Linux Random Number Generator – A New Approach

Stephan Müller <smueller@chronox.de>

October 16, 2020

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.

Contents

List of Figures

List of Tables

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 [8], the behavior of the legacy /dev/random in virtual environments [7] and presentations on /dev/random such as [6] 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,

[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.

- 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]

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.

[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.

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 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

[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.

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. 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

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

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 Advantages Introduced by LRNG

The LRNG implements at least all features of the legacy /dev/random such as NUMA-node-local DRNGs. The following advantages compared to the legacy /dev/random implementation are present:

- Sole use of crypto for data processing:
  - Exclusive use of a hash operation for conditioning entropy data with a clear mathematical description as given section 2.2 – non-cryptographic operations like LFSR are not used.
  - The LRNG uses only properly defined and implemented cryptographic algorithms unlike the use of the SHA-1 transformation in the legacy /dev/random implementation that is not compliant with SHA-1 as defined in FIPS 180-4.
  - Hash operations use NUMA-node-local hash instances to benefit large parallel systems.
  - LRNG uses limited number of data post-processing steps as documented in section 2.2 compared to the large variation of different post-processing steps in the legacy /dev/random implementation that have no apparent mathematical description (see section 4.5).
- Performance
  - Faster by up to 75% in the critical code path of the interrupt handler depending on data collection size configurable at kernel compile time - the default is about equal in performance with existing /dev/random as outlined in section 4.2.
  - Configurable data collection sizes to accommodate small environments and big environments via `CONFIG_LRNG_COLLECTION_SIZE`.

  - Entropy collection using an almost never contended lock to benefit large parallel systems – worst case rate of contention is the number of DRNG reseeds, usually the number of potential contentions per 10 minutes is equal to number of NUMA nodes.
  - ChaCha20 DRNG is significantly faster as implemented in the legacy /dev/random as demonstrated with table 2.
  - Faster entropy collection during boot time to reach fully seeded level, including on virtual systems or systems with SSDs as outlined in section 4.1.

- Testing
  - Heuristic entropy estimation is based on quantitative measurements and analysis following SP800-90B and not on coincidental underestimation of entropy applied by the legacy /dev/random as outlined in [8] section 4.4.
  - Power-on self tests for critical deterministic components (ChaCha20 DRNG, software hash implementation, and entropy collection logic) not already covered by power-up tests of the kernel crypto API as documented in section 2.14.
  - Availability of test interfaces for all operational stages of the LRNG including boot-time raw entropy event data sampling as outlined in section 2.15.
  - Fully testable ChaCha20 DRNG via a userspace ChaCha20 DRNG implementation.
  - In case of using the SP800-90A DRBG, it is fully testable and tested via the NIST ACVP test framework, for example certificates A628, and A737.
  - In case of using the kernel crypto API SHASH hash implementation, it is fully testable and tested via the NIST ACVP test framework, for example certificates A734, A737, and A738.
  - The LRNG offers a test interface to validate the used software hash implementation and in particular that the LRNG invokes the hash correctly, allowing a NIST ACVP-compliant test cycle – see section 2.15.
  - Availability of stress testing covering the different code paths for data and mechanism (de)allocations and code paths covered with locks.

- Entropy collection
  - The LRNG is fully compliant to SP800-90B requirements and is shipped with a full SP800-90B assessment and all required test tools. The legacy /dev/random implementation on the other hand has architectural limitations which does not easily allow to bring the implementation in compliance with SP800-90B as outlined in section 4.5.
  - Full entropy assessment and description is provided with chapter 3, specifically section 3.2.6.

  - Guarantee that entropy events are not credited with entropy twice (the legacy /dev/random implementation credits HID/disk and interrupt events with entropy which are a derivative of each other) and guarantee that entropy data is not reused for two different use cases (as done in the legacy /dev/random implementation when injecting a part of fast_pool into the net_rand_state).

- Configurable
  - LRNG kernel configuration allows configuration that is functionally equivalent to the legacy /dev/random. Non-compiled additional code is folded into no-ops.
  - The following additional functions are compile-time selectable independent of each other:
    - ≻ Enabling of switchable cryptographic implementation support. This allows enabling SP800-90A DRBG.
    - ≻ Enabling of using Jitter RNG noise source.
    - ≻ Enabling of noise source health tests including SP800-90B health tests.
    - ≻ Enabling of test interface allowing to enable each test interface individually.
    - ≻ Enabling of the power-up self test.
  - At boot-time, the SP800-90B health tests can be enabled as outlined in section 2.5.4.
  - At boot-time, the entropy rate used to credit the external CPU-based noise source and Jitter RNG noise source can be configured including setting an entropy rate of zero or full entropy – see sections 2.5.2 and 2.5.3.

- Run-time pluggable cryptographic implementations used for all data processing steps specified in section 2.2
  - The DRNG can be replaced with a different implementation allowing any type of DRNG to provide data via the output interfaces. The LRNG provides the following types of DRNG implementations:
    - ≻ ChaCha20-based software implementation that is used per default.
    - ≻ SP800-90A DRBG using accelerated cryptographic implementations that may sleep.
    - ≻ Any DRNG that is accessible via the kernel crypto API RNG subsystem.
  - The hash component can be replaced with any other hash implementation provided the implementation does not sleep. The LRNG provides the following types of hash implementations:
    - ≻ SHA-256 software implementation that is used per default. Due to kernel build system inconsistencies, the software SHA-1 implementation is used if the kernel crypto API is not compiled.

    - ≻ SHA-512 hash using the fastest hash implementation available via the kernel crypto API SHASH subsystem.

- Code structure
  - The LRNG source code is available for current upstream Linux kernel separate to the legacy /dev/random which means that users who are conservative can use the unchanged legacy /dev/random implementation.
  - Back-port patches are available to apply the LRNG to Linux kernel versions of 5.8, 5.4, 4.19, 4.14, 4.12, and 4.10.

1.4 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.

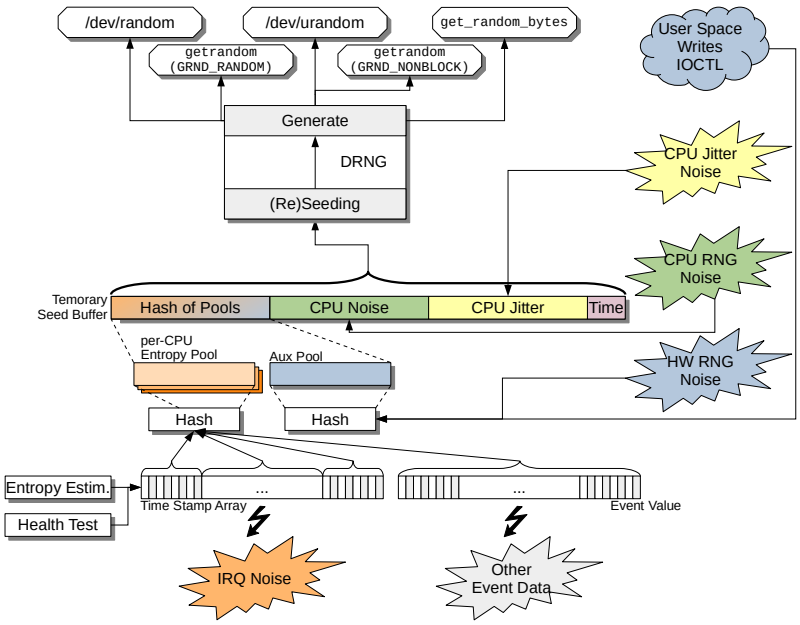

Figure 2.1: LRNG Big Picture

The colors indicate the different noise sources managed by the LRNG.

2.1 LRNG Components

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

1. The LRNG implements a DRNG. The DRNG always generates the requested amount of output. When using the SP800-90A terminology it operates without prediction resistance. The 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 DRNG is seeded from the entropy pool and additional "fast" noise sources.

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

   Depending on the used interface to request data from the DRNG, the caller may be put to sleep until the LRNG is fully seeded:

   (a) /dev/urandom and the `getrandom` system call with the `GRND_INSECURE` flag always generates data including when the LRNG is not properly seeded.

   (b) /dev/random and the getrandom system call with the `GRND_RANDOM` or with a flag of zero generates data only when the LRNG is fully seeded.

2. The DRNG is seeded by concatenating the data from the following sources:

   (a) the output of the entropy pools,

(b) the Jitter RNG if available, and

(c) the CPU-based noise source such as Intel RDSEED 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 DRNG with. The LRNG ensures, however, that the DRNG after seeding is at maximum the security strength of the used DRNG implementation of 256 bits.

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

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

(b) At runtime, the available entropy from the slow noise source is concatenated with a pre-defined amount of data from the fast noise sources. In addition, each DRNG reseed operation triggers external noise source providers to deliver one block of data.

3. 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 per-CPU entropy pool using an a hash. After boot, the used hash is SHA-256, but the used hash algorithm can be changed at runtime. The following sources of entropy are used:

   (a) When an interrupt occurs, the high-resolution time stamp is concatenated with previous time stamps. Once a given number of time stamps are received, they are hashed together with the per-CPU entropy pool to form a new state of that entropy pool. This time stamp is credited with heuristically implied entropy. 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 processed by the mentioned hash operation. During boot time, until the fully seeded stage is reached, each time stamp truncated to the 32 least significant bits instead. These 32 bits are concatenated like the 8 bit values and processed by the hash operation. More details are given in section 2.5.1.

   (b) HID event data like the key stroke or the mouse coordinates are concatenated with the time stamps and processed as outlined for the time stamps. This data is not credited with entropy by the LRNG and this data has no correlation with the data credited with entropy, i.e. the time stamp. Thus it is treated as additional data to stir the entropy pool.

   (c) Device drivers may provide data that is mixed into the auxiliary pool using the hash operation. This data is not credited with entropy by the LRNG and it is unrelated to the data credited with entropy, i.e. the time stamp. Thus it is treated as additional data to stir the entropy pool.

When the DRNG requires fresh seed data from the entropy pools, a hash of all per-CPU entropy pools and the auxiliary pool is created. The resulting message digest is used as seed data but at the same time is the new state of the auxiliary pool. The data returned as seed data is truncated to the amount of bits requested by the caller (commonly 256 bits which is equal to the security strength of the DRNG) or truncated to the available entropy, whatever is smaller. The result of this operation is that the generated data used to seed the DRNG have full entropy. The truncation operation maintains entropy as it is defined by [11] section 5.1.3.1.1 table 1 which defines that each approved hash has full security strength even though the hash operation performs a truncation (e.g. for SHA-384 or SHA-512/256).

4. Any data provided from user space either provided via /dev/random, /dev/urandom or the IOCTL of `RNDADDENTROPY` on both device files are always injected into the auxiliary 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 auxiliary 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.

   The reason for having the auxiliary pool is to allow injection of entropy data compliant to [11] section 3.1.6. As the LRNG may maintain multiple DRNGs, it would not be clear into which DRNG to inject it at the time of receipt of data from these auxiliary noise sources.

   When the entropy pool is processed with the hash function to obtain random numbers, the auxiliary pool is "read" at the same time. Thus, the hash operation generates random numbers using both pools at the same time. To support backtracking resistance, the generated hash is given to the caller but also injected into the auxiliary pool like other data by hashing it together with the existing auxiliary pool data to form the new auxiliary pool content.

The LRNG allows the DRNG mechanism and the used hash to be changed at runtime. Per default, a ChaCha20-based DRNG is used[8] together with a software implementation of SHA-256. The LRNG also offers an SP800-90A DRBG based on the Linux kernel crypto API DRBG implementation along with the most accelerated SHA-512 hash implementation from the kernel crypto API.

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

2.2 LRNG Data Processing

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

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

operation $SHA()$ refers to the hash operation using the message digest implementation that is currently present, i.e. either SHA-256 or SHA-512 (in case the kernel crypto API is not compiled, SHA-1 is used).

1. Truncation: The received time stamps are truncated to 8 least significant bits (or 32 least significant bits during boot time): t_8 (or t_{32})

2. Concatenation: The received and truncated time stamps as well as auxiliary 32 bit words a_{32} are concatenated to fill the per-CPU data array that is capable of holding 64 8-bit words[9] - the order of the data a_{32} or t_8 present in the concatenation depends on the occurrence of events - the following formula depicts one possible order for illustration - the implementation is provided with functions `_lrng_pcpu_array_add_u32` and `lrng_pcpu_array_add_slot`:

$$t_{8_{n-59}}||a_{32_n}||t_{8_{n-58}}|| \ ... \ ||t_{8_n} \tag{2.1}$$

3. Hashing: A set of concatenated time stamp data received from the interrupts are hashed together with the current existing per-CPU entropy pool state EP_{CPU}. The resulting message digest is the new per-CPU entropy pool state - the implementation is provided with function `lrng_pcpu_array_compress`:

$$EP_{CPU_n} = SHA(EP_{CPU_{n-1}}||t_{8_{n-59}}||a_{32_n}||t_{8_{n-58}}|| \ ... \ ||t_{8_n}) \tag{2.2}$$

4. Hashing: When new data D is added to the auxiliary pool AP, the data is hashed together with the auxiliary pool to form a new auxiliary pool state - the implementation is provided with function `lrng_pool_insert_aux`:

$$AP_n = SHA(AP_{n-1}||D) \tag{2.3}$$

5. Hashing: A message digest of all per-CPU entropy pools and the auxiliary pool is calculated which forms the new auxiliary pool state. At the same time, this message digest is used to fill the slow noise source output buffer S discussed in the following - the implementation is provided with function `lrng_pcpu_pool_hash`:

$$AP_n = SHA(AP_{n-1}||EP_{CPU0_n}||EP_{CPU1_n}|| \ ... \ ||EP_{CPUX_n}) \tag{2.4}$$

6. Truncation: The most-significant bits (MSB) defined by the requested number of bits (commonly equal to the security strength of the DRBG) or the entropy available transported with the buffer (which is the minimum of the message digest size and the available entropy in all entropy pools and the auxiliary pool), whatever is smaller, are obtained from the slow noise source output buffer S - the implementation is provided with function `lrng_pcpu_pool_hash`:

$$S_n = MSB_{min(entropy,security\ strength)}(AP_n) \tag{2.5}$$

[9]The LRNG collection size is compile-time configurable where 64 is a default value. When configuring a different value, the number of the concatenated data must be adjusted as needed. However, this modification has no impact to the illustration of the data processing.

7. Concatenation: The temporary seed buffer T used to seed the DRNG is a concatenation of the slow noise source buffer S, the Jitter RNG output J, the CPU noise source output C, and the current time t - the implementation is provided with function `lrng_fill_seed_buffer`:

$$T_n = S_n||J_n||C_n||t \tag{2.6}$$

2.3 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 pools.

At the time the DRNG shall be seeded, the all entropy pools and the auxiliary pool are processed with a cryptographic hash function which can be chosen at runtime. If the digest of the hash and the available entropy are larger than requested by the caller, the digest is truncated to the appropriate size.

The output of the hash function is the new auxiliary pool state to ensure backtracking resistance. The message digest is concatenated with data from the fast noise sources.

The DRNG 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 DRNG 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 DRNG, the hash function always generates 256 data bits during boot to seed the DRNG. During boot, the DRNG 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 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.

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 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 DRNG is (re)seeded with a number of random bytes from the entropy pool that is at most the amount of entropy present in the entropy pool. This means that when the entropy pool contains 128 or more bits of entropy, the DRNG is seeded with that amount of entropy as well.

Before the DRNG is seeded with 256 bits of entropy in step 3, requests of random data from /dev/random or the `getrandom` system call are not processed.

The hash operation providing random data from the entropy pool will always require that all entropy sources collectively can deliver at least 128 entropy bits.

The 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 DRNG is reseeded with whatever entropy is available, but at least 128 bits – 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 DRNG is more than 600 seconds ago[10], or
  - 2^{20} DRNG generate operations are performed, whatever comes first, or
  - the 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[11][12].

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.

2.3.1 Minimally Versus Fully Seeded Level

The LRNG's DRNG is reseeded when the first 128 bits / 256 bits of entropy are received during boot as indicated above. The 128 bits level defines that that the DRNG is considered "minimally" seeded whereas reaching the 256 bits level is defined as the DRNG is "fully" seeded.

Both seed levels have the following implications:

[10] 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 DRNG reseed requirement.

[11] 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 DRNG without a reseed operation.

[12] After boot, the 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 DRNG object and once for the atomic DRNG object.

- 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 or /dev/random 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.

2.3.2 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 into the DRNG, 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 2.5.2 and 2.5.3.

Seed Stage	Noise source data bits	Noise source entropy bits	LRNG behavior
Receipt of 32 fresh IRQs	IRQ: 256 Jitter: 256 CPU: 256	IRQ: 32 Jitter: 16 CPU: 8	/dev/random blocked `getrandom`(0) blocked /dev/urandom operational `wait_for_random_bytes` blocked `add_random_ready_callback` blocked `get_random_bytes` operational
Receipt of 104 fresh IRQs	IRQ: 256 Jitter: 256 CPU: 256	IRQ: 104 Jitter: 16 CPU: 8	/dev/random blocked `getrandom`(0) blocked /dev/urandom operational `wait_for_random_bytes` operational `add_random_ready_callback` operational `get_random_bytes` operational
Receipt of 232 fresh IRQs	IRQ: 256 Jitter: 256 CPU: 256	IRQ: 232 Jitter: 16 CPU: 8	/dev/random operational `getrandom`(0) operational /dev/urandom operational `wait_for_random_bytes` operational `add_random_ready_callback` operational `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	LRNG behavior
2000 unused IRQs in entropy pool	IRQ: 256 Jitter: 256 CPU: 256	IRQ: 256 Jitter: 16 CPU: 8	/dev/random operational `getrandom`(0) operational /dev/urandom operational `wait_for_random_bytes` operational `add_random_ready_callback` operational `get_random_bytes` operational
104 unused IRQs in entropy pool	IRQ: 104 Jitter: 256 CPU: 256	IRQ: 104 Jitter: 16 CPU: 8	/dev/random operational `getrandom`(0) operational /dev/urandom operational `wait_for_random_bytes` operational `add_random_ready_callback` operational `get_random_bytes` operational
103 unused IRQs in entropy pool	IRQ: 103 Jitter: 256 CPU: 256	IRQ: 103 Jitter: 16 CPU: 8	/dev/random operational `getrandom`(0) operational /dev/urandom operational `wait_for_random_bytes` operational `add_random_ready_callback` operational `get_random_bytes` operational
0 unused IRQs in entropy pool	IRQ: 0 Jitter: 256 CPU: 256	IRQ: 0 Jitter: 16 CPU: 8	/dev/random operational `getrandom`(0) operational /dev/urandom operational `wait_for_random_bytes` operational `add_random_ready_callback` operational `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	LRNG behavior
2000 unused IRQs in entropy pool	IRQ: 256 Jitter: 256 CPU: 256	IRQ: 256 Jitter: 0 CPU: 0	/dev/random operational `getrandom`(0) operational /dev/urandom operational `wait_for_random_bytes` operational `add_random_ready_callback` operational `get_random_bytes` operational
0 unused IRQs in entropy pool	IRQ: 0 Jitter: 256 CPU: 256	IRQ: 0 Jitter: 0 CPU: 0	/dev/random operational `getrandom`(0) operational /dev/urandom operational `wait_for_random_bytes` operational `add_random_ready_callback` operational `get_random_bytes` operational

2.3.3 NUMA Systems

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

The question now arises how are the different DRNGs seeded without re-using entropy or relying on random numbers from a yet insufficiently seeded LRNG. The LRNG seeds the DRNGs sequentially starting with the one for NUMA node zero – the DRNG for NUMA node zero is seeded with the approach of 32/128/256 bits of entropy stepping discussed above. Once the DRNG for NUMA node 0 is seeded with 256 bits of entropy, the LRNG will seed the DRNG of node one when having again 256 bits of entropy available. This is followed by seeding the 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 DRNG. If that DRNG is not yet seeded, it falls back to using the 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.

2.3.4 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. This function pointer data structure also holds the callbacks to the hash used to process the entropy pools.

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-256 hash[13] for accessing the entropy pools.

2.3.5 Covered Design Concerns of Legacy /dev/random

Starting with kernel 5.8, the legacy /dev/random implementation seeds the external random32 PRNG with data directly taken from the fast_pool where that same data is added to the entropy pool. This implies that data believed to hold entropy is used twice for different purposes which is considered to be an architectural weakness. In addition, the random32 PRNG performs a cryptographic non-secure processing of data which may leak entropy. In this case, the legacy /dev/random heuristically credits entropy to data that may have no entropy.

[13] In case `CONFIG_CRYPTO` is not selected during the kernel compilation, SHA-1 is used.

The LRNG covers this aspect by only sending data to the random32 PRNG that is not used by the LRNG.

Additional concerns regarding the design and implementation of the legacy /dev/random and their coverage in the LRNG are given in [8] section 4.4.

2.4 LRNG Data Structures

The LRNG uses three main data structures:

- The data from the interrupt noise source is processed with a per-CPU entropy pool. In addition, a per-CPU data array that can hold the concatenated time stamps is maintained. Both are accessed lockless since the currently executing CPU's entropy pool and data array is used. During access to the entropy pool, the LRNG though takes a lock since the entropy pool is also read when the hash is calculated for filling the seed buffer. As the filling of the seed buffer is very infrequently (see above for the reseed periods of the DRNG), the lock is hardly contented which allows the conclusion that the entropy collection operates quasi-lockless.

- The deterministic random number generator data structure for the DRNG holds the reference to the DRNG instance and the hash instance and associated meta data needed for its operation. The DRNG is managed with a separate data structure. When using the DRNG, a full read/write lock is used to guard (a) against replacement of the DRNG reference while operating on the DRNG state, and (b) to read/write the DRNG state. Contrarily when using the hash, only a read-lock is used to guard against the replacement of the hash reference. This implies that the hash state is kept on the stack of the calling application.

2.5 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 and stored in the operating CPU's data array. After the receipt of 64 time stamps, the data array with all concatenated time stamps is inserted into the currently executing CPU's entropy pool. During boot time until the LRNG reaches the fully seeded level, the 32 least significant bits of the data are directly inserted into the CPU's entropy pool. 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 2.5.4 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 per-CPU data array is hashed together with the same CPU's entropy pool to form the new state of the entropy pool.

4. The LRNG increases the per-CPU counter of the received interrupt events by the number of healthy interrupts stored in the per-CPU data 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 by all per-CPU entropy pools, 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 all DRNG instances are 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 DRNG.

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 2.5.1 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 2.7.

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.

2.5.1 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[14].

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 processing with a hash to generate a new entropy pool state 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 per-CPU entropy pool. 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 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.

[14] 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.

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 into the per-CPU data array 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 2.3.1 – the aforementioned runtime behavior of concatenating the 8 LSB of 64 time stamps before mixing them into the per-CPU entropy pool is enabled.

2.5.2 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`.

2.5.3 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`.

2.5.4 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_archrandom.archrandom=0`
- Jitter RNG: `lrng_jent.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 per-CPU data array. 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. All 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 2.5.4. 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 when called with a flag that does not include `GRND_INSECURE`
  - `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`
  - `getrandom(2)` system call when called with `GRND_INSECURE`
  - `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

2.6 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 concatenated to the per-CPU data array for interrupts as well.

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

2.7 DRNG Seeding Operation

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

The (re)seeding logic tries to obtain 256 bits of entropy from the noise sources. However, if less entropy can only be delivered, the DRNG reseeding is only performed if at least 128 bits of entropy collectively from all noise sources can be obtained.

The entropy pool has a size of 128 32-bit words. The value of 128 words is the default but a different size can be selected during compile time.

For efficiency reasons, the seeding operation uses a seed buffer depicted in figure 2.1 that is three blocks of 256 bits. The first block is filled with data from the hashed data from the entropy pools. 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 and third 256-bit blocks are dedicated the fast noise sources and is filled with data from those noise sources – i.e. RDSEED and the Jitter RNG. 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 DRNG. The reason is that the fast noise sources can dominate the slow noise sources when much entropic data is required.

When reading the per-CPU entropy pools, the entire entropy pool and the auxiliary pool are processed with the hash function. The result of the hash function is used as the new auxiliary pool state. 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 output returned to the DRNG reseed operation is truncated to a size equal to the amount of entropy that is present in the entropy pool. This operation is followed by reducing the assumed entropy in the pool by the amount returned by the hash operation.

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

During boot time, the number of required interrupts for seeding the DRNG is first set to an emergency threshold of one word, i.e. 32 bits. This is followed by setting the threshold value to deliver at least 128 bits of entropy. At that entropy threshold, the DRNG is considered "minimally" seeded – the value of 128 bits covers the minimum entropy requirement specified in SP800-131A ([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.

2.8 LRNG-external Noise Sources

The LRNG also supports obtaining entropy from the following data sources and noise sources that are external to the LRNG. The data is injected into the auxiliary poo by hashing the input data together with the current auxiliary pool to form the new auxiliary pool statel.

During the reseeding operation of the DRNG, any user-space entropy provider waiting via select(2) or kernel space entropy provider using the `add_hwgenerator_randomness` API call are triggered to provide one buffer full of data. This data is mixed into the auxiliary pool. This approach shall ensure that the LRNG-external noise sources may provide entropy at least once each DRNG reseed operation.

2.8.1 Kernel Hardware Random Number Generator Drivers

Drivers hooking into the kernel HW-random framework can inject entropy directly into the auxiliary pool. Those drivers provide a buffer to the entropy pool and an entropy estimate in bits. The auxiliary 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 auxiliary pool and triggers a reseed of the 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 auxiliary pool and define the amount of entropy associated with that data.

2.8.3 Auxiliary Pool

The auxiliary pool is maintained in compliance with [11] section 3.1.6 which requires that noise sources must be combined using a vetted conditioning component.

The auxiliary pool is processed with the available hash and calculates a message digest of the pool content and the newly provided data. The output of the hash operation is the new content of the auxiliary pool. In addition, it maintains an entropy estimator counting the received entropy. The entropy estimator is capped to a maximum of the digest size of the used hash as this hash cannot maintain more entropy.

The auxiliary pool is processed with the hash function when generating random numbers to seed the DRNG at the same time when the entropy pool is processed. Thus, both, the entropy pool and the auxiliary pool, are simultaneously used as noise data provider to seed the DRNG.

The entropy estimator is decreased by the amount of data read via the hash.

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.

All DRNGs 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 DRNG "generate" function. I.e. the 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 DRBG will be reseeded by using the operation documented in section 2.7 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.

2.9.2 /dev/random

The random numbers to be generated for /dev/random are defined to have a special property: it only provides data once at least 256 bits of entropy have been collected by the LRNG.

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-256 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 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 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 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 hash operation used for the hash call to generate random numbers from the entropy pool 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 DRNG is seeded. After the 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 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 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[15].

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.

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

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 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 during boot time. /dev/random behaves identical to 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 2 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/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 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 DRNG tries to reseed from the entropy pool before every generate request. I.e. the DRNG in this case acts like a DRNG with prediction resistance enabled as defined in [1].

/proc/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.

/sys/module/lrng_selftest/parameters/selftest_status Querying the status and restarting the LRNG self tests - see section 2.14 for details.

/sys/kernel/debug/lrng_testing/* Virtual files providing test interfaces as documented in section 2.15.

/sys/module/lrng_testing/parameters/* Virtual files providing test interfaces as documented in section 2.15.

/sys/module/lrng_archrandom/parameters/archrandom Interface to adjust entropy estimation from CPU noise source. See section 2.5.2 for details.

/sys/module/lrng_jent/parameters/jitterrng Interface to adjust entropy estimation from Jitter RNG noise source. See section 2.5.3 for details.

IOCTLs are implemented as documented in random(4).

2.14 LRNG Self-Tests

When compiling the LRNG with `CONFIG_LRNG_SELFTEST`, the following self-tests are performed during startup of the kernel covering all deterministic operations that are vital to collect and maintain entropy:

- The hash function used to obtain data from the entropy pool is tested. The power-on self test vector is taken from `crypto/testmgr.h` for the hash known answer test using the string "abc" as input.
- The ChaCha20 DRNG is tested by seeding its state, generating random numbers, and comparing them with expected data. The standalone user space ChaCha20 DRNG not only allows studying of the LRNG ChaCha20 DRNG in user space, but is also used to generate the known answers.

- The operation to store time stamps in the data array is tested by injecting integers in that array using the management functions and comparing the array content with expected values. The expected values for the array operation are generated during compile time in the function `lrng_data_process_selftest`.

All self tests are performed such that they do not have an impact on the regular operation of the LRNG by using separate memory locations processed by the tested deterministic operations.

All individual self tests must pass to indicate that the LRNG is successfully tested. If one self test fails, at least a warning message is printed If the kernel compilation option `CONFIG_LRNG_SELFTEST_PANIC` is enabled, the kernel will crash if a self test fails.

The status of the self-tests can be queried by reading the file `/sys/module/lrng_selftest/parameters/selftest_status`. If that file shows 0, all self-tests passed successfully. Otherwise at least one self-test failed. Writing any value into that file causes the self-tests to be repeated.

Additional self-tests that support the LRNG are:

- The SP800-90A DRBG is self-tested by the Linux kernel crypto API test manager during instantiation.

- When using a PRNG the LRNG kernel module `lrng_kcapi.ko`, its self-test is driven by the Linux kernel crypto API test manager.

- The raw noise sources are tested at runtime with at least the stuck test. In addition the SP800-90B start-up and runtime tests discussed in section 2.5.4 are performed if they are enabled.

With these tests, all aspects of the LRNG that are vital to the entropy management and random number generation are self-tested during power-up or at runtime.

2.15 LRNG Test Interfaces

During kernel compilation, the following interfaces may be enabled allowing direct access to non-deterministic aspects. It is not advisable to enable these interfaces for production systems. Yet, these interfaces are considered to be protected against misuse by allowing only the root user to access them. In addition, any data obtained through these interfaces is not used by the LRNG to feed the entropy pool. Thus, even when leaving these interfaces enabled on production systems, the impact on security is considered to be limited.

- The interface `/sys/kernel/debug/lrng_testing/lrng_raw_hires` allows reading of the raw unconditioned noise data collected while the read operation is in progress by providing the time stamps of the events collected by the LRNG that otherwise are injected into the entropy pool. When booting the kernel with the kernel command line option `lrng_testing.boot_raw_hires_test=1`, the time stamps of the first 1,024 events recorded by the LRNG are stored. The first read of the `lrng_raw_hires` file after boot provides this data in this case.

- The interface `/sys/kernel/debug/lrng_testing/lrng_raw_jiffies` allows reading of the raw unconditioned Jiffies collected while the read operation is in progress by providing the Jiffies values collected by the LRNG that otherwise are injected into the entropy pool (if no high-resolution time stamp is detected). When booting the kernel with the kernel command line option `lrng_testing.boot_raw_jiffies_test=1`, the time stamps of the first 1,024 events recorded by the LRNG are stored. The first read of the `lrng_raw_jiffies` file after boot provides this data in this case.

- The interface `/sys/kernel/debug/lrng_testing/lrng_raw_irq` allows reading of the raw unconditioned interrupt numbers collected while the read operation is in progress by providing the interrupt number values collected by the LRNG that otherwise are injected into the entropy pool (if no high-resolution time stamp is detected) or into the random32 PRNG external to the LRNG. When booting the kernel with the kernel command line option `lrng_testing.boot_raw_irq_test=1`, the time stamps of the first 1,024 events recorded by the LRNG are stored. The first read of the `lrng_raw_irq` file after boot provides this data in this case.

- The interface `/sys/kernel/debug/lrng_testing/lrng_raw_irqflags` allows reading of the raw unconditioned interrupt flags collected while the read operation is in progress by providing the interrupt flag values collected by the LRNG that otherwise are injected into the entropy pool (if no high-resolution time stamp is detected) or into the random32 PRNG external to the LRNG. When booting the kernel with the kernel command line option `lrng_testing.boot_raw_irqflag_test=1`, the time stamps of the first 1,024 events recorded by the LRNG are stored. The first read of the `lrng_raw_irqflags` file after boot provides this data in this case.

- The interface `/sys/kernel/debug/lrng_testing/lrng_raw_retip` allows reading of the raw unconditioned return instruction pointer collected while the read operation is in progress by providing the instruction pointer 32 LSB values collected by the LRNG that otherwise are injected into the entropy pool (if no high-resolution time stamp is detected) or into the random32 PRNG external to the LRNG. When booting the kernel with the kernel command line option `lrng_testing.boot_raw_retip_test=1`, the time stamps of the first 1,024 events recorded by the LRNG are stored. The first read of the `lrng_raw_retip` file after boot provides this data in this case.

- The interface `/sys/kernel/debug/lrng_testing/lrng_raw_regs` allows reading of the raw unconditioned interrupt register state collected while the read operation is in progress by providing the selected register 32 LSB values collected by the LRNG that otherwise are injected into the entropy pool (if no high-resolution time stamp is detected). When booting the kernel with the kernel command line option `lrng_testing.boot_raw_regs_test=1`, the time stamps of the first 1,024 events recorded by the LRNG are stored. The first read of the `lrng_raw_regs` file after boot provides this data in this case.

- The interface `/sys/kernel/debug/lrng_testing/lrng_raw_array` allows reading of the raw noise data that has been stored in the per-CPU data

array collected while the read operation is in progress. When booting the kernel with the kernel command line option `lrng_testing.boot_raw_array=1`, the array content of the first 1,024 events recorded by the LRNG are stored. The first read of the `lrng_raw_array` file after boot provides this data in this case.

- The interface `/sys/kernel/debug/lrng_testing/lrng_irq_perf` allows reading of the number of cycles used to process one interrupt event. This allows measuring the performance impact of the LRNG on the interrupt handler. When booting the kernel with the kernel command line option `lrng_testing.boot_irq_perf=1`, the performance data of the first 1,024 events recorded by the LRNG are stored. The first read of the `lrng_irq_perf` file after boot provides this data in this case.

- The interface `/sys/kernel/debug/lrng_testing/lrng_acvt_hash` allows sending data to the used hash operation to calculate a message digest that is returned to user space. With this interface ACVP testing can be implemented showing compliance of the hash implementation with a NIST reference implementation.

The helper tool getrawentropy.c is provided to read the files and format the data for post-processing.

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.

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 2.5.4 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 approved message digest operation is used.
- For reading the entropy pool and compressing the entropy data, the hash operation is used. The security strength of the LRNG is the minimum of

the DRBG security strength and the security strength of the hash following [11] section 3.1.5.1.1 table 1. All ciphers can be tested via ACVT, including the LRNG-builtin SHA-1 or SHA-256 hash implementation.

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 a hash which is entropy preserving. The hash operation function providing 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 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.

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: 3.452064.

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 eight bits are processed:

- Using the 8 least significant bits of the time stamps in column-wise assessment – lowest entropy value of all 1,000 column entries: 3.455504
- Using the 8 least significant bits of the time stamps in row-wise assessment – lowest entropy value of all 1,000 column entries: 3.393808
- 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 are performed at different stages as outlined in section 2.1. Although the hashing operation is used for different stages, the following discussion is applicable to all use cases.

Truncation The truncation operation ensures that the entropy in that data is at maximum the truncated hash.

The truncation of operation (1) listed in section 2.2 is not affected by the capping of the entropy, because the quantitative measurement of the existing entropy using the SP800-90B tool set is performed using that truncated input data. The LRNG implies an entropy of 1 bit per truncated time stamp and zero bits of entropy per arbitrary 32-bit word size which means that the entropy present in the data is always smaller as the data size.

The truncation operation of step (6) listed in section 2.2 verifies that the truncated data contains at most the amount of entropy as the generated data size. The remaining part of the truncated data is not exported to any external entity but remains in the auxiliary pool - when new random data is generated involving the auxiliary pool, the current auxiliary pool state is always hashed. This is a deviation from SP800-90B section 3.1.5.1.2 which requires a relative reduction of entropy. This statement is considered inconsistent with the statement implied in table 1 [11] and therefore wrong depicted with the following analogy: Assume to have a buffer of 512 bits of data having 256 bits of entropy. When hashing it with SHA-512, the resulting message digest of 512 bits has 256 bits of entropy. When truncating the digest to 256 bits, SP800-90B states the entropy is 128 bits. However, SP800-90B section 3.1.5.1.1 table 1 states that full entropy is given to approved hash functions. Assume to use a SHA-512/256 which has a digest size of 256 bits and thus could transport 256 bits of entropy following table 1. This SHA-512/256 hash operation calculates a SHA-512 hash truncated to 256 bits. Albeit the cryptographic operation of SHA-512/256 is identical to the LRNG-applied truncation[16], SP800-90B table 1 awards 256 bits of entropy to SHA-512/256 but at the same time SP800-90B would apply only 128 bits to the LRNG-applied truncation. Due to this inconsistency, the LRNG

[16]Depending on the runtime configuration the LRNG uses a hash of SHA-512 and fills a buffer of the DRNG security strength size, i.e. 256 bits.

applies the entropy behavior implicitly specified in table 1, i.e. the entropy is the minimum of the available entropy and the message digest size.

Concatenation When applying a concatenation operation, the LRNG simply adds the entropy delivered with each data entry part.

Hash The input of the hash n_{in} is fixed as it processes the existing per-CPU entropy pool(s), auxiliary pools and the per-CPU data arrays.

The output of the hash n_{out} is usually fixed to the message digest size. The on exception is the output of the hash n_{out} to provide the seed to the DRNG: it is the minimum of either the digest size of the used hash or the amount of entropy available in the processed entropy pools based on the number of "unprocessed" time stamps held in the per-CPU entropy pools.

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

- ChaCha20: SHA-256 in normal case, SHA-1 if kernel is not compiled with `CONFIG_CRYPTO`
- SP800-90A Hash DRBG: SHA-512
- SP800-90A HMAC DRBG: SHA-512
- SP800-90A CTR DRBG: SHA-512

In the following, the two hash operations, one applied to the entire pool and one applied to the auxiliary pool are analyzed separately.

The hashing operation applied with the equation 2.4 compresses the different per-CPU entropy pools and the auxiliary pool together using a vetted conditioning component to form the new state of the auxiliary pool. As mentioned above, the auxiliary pool is fed by a noise source from user space or kernel space that is totally separate from the LRNG interrupt noise source that feeds the entropy pools. Yet, the entropy of both noise sources is considered to form the entropy statement as outlined in equation 3.6. According to [11] section 3.1.6, combining multiple noise sources with a vetted conditioning component of a hash is allowed, but only one noise source is allegedly to be credited with entropy. This requirement is not applied here as its basis is not applicable due to the following:

- The two noise sources are separate SP800-90B noise sources with their own entropy assessment and resulting entropy statement. Using noise sources that are not SP800-90B compliant to provide entropy to the LRNG via the auxiliary pool is disallowed as stated in section 3.4. Thus, when the entropy pools and the auxiliary pools are filled, their entropy level are based on these independent entropy assessments.
- Using a vetted conditioning component of a hash compresses the available entropy at the same level as compresses the entropy for the different per-CPU entropy pools when new data arrives or when new data is received from the external noise sources to feed into the auxiliary pool. The same entropy compression consideration applies when hashing the per-CPU entropy pools and the auxiliary pool together as there is no difference with

respect to the entropy handling provided the noise sources are totally independent and follow SP800-90B. Naturally, the resulting entropy rate of the calculated message digest is the minimum of the message digest size, and the sum of the entropy found in all hashed entropy pools and the auxiliary pool as stated with equation 3.6.

- Disallowing the crediting of entropy of both noise sources with [11] section 3.1.6 is contradictory to the implied statement in [11] table 1 section 3.1.5.1.1 as follows. Assume that the message digest from the per-CPU entropy pools and the auxiliary pool data are concatenated to form the seed buffer handed to, say, a Hash DRBG. The first operation of the Hash DRBG is a hash_df operation, i.e. a vetted conditioning component as per table 1. The entropy rate of both concatenated buffers is now allowed to be counted towards seeding the Hash DRBG. For example, if the per-CPU entropy pools collectively deliver, say, 130 bits of entropy and the auxiliary pool delivers 140 bits of entropy, the Hash DRBG with a security strength of 256 bits is claimed to be seeded with 256 bits of entropy (130 bits + 140 bits = 270 bits of entropy used to seed the DRBG). Now, the application of a vetted conditioning component in the Hash DRBG is allowed to claim that multiple noise sources provide entropy in unison. However, when performing a vetted conditioning operation as part of the LRNG, [11] section 3.1.6 denies crediting entropy to all noise sources, provided their entropy rate is clearly determined based on separate SP800-90B analyses. This contradiction is resolved by the LRNG to follow the implied statement in table 1 together with the entropy rate applied when seeding a DRBG, i.e. considering the entropy rate of both independent noise sources of the auxiliary pool and the per-CPU entropy pools.

Note, the reason for hashing the per-CPU entropy pools together with the auxiliary pool is to ensure backtracking resistance when calculating the next round of random numbers used to fill the seed buffer used to seed the DRBG from the noise sources.

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 per-CPU data arrays receive interrupt time stamps continuously, only the entropy from 256 time stamps are required as illustrated below. Only when all per-CPU entropy pools have 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 hash as outlined above, the following approach for the entropy calculation is taken for each of the data processing steps outlined in section 2.2:

- Function 2.2:

- $n_{in_{per-CPU\ pool}}$ equals to 512 bits as the per-CPU entropy pool is 512 bits in size and 64 * 8 bits[17] of the per-CPU data array.
- $n_{out_{per-CPU\ pool}}$ is the message digest size in bits.
- $nw_{per-CPU\ pool}$ is the message digest size in bits.

- Function 2.3:
  - $n_{in_{aux\ pool}}$ equals to 512 bits as the auxiliary pool pool size is 512 bits in size plus the provided input data.
  - $n_{out_{aux\ pool}}$ is the message digest size in bits.
  - $nw_{aux\ pool}$ is the message digest size in bits.

- Function 2.4:
  - $n_{in_{hash\ pools}}$ equals to $\sum_{a=0}^{max\ CPU} n_{out_{per-CPU\ pool_a}} + n_{out_{hash\ aux}}$
  - $n_{out_{hash\ pools}}$ is the message digest size in bits.
  - $nw_{hash\ pools}$ is the message digest size in bits.

3.2.6 SP800-90B Section 3.1.5.1

The hash operation is either SHA-512, SHA-256, or SHA-1 as outlined above is considered to be a vetted conditioning component. Thus the entropy rate of the hash output is calculated as follows using the aforementioned variables for the hash function. In addition, the following consideration applies:

- The entropy content of the input $h_{in_{per-CPU\ pool}}$: The input entropy of the hash used to process the per-CPU entropy pool is equal to the entropy provided by the per-CPU data array and the entropy already present in the per-CPU entropy pool considering that both data components are hashed at the same time to form a new per-CPU entropy pool state. Of course, the entropy held in the per-CPU entropy pool will never be larger than the digest size of the used hash which is compliant to [11] section 3.1.5.1.1 table 1.

- The entropy content of the input $h_{in_{aux\ pool}}$: The input entropy of the hash used to process the auxiliary pool is equal to the entropy provided by the noise source and the already collected entropy in the auxiliary pool considering that both data components are hashed at the same time to form a new auxiliary pool state. Of course, the entropy held in the auxiliary pool will never be larger than the digest size of the used hash which is compliant to [11] section 3.1.5.1.1 table 1.

- The entropy content of the input $h_{in_{hash\ pools}}$: The input entropy of the hash used to process the entire entropy pool is equal to the entropy found in all per-CPU entropy pools managed by the hash operation and the auxiliary pool. Again, the entropy generated by the hash will never be larger than the digest size of the used hash which is compliant to [11] section 3.1.5.1.1 table 1.

[17] Section 4.2 outlines that the LRNG collection size can be modified at compile time where the default is 64. When a different collection size is chosen, the value needs to be adjusted accordingly. Yet, such modified value has no impact to the entropy analysis.

As stated in [11] section 3.1.5.1.2, vetted conditioning components are allowed to claim full entropy. In case of full entropy, the following is applied:

- $h_{out_{SHA-512}} = nw_{SHA-512} = n_{out_{SHA-512}} = 512$,
- $h_{out_{SHA-256}} = nw_{SHA-256} = n_{out_{SHA-256}} = 256$, or
- $h_{out_{SHA-1}} = nw_{SHA-1} = n_{out_{SHA-1}} = 160$.

Based on that conclusion, the entropy rate for each processing step given in section 2.2 can be illustrated in the following. This entropy assessment uses n_{out} which depends on the chosen hash operation with the respective value listed above.

The entropy for the individual time stamps is defined with the following equation applicable when a high-resolution timer is present – the absence of a high-resolution timer automatically implies the LRNG is treated as non-compliant to SP800-90B:

$$h_{t_8} = h_{t_{32}} = 1 \tag{3.1}$$

The entropy present in the arbitrary 32 bit word that may be added to the per-CPU data array is defined with:

$$h_{a_{32}} = 0 \tag{3.2}$$

The entropy in the concatenated time stamps found in the per-CPU data array is calculated as the sum of all time stamps (truncated or not) present in the per-CPU data array:

$$h_{per-CPU\ data\ array} = \sum_{n=0}^{number\ time\ stamps} (h_{t_{\{8,32\}}})_n \tag{3.3}$$

For the maintenance of the per-CPU entropy pool as specified by equation 2.2, the following entropy rate applies. This formula implies that each per-CPU entropy pool holds the sum of the entropy of the received data capped by the message digest size. This operation implies that the used hash compresses of the entropy available in the different input data.

$$h_{per-CPU\ pool} = min(h_{per-CPU\ data\ array} + h_{per-CPU\ pool}, n_{out}) \tag{3.4}$$

Similarly, the following equation applies to the entropy of the auxiliary pool maintenance as specified by equation 2.3. This formula implies that auxiliary pool holds the sum of the entropy of the received data capped by the message digest size. Again, this operation implies that the used hash compresses the entropy available in the different input data.

$$h_{aux\ pool} = min(h_{in_{aux\ pool}} + h_{aux\ pool}, n_{out}) \tag{3.5}$$

The following equation applies when calculating the slow noise source output buffer before its truncation as specified by equation 2.4. The formula implies that the slow noise source buffer before truncation holds the sum of the entropy of all per-CPU entropy pools plus the auxiliary pool capped by the message

digest size. Again, this operation implies that the used hash compresses the entropy available in the different input data.

$$h_{hash\ pools} = min(\sum_{n=0}^{max\ CPU} h_{per-CPU\ pool_n} + h_{aux\ pool}, n_{out}) \tag{3.6}$$

The entropy present in the truncated slow noise source buffer is the minimum of the entropy found in the pools and the requested amount of bits which is equal to the security strength of the DRBG:

$$requested\ size_s = security\ strength = 256 \tag{3.7}$$

$$h_s = min(h_{hash\ pools}, requested\ size_s) \tag{3.8}$$

The result of the formulas show that the entropy is simply a sum of the entropy of all input events capped to the message digest size of the used hash operation.

When generating the random numbers filling the slow noise source buffer, the entropy is debited in the following steps. First the entropy estimator of the auxiliary pool is reduced as much as possible: either by h_s or at most to zero. If not all entropy of h_s could have been debited from the auxiliary pool entropy estimator, then the yet not debited part of h_s is debited from the per-CPU entropy pool entropy estimators.

For example, assume that after the generation of random numbers and filling the slow noise source buffer its entropy is $h_s = 256$. Assume further, the auxiliary pool contains $h_{aux\ pool} = 155$ and the per-CPU entropy pools of the assumed 2 CPUs contain $h_{per-CPU\ pool_{CPU0}} = 80$ and $h_{per-CPU\ pool_{CPU1}} = 123$. The debit operation performs:

1. $h_{aux\ pool} = 155 - 155 = 0$ leaving $h_{s_{not\ debited}} = 256 - 155 = 101$
2. $h_{per-CPU\ pool_{CPU0}} = 80 - 80 = 0$ leaving $h_{s_{not\ debited}} = 256 - 155 - 80 = 21$
3. $h_{per-CPU\ pool_{CPU1}} = 123 - 21 = 102$

3.2.7 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 2.5.4.

- 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 2.5.4.

- A user-space RNGD is allowed to feed entropy into the LRNG via the `RNDADDENTROPY` IOCTL. This data is received and processed by the LRNG with an approved hash. The data is stored in the auxiliary pool. Similarly all other user space data is fed into the auxiliary pool.

- A kernel space hardware noise source is allowed to feed entropy into the LRNG via the `add_hwgenerator_randomness` function. The data is processed identically to user space entropy data by applying an approved hash and storing it in the auxiliary pool.

Additional data that is not treated as noise source data can be injected into the LRNG but that is not credited with entropy, but received from in-kernel sources such key codes from HID devices. This additional data is processed by the vetted conditioning component of the hash before it is injected as seed data into the DRNG. Thus, this operation complies with the last paragraph of section 3.1.6.

All random data from all noise sources are either concatenated or are processed by a vetted conditioning component before it is used to seed the DRNGs as allowed by SP800-90C [2] section 5.3.4.

3.2.8 SP800-90B Section 3.2.1 Requirement 1

This entire document is intended to provide the required analysis.

3.2.9 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.

3.2.10 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.

3.2.11 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.

3.2.12 SP800-90B Section 3.2.1 Requirement 5

The per-CPU entropy pools as processed by the hash is the output of the interrupt noise source. I.e. the entropy pools maintained by the hashing operation holds the data that is given to the DRNG when requesting seeding.

The noise source output without the hashing operation 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 per-CPU data array 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_RAW_HIRES_ENTROPY`. This option should not be set in production kernels.

3.2.13 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.

3.2.14 SP800-90B Section 3.2.1 Requirement 7

See section 3.2.4 for a description of the restart test.

3.2.15 SP800-90B Section 3.2.2 Requirement 1

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

3.2.16 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.

3.2.17 SP800-90B Section 3.2.2 Requirement 3

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

A stochastic model is not provided.

3.2.18 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.

3.2.19 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.

3.2.20 SP800-90B Section 3.2.2 Requirement 6

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

Although the hash 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.

3.2.21 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.

3.2.22 SP800-90B Section 3.2.3 Requirement 1

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

3.2.23 SP800-90B Section 3.2.3 Requirement 2

The used hash implementations for the conditioning components functions are all ACVP-testable. The LRNG offers an ACVP interface to ensure also the built-in SHA-256 and SHA-1 implementations are testable.

3.2.24 SP800-90B Section 3.2.3 Requirement 3

For the defined hashes, no key is required.

3.2.25 SP800-90B Section 3.2.3 Requirement 4

For the defined hashes, no key is required.

3.2.26 SP800-90B Section 3.2.3 Requirement 5

The conditioning component is the hash operation. See section 3.2.6 for a discussion of the narrowest internal width and the output block size.

3.2.27 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.

3.2.28 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.

3.2.29 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.

3.2.30 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.

3.2.31 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.

3.2.32 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.

3.2.33 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.

3.2.34 SP800-90B Section 4.3 Requirement 1

The implemented health tests comply with SP800-90B sections 4.4 as described in section 3.2.43.

3.2.35 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.

3.2.36 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. $H = 1$[18].

3.2.37 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.

3.2.38 SP800-90B Section 4.3 Requirement 5

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

3.2.39 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 per-CPU data array and further processed with the hash conditioning component.

3.2.40 SP800-90B Section 4.3 Requirement 7

The health tests are documented with section 2.5.4.

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. If the test fails, the entire LRNG is reset to drop all existing entropy and the startup testing is performed again.

3.2.41 SP800-90B Section 4.3 Requirement 8

There are no currently known suspected noise source failure modes.

3.2.42 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.

3.2.43 SP800-90B Section 4.4

The health tests described in section 2.5.4 are applicable to cover the requirements of SP800-90B health tests.

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

[18] Note, the referenced Excel function seems to be imprecise when calculating the value. The data has been obtained using R-Project with the formula of $1 + qbinom(1 - 2^{-30}, 512, 2^{-1})$.

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 31 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 concatenated with the per-CPU data array. 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[19].

- Filling up the LRNG with entropy using either a user-space RNGD via the IOCTL `RNDADDENTROPY` or a kernel-space via the function `add_hwgenerator_randomness` is allowed. However, the caller is only allowed to claim entropy associated with the data and thus increase the LRNG entropy estimation if the noise source is SP800-90B compliant with its own entropy assessment.

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

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.

Only data obtained from the potentially blocking output interfaces of the LRNG are SP800-90B compliant. The following interfaces are DRG.3 compliant:

- /dev/random,

- `getrandom` system call invoked with a zero flag value,

- invoking the in-kernel `get_random_bytes` or `get_random_bytes_full` API call when the callback registered with `add_random_ready_callback` was invoked.

Any other interface is not considered to provide SP880-90B compliant data.

Note, invoking the in-kernel `get_random_bytes` API call after the `wait_for_random_bytes` API call returns is not considered to be SP800-90B compliant because this call does not validate whether the SP800-90B startup tests are complete. This function could be transformed to be SP800-90B compliant by changing the code to wait for `lrng_state_operational` instead of `lrng_state_min_seeded`.

[19] At the time of writing, the user space Jitter RNG is SP800-90B compliant. Patches ensuring the in-kernel variant is SP800-90B compliant as well when into the kernel for version 5.8.

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/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 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 DRBG is instantiated using input from the noise sources.

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.

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 full entropy definition is not applied for seeding the DRBG. This means that process is described in section 9.4.2 of [2] is not used to seed the 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 DRBG in `lrng_drng_seed` requires calling of the entropy pool's hash 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 2.5.4.

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 per-CPU entropy pools with their hash output function is an NTG.1 which uses the interupt entropy source. Each per-CPU entropy pool has

an entropy estimation associated with it. The generation of the data the deterministic random number generator instances 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 is the hash operation and output function ψ is the hash function operated with the chosen hash. When obtaining data from the per-CPU entropy pools, the LRNG ensures that each generated random number must be backed by an equal amount of entropy that was mixed into the per-CPU entropy pool. Hence, the data derived from the per-CPU entropy pools are backed by information theoretical entropy.

- The blocking output interfaces of the LRNG are 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. The following interfaces are DRG.3 compliant:
  - /dev/random,
  - `getrandom` system call invoked with a zero flag value,
  - invoking the in-kernel `get_random_bytes` API call when the callback registered with `add_random_ready_callback` was invoked,
  - invoking the in-kernel `get_random_bytes` API call after the `wait_for_random_bytes` API call returns – note, service functions like the `get_random_XXX_wait` API call family where XXX is either u32, u64, int or long fall into this category.

  Any other interface is not considered to provide DRG.3 compliant data.

3.7.1 NTG.1 Compliant Configuration

To due to the non-blocking behavior of the LRNG, it is not considered to operate as an NTG.1. The code provided with the function generate_ntg1 is considered to be very close to an NTG.1. Yet an inherent race-condition does not guarantee that the caller triggering the reseed will also obtain the first random numbers from the LRNG. Thus, the solution is not fully and exactly NTG.1 compliant.

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 DRNG is minimally seeded 1.7 seconds after boot. This is around the time when the initramfs is started. The DRNG 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.

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_drng: force reseed of 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 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	90
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

Table 1: Average Cycle Count To Process One Interrupt Depending on Enabled Functionality

The LRNG allows a compile-time option to set the collection size which defines the size of the per-CPU data array. The table above shows the measured number for the default collection size of 64 entries and the use of the accelerated AVX2 SHA-512 hash operation. The following graph shows the average cycle count for processing an interrupt depending on the collection size, the used hash implementation (either the software SHA-256 provided with the ChaCha20 DRNG or the AVX2 SHA-512 implementation used with the DRBG). Finally, the graph shows the legacy /dev/random value as reference.

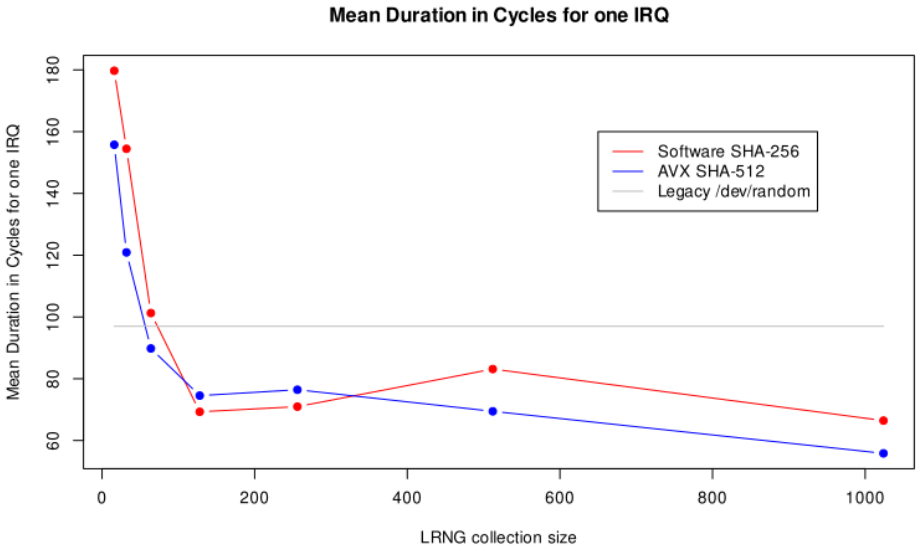

Figure 4.1: Average Cycle Count To Process One Interrupt Depending on Collection Size

The graph shows that when using an accelerated hash implementation, the average cycle count decreases. When increasing the collection size, the average cycle count increases as well. Finally, the graph shows that the default collection size shows about the same performance as the legacy /dev/random. The question must be raised, why not use the largest supported collection size as default? The reason is the goal that the LRNG shall deliver entropy fast

during boot time. The collected entropy is only available to the LRNG when it is injected into the per-CPU entropy pool. The injection occurs only when the per-CPU data array is completely filled. When the data array is large, it takes longer before the entropy is available to the LRNG to seed the DRNG. Thus, the default collection size is chosen to show a performance en-par with the legacy /dev/random which also ensures a fast entropy collection during boot time. Yet, a user can select a different size during compile time as needed.

4.3 LRNG Output 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` 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 2. 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	13.1 MB/s
Hash DRBG	SHA-512	AVX2	32 bytes	25.9 MB/s
Hash DRBG	SHA-512	AVX2	64 bytes	51.1 MB/s
Hash DRBG	SHA-512	AVX2	128 bytes	83.3 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	24.4 MB/s
CTR DRBG	AES-256	AES-NI	32 bytes	49.3 MB/s
CTR DRBG	AES-256	AES-NI	64 bytes	96.2 MB/s
CTR DRBG	AES-256	AES-NI	128 bytes	177.1 MB/s
CTR DRBG	AES-256	AES-NI	4096 bytes	1.247 GB/s
ChaCha20	ChaCha20	C	16 bytes	42.0 MB/s
ChaCha20	ChaCha20	C	32 bytes	84.5 MB/s
ChaCha20	ChaCha20	C	64 bytes	131.0 MB/s
ChaCha20	ChaCha20	C	128 bytes	194.7 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 2: LRNG performance on 64-bit

In addition, table 3 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 3: LRNG 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 Legacy /dev/random Non-Compliance with SP800-90B

In addition to the general concerns regarding the design and implementation of the legacy /dev/random and their coverage in the LRNG given in [8] section 4.4, the following list enumerates the areas of non-compliance of the legacy /dev/random with SP800-90B. As this document does not claim to provide an SP800-90B entropy analysis of the legacy /dev/random, it is possible that more areas of non-compliance are identified.

The legacy /dev/random implementation does not contain a repetitive count test (RCT) and adaptive proportion test (APT) or a suitable alternative as mandated in [11] sections 4.4 and 4.5. This includes neither a start-up health test nor a run-time health test.

As mandated in [11] section 3.1.6, multiple noise sources are allowed but only one noise source is to be credited with entropy. In particular the second paragraph prohibits the crediting of entropy to closely related noise sources.

The legacy /dev/random credits entropy to HID and block device events and at the same time interrupt events. However, each HID and block device event will always show up as an interrupt event as well considering that each HID and block device is interacted with using interrupts. Thus, HID and block device events are derivatives of interrupt events with respect to their entropy. Such double counting of entropy events are prohibited by [11] section 3.1.6.

When using multiple noise sources such as `add_disk_randomness`, `add_input_randomness` or `add_interrupt_randomness`, [11] section 3.1.6 requires the use of a vetted conditioning component. However, the legacy /dev/random does not use any vetted conditioning component.

To comply with SP800-90B, [11] section 3.1.5 requires an estimation of the entropy behavior of the conditioning components. Such estimation is considered to be a challenge to obtain due to the following different conditioning components implemented by the legacy /dev/random and applied to data believed to contain entropy:

- Some form of LSFR is implemented in the function `crng_slow_load`.
- The LFSR applied to the fast_pool state with 4 words when injecting new data must be assessed.
- The LFSR used for the input_pool must be assessed.
- The conditioning component provided with the SHA-1 operation reading the input_pool whose output is folded in half must be assessed. This operation is almost a vetted conditioning component compliant to [11] section 3.1.5.1.1 with the exception that the output of the SHA-1 operation is folded in half. Although this document does not contain any analysis of the legacy conditioning components, the reader is reminded of table 1 of [11] section 3.1.5.1.1 which outlines the narrowest internal width of a vetted conditioning component. For the hash operation it is marked as the hash-function output size. Considering that the used operation is almost a vetted conditioning component where only the output size is 80 bits due to the folding operation, a careful analysis must be applied whether the SHA-1 operation and its output-folding operation only delivers 80 bits of output length as listed in table 1. If this is the case, it is very likely that the legacy /dev/random entropy rate is limited to 80 bits of entropy due to this operation. It is also to be noted that the SHA-1 operation does not comply to the specification, such as that it does not use the correct initialization vector and it does not perform the finalization operation including the padding specified in section 5.1.1 [10].
- The ChaCha20 DRNG used to provide random numbers via the output interfaces must be assessed as well.

Starting with kernel 5.8, a patch is added to the legacy /dev/random which reads one 32-bit word straight from one fast_pool and injects that data into the external random32 random number generator every time an interrupt is received. Yet, the legacy /dev/random uses that same data to update its input_pool with that data. The external random32 random number generator is a non-cryptographic RNG using its data for network related operations where the generated random numbers are visible to external entities. It is unclear

how much entropy is lost due to this operation. Yet, the fact that data that is believed to hold entropy is extracted from the legacy /dev/random while being processed and at the same time being credited with entropy by the legacy /dev/random is considered to violate basic fundamental design requirements in [11] section 2.2.

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 SP800-90B Entropy Measurements

The following table presents the SP800-90B entropy measurements indicating whether the found entropy is sufficiently high to support the entropy analysis given in section 3.2.5. Entropy values are given in bits and apply to the entropy found in one time stamp generated when receiving an interrupt event. The testing shown in this section provides the quantiative foundation of the entropy analysis compliant to sections 3.2.6 as well as all other assessments required for SP800-90B.

The testing collected raw unconditioned time stamps as delivered by the file `/sys/kernel/debug/lrng_testing/lrng_raw_hires`. The entropy calculation is based on 1,000,000 raw time stamps collected by the LRNG. To speed up the raw time stamp collection as well as to obtain a worst-case assessment, all test systems were either ping-flooded or within an SSH-session a `find /` was executed to generate a large number of interrupts in a short amount of time. The ping-flood generator was in close network proximity (e.g. KVM host, or a system at most one switch away from the test system).

The entropy result listing in the table below is generated as follows. The time stamps generated by the LRNG for each interrupt event is extracted and concatenated to form a bit-stream. This bit stream is processed by the NIST SP800-90B entropy analysis tool to obtain an entropy rate. This entropy rate is listed below. As the 8 least significant bits (LSB) of the time stamp are used and the other bits are ignored by the LRNG, the entropy rate applies to those 8 data bits. As discussed in sections 3.2.6, the LRNG assumes that each time stamp provides at least slightly more than one bit of entropy. As all values

in the table below show significantly more entropy even with the worst-case measurement of 8 LSB, the LRNG underestimates the entropy existing in the respective system. Thus, the LRNG is considered to operate securely on these systems. The test complies with SP800-90B outlined in section 3.2.3.

Test System	Entropy of 1,000,000 Traces	Sufficient Entropy
ARMv7 rev 5	1.9344	Y
ARMv7 rev 5 (Freescale i.MX53)[22]	7.07088	Y
ARM 64 bit AppliedMicro X-Gene Mustang Board	5.599128	Y
Intel Atom Z530 – using GUI	3.38584	Y
Intel i7 7500U Skylake - 64-bit KVM environment	3.452064	Y
Intel i7 8565U Whiskey Lake – 64-bit KVM environment	7.400136	Y
Intel i7 8565U Whiskey Lake – 32-bit KVM environment	7.405704	Y
Intel i7 8565U Whiskey Lake	6.871	Y
Intel Xeon Gold 6234	4.434168	Y
IBM POWER 8 LE 8286-42A	6.830712	Y
IBM POWER 7 BE 8202-E4C	4.233912	Y
IBM System Z z13 (machine 2964)	4.366368	Y
IBM System Z z15 (machine 8561)	5.691832	Y
MIPS Atheros AR7241 rev 1[23]	7.157064	Y

Table 5: LRNG Entropy Testing Results on Different Hardware

Some of the tested systems are quite old or are small embedded devices demonstrating that even on older and smaller systems the LRNG does not overestimate the available entropy when applying worst case conditions.

I am looking for test data from all kinds of systems. The less common a system is the more I am interested in the data to verify that the basic entropy estimate underlying the LRNG is correct. If you want to provide support, please generate data using the LRNG test tool set specifically the test as documented in `sp80090b/recording/raw_entropy/README.md`.

D Auxiliary Testing

In addition to the testing conducted in appendix C, the following tests were executed on all systems.

Stress testing (provided with the `swap_stress.sh` test script): A continuous read operation on /dev/urandom is started with as many parallel threads as CPUs, one continuous read operation on /dev/random is started, and one

[22] USBArmory MK I

[23] Ubiquiti Nanostation M5 (xm)

continuous read operation on /proc/lrng_type is started. While the read operations are performed, 5,000 insmod / rmmod operations of the `lrng_drbg.ko` kernel module is performed to change the DRNG type and the read hash of the entropy pool. A test that runs to completion shows that the locking of the LRNG does not show deadlocks or unprotected critical code paths.

Performance testing (provided with `lrng_get_speed.sh` and `speedtest.c` test code): The performance of the legacy /dev/random as well as the LRNG for its ChaCha20 and all SP800-90A DRBG types is recorded. The LRNG ChaCha20 DRNG is commonly significantly faster compared to the legacy DRNG. The performance of the different DRBGs depends on the availability of accelerated cryptographic support. If such support is present, the DRBG may reach the ChaCha20 performance and the CTR DRBG for larger block sizes it may greatly exceed the ChaCha20 performance.

The self tests implemented when enabling `CONFIG_LRNG_SELFTEST` are verified to run successfully.

The boot process was analyzed to verify that the LRNG is fully seeded on all systems around the time when the hard disks are mounted by the boot environment. This implies that a fully seeded LRNG is available at the time cryptographic user space services such as OpenSSH are started.

E 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. /dev/random and sp800-90b. International Cyptographic Module Conference (ICMC), 2015.

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

[8] Stephan Müller. *Documentation and Analysis of the Linux Random Number Generator*. 4.1 edition, 2020.

[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] NIST. *FIPS PUB 180-4 Secure Hash Standard (SHS).* 2011.

[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.

F License