and shall be) at least 1 bit of entropy per time stamp. When using high resolution time stamps with a frequency of 1GHz or more, the assumed entropy is much larger than 1 bit. Yet, for a worst case assessment presented in this section the 257 time stamps are assumed to deliver at least 257 bits of entropy.
- The obtained entropy estimate h': more than 1 bit for a time stamp.

When using these values to calculate the Output_Entropy using the minimum estimated entropy in one time stamp (1 bit of entropy per time stamp), the following lower boundary of the Output_Entropy value is calculated:

1. $P_{high} = 2^{-257}$ and $P_{low} = \frac{1-2^{-257}}{2^{2056}-1}$
2. $n = min(4096, 4096) = 4096$
3. $\psi = 2^{2056-4096} \cdot \frac{1-2^{-257}}{2^{2056}-1} + 2^{-257} \approx \frac{1-2^{-257}}{2^{4096}} + 2^{-257} \approx 2^{-257}$
4. $U = 2^{2056-4096} + \sqrt{2 \cdot 4096 \cdot 2^{2056-4096} \cdot ln(2)} = 2^{-2040} + \sqrt{2^{13} \cdot 2^{-2040} \cdot ln(2)} = 2^{-2040} + 2^{-\frac{2027}{2}} \cdot \sqrt{ln(2)} \approx 2^{-1013}$
5. $\omega = 2^{-1013} \times \frac{1-2^{-257}}{2^{2056}-1} \approx \frac{1-2^{-257}}{2^{3069}}$
6. $\omega < \psi \Rightarrow h_{out} = -log_2(\psi) = 257$

Thus, the LFSR possesses the following entropy based on section 3.1.5.2 SP800-90B:

$$h_{out} = min(257, 0.999 \cdot 4096, 3.543896 \times 4096) = 257$$

The output block of the LFSR which received 257 time stamps that are assumed to have one bit of entropy each is awarded with 257 bits of entropy which is fed into the hash_df.

3.2.8 SP800-90B Section 3.1.6

The LRNG uses the following noise sources:

- The noise source of the timing of the occurrence of interrupts. The entire SP800-90B analysis covers this one noise source. Thus, the requirements in this section for the interrupt noise source are trivially met.
- The Jitter RNG: This noise source is a complete stand-alone noise source whose compliance to SP800-90B is vetted independently. If the user considers this noise source to be not SP800-90B compliant, it may credit it with zero bits of entropy as outlined in section 2.4.4.
- The CPU-based noise source like Intel RDRAND: Like the Jitter RNG, this noise source is a complete stand-alone noise source whose SP800-90B compliance is vetted independently. If the user considers this noise source to be not SP800-90B compliant, it may credit it with zero bits of entropy as outlined in section 2.4.4.

All random data from all noise sources are concatenated to seed the DRNGs as allowed by SP800-90C [2] section 5.3.4.

This explanation shows that the assessed noise source of the occurrence of interrupt does not have "additional noise sources" in terms of SP800-90B section 3.1.6.

3.2.9 SP800-90B Section 3.2.1 Requirement 1

This entire document is intended to provide the required analysis.

3.2.10 SP800-90B Section 3.2.1 Requirement 2

This entire document in general and chapter 3 in particular is intended to provide the required analysis.

3.2.11 SP800-90B Section 3.2.1 Requirement 3

There is no specific operating condition other than what is needed for the operating system to run since the noise source is a complete software-based noise source.

The only dependency the noise source has is a high-resolution timer which does not change depending on the environmental conditions.

3.2.12 SP800-90B Section 3.2.1 Requirement 4

This document explains the architectural security boundary.

The boundary of the implementation is the source code files provided as part of the software delivery. This source code contains API calls which are to be used by entities using the LRNG.

3.2.13 SP800-90B Section 3.2.1 Requirement 5

The output of the LFSR as processed by the hash_df is the output of the interrupt noise source. I.e. the entropy pool maintained by the LFSR holds the data that is given to the DRNG when requesting seeding.

The noise source output without the LFSR is accessed with specific tools which add interfaces that are not present and thus not usable when employing the LRNG in production mode. These additional interfaces are used for gathering the data used for the analysis documented in section 3.2.3. These interfaces perform the following operation:

1. Switch the LRNG into raw entropy generation mode. This implies that each raw entropy event is fed to the raw entropy collection interface and not processed by the LFSR or otherwise used.

2. When an interrupt event is received, forward the time stamp holding the entropy to a ring buffer. This operation is performed repeatedly until the ring buffer is full or the user space application read that ring buffer.

3. When an application requests the reading of the ring buffer, the data is extracted from the kernel and the ring buffer is cleared.

With this approach, the actual interrupt events which would be processed by the LRNG are obtained.

The kernel interface is only present if the kernel is compiled with the option `CONFIG_LRNG_TESTING`. This option should not be set in production kernels.

3.2.14 SP800-90B Section 3.2.1 Requirement 6

Please see section 3.1.3 for details how and why the raw entropy extraction does not substantially alter the noise source behavior.

3.2.15 SP800-90B Section 3.2.1 Requirement 7

See section 3.2.4 for a description of the restart test.

3.2.16 SP800-90B Section 3.2.2 Requirement 1

This entire document provides the complete discussion of the noise source.

3.2.17 SP800-90B Section 3.2.2 Requirement 2

N/A - not mandated by FIPS IG 7.18. The lowest entropy yield is analyzed with the lower boundary of the raw entropy assessment.

3.2.18 SP800-90B Section 3.2.2 Requirement 3

See sections 3.2.6 and 3.2.7 for a discussion of the entropy provided by the interrupt noise source.

A stochastic model is not provided.

3.2.19 SP800-90B Section 3.2.2 Requirement 4

The noise source is expected to execute in the kernel address space. This implies that the operating system process isolation and memory separation guarantees that adversaries cannot gain knowledge about the LRNG operation.

3.2.20 SP800-90B Section 3.2.2 Requirement 5

The output of the noise source is non-IID as it rests on the execution time of a fixed set of CPU operations and instructions.

3.2.21 SP800-90B Section 3.2.2 Requirement 6

The noise source generates the data via the hash_df generation function as outlined in section 3.2.5.

Although the hash_df commonly generates a fixed-length string, this string length may be reduced by the amount of available entropy as outlined in section 3.2.6.

3.2.22 SP800-90B Section 3.2.2 Requirement 7

N/A as no additional noise source is implemented with the interrupt noise source.

Though, the LRNG employs complete self-contained other noise sources which may be compliant to SP800-90B by itself. To seed the DRNG maintained by the LRNG, the output of all noise sources are concatenated compliant to SP800-90C [2] section 5.3.4.

3.2.23 SP800-90B Section 3.2.3 Requirement 1

The conditioning component is the LFSR. See section 3.2.5 for a discussion of the input and output sizes.

3.2.24 SP800-90B Section 3.2.3 Requirement 2

The LFSR is a non-vetted conditioning component and thus not subject to this requirement.

The used hash implementations for the hash_df functions are all ACVP-testable.

For the CMAC-AES, a static key is used. This key is set during the initialization of the CMAC-AES instance by reading an uninitialized variable and thus random memory data. However, that key is not required to have a specific security strength. By using a static key, the CMAC-AES is turned into a hash.

3.2.25 SP800-90B Section 3.2.3 Requirement 3

The CMAC-AES key is fixed and set during initialization.

3.2.26 SP800-90B Section 3.2.3 Requirement 4

The CMAC-AES key is not credited to have any entropy. The key is derived from data not otherwise processed by the conditioning element.

3.2.27 SP800-90B Section 3.2.3 Requirement 5

The conditioning component is the LFSR. See section 3.2.7 for a discussion of the narrowest internal width and the output block size.

An LFSR with a primitive and irreducible polynomial is considered to not diminish the entropy.

The LFSR processes the non-IID data of the 64 bit time stamps values word-wise. That means that one LFSR operation is performed for each received time stamp value. This approach is not adversely affected when the input data to the LFSR is non-IID.

The polynomial is tested with magma for being primitive and irreducible.

3.2.28 SP800-90B Section 3.2.4 Requirement 1

Test tools for measuring raw entropy are provided at the LRNG web page. These tools can be used by everybody without further knowledge of the LRNG.

3.2.29 SP800-90B Section 3.2.4 Requirement 2

The operation of the test tools for gathering raw data are discussed in section 3.2.3. This explanation shows that the raw unconditioned data is obtained.

3.2.30 SP800-90B Section 3.2.4 Requirement 3

The provided tools for gathering raw entropy contains exact steps how to perform the tests. These steps do not require any knowledge of the noise source.

3.2.31 SP800-90B Section 3.2.4 Requirement 4

The raw entropy tools can be executed on the same environment that hosts the LRNG. Thus, the data is generated under normal operating conditions.

The set of test tools contains the LFSR test injecting a monotonic counter and obtaining its output to demonstrate that the LFSR generates IID data.

3.2.32 SP800-90B Section 3.2.4 Requirement 5

The raw entropy tools can be executed on the same environment that hosts the LRNG. Thus, the data is generated on the same hardware and operating system that executes the LRNG.

3.2.33 SP800-90B Section 3.2.4 Requirement 6

The test tools are publicly available at LRNG web page allowing the replication of any raw entropy measurements.

3.2.34 SP800-90B Section 3.2.4 Requirement 7

Please see section 3.1.3 for details how and why the raw entropy extraction does not substantially alter the noise source behavior.

3.2.35 SP800-90B Section 4.3 Requirement 1

The implemented health tests comply with SP800-90B sections 4.4 as described in section 3.2.44.

3.2.36 SP800-90B Section 4.3 Requirement 2

When either health test fails, the kernel:

- Emits a failure log,
- Resets the noise source, and
- Restarts the SP800-90B startup health tests.

This implies that no data is produced by the LRNG (including its DRNG) when using the SP800-90B compliant external interfaces.

Both health test failures are considered permanent failures and thus trigger a full reset.

3.2.37 SP800-90B Section 4.3 Requirement 3

The following false positive probability rates are applied:

- RCT: The false positive rate is $\alpha = 2^{-30}$ and therefore complies with the recommended false positive probability.
- APT: The cut-off value is set to 325 compliant to SP800-90B section 4.4.2 for non-binary data at a significance level of $\alpha = 2^{-30}$ with time stamp is assumed to at least provide one bit of entropy, i.e. $H = 1$.

3.2.38 SP800-90B Section 4.3 Requirement 4

The LRNG applies a startup health test of 1,024 noise source samples. Additional tests are applied. The collected noise source samples are re-used for the generation of random numbers if the startup test was successful.

3.2.39 SP800-90B Section 4.3 Requirement 5

The noise source supports on-demand testing in the sense that the caller may restart the kernel.

3.2.40 SP800-90B Section 4.3 Requirement 6

The health tests are applied to the raw, unconditioned time stamp data directly obtained from the noise source before they are injected into the LFSR conditioning component.

3.2.41 SP800-90B Section 4.3 Requirement 7

The health tests are documented with section 2.4.4.

The tests are executed as follows:

- During startup, the RCT and the APT are applied to 1,024 samples. The startup test can be triggered again when the caller reboots the kernel.

- At runtime, the RCT is applied to each received time stamp. The APT-collects 512 time stamps. The APT is calculated over all 512 time stamps. The result of both tests is enforced the fact that only with a successful testing, the entropy counter for the LFSR is increased by 512. If the test fails, the entire LRNG is reset to drop all existing entropy and the startup testing is performed again.

3.2.42 SP800-90B Section 4.3 Requirement 8

There are no currently known suspected noise source failure modes.

3.2.43 SP800-90B Section 4.3 Requirement 9

N/A as the noise source is pure software. The software is expected to execute on hardware operating in its defined nominal operating conditions.

3.2.44 SP800-90B Section 4.4

The health tests described in section 2.4.4 are applicable to cover the requirements of SP800-90B health tests.

The SP800-90B compliant health tests are implemented with the following rationale:

RCT The Repetition Count Test implemented by the LRNG compares two back-to-back time stamps to verify that they are not identical. If the number of identical back-to-back time stamps reaches the cut-off value of 30, the RCT test raises a failure that is reported and causes a reset the LRNG. The RCT uses the a cut-off value that is based on the following: $\alpha = 2^{-30}$ compliant to FIPS 140-2 IG 9.8 and compliant to SP800-90B which mandates this value to be in the range $2^{-20} \leq \alpha \leq 2^{-40}$. In addition, one time stamp is assumed to at least provide one bit of entropy, i.e. $H = 1$. When applying these values to the formula given in SP800-90B section 4.4.1, the cut-off value of 30 is calculated.

When the RCT passes, the counter is set to zero for the next time delta to arrive. In mathematical terms, the verification of back-to-back values being not identical is the calculation of the first discrete derivative of the time stamp to show that it is not zero. In addition, the LRNG enhances the RCT by calculating also the second and third discrete derivative of the time stamp to be injected into the entropy pool by the LFSR. With that, up to 8 consecutive time stamp values are assessed. All derivatives must always be non-zero in order to pass the RCT. If one discrete derivative shows a zero, the RCT counter is increased. Thus, the addition of the second and third derivative makes the RCT even more conservative. Hence, the first discrete derivative is considered to be identical to the "approved" RCT specified in SP800-90B section 4.4. In addition, linear and exponential patterns are identified with the second and third discrete derivative, respectively. As the additional pattern recognition do not invalidate the mandatory pattern recognition, this RCT approach therefore is considered to be an enhanced version of the "approved" RCT and thus meets the requirement (a) of SP800-90B section 4.5.

APT The LRNG implements the Adaptive Proportion Test as defined in SP800-90B section 4.4.2. As explained in other parts of the document, one time stamp value is assumed to have (at least) one bit of entropy. Thus, the cut-off value for the APT is 325 compliant to SP800-90B section 4.4.2 for non-binary data with a significance level of $\alpha = 2^{-30}$. The APT is calculated using the four least significant bits of the time stamp. During initialization of the APT, a time stamp is set as a base. All subsequent time stamps are compared to the base time stamp. If both values are identical, the APT counter is increased by one. The window size for the APT is 512 time stamps. The implementation therefore provides an "approved" APT.

3.3 NIST Clarification Requests

In addition to complying with the requirements of FIPS 140-2 and SP800-90B, NIST requests the clarification of the following questions.

3.3.1 Sensitivity of Interrupt Timing Measurements

The question that needs to be answered is whether the logic that measures the interrupt timing is sensitive enough to pick up the variances of the interrupt timing.

The sensitivity implies that timing variations are picked up and measured. This is enforced by the stuck test enforced on each interrupt time stamp. That stuck test requires that the first, second and third discrete derivative of the time stamp must always be non-zero to accept that time stamp. Therefore, the time stamp must vary for the received and processed interrupts which implies that the LRNG health test ensures that the sensitivity of the time stamp mechanism is sufficient.

3.3.2 Dependency Between Interrupt Timing Measurements

Another question that is raised by NIST asks for a rationale why there are no dependencies between individual Jitter measurements.

The interrupts are always created by either explicit or implicit human actions. The LRNG measures the time stamp of the occurrence of these interrupts. Thus, the LRNG measures the effects of operations triggered by human interventions. With the presence of a high-resolution time stamp that operates in the nanosecond range and the assumption that only one bit of entropy is present in one nanosecond time stamp of one interrupt event, the dependency discussion therefore focuses on the one (or maybe up to four) least significant bit of the nanosecond time stamp. With such high-resolution time stamps and considering that only the least significant bit(s) is/are relevant for the LRNG, dependencies are considered to be not present for these bits.

3.4 SP800-90B Compliant Configuration

In order to use the LRNG SP800-90B compliant, the following configurations and settings must be made. These settings are cover requirements for the compile-time options found in the kernel configuration file `.config` of the running kernel. In addition, runtime configurations are to be considered as well.

The following compile-time settings must be observed:

- `CONFIG_LRNG` must be set to Y.
- `CONFIG_LRNG_HEALTH_TESTS` must be set to Y.
- `CONFIG_LRNG_ARCHRANDOM_TRUST_CPU_STRENGTH` must not be set unless the CPU-based noise source (e.g. RDSEED or RDRAND on Intel) have an SP800-90B compliant entropy assessment and comply with all requirements from SP800-90B.
- `CONFIG_RANDOM_TRUST_BOOTLOADER` must not be set unless the data provided by the boot loader have an SP800-90B compliant entropy assessment and comply with all requirements from SP800-90B.
- All kernel code that uses the `add_hwgenerator_randomness` must either invoke the function with a zero for the `entropy_bits` parameter or must have an SP800-90B compliant entropy assessment and comply with all requirements from SP800-90B. For example, this call is invoked by the ATH9K driver or the hardware random number generator driver framework.

The following requirements apply to the runtime configuration:

- The kernel must be booted with the kernel command line option of `fips=1` to enable the SP800-90B health test.
- The kernel must be booted with the kernel command line option of `lrng_archrandom.archrandom=0` unless the CPU-based noise source (e.g. RDSEED or RDRAND on Intel) have an SP800-90B compliant entropy assessment and comply with all requirements from SP800-90B.
- The kernel must be booted with the kernel command line option of `lrng_jent.jitterrng=0` unless the Jitter RNG noise source has an SP800-90B compliant entropy assessment and comply with all requirements from SP800-90B[18].

To verify that the SP800-90B compliance is achieved, the file /proc/sys/kernel/random/lrng_type provides an appropriate status indicator.

SP800-90A and SP800-90B Compliant Configuration

To achieve a compliant configuration to SP800-90A and SP800-90B, the following requirements must be met:

- All requirements for SP800-90B documented in section 3.4 must be met.
- The Linux kernel configuration option of `CONFIG_LRNG_DRBG` must either be set to Y or to M. If it is set to M (compile the code as loadable kernel module), the kernel module lrng_drbg.ko must be loaded into the kernel before any caller to the LRNG requiring SP800-90A compliance is active.

[18] At the time of writing, the user space Jitter RNG is SP800-90B compliant. Patches are developed to make the in-kernel variant SP800-90B compliant as well.

3.5 Reuse of SP800-90B Analysis

To reuse the SP800-90B analysis provided in this document the following steps must be performed on the target platform:

1. Obtain raw noise data through the raw noise source interface on the intended target platform as explained in section 3.2.3. The obtained raw noise data must be processed by the SP800-90B tool to obtain an entropy rate which must be above 1 bit of entropy per time delta.

2. Obtain the restart noise data through the raw noise source interface on the intended target platform as explained in section 3.2.3. The obtained raw noise data must be processed by the SP800-90B tool to verify:

   (a) the sanity test to apply to the noise restart data must pass, and

   (b) the minimum of the row-wise and column-wise entropy rate must not be less than half of the entropy rate from measurement (1) and the entropy assessment of the noise source based on the restart data must be at least 1 bit of entropy per time stamp.

If these steps are successfully mastered the user would now satisfy all SP800-90B criteria and thus does not need to prepare his own SP800-90B analysis since the document we discuss here covers all other aspects of the SP800-90B analysis.

3.6 SP800-90C

The specification of SP800-90C as provided in [2] 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 [2]. 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 [2] [19]. Although the entropy source is not always "live" as referenced in [2], 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 [2].

  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

[19] 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 [2]. As discussed below, the "full entropy" definition of section 5.2 and hence the process specified in 9.4.2 from [2] 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 [2]. 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 output of the different noise sources maintained by the LRNG are processed as follows which shows full compliance to section 5.3.4:

  - The interrupt noise source, the Jitter RNG and the CPU-based noise source outputs are all concatenated with a time stamp. This concatenated bit stream is the seed data used to seed the DRNG.

  - Data obtained from architecture-specific noise sources via the `add_hwgenerator_randomness` API call is inserted into the entropy pool like the interrupt data.

The requirements of the security of an RNG defined in section 4.1 of [2] are considered to be covered as follows:

1. The entropy source of the interrupt noise source complies with SP800-90B [11] as assessed in section 3.2. For the CPU noise sources, no statement can be made as no access to the design and implementations are given. The Jitter RNG noise source provides its own self-contained SP800-90B assessment.

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 [2], 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 [2].

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 [2] 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 [11] and chapter 5 [2], the interrupt noise source implemented by the LRNG is subject to a health test. This health tests are documented in section 2.4.4.

Chapter 7 [2] 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 [2] where each noise source is accessed to obtain a bit stream and a value of the assessed entropy.

3.7 AIS 20 / 31

The German BSI defines construction methods of RNGs with AIS 20/31 [5]. 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 TRNG 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 φ and output function ψ are provided with the chosen DRNG. By using the TRNG to serve /dev/random, the LRNG ensures that each random number generated by the TRNG 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.3. It uses a DRNG for the state transition function φ and output function ψ to ensure enhanced backward secrecy which is the prerequisite for a DRG.3.

3.7.1 NTG.1 Compliant Configuration

To ensure the LRNG is operated as an NTG.1 following the requirements of AIS 31, the following kernel compile time options must be set:

- `CONFIG_LRNG` must be set to Y.
- `CONFIG_LRNG_HEALTH_TESTS` must be set to Y to enable at least the basic health tests.
- `CONFIG_LRNG_ARCHRANDOM_TRUST_CPU_STRENGTH` must not be set unless the CPU-based noise source (e.g. RDSEED or RDRAND on Intel) is accepted by BSI to deliver entropy.
- `CONFIG_RANDOM_TRUST_BOOTLOADER` must not be set unless the data provided by the boot loader is accepted by BSI to deliver entropy.
- All kernel code that uses the `add_hwgenerator_randomness` must either invoke the function with a zero for the `entropy_bits` parameter or must be accepted by BSI to deliver the amount of entropy defined in the function invocation. For example, this call is invoked by the ATH9K driver or the hardware random number generator driver framework.

The following requirements apply to the runtime configuration:

- The kernel must be booted with the kernel command line option of `lrng_archrandom.archrandom=0` unless the CPU-based noise source (e.g. RDSEED or RDRAND on Intel) is accepted by BSI to deliver entropy.
- The kernel must be booted with the kernel command line option of `lrng_jent.jitterrng=0` unless the Jitter RNG noise source is accepted by BSI to deliver entropy.

4 LRNG Comparison to legacy /dev/random

Tests to compare the LRNG with the legacy /dev/random are conducted to analyze whether the LRNG brings benefits over the legacy implementation.

4.1 Time Until Fully Initialized

The legacy /dev/random implementation feeds all entropy directly into the CRNG until the kernel log message is recorded that the CRNG 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 LRNG 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 1: Time until fully initialized -- LRNG using all noise sources

```
$ dmesg | grep "LRNG minimally seeded"
[    1.718705] lrng_pool: LRNG minimally seeded with 128 bits of entropy
--- 0 sm@x86-64 ~ ------------------------------------------------------------
```

```
$ dmesg | grep "LRNG fully seeded"
[    2.056685] lrng_pool: LRNG fully seeded with 256 bits of entropy
--- 0 sm@x86-64 ~ ---------------------------------------------------------------
$ dmesg | grep "random: crng init done"
[   20.932050] random: random: crng init done
```

The test shows that the TRNG is minimally seeded 1.7 seconds after boot. This is around the time when the initramfs is started. The TRNG is fully seeded 2 seconds after boot which is long 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 TRNG 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 CRNG 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.archrandom=0` and `lrng_jent.jitterrng=0` to disable the fast noise sources.

Listing 2: Time until fully initialized -- LRNG using only interrupt noise source

```
$ cat /sys/module/lrng_archrandom/parameters/archrandom
0
--- 0 sm@x86-64 ~ ---------------------------------------------------------------
$ dmesg | grep "LRNG minimally seeded"
[    1.683981] lrng_pool: LRNG minimally seeded with 128 bits of entropy
--- 0 sm@x86-64 ~ ---------------------------------------------------------------
$ dmesg | grep "LRNG fully seeded"
[    2.110482] lrng_pool: LRNG fully seeded with 256 bits of entropy
--- 0 sm@x86-64 ~ ---------------------------------------------------------------
[    3.075414] lrng_sdrng: force reseed of secondary DRNG on node 0
```

Even when the fast noise sources are disabled, the LRNG is minimally and fully initialized at the time the initramfs started.

During all testing, the LRNG was fully seeded before user space injected the seed data into /dev/random as mandated by the legacy /dev/random implementation. This point in time is identifiable with the forced reseeding of the secondary DRNG. 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.

4.2 Interrupt Handler Performance

The LRNG is invoked from the interrupt handler. Therefore, it is mandatory that the code executed by the interrupt handler is as fast as possible. To illustrate the performance, the following measurement is made. The execution time in CPU cycles is measured on one particular test system. Since the cycle count is subject to some variations, an average cycle count is calculated.

RNG Options	Average Cycle Count
LRNG with functionality compliant to legacy /dev/random and using 8 LSB of time stamp	65
LRNG with health tests enabled, but no SP800-90B compliance and using 8 LSB of time stamp	195
LRNG with SP800-90B compliant health tests	230
Legacy /dev/random implementation	97

4.3 /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 Whiskey Lake. CPU[20], the following read speeds using the getrandom(2) system call are obtained with different read sizes indicated in the following tables. These numbers give an indication on how much one DRNG performs better over another[21] and are presented in table 1. 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.

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

[21]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.

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	4.7 MB/s
HMAC DRBG	SHA-512	AVX2	32 bytes	11.6 MB/s
HMAC DRBG	SHA-512	AVX2	64 bytes	23.3 MB/s
HMAC DRBG	SHA-512	AVX2	128 bytes	38.3 MB/s
HMAC DRBG	SHA-512	AVX2	4096 bytes	92.1 MB/s
Hash DRBG	SHA-512	C	64 bytes	27.9 MB/s
Hash DRBG	SHA-512	AVX2	16 bytes	12.5 MB/s
Hash DRBG	SHA-512	AVX2	32 bytes	25.0 MB/s
Hash DRBG	SHA-512	AVX2	64 bytes	49.4 MB/s
Hash DRBG	SHA-512	AVX2	128 bytes	79.6 MB/s
Hash DRBG	SHA-512	AVX2	4096 bytes	217.8 MB/s
CTR DRBG	AES-256	C	16 bytes	15.4 MB/s
CTR DRBG	AES-256	AES-NI	16 bytes	21.8 MB/s
CTR DRBG	AES-256	AES-NI	32 bytes	38.2 MB/s
CTR DRBG	AES-256	AES-NI	64 bytes	81.8 MB/s
CTR DRBG	AES-256	AES-NI	128 bytes	160.6 MB/s
CTR DRBG	AES-256	AES-NI	4096 bytes	1.203 GB/s
ChaCha20	ChaCha20	C	16 bytes	37.6 MB/s
ChaCha20	ChaCha20	C	32 bytes	76.9 MB/s
ChaCha20	ChaCha20	C	64 bytes	119.8 MB/s
ChaCha20	ChaCha20	C	128 bytes	191.1 MB/s
ChaCha20	ChaCha20	C	4096 bytes	550.3 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 1: LRNG /dev/urandom performance on 64-bit

In addition, table 2 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	1.4 MB/s
HMAC DRBG	SHA-512	C	32 bytes	2.1 MB/s
HMAC DRBG	SHA-512	C	64 bytes	5.5 MB/s
HMAC DRBG	SHA-512	C	128 bytes	9.0 MB/s
HMAC DRBG	SHA-512	C	4096 bytes	22.8 MB/s
Hash DRBG	SHA-512	C	16 bytes	3.6 MB/s
Hash DRBG	SHA-512	C	32 bytes	7.2 MB/s
Hash DRBG	SHA-512	C	64 bytes	14.5 MB/s
Hash DRBG	SHA-512	C	128 bytes	22.5 MB/s
Hash DRBG	SHA-512	C	4096 bytes	46.3 MB/s
CTR DRBG	AES-256	AES-NI	16 bytes	10.3 MB/s
CTR DRBG	AES-256	AES-NI	32 bytes	22.7 MB/s
CTR DRBG	AES-256	AES-NI	64 bytes	45.5 MB/s
CTR DRBG	AES-256	AES-NI	128 bytes	84.2 MB/s
CTR DRBG	AES-256	AES-NI	4096 bytes	397.4 MB/s
ChaCha20	ChaCha20	C	16 bytes	18.8 MB/s
ChaCha20	ChaCha20	C	32 bytes	38.0 MB/s
ChaCha20	ChaCha20	C	64 bytes	61.9 MB/s
ChaCha20	ChaCha20	C	128 bytes	102.5 MB/s
ChaCha20	ChaCha20	C	4096 bytes	346.5 MB/s
Legacy /dev/random	SHA-1	C	10 bytes	9.4 MB/s
Legacy /dev/random	ChaCha20	C	16 bytes	16.8 MB/s
Legacy /dev/random	ChaCha20	C	32 bytes	32.9 MB/s
Legacy /dev/random	ChaCha20	C	64 bytes	43.3 MB/s
Legacy /dev/random	ChaCha20	C	128 bytes	61.7 MB/s
Legacy /dev/random	ChaCha20	C	4096 bytes	153.2 MB/s

Table 2: 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 2.9.1.

4.4 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 3: 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.

4.5 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`[22] 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.

[22] 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`.

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*. Revision 1 edition, 2015.

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

[3] 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.

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

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

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

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

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

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

[10] Wayne Stahnke. *Primitive Binary Polynomials*. 1973.

[11] Meltem Sönmez Turan, Elaine Barker, John Kelsey, Kerry A. McKay, Mary L. Baish, and Mike Boyle. *NIST Special Publication 800-90B Recommendation for the Entropy Sources Uses for Random Bit Generation*. 2018.

[12] Roy Ward and Tim Molteno. *Table of Linear Feedback Shift Registers*. October 26, 2007.

D License

IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF NOT ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.