Linux /dev/random – A New Approach

Stephan Müller <smueller@chronox.de>

March 21, 2024

Abstract

The kernel crypto API contains deterministic random number generators (DRNG) which a caller must seed and reseed. The task of seeding a DRNG is a non-trivial task requiring the consideration of a significant number of aspects. The Linux Random Number Generator (LRNG) fills that gap to transparently seed and reseed DRNGs. A user of the LRNG obtains random numbers from an appropriately seeded and initialized DRNG. Further, the LRNG controls various entropy sources guaranteeing that they are properly initialized and managed.

Contents

List of Figures

List of Tables

1 Introduction

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

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

1. The LRNG manages the proper seeding and reseeding of DRNGs. In addition, it provides internal entropy sources which the LRNG fully controls as well as interfaces to obtain data from external entropy sources.

2. 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 6 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.

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 interrupt entropy source of the LRNG is hooked into the interrupt handler and therefore should finish the code path in interrupt context very fast. Similarly, the scheduler entropy source hooks into the scheduler operation requiring to be very lightweight.

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 conditioning.

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 entropy source maintenance to allow a replacement of these components.

11. The LRNG shall offer flexible configurations allowing vendors to apply the settings applicable to their environment.

12. When enabling the LRNG interface to behave like /dev/random, it provides an API and ABI compliant implementation of all interfaces allowing to serve as a drop-in replacement.

1.1 Properties Offered by the LRNG

Apart from the fact that a user does not need to manage the DRNG and its seeding status, the LRNG provides the following properties making the LRNG a contemporary and future-proof entropy source and DRNG management framework:

- 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 6.5).

- Performance
  - Faster by up to 130% in the critical code path of the interrupt handler depending on data collection size configurable at kernel compile time as outlined in section 6.2
  - Block device data path is not instrumented by LRNG which implies that the LRNG does not add any delays compared to the legacy /dev/random.
  - 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 3.
  - Faster entropy collection during boot time to reach fully seeded level, including on virtual systems or systems with SSDs as outlined in section 6.1.
  - High-performance implementation of scheduler entropy source requiring only a few cycles to collect entropy.

- Testing
  - Availablility of run-time health tests of the raw unconditioned entropy source data of the interrupt entropy source to identify degradation of the available entropy as documented in section 2.5.2. Such health tests are important today due to virtual machine monitors reducing the resolution of or disabling the high-resolution timer.
  - Heuristic entropy estimation for the interrupt entropy source 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 [2] 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 2.16.

  - Availability of test interfaces for all operational stages of the LRNG including boot-time raw entropy event data sampling as outlined in section 2.17.
  - 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 2.17.
  - Availability of stress testing covering the different code paths for data and mechanism (de)allocations and code paths covered with locks.
  - Availability of fully automated regression testing covering different LRNG functions in different configurations allow an unattended functional verification testing.

- Entropy collection of the internal interrupt and scheduler-based entropy source
  - 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 6.5.
  - Full entropy assessment and description is provided with chapter 3, specifically section 3.3.6.
  - The LRNG provides a configuration to be compliant to SP800-90C following RBG2(NP) as well as RBG2(P) construction methods.
  -

- 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.
    - ≻ Independent enabling of the available entropy source.
    - ≻ Enabling of internal entropy 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.
    - ≻ Selectively enabling of each entropy source and configuring their entropy rate.
    - ≻ The entropy rate used to credit the internal interrupt entropy source, the external CPU-based entropy source and Jitter RNG entropy source can be configured including setting an entropy rate of zero or full entropy – see sections 2.10.1 and 2.9.1. A rate of zero implies that the entropy source still provides data which is credited with zero bits of entropy.
    - ≻ Enabling the API and ABI compatible drop-in replacement for /dev/random.
  - At boot-time, the SP800-90B health tests for the internal interrupt entropy source can be enabled as outlined in section 2.5.2.At compile time, the entropy rate used to credit the internal and external entropy sources can be configured including setting an entropy rate of zero or full entropy – see sections 2.5.1, 2.10.1 and 2.9.1.
  - Configurable seeding strategies are provided following different concepts.

- 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.
    - ≻ Default of SHA-512 hash using the fastest hash implementation available via the kernel crypto API SHASH subsystem. During boot time or runtime, other hashes can be selected.

1.2 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 of the internal interrupt entropy source including the SP800-90B compliance assessment is provided in chapter 3.
- The statistical testing of the internal scheduler entropy source including the SP800-90B compliance assessment is provided in chapter 4.
- The discussion of various configurations offered by the LRNG is given in chapter 5.
- The comparison of the LRNG with the legacy /dev/random is covered in chapter 6.
- 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.

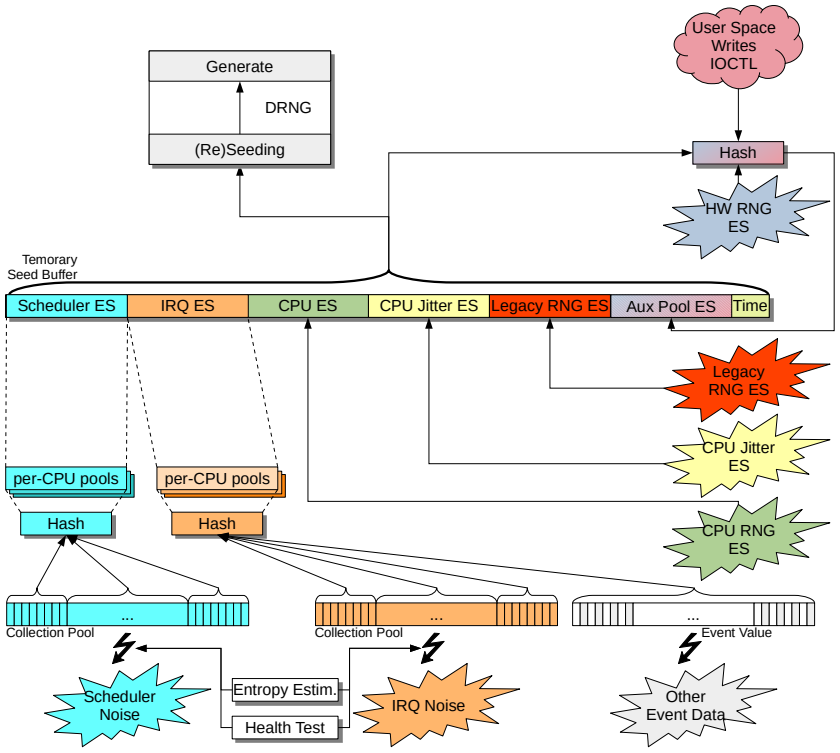

Figure 2.1: LRNG Big Picture

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

The LRNG introduces the concept of slow and fast entropy sources. Fast entropy sources provide entropy at the time of request. A slow entropy source collects data over time into an entropy pool.

The entropy source that is under full control of the LRNG, also called the internal entropy source, comprises of the:

- Orange marked parts consisting of the IRQ entropy source that feeds into the per-CPU entropy pool from which the hash of pools is derived.
- The indigo marked parts refer to the scheduler entropy source that feeds per-CPU collection pools. These collection pools are hashed into per-CPU entropy pools when seeding of the DRNG is to be performed.

The following external entropy sources are present. These entropy sources are expected to be fully self-contained. The LRNG only requests data from them and expects that the entropy estimate provided with the data is correct:

- The auxiliary pool collects data from external entropy sources which deliver data at times not controllable by the LRNG.
- The CPU entropy source obtains data from a potentially existing source in the CPU like RDSEED on Intel CPUs.
- The Jitter RNG entropy source is another external entropy source.
- The legacy random number generator provided with the kernel file `random.c` can be enabled at compile time to serve as an entropy source. Its implementation of `get_random_bytes` is used to provide data to the LRNG. As the legacy RNG rests on collecting interrupts and its derivatives, the LRNG's internal entropy source recording interrupt occurrences is automatically deactivated.

The LRNG treats all external and internal entropy sources equally. It can handle the situations where one or more entropy sources returns little or no entropy.

The IRQ entropy source is considered to be the internal entropy source which delivers entropy as assessed in chapter 3. In addition, the scheduler entropy source is assessed in chapter 4 which provides its complete entropy assessment. All other entropy sources are expected to provide their own entropy assessment supporting the claim of the supplied entropy that is credited by the LRNG for these sources. The LRNG treats these additional entropy sources as black boxes and take their claimed entropy rate at face value. The LRNG, however, guarantees that all entropy sources are processed in compliance with defined standards as documented in chapter 3.

The "other event data" is considered to provide additional data that contains no entropy. However, the LRNG collects it to ensure further mixing of the internal state.

The different colors used in figure 2.1 depict the different entropy sources mentioned before.

The LRNG offers various interfaces to obtain random numbers as depicted in figure 2.2.

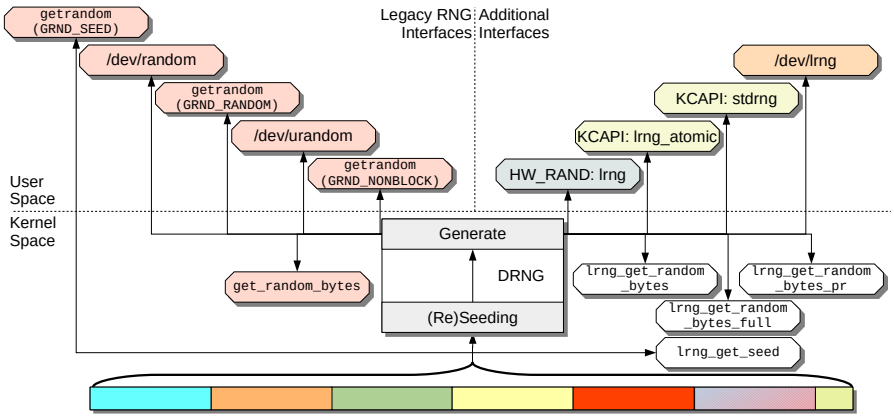

Figure 2.2: LRNG Interfaces To Obtain Random Numbers

Based on this figure, the following types of interfaces are available which are marked with the different colors in figure 2.2. Each type of interface can be selectively enabled and disabled at compile time:

- Legacy RNG interfaces: The common interfaces found in today's kernels include /dev/random, /dev/urandom, the getrandom system call or the `get_random_bytes` kernel-internal interface. These interfaces behave identically to the legacy RNG and thus allow the LRNG to act as a drop-in replacement. When not compiling those, the LRNG can be operated in parallel with the legacy RNG.

- Hardware Random Number Generator framework: The HW_RAND framework can access the LRNG which is named "LRNG". This implies that user space can access the LRNG via /dev/hwrng.

- Kernel Crypto API RNG framework: The kernel crypto API RNG framework may access the LRNG via the names of either "stdrng" – providing access to a guaranteed fully seeded and fully initialized LRNG – and "LRNG_atomic" which accesses the LRNG interface that can be used in atomic contexts.

- The device file of /dev/lrng behaves identically to the /dev/random device file including read/write operations, sleeping on it with the `select(2)` system call or using all IOCTLs defined for /dev/random. This file is intended for the case where the legacy RNG is still compiled but user space shall be capable of accessing the LRNG with all its properties.

- When reading data from /dev/random that was opened with `O_SYNC`, or using `getrandom(2)` with `GRND_RANDOM` a special DRNG instance that is continuously reseeded is used. This implies the DRNG operates with prediction-resistance (SP800-90A terminology) or is NTG.1 compliant (German BSI AIS 20/31 from 2011). In addition, when the kernel executes in FIPS mode, the behavior complies with all FIPS requirements allowing the output of the LRNG to be claimed to be from a vetted conditioning component. I.e. the caller may safely use this data to seed another

DRBG. To achieve this, the following checks are applied in the function `lrng_drng_get`:

  - Require a reseed - in FIPS mode, the LRNG requires the availability of at least 256 bits of entropy in its entropy sources for the reseed to commence.
  - Produce at most only 256 bits of random bits from the DRNG (i.e. the security strength of the DRNG).

- The kernel-internal interfaces of `lrng_get_random_bytes`, `lrng_get_random_bytes_full` and `lrng_get_random_bytes_pr` are always provided.

2.1 LRNG Components

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

1. The LRNG implements a DRNG. Unless using the interface providing prediction resistance, 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 sources with the available entropy.

   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) All interfaces using the `lrng_get_entropy_bytes` function always generates data including when the LRNG is not properly seeded.

   (b) All interfaces using the `lrng_get_entropy_bytes_full` function generate data only when the LRNG is fully seeded and fully initialized.

   (c) All interfaces using the `lrng_get_entropy_bytes_pr` function generate data only when the LRNG is fully initialized and received a reseeding from the entropy sources immediately before the random bits are generated. Only an amount of random bits are generated that equals to the amount of entropy the DRNG was seeded with.

2. The DRNG is seeded by concatenating the data from the following sources in case they are enabled at kernel compile time:

   (a) the output of the auxiliary pool,

   (b) the output of the per-CPU interrupt entropy pools,

   (c) the output of the per-CPU scheduler entropy pools,

   (d) the Jitter RNG if available,

   (e) the CPU-based entropy source such as Intel RDSEED if available, and

   (f) the legacy RNG output.

The entropy estimate of the data of all entropy 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 entropy sources can dominate the other entropy sources to provide seed data to the DRNG due to the following:

(a) During boot time, the amount of available entropy is the trigger point to (re)seed the DRNG following the explanation in the next section.

(b) At runtime, the the DRNG reseed is triggered by either the DRNG due to hitting the aforementioned thresholds or by a user space caller. The reseed is never triggered by the entropy sources.

3. To support backward secrecy, the following steps are applied:

(a) The temporary seed buffer holding the concatenation of data from all entropy sources to seed the DRNG is 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 injection of the temporary seed buffer will not alter the entropy estimation of the auxiliary pool.

(b) The message digest created for each per-CPU entropy pool of the scheduler and interrupt entropy sources is inserted into the corresponding per-CPU entropy pool.

The LRNG allows the DRNG mechanism and the used hash to be changed at runtime. Per default, a ChaCha20-based DRNG is used[1] 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 different entropy source before injecting them into the DRNG is performed with the following mathematical operations. The 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).

2.2.1 Scheduler and Interrupt Entropy Sources

1. Truncation: The time stamps received by the IRQ as well as the scheduler entropy sources are truncated to 8 least significant bits (or 32 least significant bits during boot time) – note the GCD is a value calculated during initialization and is fixed thereafter which implies that the time stamp divided by the GCD is the raw entropy value: t_8 (or t_{32})

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

2. Concatenation: The time stamps received and truncated by the IRQ and scheduler entropy sources as well as auxiliary 32 bit words a_{32} are concatenated to fill the per-CPU collection pool that is capable of holding 1,024 8-bit words[2] - 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`:

$$CP = t_{8_{n-1019}}||a_{32_n}||t_{8_{n-1018}}|| \ ... \ ||t_{8_n} \tag{2.1}$$

Note: In case the continuous compression operation is disabled for the IRQ entropy source, the auxiliary 32 bit words a_{32} are discarded and are not injected into the collection pool. This approach is taken to prevent non-entropy data to potentially overwrite entropy data in the collection pool when the array wraps. The scheduler entropy source only records time stamps.

2.2.2 Interrupt Entropy Source

1. Hashing: For the IRQ entropy source, all concatenated time stamp data received from the interrupts since the last output generation of the per-CPU entropy pool $EP_{CPU_{n-1}}$ are hashed together with that last output $EP_{CPU_{n-1}}$ to generate new per-CPU entropy pool output of EP_{CPU_n}. The following steps are performed:

   (a) One filled per-CPU collection pool for the interrupt entropy source CP_{IRQ_m} is inserted into the per-CPU entropy pool using a hash update operation.

   (b) To generate data from the entropy pool EP_{CPU_n} as used by function 2.3, a hash final operation is performed.

   (c) Once a hash final operation is performed it is followed by an immediate re-initialization of the hash state with a hash init operation and adding the just calculated message digest with the first hash update.

   The implementation is provided with function `LRNG_pcpu_array_compress` together with the function `LRNG_pcpu_pool_hash_one` generating data from the per-CPU entropy pool:

$$EP_{CPU_n} = SHA(EP_{CPU_{n-1}}||CP_{IRQ_{m-(n-1)}}||...||CP_{IRQ_{m-1}}||CP_{IRQ_m}) \tag{2.2}$$

   Note: The hash update operation is performed at the following occasions:

   (a) Continuous compression enabled: The hash update is performed every time the collection pool is full. This operation therefore is performed in interrupt context. In addition, the operation is performed at the time operation of equation 2.3 is invoked which is in process context.

[2]The LRNG collection size is compile-time configurable where 1,024 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.

(b) Continuous compression enabled and disabled: The hash update is performed at the time the operation of equation 2.3 is invoked, i.e. at the time the DRNG is reseeded. This operation therefore is performed in process context. This guarantees that all unprocessed entropy data in the collection pool is added to the entropy pool at the time the entropy pool is requested for random data.

This implies that in case of disabled continuous compression, the oldest entries in the collection pool are overwritten with newer entropy event data when more entropy events are collected than can be held in the collection pool between DRNG reseeds.

2. Hashing: For the IRQ entropy source, a message digest of all per-CPU entropy pools is calculated. This message digest is used to fill the interrupt entropy source output buffer S discussed in the following - the implementation is provided with function `LRNG_pcpu_pool_hash`:

$$EP_{all_n} = SHA(EP_{CPU0_n}||EP_{CPU1_n}|| \ ... \ ||EP_{CPUX_n}) \tag{2.3}$$

3. Truncation: For the interrupt entropy pool, 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), whatever is smaller, are obtained from the interrupt entropy source output buffer S - the implementation is provided with function `LRNG_pcpu_pool_hash`:

$$E_n = MSB_{min(entropy,security\ strength)}(EP_{all_n}) \tag{2.4}$$

2.2.3 Scheduler Entropy Source

1. Hashing: For the scheduler-based entropy source, all concatenated time stamp data received from the interrupts since the last output generation of the per-CPU entropy pool $SP_{CPU_{n-1}}$ are hashed together with that last output $SP_{CPU_{n-1}}$ to generate new per-CPU entropy pool output of SP_{CPU_n}. The following steps are performed:

   (a) One filled per-CPU collection pool for the interrupt entropy source CP_{SCHED_m} is inserted into the per-CPU entropy pool using a hash update operation.

   (b) To generate data from the entropy pool SP_{CPU_n} as used by function 2.6, a hash final operation is performed.

   (c) Once a hash final operation is performed it is followed by an immediate re-initialization of the hash state with a hash init operation and adding the just calculated message digest with the first hash update.

   The implementation is provided with function `lrng_sched_pool_hash_one` generating data from the per-CPU entropy pool:

$$SP_{CPU_n} = SHA(SP_{CPU_{n-1}}||CP_{SCHED_{m-(n-1)}}||...||CP_{SCHED_{m-1}}||CP_{SCHED_m}) \tag{2.5}$$

Note: The hash update is performed at the time the operation of equation 2.6 is invoked, i.e. at the time the DRNG is reseeded. This operation therefore is performed in process context. This guarantees that all unprocessed entropy data in the collection pool is added to the entropy pool at the time the entropy pool is requested for random data. This implies that he oldest entries in the collection pool are overwritten with newer entropy event data when more entropy events are collected than can be held in the collection pool between DRNG reseeds.

2. Hashing: For the scheduler-based entropy source, a message digest of all per-CPU entropy pools is calculated. This message digest is used to fill the interrupt entropy source output buffer S discussed in the following - the implementation is provided with function `lrng_sched_pool_hash`:

$$SP_{all_n} = SHA(SP_{CPU0_n}||SP_{CPU1_n}|| \ ... \ ||SP_{CPUX_n}) \tag{2.6}$$

3. Truncation: Just like the interrupt entropy source, the scheduler entropy source applies a truncation to the generated data as implemented by the function `lrng_sched_pool_hash`:

$$S_n = MSB_{min(entropy,security\ \ strength)}(SP_n) \tag{2.7}$$

2.2.4 Auxiliary Entropy Pool

1. Hashing: When new data D_m is added to the auxiliary pool AP, the data is inserted into the auxiliary pool with a hash update operation - the implementation is provided with function `LRNG_pool_insert_aux`. The message digest generation is performed at the time entropy from the auxiliary pool is requested. To ensure backward secrecy, the temporary seed buffer T_{n-1} that holds among others the auxiliary pool digest from the previous generation round as depicted with equation 2.12 is concatenated with the received data:

$$AP_n = SHA(T_{n-1}||D_{m-(n-1)}||...||D_{m-1}||D_m) \tag{2.8}$$

2. Truncation: The MSB of the auxiliary pool of the size of the DRNG security strength are used for the seed buffer:

$$A_n = MSB_{min(digest\ size,security\ \ strength)}(AP_n) \tag{2.9}$$

2.2.5 CPU Entropy Source

1. Hashing: If the CPU entropy source provides less than full entropy, a message digest of the amount of data to be requested from it is calculated:

$$C_{cond} = SHA(C_1|| \ ... \ ||C_m) \tag{2.10}$$

2. Truncation: If the CPU entropy source provides less than full entropy, the MSB defined by the requested number of bits (commonly equal to the security strength of the DRBG) or the applied message digest size, what ever is smaller, are obtained - the implementation is provided with function `LRNG_get_arch_data_compress` – otherwise C_n is the data obtained directly from the CPU entropy source:

$$C_n = MSB_{min(digest\ size,security\ strength)}(C_{cond}) \tag{2.11}$$

2.2.6 Temporary Seed Buffer Construction

1. Concatenation: The temporary seed buffer T used to seed the DRNG at the time n is a concatenation of one or more of the following entropy source data sets, depending on the compile-time configuration:

   (a) the auxiliary pool entropy source A,

   (b) the interrupt entropy source buffer E,

   (c) the scheduler entropy source buffer S,

   (d) the Jitter RNG output J,

   (e) the CPU entropy source output C,

   (f) the legacy RNG entropy source output L, and

   (g) the current time t

   with the implementation is provided with function `lrng_fill_seed_buffer`:

$$T_n = A_n||E_n||S_n||J_n||C_n||L_n||t \tag{2.12}$$

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 entropy sources is collected and accumulated in the entropy pools which contain already compressed entropy data, supported by the collection pools which contain uncompressed, but only concatenated entropy data.

At the time the DRNG shall be seeded, the all entropy pools, any non-compressed data in the collection pools and the auxiliary pool are processed with a cryptographic hash function which can be chosen at runtime.

For the entropy pool, 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. For the auxiliary pool, always 256 bits of data are returned irrespective of the entropy rate of this pool. This ensures that also data that is not credited with entropy but injected into the LRNG is used to stir the seed for the DRNG.

The DRNG always tries to seed itself with 256 bits of entropy, except during boot. In any case, if the entropy 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 sources if all entropy sources collectively have at least 32 bits of entropy available. The goal of this step is to ensure that the DRNG receive some initial entropy as early as possible.

2. The DRNG is reseeded from the entropy sources if all entropy sources collectively have at least 128 bits of entropy available.

3. The DRNG is reseeded from the entropy sources if all entropy sources collectively have at least 256 bits of entropy available.

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.

The LRNG also implements a forced seeding operation. This is performed when a blocking interface has blocked a request 5 times because the DRNG is not fully seeded. The forced seeding performs a continuous reseeding of the DRNG in an atomic operation with existing entropy. This means that a regular reseeding is triggered repeatedly as often as needed to obtain 256 bits of entropy to reach the fully seeded level. In this case, the entropy obtained from the entropy sources is added up. This is permissible because the multiple reseeding operations are performed while the DRNG is locked, i.e. the DRNG will not produce output in that time.

Before the DRNG is seeded with 256 bits of entropy in step 3, requests of random data from the blocking interfaces are not processed.

At runtime 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 backward secrecy.

- The DRNG is reseeded with whatever entropy is available, but at least 128 bits (256 bits if SP800-90C compliance is enabled) – in the worst case where no additional entropy can be provided by the entropy 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[3], 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.

[3]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 entropy 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 entropy source within that time frame, satisfying the DRNG reseed requirement.

The chosen values prevent high-volume requests from user space to cause frequent reseeding operations which drag down the performance of the DRNG[4][5].

- If the DRNG was not reseeded for the last 2^{30} DRNG generate operations – i.e. the reseeding requests discussed in the previous bullets were unsuccessful – the DRNG reverts back to an unseeded state. This applies that the DRNG will not produce random numbers when accessed via the blocking interfaces. In this case, the DRNG behaves like during boot time.

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:

- 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` is woken up.

- 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. Also in-kernel callers waiting with `add_random_ready_callback` are woken up. Using the LRNG API of `LRNG_get_random_bytes_full`, the caller is waiting synchronously until the fully seeded level is reached. Finally, in-kernel callers can poll `rng_is_initialized` which returns true if the LRNG is fully seeded and initialized.

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

[4]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.

[5]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.

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 entropy sources as outlined in sections 2.10.1 and 2.9.1. In addition, this table considers the heuristic entropy rate outlined in equation 3.2 which implies that the time stamp delivered with one interrupt event provides one bit of entropy.

Seed Stage	Entropy source data bits	Entropy 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` blocked `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 entropy sources have the default entropy levels.

Seed Stage	Entropy source data bits	Entropy 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 entropy sources are configured to deliver zero bits of entropy.

Seed Stage	Entropy source data bits	Entropy 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. Figure 2.3 illustrates the seeding strategy showing that each DRNG instance is freshly seeded with a separate seed buffer.

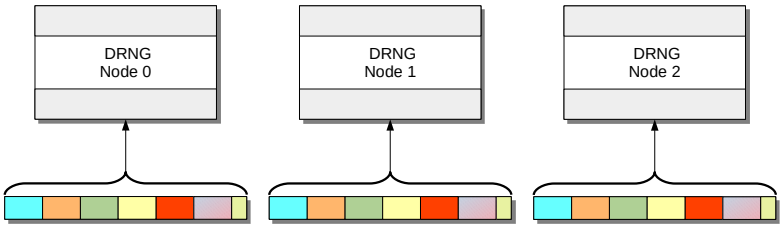

Figure 2.3: DRNG Instances on NUMA systems with seeding strategy

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_drng_cb` in "lrng.h" must be implemented. These functions cover the allocation/deallocation of the DRNG as well as its usage. Similarly, all functions in `struct lrng_hash_cb` from "lrng.h" must be implemented to provide the the conditioning hash.

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[6] for accessing the entropy pools.

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

In addition, the LRNG allows the addition of new entropy sources. The documentation provided in README.hacking.md in the LRNG source code tree documents how to add a new entropy source.

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. The LRNG covers this aspect by only sending data to the random32 PRNG that is not used by the LRNG. This change has been reverted with 5.10.

Additional concerns regarding the design and implementation of the legacy /dev/random and their coverage in the LRNG are given in [2] section 4.4.

2.4 LRNG Data Structures

The LRNG uses the following main data structures:

- The data from the interrupt entropy source is processed with a per-CPU entropy pool. In addition, a per-CPU collection pool that can hold the concatenated time stamps is maintained. Both are accessed lockless since the currently executing CPU's entropy pool and collection pool 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 scheduler entropy source also defines per-CPU collection pools like the interrupt entropy source. However, no entropy pools are maintained as compression of scheduler-based entropy sources is only performed when the DRNG shall be (re)seeded. The entire scheduler-based entropy source operates lock-less.

- 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 - LRNG-internal Entropy Source

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.

Figure 2.4 illustrates the interrupt processing performed by the LRNG. The figure specifies which parts of the interrupt processing execute in IRQ context and which executes in process context. The operations executed in interrupt context are all completely listed in this section. All steps executed in process context are illustrated in section 2.12.

The figure depicts the example when one interrupt arrives on CPU 0. If an interrupt arrives on another CPU, the same operation is applied, but the respective CPU-local collection pool and entropy pool is used. The entropy pools from other CPUs in the figure therefore are filled with the same processing steps, which, however, are not shown.

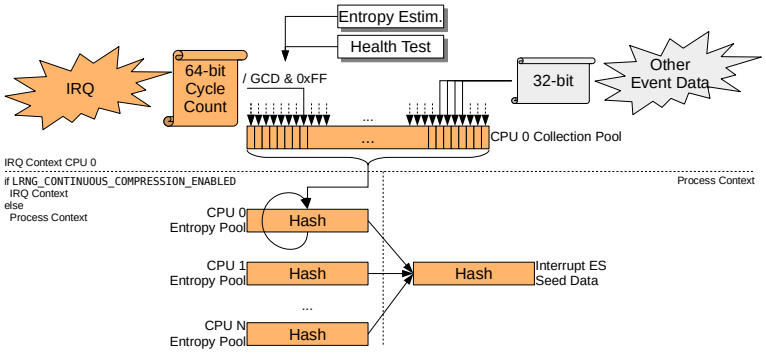

Figure 2.4: Interrupt Processing

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. This integer value is divided by a GCD to eliminate bits that do not change. 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 collection pool. After the receipt of 1,024 time stamps, the data collection pool with all concatenated time stamps is inserted into the currently executing CPU's entropy pool. During boot time until the LRNG completed the calculation of the GCD, 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 2.5.2 are performed on each received time stamp where the truncated time stamp value is forwarded to the health test. Unless 1,024 time stamps have been received, the processing of an interrupt stops now.

3. The per-CPU collection pool is added to the same CPU's entropy pool by performing a hash update operation. This approach works as the per-CPU entropy pool is managed as the message digest state. When data of the per-CPU entropy is to be extracted, a hash final operation is performed followed by an immediate re-initialization of the state buffer using the message digest of the previous extraction. This operation is depicted in figure 2.5 for the entropy pool maintained by CPU 0. The other CPUs perform the same processing with their independent copy of the collection pool and the entropy pool. In case the continuous compression support is disabled, the hash operation is not performed. Instead, the oldest entropy values in the collection pool are overwritten with the latest entropy value. In case the continuous compression operation is disabled, the hash update operation is conducted in process context at the time of obtaining random numbers from the entropy pool requested to seed the DRNG documented in section 2.12.

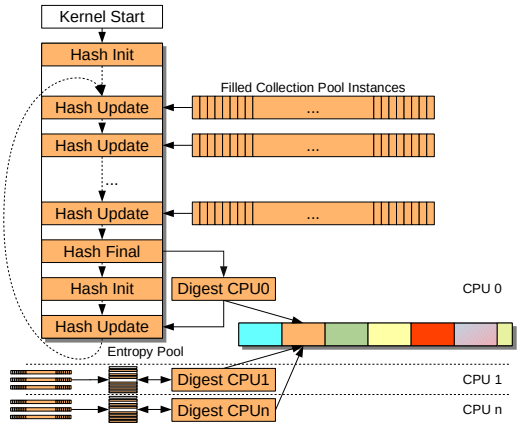

Figure 2.5: Collection Pool Processing

4. The LRNG increases the per-CPU counter of the received interrupt events by the number of healthy interrupts stored in the per-CPU collection pool. 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 entropy sources 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 2.12.

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 entropy source. The current implementation implicitly assumes one bit of entropy per time stamp obtained for one interrupt[7].

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.

With the kernel compile time parameter of CONFIG_LRNG_IRQ_ENTROPY_RATE the number of interrupts that must be collected to obtain 256 bits of entropy can be specified. This value is forced by the LRNG to be at least the aforementioned limit, i.e. 256 interrupts. This value is subject to the increase by the oversampling factor, if no high-resolution timer is found. The entropy value can be altered by setting the kernel command line option of LRNG_es_irq.irq_entropy if the kernel is compiled with CONFIG_LRNG_RUNTIME_ES_CONFIG.

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 3.3. Although only one bit of entropy is assumed to be delivered with a given time stamp the LRNG uses the 8 least significant

[7] 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.

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 1,024 time stamps received by the current CPU before those 1,024 bytes are injected into the per-CPU entropy pool. The performance of this approach is demonstrated with the measurements shown in section 6.2. 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.

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 (after the time stamp was divided by its GCD).

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 collection pool when the time stamp is collected – the LRNG still awards this time stamp one bit of entropy. Once the LRNG completed the calculation of the GCD the aforementioned runtime behavior of concatenating the 8 LSB of 1,024 time stamps before mixing them into the per-CPU entropy pool is enabled.

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

- Sources feeding the auxiliary entropy pool: The interface functions to provide entropy data (add_hwgenerator_randomness or the IOCTL `RNDADDENTROPY` have to be invoked with the value 0 for the entropy rate.
- CPU-based entropy source: `CONFIG_LRNG_CPU_ENTROPY_RATE=0`
- Jitter RNG: `CONFIG_LRNG_JENT_ENTROPY_RATE=0`

These options ensure that random data from the entropy 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 entropy 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 collection pool. Only if all three values are non-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 [5] 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 [5] 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 2.5.2. 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 entropy 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` as well as `LRNG_get_random_bytes` kernel-internal interface when being triggered by the callback registered with `add_random_ready_callback`
  - `LRNG_get_random_bytes_full` kernel-internal interface
- 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 collection pool for interrupts as well. This is depicted in figure 2.4 where the HID data is processed as the "other event data".

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

2.7 Scheduler Events - LRNG-internal Entropy Source

The LRNG hooks into the scheduling operation of the Linux kernel which is triggered every time a context switch is performed as initiated by the scheduler. The LRNG handling of a scheduling event is depicted with figure 2.6.

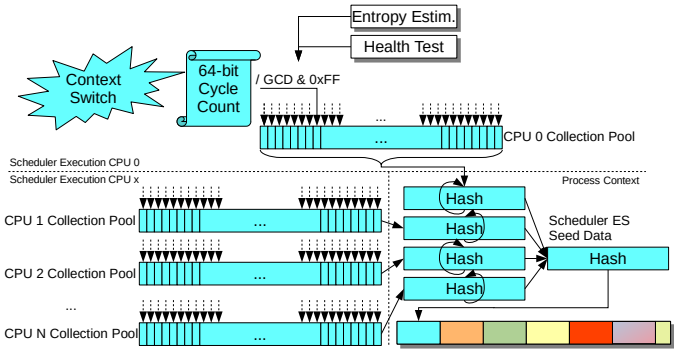

Figure 2.6: Scheduler-event Processing

When a context switch occurs, the LRNG callback is invoked which obtains a high-resolution time stamp that is concatenated into the CPU-local collection pool. The processing of the time stamp is identical to the interrupt entropy source as outlined in section 2.5. When the CPU-local collection pool is full, the oldest entries are overwritten by the latest time stamp.

At the time the DRNG shall be (re)seeded, a message digest of each scheduler per-CPU collection pool is calculated to fill the per-CPU scheduler entropy pool as outlined in figure 2.7. This figure shows that first the collection pool is inserted into a per-CPU scheduler entropy pool which then is inserted into the temporary seed buffer. This guarantees that "unused entropy" is appropriately protected by the per-CPU scheduler entropy pool. The data processing concepts between the scheduler ES and the interrupt ES are identical when comparing the figures in this section to section 2.5.

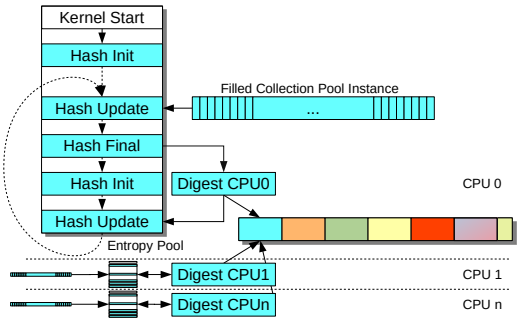

Figure 2.7: Scheduler per-CPU Entropy Pool Management

This entire message digest calculation is performed in the process context. Therefore, the hashing operation is not performed as part of the scheduling operation. The message digest is truncated to the entropy available in all scheduler per-CPU collection pools or the requested amount of entropy, whatever is smaller. Note, the requested amount of entropy is always smaller or equal to the size of the message digest of the used hashing algorithm.

As a side note, the scheduler entropy source potentially has some relationship with the IRQ entropy source because an IRQ ma trigger a scheduler event (e.g.

with the flag TIF_NEED_RESCHED). Therefore, both entropy sources cannot be used at the same time and credit entropy to both. It is permissible to use both, but only one is credited with entropy.

2.7.1 Entropy Amount of Scheduling Events

The discussion of the entropy content of interrupt events in section 2.5.1 applies to the scheduler-based entropy. The only difference is the use of a different compile-time entropy value of CONFIG_LRNG_SCHED_ENTROPY_RATE.

2.7.2 Health Tests

The health tests documented in section 2.5.2 are applied to the scheduler entropy source as well. The LRNG ensures that the health tests for the interrupt and the scheduler entropy sources are strictly separated. This separation applies to the SP800-90B as well.

2.8 Auxiliary Entropy Pool - LRNG-external Entropy Sources

The LRNG also supports obtaining entropy from the following data sources and entropy sources that are external to the LRNG. The data is injected into the auxiliary pool.

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 entropy 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 as a separate entropy source that eventually is concatenated with all other entropy sources in compliance with SP800-90C.

The auxiliary pool is processed with the available hash as follows:

1. Data is inserted the same way as data is added into the per-CPU entropy pools. The auxiliary pool technically is the message digest state where new data is inserted into the pool by performing a hash update operation.

2. When entropy is to be extracted from the auxiliary pool, a hash final operation is performed which is immediately followed by a hash init operation to initialize the hash context for new data.

3. The generated message digest is truncated to the amount of data requested by the DRNG (e.g. either 256 or 384 bits) and returned to the caller. Note, the auxiliary pool output is not truncated to the amount of entropy the data contains because the entropy provider may add data to the auxiliary pool without entropy, e.g. by simply writing to /dev/random. Thus, it may be possible that the auxiliary pool contains zero bits of entropy but yet contains data that should be used to "stir" the DRNG state.

Figure 2.8 illustrates the auxiliary pool operation for the case when user space inserts two separate buffers and a kernel driver uses the add_hwgenerator_randomness function.

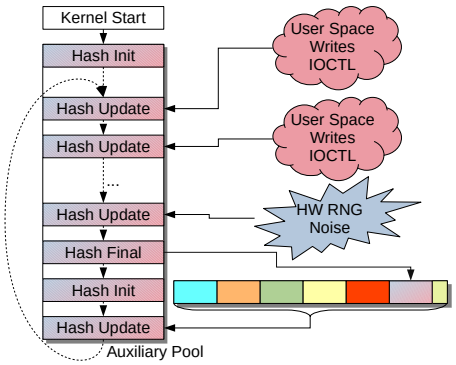

Figure 2.8: Auxiliary Pool Processing

In addition, the LRNG maintains an entropy estimator for the auxiliary pool 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 message digest is copied into the seed buffer when generating random numbers to seed the DRNG. The entire seed buffer is mixed back into the auxiliary pool for backward secrecy as shown in figure 2.8. The copy operation as well as the backtracking operation is atomic with respect to the auxiliary pool. This implies that both operations will always be fully completed before the next operation can commence. This ensures that the same auxiliary pool state can only be used once for a given seeding operation. 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 message digest.

2.9 Jitter RNG - LRNG-external Entropy Source

The Jitter RNG is treated as an external entropy source which is requested for random bits at the time the DRNG shall be seeded.

To comply with [5] section 4.3, the LRNG treats a health test error of the Jitter RNG as an intermittent error. The permanent error is handled the same way as outlined in section 3.3.37. Thus the rationale of section 3.3.37 applies to the Jitter RNG entropy source as well.

The Jitter RNG is by default used in an asynchronous operation where a separate kernel thread fills a buffer with entropy data. When the LRNG requires entropy it obtains the data from the buffer instead of waiting for the Jitter RNG to complete its operation. If the buffer is about to run out of data, the mentioned thread generates new data.

2.9.1 Entropy of CPU Jitter RNG Entropy Source

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

The entropy value can be altered by changing the kernel configuration option of `CONFIG_LRNG_JENT_ENTROPY_RATE`.

In addition, the value can be changed by writing an integer into /sys/module/LRNG_es_jent/parameters/jent_entropy or by setting the kernel command line option of `LRNG_es_jent.jent_entropy` if the kernel is compiled with `CONFIG_LRNG_RUNTIME_ES_CONFIG`.

2.10 CPU-base Entropy Source - LRNG-external Entropy Source

The CPU-based entropy source is treated as an external entropy source which is requested for random bits at the time the DRNG shall be seeded. Depending on the underlying CPU, only one such source is available like RDSEED on Intel x86 (or RDRAND if RDSEED is not available), the POWER CPU DARN instruction, etc.

Depending whether the the CPU entropy source is documented to full entropy, the following data collection methods are applied. This approach is orthogonal to the amount of entropy the LRNG awards to the CPU entropy source.

- CPU entropy source provides full entropy: The CPU entropy source is queried for the amount of data which is stored in the temporary seed buffer.

- CPU entropy source provides less than full entropy: For this entropy source, the LRNG contains the information about how much data must be fetched from the CPU entropy source to get full entropy. The required amount of data is pulled from the CPU entropy source and conditioned

with the hash currently in use by the LRNG. The calculated message digest is truncated to the requested amount of data which is stored in the temporary seed buffer.

2.10.1 Entropy of CPU Entropy Source

The entropy 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 entropy sources is not commonly known and reviewable. The entropy value can be altered by changing the kernel configuration option of `CONFIG_LRNG_CPU_ENTROPY_RATE`.

In addition, the value can be changed by writing an integer into /sys/-module/LRNG_es_cpu/parameters/cpu_entropy or by setting the kernel command line option of `LRNG_es_cpu.cpu_entropy` if the kernel is compiled with `CONFIG_LRNG_RUNTIME_ES_CONFIG`.

2.11 Legacy RNG Entropy Source - LRNG-external Entropy Source

The LRNG offers the use of the legacy RNG as an entropy source. This requires the compilation of the random.c along with the LRNG. The legacy RNG derives its entropy from sampling of interrupts. Therefore, the LRNG interrupt entropy source is not enabled if the legacy RNG entropy source is enabled, and vice versa.

The legacy RNG is queried for random numbers using the `get_random_bytes` function implemented in random.c.

2.11.1 Entropy of Legacy RNG Entropy Source

When the legacy RNG is enabled, the LRNG applies the entropy rate defined at compile time with `CONFIG_LRNG_LEGACY_RNG_ENTROPY_RATE` which is a value between 0 and 256 bits of entropy when 256 data bits are pulled from the legacy RNG.

If the kernel is booted with FIPS mode enabled, i.e. the kernel command line contains "fips=1", the legacy RNG entropy source's entropy rate is set to zero. The reason is that the legacy RNG is known to not comply with FIPS 140 rules like SP800-90B and thus must be assumed to provide no entropy.

2.12 DRNG Seeding Operation

The seeding operation obtains random data from all available entropy sources.

The (re)seeding logic tries to obtain 256 bits of entropy from the entropy 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 entropy sources can be obtained.

For efficiency reasons, the seeding operation uses a seed buffer depicted in figure 2.1 that is the following set of blocks of 256 bits each. If SP800-90C compliance is enabled, the initial seeding of the DRNG is seeded with a seed buffer that pulls 384 bits in the following blocks. Each block is dedicated to an entropy source. As each entropy source can be disabled at compile time, the

different bullets in the following list only applies if the corresponding entropy source is enabled.

1. One block is filled with the message digest from the auxiliary pool.

2. One block contains the message digest calculated from all per-CPU interupt entropy pools as depicted in figure 2.4 That buffer receives as much data from the hash operation as entropy can be pulled from the entropy pools. In the worst case when no new interrupts are received a zero buffer will be injected into the DRNG. This is performed by iterating over all per-CPU entropy pools and:

   (a) Perform a hash update operation to inject the current content of the per-CPU collection pool into the per-CPU entropy pool.

   (b) Perform a hash final operation on the per-CPU entropy pool to obtain the message digest.

   (c) That message digest is used to re-initialize the per-CPU entropy pool with a hash init and hash update operation to ensure backward secrecy.

   (d) Also, the message digest is fed into the hash operation to collect the output from all entropy pools.

   Once the message digest from all is obtained, it is truncated to the amount of entropy present in all entropy pools.

3. Another block is filled with the message digest of all scheduler per-CPU entropy pools.

4. One block is filled with the data from the legacy RNG entropy source.

5. The next block is filled by the Jitter RNG entropy source.

6. Finally, a block is filled by the fast entropy source of the CPU entropy source.

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.

The filled seed buffer is handed to the DRNG as a seed string. In addition, the seed buffer is inserted back into the auxiliary pool for backward secrecy. The seed buffer will not alter the entropy estimation of the auxiliary pool.

2.12.1 DRNG May Become Not Fully Seeded

The LRNG maintains a counter for each DRNG instance how many generate operation are performed without performing a reseed that has full entropy. If this counter exceeds the threshold of 2^{30} generate operations, i.e. the DRNG did not receive a seed with full entropy for that many generate operations, the DRNG is set to not fully seeded. This setting implies that the DRNG instance will not be used any more for generating random numbers until the LRNG received sufficient entropy to reseed the DRNG with full entropy.

If the DRNG that becomes not fully seeded is the initial DRNG instance that was seeded during boot time as outlined in section 2.3.1, the entire LRNG

is marked as not operational. This setting blocks all blocking interfaces just like during boot time when the LRNG is not yet fully seeded.

The LRNG automatically tries to recover from it when it received sufficient entropy.

2.13 Cryptographic Primitives Used By LRNG

The following subsections explain the cryptographic primitives that may be used by the LRNG.

2.13.1 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 entropy 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.

/dev/urandom and `get_random_bytes` Users that want to obtain data via the /dev/urandom user space interface or the `get_random_bytes` in-kernel API are delivered data that is obtained from the 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 2.12 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.

/dev/random and `lrng_get_random_bytes_full` The random numbers to be generated for /dev/random as well as `LRNG_get_random_bytes_full` are defined to have a special property: it only provides data once at least 256 bits of entropy have been collected by the LRNG. In addition, the LRNG must be fully initialized before random numbers are generated, including the completion of the SP800-90B heath test if entropy from internal entropy sources is gathered.

2.13.2 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 [3], section 2.3.

The operation of the ChaCha20 DRNG can be characterized with figure 2.9. This figure outlines the initialization of the DRNG, its seeding using the state update operation and the invocation of one generate operation that is requested to obtain more than 512 bits of data.

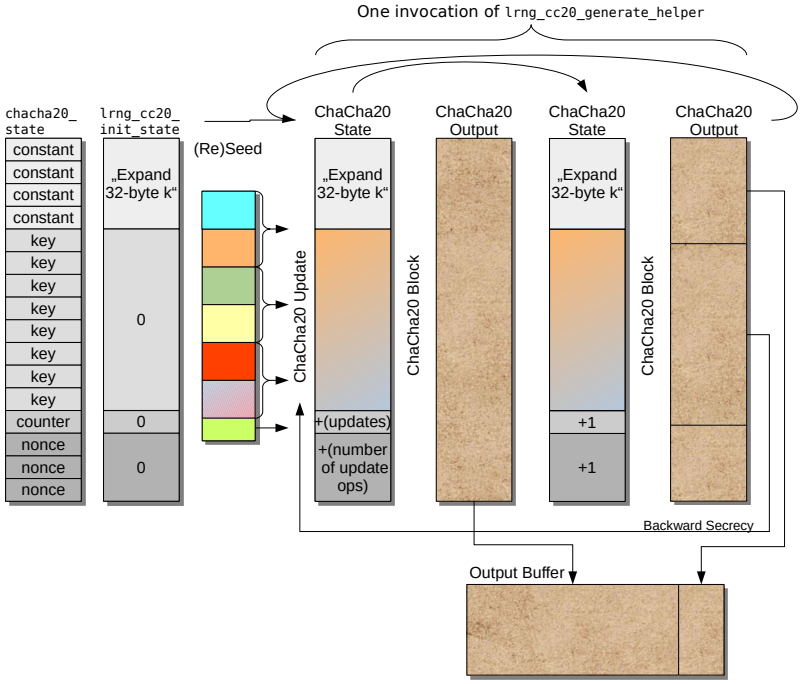

Figure 2.9: ChaCha20 DRNG Operation

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 [3] chapter 4.

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.

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 backward secrecy of the ChaCha20 state that was used to generate the random numbers.

2.13.3 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.

2.14 `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 `lrng_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 `lrng_get_random_bytes_full` accesses the DRNG instances that are allocated by the switchable DRNG support. This implies that `lrng_get_random_bytes_full` must be expected to sleep.

2.15 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 entropy 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.

`lrng_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. In addition, it guarantees that the LRNG is fully seeded and fully initialized before random numbers are generated. Yet, this function may sleep and thus is inappropriate for atomic contexts.

`lrng_get_random_bytes_min` 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. In addition, it guarantees that the LRNG is minimally seeded. Though, it is not guaranteed that the LRNG is fully initialized yet, specifically it is not guaranteed that the SP800-90C oversampling is applied. Yet, this function may sleep and thus is inappropriate for atomic contexts.

`lrng_get_random_bytes_pr` 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. In addition, it guarantees that the LRNG is fully initialized before random numbers are generated. Also, immediately before the generation of random bits, the DRNG is reseeded from the entropy sources and will only return as many random bits as entropy was received during the reseed. Yet, this function may sleep and thus is inappropriate for atomic contexts.

`lrng_get_seed` This function accesses the entropy sources to return a copy of the temporary seed buffer to the caller. The difference between `lrng_get_seed` and `lrng_get_random_*` APIs can be compared to the difference between RDSEED and RDRAND instructions from the Intel CPU.

`get_random_bytes_arch` In-kernel service function to safely call CPU entropy sources directly and ensure that the LRNG is used as a fallback if the CPU entropy 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[8].

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

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

For network ports/cookies, stack canaries, PRNG seeds, address space layout randomization, session *authentication* keys, or other applications

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

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. When opening the device with `O_SYNC`, the behavior discussed for `lrng_get_random_bytes_pr` is applied.

/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 3 for an indication of how important that factor is.

/proc/sys/kernel/random/poolsize Size of the entropy pool in bits. The value shows the size of the auxiliary entropy pool, i.e. the entropy pool that can be managed from user space. Thus, the available pool size is calculated by (*hash digestsize*).

/proc/sys/kernel/random/entropy_avail Amount of entropy present into the into the auxiliary pool.

/proc/sys/kernel/random/write_wakeup_threshold When entropy_avail falls below that threshold, suppliers of entropy are woken up. This value is given in bits.

/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_drng/parameters/max_wo_reseed Interface to read the maximum number of DRNG generate operations without receiving reseed that contains full entropy. This value can be set with `lrng_drng.max_wo_reseed` kernel command line option. This interface is only available when the kernel configuration option of `CONFIG_LRNG_RUNTIME_MAX_WO_RESEED` is set.

/sys/module/lrng_selftest/parameters/selftest_status Querying the status and restarting the LRNG self tests - see section 2.16 for details.

/sys/kernel/debug/lrng_testing/* Virtual files providing test interfaces as documented in section 2.17.

/sys/module/lrng_testing/parameters/* Virtual files providing test interfaces as documented in section 2.17.

/sys/module/lrng_es_irq/parameters/irq_entropy Interface to read the number of interrupts required to obtain 256 bits of entropy. See section 2.5.1 for details how to adjust it.

/sys/module/lrng_es_cpu/parameters/cpu_entropy Interface to adjust entropy estimation from CPU entropy source. See section 2.10.1 for details.

/sys/module/lrng_es_jent/parameters/jent_entropy Interface to adjust entropy estimation from Jitter RNG entropy source. See section 2.9.1 for details.

/sys/module/lrng_es_irq/parameters/lrng_pcpu_continuous_compression Interface is only present if the LRNG is compiled with `CONFIG_LRNG_SWITCHABLE_CONTINUOUS_COMPRESSION`. This interface allows reading the state whether the continuous compression operation is enabled. During boot time, this option may be used to either enable or disable the continuous compression operation. For details about the continuous compression operation, see section 2.2.

/sys/module/lrng_es_mgr/parameters/ntg.1 Interface is only present of the LRNG is compiled with `CONFIG_LRNG_AIS2031_NTG1_SEEDING_STRATEGY`. This interface is read-only and shows whether the German AIS20/31 NTG.1 (2022) compliance is enabled. See section 5.3.1 for details on such configuration.

IOCTLs are implemented as documented in random(4).

2.15.1 LRNG Interface to Linux Kernel Crypto API

The LRNG offers a kernel module to register with the kernel crypto API random number generator framework. This allows the LRNG to be used via this framework. It registers the name "stdrng" with the highest priority which implies that the LRNG would be used as default by this framework.

This registration also allows the LRNG to be accessed by user space via the AF_ALG interface wrapped by the libkcapi user space library.

The LRNG registers the interface using the function `lrng_get_random_bytes_full` guaranteeing that only random numbers are generated once the LRNG is fully seeded and fully initialized.

2.15.2 LRNG Interface to Hardware Random Number Generator Framework

In addition, the LRNG offers a kernel module to register with the kernel's hardware random number generator framework. Using this framework, the LRNG can be accessed by user space via /dev/hwrng.

The LRNG registers the interface using the function `lrng_get_random_bytes_full` guaranteeing that only random numbers are generated once the LRNG is fully seeded and fully initialized.

2.15.3 LRNG Device File

The device file of /dev/lrng behaves identically to the /dev/random device file including read/write operations, sleeping on it with the `select(2)` system call or using all IOCTLs defined for /dev/random. This file is intended for the case where the legacy RNG is still compiled but user space shall be capable of accessing the LRNG with all its properties.

The output of the file uses the function `lrng_get_random_bytes_full` guaranteeing that only random numbers are generated once the LRNG is fully seeded and fully initialized. When opening the device file with `O_SYNC`, it behaves like `lrng_get_random_bytes_pr` discussed before.

2.16 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 collection pool 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_drng_kcapi.ko`, its self-test is driven by the Linux kernel crypto API test manager.

- The raw noise source of the interrupt timing is tested at runtime with at least the stuck test. In addition the SP800-90B start-up and runtime tests discussed in section 2.5.2 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.

2.17 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.

- Interrupt Entropy Source:

  - 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 col-

lected 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 collection pool 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.

- Scheduler Entropy Source:
  - The interface `/sys/kernel/debug/lrng_testing/lrng_raw_sched_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_sched_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_sched_hires` file after boot provides this data in this case.
  - The interface `/sys/kernel/debug/lrng_testing/lrng_raw_sched_pid` allows reading of the PID value of the task that are about to be scheduled to collected while the read operation is in progress. The PID values are extracted which are 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_sched_pid_test=1`, the time stamps of the first 1,024 events recorded by the LRNG are stored. The first read of the `lrng_raw_sched_pid` file after boot provides this data in this case.
  - The interface `/sys/kernel/debug/lrng_testing/lrng_raw_starttime_pid` allows reading of the start time value of the task that are about to be scheduled to collected while the read operation is in progress. The start time values are extracted which are 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_sched_starttime_test=1`, the time stamps of the first 1,024 events recorded by the LRNG are stored. The first read of the `lrng_raw_starttime_pid` file after boot provides this data in this case.

- The interface `/sys/kernel/debug/lrng_testing/lrng_raw_nvcsw_pid` allows reading of the numbers of context switches of the task that are about to be scheduled to collected while the read operation is in progress. The context switch values are extracted which are 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_sched_nvcsw_test=1`, the time stamps of the first 1,024 events recorded by the LRNG are stored. The first read of the `lrng_raw_nvcsw_pid` file after boot provides this data in this case.
- The interface `/sys/kernel/debug/lrng_testing/lrng_sched_perf` allows reading of the number of cycles used to process one scheduler event. This allows measuring the performance impact of the LRNG on the scheulder. When booting the kernel with the kernel command line option `lrng_testing.boot_sched_perf=1`, the performance data of the first 1,024 events recorded by the LRNG are stored. The first read of the `lrng_sched_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 Interrupt Entropy Source Assessment

This section documents the entropy assessment and the compliance with various standards of the interrupt entropy source. All other entropy sources are treated as black boxes by the LRNG and must therefore deliver their own entropy assessment.

The entropy sources of the auxiliary pool, the Jitter RNG and the CPU-based entropy source are all not related to the interrupt entropy source. Thus, the combination of all those entropy sources is subject to the SP800-90C assessment provided in section 5.1

3.1 Noise Source Behavior

The noise source component of an entropy source contains the non-deterministic, entropy-producing activity. For the IRQ entropy source this is the timing of the

arrival of interrupts. The general behavior of the noise source can be characterized by analyzing the time stamps of the arrival time. Before performing this analysis, a recap of the noise source is important:

- When an interrupt occurs, the LRNG obtains a high-resolution time stamp. Technically it may be a cycle counter offered by the CPU that produces the time stamp which is unrelated to time, but provides a fast-moving counter.
- The time stamp is divided by the greatest common divisor (GCD) that is calculated during boot time by using the first 100 time stamps.
- After the division with the GCD, the 8 least-significant bits of the time stamp are used. The higher-order bits are discarded. These 8 bits are now subject to post-processing via the collection pool, the entropy pool and the DRNG.

Considering these static and never-changing steps, the noise source rests on the gathering of the 8 bits from the last step. An analysis of the noise source therefore focuses on these 8 bits. These 8 bits are subsequently referenced as "raw noise source data".

3.1.1 Distribution of Raw Data

If the noise source would operate perfectly, an equi-distribution of the raw noise source data is to be expected. Various tests have been conducted on most major CPUs as outlined in appendix C. Using the runtime data, a common pattern of the raw noise source data emerges that can be seen with figure 3.1. This figure uses the 1,000,000 traces of the raw noise source data from the IRQ entropy source on the RISC-V, i.e. a system with less-then-average entropy rate.

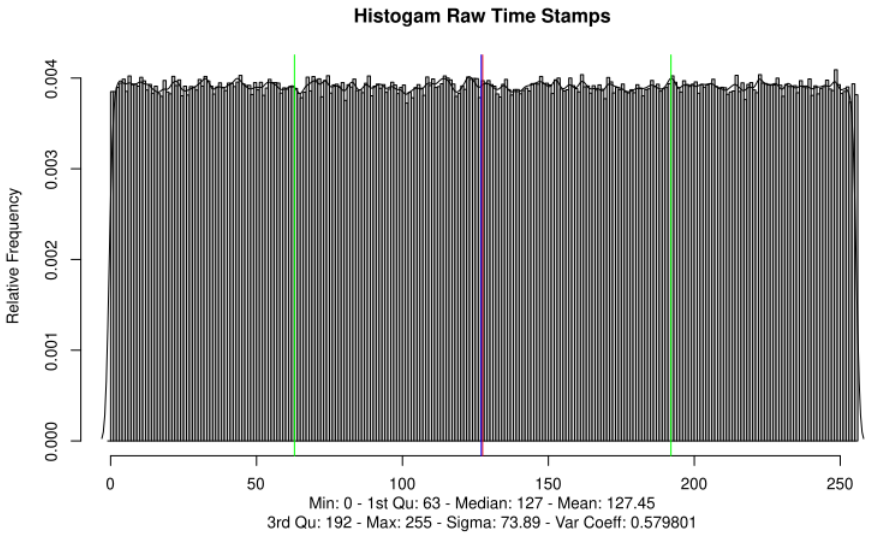

Figure 3.1: RISC-V Raw Noise Source Data Distribution

Figure 3.1 shows the following characteristics:

- A histogram of the number of time stamps received for each of the 256 possible time stamp values is depicted.
- The 25% and 75% quartiles of the distribution are marked with the two green lines. They are at the time stamp value of 63 and 192, respectively
- The mean of the distribution is marked with the red line which is 127.45.
- The median of the distribution is marked with the blue line at 128.
- The black line marks the distribution of the time stamps.
- The standard derivation is not depicted, but it is at 73.89.
- The variation coefficient is not shown in the figure, but it is 0.579801.

The figure together with the mentioned statistical values clearly shows that it is close to an equi-distribution. This means that the raw noise source data for an entropy source instance on a less-than-ideal hardware still exhibit a distribution that is close to the expected entropy source for a perfect operation. The distributions of most tested systems are always close to an equi-distribution as shown with the table below.

To verify that the distribution is an equi-distribution, a Chi-Squared Goodness-of-Fit test is applied. For the mentioned RISC-V system, the following values are observed:

- Asymptotic significance P: 0.2343
- Degrees of freedom: 255

The degrees of freedom shows that indeed all 256 possible timing values are covered. Applying an alpha of 5%, the Chi-Squared test indicates that the observed data set is an equi-distribution.

With the table below, the statistical properties for the different tested systems are listed.

Test System	25% Quartile	Median	Mean	75% Quartile	Std. Deriv.	Var. Coeff.	χ^2 P	χ^2 DF
AMD Ryzen 5950X - 64-bit KVM environment	64	128	127.6	192	73.88	0.58	0.0523	255
AMD EPYC Milan 7713 2 sockets 128 cores 8-way NUMA	63	127	127.48	192	73.91	0.58	0.4458	255
ARMv7 rev 5	68	128	121.05	191	72.95	0.60	0	255

Test System	25% Quartile	Median	Mean	75% Quartile	Std. Deriv.	Var. Coeff.	χ^2 P	χ^2 DF
ARMv7 rev 5 (Freescale i.MX53)[9]	63	128	127.52	192	74	0.58	0.4693	255
ARMv7 rev 5 (Freescale i.MX6 Ultralite)[10]	63	127	127.15	191	73.96	0.58	0	255
ARM 64 bit AppliedMicro X-Gene Mustang Board	63	128	127.54	192	73.94	0.58	0.7229	255
Intel Atom Z530 – using GUI	64	128	127.58	192	73.93	0.58	0.3181	255
Intel Sandy Bridge Clang Compile	62	125	125.86	187	73.24	0.58	0.9347	252
Intel i7 8565U Whiskey Lake – 32-bit KVM environment	64	128	127.53	192	73.91	0.58	0.2893	255
Intel i7 8565U Whiskey Lake	63	127	127.37	191	73.86	0.58	0	255
Intel Xeon E7 4870 8 sockets 160 CPUs 8-way NUMA	64	127	127.44	191	73.89	0.58	0	255
Intel Xeon Gold 6234	64	126	126.75	190	73.88	0.58	0	127[11]
IBM POWER 8 LE 8286-42A	63	127	127.66	191	73.86	0.58	0	255
IBM POWER 7 BE 8202-E4C	64	128	130.07	192	73.96	0.57	0	255
IBM System Z z13 (machine 2964)	56	120	118.27	184	73.58	0.62	0	235
IBM System Z z15 (machine 8561)	58	122	124.3	192	73.54	0.59	0	239
MIPS Atheros AR7241 rev 1[12]	64	128	127.55	191	73.88	0.58	0.5646	255

[9] USBArmory MK I
[10] USBArmory MK II
[11] Tested without GCD.
[12] Ubiquiti Nanostation M5 (xm)

Test System	25% Quar-tile	Median	Mean	75% Quar-tile	Std. Deriv.	Var. Coeff.	χ^2 P	χ^2 DF
MIPS Lantiq 34Kc V5.6[13]	64	127	127.49	191	73.85	0.58	0	255
Qualcomm IPQ4019 ARMv7[14]	63	127	127.4	191	73.9	0.58	0	255
SiFive HiFive Unmatched RISC-V U74	63	127	127.45	192	73.89	0.58	0.2343	255

The Chi-Squared asymptotic significance shows for some systems that an equi-distribution is not applicable, i.e. when the χ^2 P value is less than 0.05. Yet, when considering the other statistical values, the actual distribution is neither skewed nor otherwise loop-sided. When looking at the distribution, it becomes evident that some time stamps have a higher likelihood than others which hint to special properties of the system. This observation applies to all measurements which do not follow an equi-distribution based on the Chi-Squared Goodness-of-Fit test. This allows the conclusion that the min-entropy estimates given in appendix C can be considered to illustrate the real entropy rate.

For example, the distribution for the USB Armory Mark II system shown with figure 3.2 indicates that all time stamp values divisible by 8 are only chosen two-thirds as often as the rest.

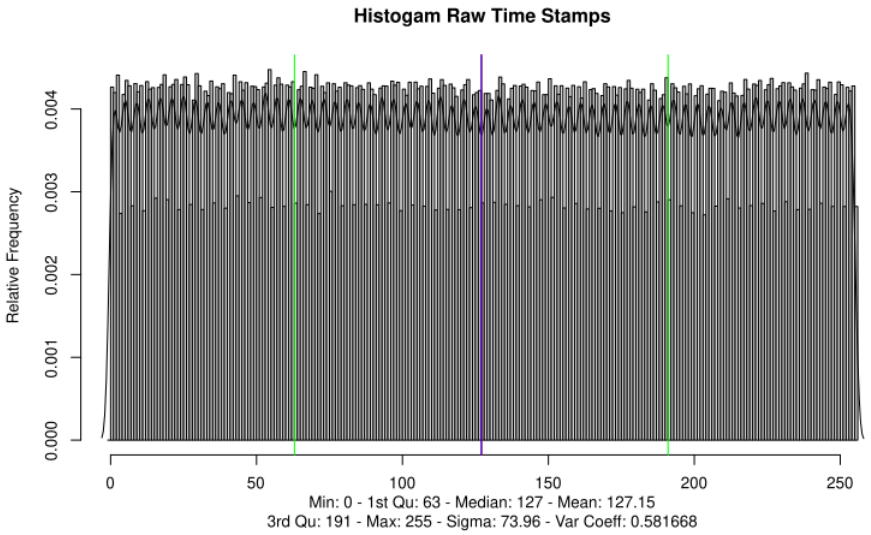

Figure 3.2: USB Armory Mark II: Raw Noise Source Data

[13] AVM Fritz Box 7490
[14] AVM Fritz Box 7520

Another example is a specific ARMv7 system which contains a very periodic timer interrupt. Figure 3.3 shows that the time stamp has a pattern but still exhibits a distribution that does not contradict the expectation to deliver entropy at the rate calculated in appendix C. In the worst case that the timer interrupt only causes the raw noise source data which would exhibit a clear pattern, the stuck health test would identify this pattern with the second discrete derivative and disregards time stamp for entropy collection.

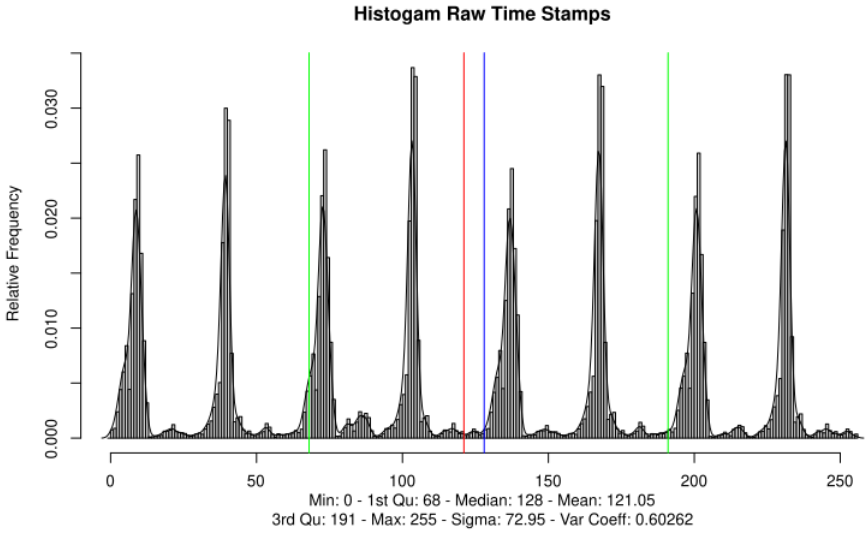

Figure 3.3: Periodic timer interrupt: Specific ARMv7 System Raw Noise Source Data

A similar effect is visible on an IBM System Z z13 system shown with figure 3.4. Again, the shown pattern does not contradict the entropy rate calculated for this system in appendix C.

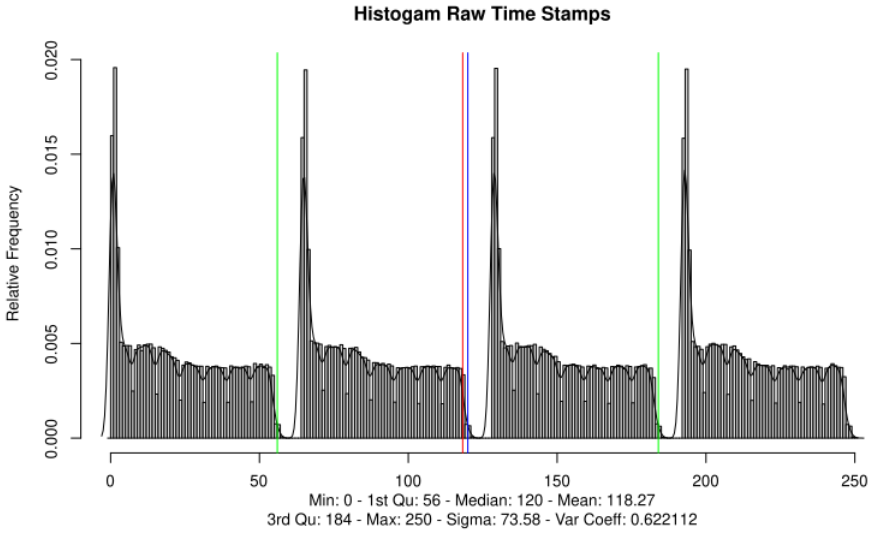

Figure 3.4: IBM System Z Raw Noise Source Data

3.1.2 Greatest Common Divisor Assessment

The behavior of the GCD application can be clearly seen with the following figures and numbers obtained for an Intel Atom Z530 system whose GCD is 4.

Without the application of the GCD, the distribution of the time stamp is given with figure 3.5

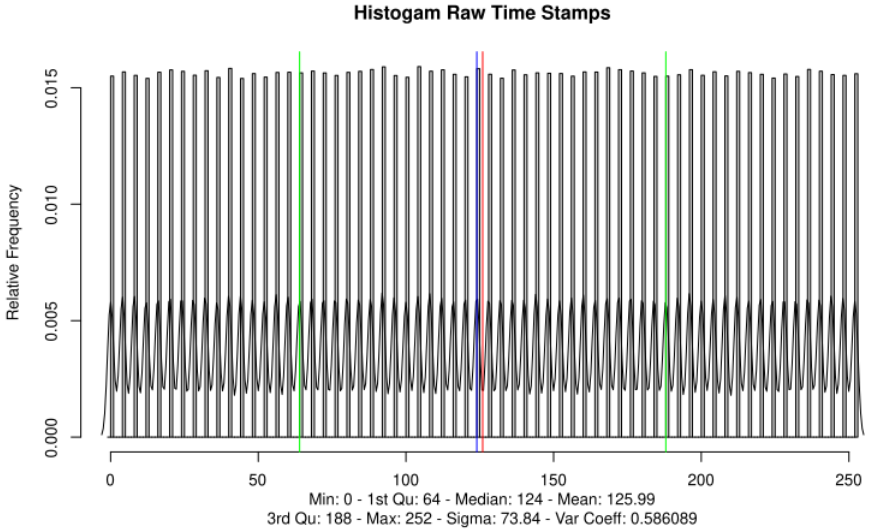

Figure 3.5: Without GCD - Raw Noise Source Data Distribution

Figure 3.5 shows the following characteristics:

- A histogram of the number of time stamps received for each of the 256 possible time stamp values is depicted.
- The 25% and 75% quartiles of the distribution are marked with the two green lines. They are at the time stamp value of 63 and 188, respectively
- The mean of the distribution is marked with the red line which is 125.99.
- The median of the distribution is marked with the blue line at 128.
- The standard derivation is not depicted, but it is at 73.84.
- The variation coefficient is not shown in the figure, but it is 0.586089.
- χ^2 P value for equi-distribution Goodness-of-Fit test: 0.4263
- χ^2 degrees of freedom for equi-distribution Goodness-of-Fit test: 63

When applying the GCD, and obtaining a new measurement, the distribution shown with figure 3.6 emerges:

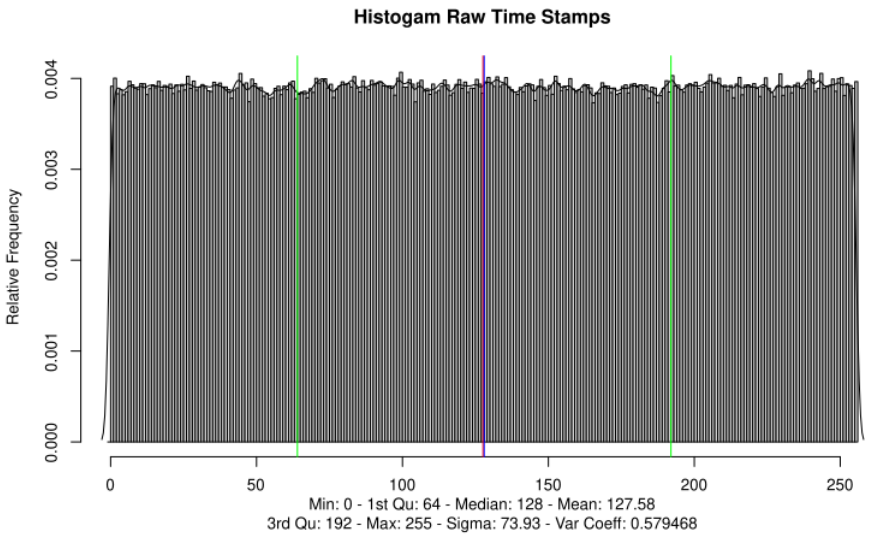

Figure 3.6: With GCD - Raw Noise Source Data Distribution

Figure 3.6 shows the following characteristics:

- A histogram of the number of time stamps received for each of the 256 possible time stamp values is depicted.
- The 25% and 75% quartiles of the distribution are marked with the two green lines. They are at the time stamp value of 64 and 192, respectively
- The mean of the distribution is marked with the red line which is 127.58.
- The median of the distribution is marked with the blue line at 124.

- The standard derivation is not depicted, but it is at 73.93.
- The variation coefficient is not shown in the figure, but it is 0.579468.
- χ^2 P value for equi-distribution Goodness-of-Fit test: 0.3181
- χ^2 degrees of freedom for equi-distribution Goodness-of-Fit test: 255

This shows that the application of the GCD removes the "unused" time stamp values without changing the overall distribution.

3.1.3 Worst and Regular Case Distribution

The measurements shown in appendix C commonly are obtained by applying a worst-case which triggers as much interrupts as possible in the shortest amount of time.

To compare the distributions of the time stamp between the worst case and a regular case of normal system use, the USB Armory mark I system was tested twice with the following result values:

- Worst case
  - 25% quartile: 64
  - Median: 128
  - Mean: 127.6
  - 75% quartile: 192
  - Standard derivation: 73.91
  - Variation coefficient: 0.579221
  - χ^2 P value for equi-distribution Goodness-of-Fit test: 0.9445
  - χ^2 degrees of freedom for equi-distribution Goodness-of-Fit test: 255
- Regular case
  - 25% quartile: 63
  - Median: 128
  - Mean: 127.52
  - 75% quartile: 192
  - Standard derivation: 74
  - Variation coefficient: 0.580276
  - χ^2 P value for equi-distribution Goodness-of-Fit test: 0.4693
  - χ^2 degrees of freedom for equi-distribution Goodness-of-Fit test: 255

The values indicate no statistical significant difference allowing the conclusion that both distributions are very similar.

3.2 FIPS 140-3 Compliance

FIPS 140-3 specifies entropy source compliance in FIPS 140-3 IG D.K. This section analyzes each requirement for compliance. The general requirement to comply with SP800-90B [5] is analyzed in section 3.3.

3.2.1 FIPS 140-3 IG D.K 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.

3.2.2 FIPS 140-3 IG D.K 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 3.3.

3.2.3 FIPS 140-3 IG D.K Additional Comment 1

The first test referenced in section 3.2.1 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.

3.2.4 FIPS 140-3 IG D.K 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 2.5.2 explains the applied actions.

3.2.5 FIPS 140-3 IG D.K Additional Comment 3

The LRNG uses the following conditioning components:

- For collecting of entropy data from the entropy source, 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 [5] 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.

3.2.6 FIPS 140-3 IG D.K Additional Comment 4

The restart test is covered by the first test documented in section 3.2.1.

3.2.7 FIPS 140-3 IG D.K 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 3.2.1 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.

3.2.8 FIPS 140-3 IG D.K 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 3.2.1.

3.3 SP800-90B Compliance

This chapter analyzes the compliance of the LRNG to the SP800-90B [5] standard considering the FIPS 140-2 implementation guidance 7.18 which alters some of the requirements mandated by SP800-90B.

3.3.1 SP800-90B Section 3.1.1

The collection of raw data for the SP800-90B entropy testing documented in section 3.2.1 uses 1,000,000 consecutive time stamps obtained in one execution round.

The restart tests documented in section 3.2.1 perform 1,000 restarts collecting 1,000 consecutive time stamps.

3.3.2 SP800-90B Section 3.1.2

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

3.3.3 SP800-90B Section 3.1.3

Please see section 3.2.7: The entropy of the raw noise source data 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. Yet, this rate can be configured at compile time to be lower than one bit of entropy per interrupt event.

The first test referenced in section 3.2.1 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.

3.3.4 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.

3.3.5 SP800-90B Section 3.1.5

The conditioning component applied to the interrupt entropy 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 per-CPU entropy pools - when new random data is generated involving the entropy pools, the current entropy pool states are 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 [5] 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[15], 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 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. Furthermore, applying the Output_Entropy formula for a vetted conditioning component of a truncated hash, the following calculation applies. This calculation shows the entropy rate of a SHA-512 hash processing a buffer with 1024 bits that contains, say, 384 bits of entropy and truncating it to 256 bits. This means, the formula for $h_{out_{SHA-512\ trunc}} = Output\_Entropy_{SHA-512\ trunc}(1024, 256, 512, 384)$ following [5] section 3.1.5.1.2 is calculated:

$$P_{high} = 2^{-384}$$

$$P_{low} = \frac{(1 - 2^{-384})}{2^{1024} - 1} \approx 2^{-1024}$$

$$n = min(256, 512) = 256$$

$$\psi = 2^{1024-256} \cdot 2^{-1024} + 2^{-384} = 2^{-256} + 2^{-384} \approx 2^{-256}$$

$$U = 2^{1024-256} + \sqrt{2 \cdot 256 \cdot (2^{1024-256}) \cdot ln(2)} = 2^{768} + \sqrt{2^{777} \cdot ln(2)} \approx 2^{768}$$

$$\omega = 2^{768} \times 2^{-1024} = 2^{-256}$$

$$\underline{Output\_Entropy_{SHA-512\ trunc}(1024, 256, 512, 384) = -log_2(max(2^{-256}, 2^{-256})) = 256}$$

Even after calculating other entropy rates using the same formula, the following conclusion for the truncation can be applied:

$$\underline{Output\_Entropy_{trunc}(n_{in}, n_{out}, nw, h_{in}) = min(n_{in}, n_{out}, nw, h_{in})}$$

This function is applied by the LRNG for hash truncation.

[15] 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.

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 pool and the per-CPU collection pools.

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 different hash operations specified in section 2.2 are applied as follows:

- Compression of entropy delivered from the interrupt entropy source when adding the entropy into the per-CPU entropy pools.
- Compression of entropy delivered from one or more LRNG-external entropy sources when adding the entropy into the auxiliary pool.
- Compression of entropy delivered by the different per-CPU entropy pools when "reading" the entropy pools.

The requirement of [5] section 3.1.6 states that when combining two or more noise sources using a vetted conditioning component, only one noise source is to be credited with entropy. This requirement is met as follows: according to [5] section 2.2 a noise source is the phenomenon delivering entropy. The noise source data is post-processed with conditioning components and health tests to form an entropy source. Based on this statement, the collection of the per-CPU entropy pools together form one entropy source that is compliant to SP800-90B.

Note, the reason for hashing the per-CPU entropy pools together with the auxiliary pool is to ensure backward secrecy when calculating the next round of random numbers used to fill the seed buffer used to seed the DRBG from the entropy sources 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 collection pools 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 8,192 bits as the per-CPU entropy pool obtains its input data from the collection pool that has a size of 8,192 bits (1,024 * 8 bits)[16] .
  - $n_{out_{per-CPU\ pool}}$ is the message digest size in bits.
  - $nw_{per-CPU\ pool}$ is the message digest size in bits.
- Function 2.8:
  - $n_{in_{aux\ pool}}$ 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 2.3:
  - $n_{in_{hash\ pools}}$ 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.

3.3.6 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 collection pool 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 [5] section 3.1.5.1.1 table 1.

[16]Section 6.2 outlines that the LRNG collection size can be modified at compile time where the default is 1,024. 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.

- 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 [5] 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 [5] section 3.1.5.1.1 table 1.

Function 2.2 Output_Entropy To perform a calculation of the Output_Entropy of a conditioning component, the input entropy must be considered. The heuristic input entropy awarded for one time stamp processed by the LRNG is given in equation 3.2. Due to the concatenation operation of time stamps, the entropy of multiple time stamps can be added.

For the function 2.2, the entropy when 1,024 time stamps are received is $h_{out_{SHA-512}} = Output\_Entropy_{SHA-512}(8192, 512, 512, 1024)$ following [5] section 3.1.5.1.2. Therefore, the following calculation is applicable:

$$P_{high} = 2^{-1024}$$

$$P_{low} = \frac{(1 - 2^{-1024})}{2^{8192} - 1} \approx 2^{-8192}$$

$$n = min(512, 512) = 512$$

$$\psi = 2^{8192-512} \cdot 2^{-8192} + 2^{-1024} = 2^{-512} + 2^{-1024} \approx 2^{-512}$$

$$U = 2^{8192-512} + \sqrt{2 \cdot 512 \cdot (2^{8192-512}) \cdot ln(2)} = 2^{7680} + \sqrt{2^{7690} \cdot ln(2)} \approx 2^{7680}$$

$$\omega = 2^{7680} \times 2^{-8192} = 2^{-512}$$

$$\underline{Output\_Entropy_{SHA-512}(8192, 512, 512, 1024) = -log_2(max(2^{-512}, 2^{-512})) = 512}$$

As a complement, the same calculation is provided when only one time stamp is received for the formula $h_{out_{SHA-512}} = Output\_Entropy_{SHA-512}(8, 512, 512, 1)$

$$P_{high} = 2^{-1}$$

$$P_{low} = \frac{(1 - 2^{-1})}{2^{8} - 1} \approx 2^{-9}$$

$$n = min(512, 512) = 512$$

$$\psi = 2^{8-512} \cdot 2^{-9} + 2^{-1} = 2^{-513} + 2^{-1} \approx 2^{-1}$$

$$U = 2^{8-512} + \sqrt{2 \cdot 512 \cdot (2^{8-512}) \cdot ln(2)} = 2^{-504} + \sqrt{2^{-494} \cdot ln(2)} \approx 2^{-247} \cdot \sqrt{ln(2)} \approx 2^{-248}$$

$$\omega = 2^{-248} \times 2^{-9} = 2^{-257}$$

$$\underline{Output\_Entropy_{SHA-512}(8, 512, 512, 1) = -log_2(max(2^{-257}, 2^{-1})) = 1}$$

The calculation can be generalized with the following formula:

$$h_{out_{SHA-512}} = Output\_Entropy_{SHA-512} = min(h_{in}, n_{out_{SHA-512}})$$

When using SHA-256, the same type of calculation can be provided. The first set of formulas show the case when 1,024 time stamps are received and thus for $h_{out_{SHA-256}} = Output\_Entropy_{SHA-256}(8192, 256, 256, 1024)$:

$$P_{high} = 2^{-1024}$$

$$P_{low} = \frac{(1 - 2^{-1024})}{2^{8192} - 1} \approx 2^{-8192}$$

$$n = min(256, 256) = 256$$

$$\psi = 2^{8192-256} \cdot 2^{-8192} + 2^{-1024} = 2^{-256} + 2^{-1024} \approx 2^{-256}$$

$$U = 2^{8192-256} + \sqrt{2 \cdot 256 \cdot (2^{8192-256}) \cdot ln(2)} = 2^{7936} + \sqrt{2^{7945} \cdot ln(2)} \approx 2^{7936}$$

$$\omega = 2^{7936} \times 2^{-8192} = 2^{-256}$$

$$\underline{Output\_Entropy_{SHA-256}(8192, 256, 256, 1024) = -log_2(max(2^{-256}, 2^{-256})) = 256}$$

As a complement, the same calculation is provided when only one time stamp is received for the formula $h_{out_{SHA-512}} = Output\_Entropy_{SHA-256}(8, 256, 256, 1)$

$$P_{high} = 2^{-1}$$

$$P_{low} = \frac{(1 - 2^{-1})}{2^{8} - 1} \approx 2^{-9}$$

$$n = min(256, 256) = 256$$

$$\psi = 2^{8-256} \cdot 2^{-9} + 2^{-1} = 2^{-257} + 2^{-1} \approx 2^{-1}$$

$$U = 2^{8-256} + \sqrt{2 \cdot 256 \cdot (2^{8-256}) \cdot ln(2)} = 2^{-248} + \sqrt{2^{-239} \cdot ln(2)} \approx 2^{-120} \cdot \sqrt{ln(2)} \approx 2^{-121}$$

$$\omega = 2^{-121} \times 2^{-9} = 2^{-130}$$

$$\underline{Output\_Entropy_{SHA-256}(8, 256, 256, 1) = -log_2(max(2^{-130}, 2^{-1})) = 1}$$

Again, the calculation can be generalized with the following formula:

$$h_{out_{SHA-256}} = Output\_Entropy_{SHA-256} = min(h_{in}, n_{out_{SHA-256}})$$

Comparing the conclusions for SHA-512 and SHA-256, both allow to draw the general conclusion that underlies the entire entropy assessment and therefore data entropy management applied by the LRNG for all vetted conditioning operations:

$$h_{out_{vetted}} = Output\_Entropy_{vetted} = min(h_{in}, n_{out_{vetted}}) \tag{3.1}$$

Function 2.8 The function 2.8 uses the same hash operation as the discussed in the preceding section. Thus, the conclusion drawn with equation 3.1 applies here as well.

Function 2.3 The function 2.3 uses the same hash operation as the discussed in the preceding section. Thus, the conclusion drawn with equation 3.1 applies here as well.

Conclusions for Output_Entropy As stated in [5] section 3.1.5.1.2, vetted conditioning components are allowed to claim full entropy. In case of full entropy, the following is applied which matches exactly analysis and conclusion of equation 3.1:

- $h_{out_{SHA-512}} = min(h_{in}, n_{out_{SHA-512}})$,
- $h_{out_{SHA-256}} = min(h_{in}, n_{out_{SHA-256}})$, or
- $h_{out_{SHA-1}} = min(h_{in}, n_{out_{SHA-160}})$.

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 for the chosen hash.

Heuristic Entropy Assessment The heuristic entropy value 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:

$$h_{t_8} = h_{t_{32}} = 1 \tag{3.2}$$

Note, in order to assess whether the LRNG heuristic entropy value is appropriate, it must be compared with the entropy analysis result received from practical measurements such as outlined in sections 3.3.3 and 3.3.4. This comparison must show that the heuristic entropy value is always lower and thus more conservative than what the measurements show on target devices.

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

$$h_{a_{32}} = 0 \tag{3.3}$$

The entropy in the concatenated time stamps found in the interrupt as well as scheduler per-CPU collection pool is calculated as the sum of all time stamps (truncated or not) present in the interrupt as well as scheduler per-CPU collection pool of 1,024 bytes per default – if a different collection pool size is used, the right-hand value of the following equation must be adjusted accordingly:

$$h_{per-CPU\ CP} = min(\sum_{n=0}^{number\ time\ stamps} (h_{t_{\{8,32\}}})_n, 1024) \tag{3.4}$$

For the maintenance of the interrupt per-CPU entropy pool as specified by equation 2.2, the following entropy rate applies when continuous compression support is enabled. This formula implies that each output of the interrupt per-CPU entropy pool holds the sum of the entropy of the received per-CPU collection pool since last generation of the per-CPU output data and the entropy remained in the per-CPU entropy pool capped by the message digest size. This operation implies that the used hash compresses of the entropy available in the different input data.

$$h_{per-CPU\ pool_n} = min(\sum_{m=0}^{m=n-1} h_{per-CPU\ CP_m} + h_{per-CPU\ pool_{n-1}}, n_{out}) \quad (3.5)$$

When continuous compression support is disabled as well as for the scheduler-based entropy source, the per-CPU entropy pool maintenance specified by equation 2.2 shows the following entropy rate. The formula implies that the maximum amount of entropy that can be held depends on the size of the collection pool depicted with equation 2.1 since additional entropy received by the collection pool overwrites old entropy data. The collection pool can hold the maximum amount of entropy event data as defined with its size. After converting the number of received entropy event data into an entropy statement using equation 3.2, the maximum amount of entropy held in the collection pool is available.

$$h_{per-CPU\ pool_n} = min(h_{per-CPU\ CP_n} + h_{per-CPU\ pool_{n-1}}, n_{out}) \quad (3.6)$$

Similarly, the following equation applies to the entropy of the auxiliary pool maintenance as specified by equation 2.8. Note, although this entropy source is not considered to be modeled in this chapter, the formula is still provided illustrating the use of a vetted conditioning component. 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.

$$h_{aux\ pool} = min(h_{in_{aux\ pool}} + h_{aux\ pool}, n_{out}) \quad (3.7)$$

The following equation applies when calculating the interrupt and scheduler-based entropy source output buffer before its truncation as specified by equation 2.3. The formula implies that the interrupt entropy 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.

$$h_{hash\ pools} = min(\sum_{c=0}^{max\ CPU} h_{per-CPU\ pool_c}, n_{out}) \quad (3.8)$$

The entropy present in the truncated interrupt entropy 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:

$$requested\ size_s = security\ strength = 256 \quad (3.9)$$

$$h_s = min(h_{hash\ pools}, requested\ size_s) \tag{3.10}$$

The entropy of the temporary seed buffer following equation 3.1 is simply an addition of the entropy values credited for each of the entropy source with h_A denominating the auxiliary entropy entropy rate, h_E references the IRQ ES entropy rate, , h_S references the scheduler ES entropy rate, , h_J covers the Jitter RNG ES entropy rate, h_C references the CPU ES entropy rate, and h_K references the kernel ES entropy rate::

$$h_T = h_A + h_E + h_S + h_J + h_C \tag{3.11}$$

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 interrupt entropy 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 per-CPU entropy pools of the assumed 2 CPUs contain $h_{per-CPU\ pool_{CPU0}} = 185$ and $h_{per-CPU\ pool_{CPU1}} = 123$. The debit operation performs:

1. $h_{per-CPU\ pool_{CPU0}} = 185 - 185 = 0$ leaving $h_{s_{not\ debited}} = 256 - 185 = 71$
2. $h_{per-CPU\ pool_{CPU1}} = 123 - 71 = 52$

3.3.7 SP800-90B Section 3.1.6

The LRNG uses the following noise sources for the interrupt entropy source:

- 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.
- Auxiliary data delivered as part of interrupts: HID event data. This data is concatenated with the interrupt time stamps into the collection pool. Yet it is always credited with zero bits of entropy.

All 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.

3.3.8 SP800-90B Section 3.2.1 Requirement 1

This entire document is intended to provide the required analysis.

3.3.9 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.

3.3.10 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.

3.3.11 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.

3.3.12 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 3.3.3. 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 collection pool 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.

3.3.13 SP800-90B Section 3.2.1 Requirement 6

Please see section 3.2.3 for details how and why the raw entropy extraction does not substantially alter the noise source behavior.

3.3.14 SP800-90B Section 3.2.1 Requirement 7

See section 3.3.4 for a description of the restart test.

3.3.15 SP800-90B Section 3.2.2 Requirement 1

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

3.3.16 SP800-90B Section 3.2.2 Requirement 2

The noise source is based on the receipt of interrupts. The receipt of interrupts follows the usage of the system. The more I/O is performed with the system, the more interrupts are received by the LRNG. The entropy rate only is a function of the received I/O events and the timer and does not depend on any other system property such as physical characteristics (e.g. temperature variations or voltage/current variations). This finding is consistent with the fact that the noise source is a pure software-based noise source which relies on the presence of a high-resolution timer. Note, the used timer is a cycle counter that increments with a given rate.

3.3.17 SP800-90B Section 3.2.2 Requirement 3

See sections 3.3.6 for a discussion of the entropy provided by the interrupt noise source.

A stochastic model is not provided.

3.3.18 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.

3.3.19 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.

3.3.20 SP800-90B Section 3.2.2 Requirement 6

The noise source generates the data via the hash generation function as outlined in section 3.3.5.

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 3.3.6.

3.3.21 SP800-90B Section 3.2.2 Requirement 7

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

Though, the LRNG employs complete self-contained other entropy sources which may be compliant to SP800-90B by itself. To seed the DRNG maintained by the LRNG, the output of all entropy sources are concatenated compliant to SP800-90C as outlined in section 5.1.

3.3.22 SP800-90B Section 3.2.3 Requirement 1

The conditioning component is the hash operation. See section 3.3.5 for a discussion of the input and output sizes.

3.3.23 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.

3.3.24 SP800-90B Section 3.2.3 Requirement 3

For the defined hashes, no key is required.

3.3.25 SP800-90B Section 3.2.3 Requirement 4

For the defined hashes, no key is required.

3.3.26 SP800-90B Section 3.2.3 Requirement 5

The conditioning component is the hash operation. See section 3.3.6 for a discussion of the narrowest internal width and the output block size.

3.3.27 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.

3.3.28 SP800-90B Section 3.2.4 Requirement 2

The operation of the test tools for gathering raw data are discussed in section 3.3.3. This explanation shows that the raw unconditioned data is obtained.

3.3.29 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.

3.3.30 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.

3.3.31 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.

3.3.32 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.

3.3.33 SP800-90B Section 3.2.4 Requirement 7

Please see section 3.2.3 for details how and why the raw entropy extraction does not substantially alter the noise source behavior.

3.3.34 SP800-90B Section 4.3 Requirement 1

The implemented health tests comply with SP800-90B sections 4.4 as described in section 3.3.44.

3.3.35 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.

3.3.36 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. $H = 1$[17].

3.3.37 SP800-90B Section 4.3 Intermittent vs. Permanent Failures

To comply with the requirement of FIPS 140-3 that a health failure is treated as a FIPS module error[18] in case the LRNG is considered part of a FIPS module, the following approach is taken:

[17] 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 $1 + qbinom(1 - 2^{-30}, 512, 2^{-1})$.

[18] A module error implies that all services of the FIPS module must cease to operate. For the kernel crypto API that is achieved by stopping the kernel, a situation that should be avoided unless a catastrophic event happened.

1. The APT/RCT alpha values that are specific to the entropy source are considered intermittent errors as defined by [5] section 4.3 bullet 2. This means that in case a health test error is identified for the entropy source, a reset of the entropy source is performed. This implies that all potentially collected entropy is considered to be removed, the health test is reset and a complete new power-on health test is performed. The entropy source does not produce data in this case and reports zero entropy to the LRNG. Yet, after a successful power-on health test, the entropy source becomes available again to serve entropy to the LRNG.

2. The LRNG implements a permanent APT/RCT failure with an alpha rate of $\alpha = 2^{-60}$ that causes the kernel to stop. This permanent failure is implemented by identifying two back-to-back intermittent health failures. The chosen alpha value is considered appropriate considering it is to be interpreted as a false-positive rate according to [5] section 4.2. Assuming the LRNG is used in larger numbers of Linux kernels which run all concurrently, the false-positive rate must be applied to all those systems running concurrently which implies that the likelihood of a false-positive to occur is much higher. For example, if 2^{20} Linux kernels with the LRNG execute concurrently, the a false-positive permanent failure now happens with a probability of $2^{20} \cdot \alpha = 2^{-40}$ which again falls into the range of alpha stipulated by [5] section 4.3.

The intermittent failure will always trigger a reset of the offending entropy source. The permanent failure is therefore only intended in case the LRNG operates as part of a FIPS 140-3 module. If the LRNG is outside of a FIPS module, the permanent failure does not need to be considered. The reason is that with constant health test failures, the entropy source is effectively disabled anyway as it does not produce data. Yet, it has the capability to recover in case the environment changes such that no health test errors occur.

When the LRNG operates as part of a FIPS module, the permanent health failure, the entire FIPS module must cease to operate. One way to ensure that is by setting the kernel boot time variable of `lrng_es_mgr.lrng_panic_on_permanent_health_failure` to true. This will `panic` the kernel when a permanent health failure is detected which ensures that the FIPS module will cease to be operational.

3.3.38 SP800-90B Section 4.3 Requirement 4

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

3.3.39 SP800-90B Section 4.3 Requirement 5

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

3.3.40 SP800-90B Section 4.3 Requirement 6

The health tests are applied to the raw, unconditioned time stamp data directly obtained from the noise source before they are injected into the per-CPU collection pool and further processed with the hash conditioning component.

3.3.41 SP800-90B Section 4.3 Requirement 7

The health tests are documented with section 2.5.2.

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.

3.3.42 SP800-90B Section 4.3 Requirement 8

There are no currently known suspected noise source failure modes.

3.3.43 SP800-90B Section 4.3 Requirement 9

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

3.3.44 SP800-90B Section 4.4

The health tests described in section 2.5.2 are applicable to cover the requirements of SP800-90B health tests.

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

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

3.4 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.

3.4.1 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.

3.4.2 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.

3.5 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_IRQ` must be set to Y to enable the interrupt entropy source to which the entropy discussion of this chapter applies to.
- `CONFIG_LRNG_HEALTH_TESTS` must be set to Y.
- Set the configuration option of `CONFIG_LRNG_IRQ_ENTROPY_RATE` to the rate resulting from the entropy assessment outlined in section 3.6.

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.

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 3.5 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_drng_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. If SP800-90C compliance is requested, these interfaces are also providing SP800-90C compliant output. Finally, the following interfaces are DRG.3 compliant:

- /dev/random,
- `getrandom` system call invoked with a zero flag value,
- invoking the in-kernel `lrng_get_random_bytes_full` API,
- invoking the in-kernel `lrng_get_random_bytes_pr` API,
- invoking the in-kernel `get_random_bytes` 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`.

3.6 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 3.3.3. The obtained raw noise data must be processed by the SP800-90B tool to obtain an entropy rate which must be above the entropy rate per time delta that is configured with `CONFIG_LRNG_IRQ_ENTROPY_RATE`: the entropy rate must be above $^{256}/_{\texttt{CONFIG\_LRNG\_IRQ\_ENTROPY\_RATE}}$.

2. Obtain the restart noise data through the raw noise source interface on the intended target platform as explained in section 3.3.3. 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 entropy rate per time stamp mentioned in (1).

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.

The tool set provided as part of LRNG library code distribution provides the measurements and validation tools.

4 Scheduler Entropy Source Assessment

TBD

5 LRNG Specific Configurations

The LRNG offers a secure and appropriate set of features with the default configuration. Yet, use cases may arise where the LRNG should exhibit a different behavior. The flexibility of the LRNG allows a various configurations that are intended to meet different requirements.

5.1 SP800-90C Compliance

The specification of SP800-90C defines construction methods to design non-deterministic as well as deterministic RNGs. As the specification is currently in draft form, the latest available draft from January 21, 2021 is applied.

The specification defines different types of RNGs where the following mapping to the LRNG applies:

- The LRNG follows the construction of RBG2(NP) with the following entropy sources whose outputs are concatenated:

- Auxiliary external entropy sources, such as user-space rngd or the in-kernel `add_hwgenerator_randomness` API, if available. The administrator is responsible to guarantee that this entropy source is compliant to SP800-90B if it alters the entropy estimator maintained by the LRNG via the `RNDADDENTROPY` IOCTL or by providing an entropy estimate via `add_hwgenerator_randomness`. Depending on the selected entropy source, this may be a physical or non-physical entropy source. Note, all data maintained by the auxiliary pool are processed with a vetted conditioning component. Thus, to achieve full SP800-90C compliance for such entropy sources, only one should feed data credited with entropy into the auxiliary pool.
- Interrupt entropy source is the only entropy source that is fully maintained as part of the LRNG and is subject to a full entropy analysis following section 3.3. Furthermore, it is claimed to be fully SP800-90B compliant.
- Scheduler entropy source is the only entropy source that is fully maintained as part of the LRNG and is subject to a full entropy analysis following 4. Furthermore, it is claimed to be fully SP800-90B compliant.
- The Jitter RNG entropy source is used by the LRNG. This entropy source is a fully self-contained SP800-90B entropy source. Its SP800-90B compliance must be assessed separately. If SP800-90B compliance cannot be demonstrated, it must be awarded to credit zero bits of entropy with the configuration documented in section 2.9.1 or by completely disabling it by not selecting the kernel configuration option of `CONFIG_LRNG_JENT`.
- The CPU entropy source is used by the LRNG, if available. On Intel, this uses RDSEED. This entropy source is a fully self-contained SP800-90B entropy source. Its SP800-90B compliance must be assessed separately. If SP800-90B compliance cannot be demonstrated, it must be awarded to credit zero bits of entropy with the configuration documented in section 2.10.1 or by completely disabling it by not selecting the kernel configuration option of `CONFIG_LRNG_CPU`.

By applying Method 2 of section 3.3 in SP800-90C, the entropy provided by all entropy sources can be added which is applied when the LRNG constructs the temporary seed buffer as shown with equation 3.11.

- During instantiation of the DRBG, the LRNG tries to seed the DRBG with at least (DRBG security strength) + 128 bits = 384 bits of entropy. Only when this amount of entropy was obtained to seed the DRBG is considered to be fully seeded and is allowed to produce output. Note, the the following DRBG types are provided by the LRNG:
  - CTR DRBG with AES 256 using a derivation function (selected by default)
  - Hash DRBG with SHA2-512
  - HMAC DRBG with SHA2-512

If the SP800-90C construction is to be used as the "randomness source" following bullet 5 of section "Note to Reviewer" in the SP800-90C document to seed another DRBG with a security strength of 256 bits, either the hash or HMAC DRBG should be used as they are capable of transporting up to 512 bits of entropy and are initially seeded with 384 bits of entropy. Thus, the SP800-90C seeding requirement of providing 384 bits of entropy can be satisfied.

- Reseeding of the DRBG can be triggered by either writing data into /dev/random or by the `RNDRESEEDCRNG` IOCTL. The LRNG guarantees that the reseed operation is only performed if at least 128 bits of entropy is available. If this is not available, the reseed is attempted during the next generate operation. Yet, the generate operation is conducted in any case. Nonwithstanding, the LRNG always attempts to obtain 256 bits of entropy for reseeding. This is considered appropriate because the upper limit when a reseed is ultimately triggered is 2^{20} generate operations or after 10 minutes, whatever is reached earlier. If the DRNG cannot be reseeded it will continue to operate until the next time this threshold is reached. The DRNG will revert to an unseeded stage if it cannot be reseeded by the time it serviced 2^{30} generate requests since the last successful seeding operation.

The requirements from section 6.3 SP800-90C are met as follows:

1. The administrator must use the SP800-90A DRBG LRNG extension as mentioned above to satisfy the requirement.

2. The DRBG can be ACVTS-tested to show compliance to SP800-90A. The entropy sources are to be assessed pursuant to SP800-90B as outlined in the above listing.

3. See the above listing for the reseeding support.

4. If an entropy source is not validated, its entropy estimation must be set to zero as outlined in the above listing.

5. N/A as the LRNG is claimed to conform with RBG2(NP).

6. The entropy sources are listed above. By using concatenation of the output of all entropy sources, the Method 2 SP800-90C is implemented. The entropy of all entropy sources are added.

7. Technically it is possible that all DRBG security strengths can be chosen as the DRBG supports all security strengths. Yet, the LRNG interfaces currently only support the highest security strength of 256 bits to ensure that it can be used for all use cases.

8. The entropy source output is destroyed immediately after it was used to (re)seed the DRBG. Note, the use of the seed for backward secrecy by injecting it into the auxiliary pool via the vetted conditioning operation is considered to not violate the requirement as the seed data is unrecoverable. Furthermore, the seed data is not credited with any entropy during the backward secrecy operation. Therefore, the seed data is only used to further mix the internal state of the LRNG.

9. N/A as the LRNG does not use a CTR DRBG without derivation function.

10. The LRNG attempts to instantiate a DRBG with 3/2s bits of entropy. Only if this succeeds, the DRBG becomes available. The LRNG attempts to reseed a DRBG with s bits of entropy. See the rationale above for the discussion about the minimum entropy size of the reseeding operation of 128 bits.

11. The DRBG only provides output once it is fully seeded as mandated by SP800-90C.

12. An error occurring in the interrupt entropy source triggers a full reset of the LRNG as outlined in section 2.5.2. If the other entropy sources are subject to a health test failures, SP800-90B mandates that they do not produce entropy. Before the first initialization of the DRBG, it is subject to a power-on self test. The LRNG performs power-up self tests as outlined in section 2.16.

13. This requirement is implicitly met by the fact that the LRNG only provides DRBGs with the maximum security strength of 256 bits.

5.1.1 RBG2(P) Construction Method

It is possible to convert the LRNG into the SP800-90C type of RBG2(P). This approach requires that only physical entropy sources are credited with entropy. The following specific settings must be applied in addition to the general configurations listed in the next section:

- Configure the interrupt entropy source to not credited entropy: compile the kernel with the kernel configuration option of `CONFIG_LRNG_IRQ_ENTROPY_RATE=4294967295`. This setting ensures that the interrupt entropy source still collects data, but it is not credited with entropy. If it shall be completely disabled, the compile time option of `CONFIG_LRNG_IRQ` must be unset.

- Configure the scheduler entropy source to not credited entropy: compile the kernel with the kernel configuration option of `CONFIG_LRNG_SCHED_ENTROPY_RATE=4294967295`. This setting ensures that the scheduler entropy source still collects data, but it is not credited with entropy. If it shall be completely disabled, the compile time option of `CONFIG_LRNG_SCHED` must be unset.

- Configure the Jitter RNG entropy source to not credit entropy: compile the kernel with the kernel configuration option of `CONFIG_LRNG_JENT_ENTROPY_RATE=0`. If the value is set to 0, the entropy source still collects data, but it is not credited with entropy. If it shall be completely disabled, the compile time option of `CONFIG_LRNG_JENT` must be unset.

- Configure the legacy RNG entropy source to not credit entropy: compile the kernel with the kernel configuration option of `CONFIG_LRNG_LEGACY_RNG_ENTROPY_RATE=0`. If the value is set to 0, the entropy source still collects data, but it is not credited with entropy. If it shall be completely disabled, the compile time option of `CONFIG_LRNG_LEGACY_RNG` must be unset.

- Adjust the entropy rate of the CPU entropy source as needed: compile the kernel with the kernel configuration option of `CONFIG_LRNG_CPU_ENTROPY_RATE=256` when full entropy is assumed to be provided by the CPU entropy source. In general, set any value between 0 and 256 if the default value is not appropriate. If the value is set to 0, still collects data, but it is not credited with entropy. If it shall be completely disabled, the compile time option of `CONFIG_LRNG_CPU` must be unset.

- If other hardware entropy sources are considered deliver entropy, they may inject data into the LRNG via either the `add_hwgenerator_randomness` or via the IOCTL `RNDADDENTROPY`. Note, only physical entropy sources must provide data that is credited with entropy. Any other data fed into the LRNG via those interfaces must not be credited with entropy.

Important note: The entropy rate provided by the CPU entropy source plus all other physical entropy sources together must ensure they provide sufficient entropy. "Sufficient entropy" is provided when the entropy rate equals the hash type used by the LRNG. For example, if /proc/lrng_type shows that SHA2-256 is used, than 256 bits of entropy is sufficient. If SHA2-512 is used, 512 bits of entropy should be provided.

Naturally, the hardware entropy sources that are credited with entropy must be compliant to SP800-90B.

In case the RBG2(P) construction method is achieved, the following additional requirement from section 6.3 SP800-90C is met:

- Requirement 5: With the required configuration mentioned before the LRNG will only count the physical entropy sources towards fulfilling the requested amount of entropy.

5.1.2 SP800-90C Compliant Configuration

SP800-90C compliance is only achieved when all of the following settings are achieved.

The following compile-time settings must be observed:

- The option `CONFIG_RANDOM_DEFAULT_IMPL` must not be set to ensure that the LRNG provides the implementation of all interfaces like /dev/random, /dev/urandom or `getrandom`.

- The kernel configuration option `CONFIG_LRNG_OVERSAMPLE_ENTROPY_SOURCES` is set. This option guarantees the following:

  - The final conditioning operation applied to the interrupt entropy source as well as to the auxiliary pool require 64 additional bits of entropy when obtaining data for the temporary seed buffer. This complies with the requirement specified in section 4.3.2 SP800-90C about vetted conditioning components. This ensures the conditioning components provide full entropy.

  - When the DRNG is initially seeded, it is attempted to be seeded with 384 bits of entropy at least. This complies with the requirement specified in section 6.2.1 bullet 2 of SP800-90C requiring 3/2s bits

of entropy for the initial seeding. Note, the LRNG applies a stepwise seeding of 32, 128 and 256 bits of entropy during initialization. When the final step of 256 bits is to be performed, the LRNG will guarantee that at least 384 bits of entropy are collectively pulled from all entropy sources. Only if this is achieved, the SP800-90C compliant-marked interfaces of the LRNG specified in section 3.5 will produce random numbers.

- The option `CONFIG_LRNG_DRBG` must be either set to yes or module. This option ensures:
  - The used hash for the conditioning function must be capable of maintaining at least 384 bits of entropy. This is commonly only achieved with SHA-384 or SHA-512. With this option, SHA-512 is used.
  - SP800-90C implies that the DRNG is always SP800-90A compliant which is provided with this option.
  - For an absolute strict compliance to SP800-90C, the SP800-90C DRBG must be initialized with the 384 bits of entropy instead of being initialized with 32/128 bits as applied during early boot time. If you want to absolutely guarantee that a DRBG is initialized with 384 bits of entropy, you must compile the SP800-90A support as module (the configuration `CONFIG_LRNG_DRBG=m`) and you must use the following approach to load the kernel module of `lrng_drng_drbg.ko`:
    1. Either open /dev/random and read at least one byte of data or use the getrandom(2) system call without any specific option and read one byte. When the request is satisfied, the initial seeding steps of 32 and 128 bits are completed.
    2. Insert the `lrng_drng_drbg.ko` kernel module. The LRNG goes back to an unseeded stage until at least 384 bits of entropy are present to initial seed the first DRBG. All subsequent DRBGs are equally seeded once 384 bits of entropy are present for each.

- `CONFIG_RANDOM_TRUST_CPU` must not be set unless the CPU-based entropy source (e.g. RDSEED 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.

- Enable `CONFIG_LRNG_IRQ` configure the interrupt entropy source compliant to SP800-90B as outlined in section 3.5 if the interrupt entropy source is considered to be compliant to SP800-90B by accepting the assessment in chapter 3 and by applying the testing of the entropy source as outlined in section 3.6 on the target system.

- Enable `CONFIG_LRNG_SCHED` configure the scheduler entropy source compliant to SP800-90B as outlined in section 4 if the scheduler entropy source is considered to be compliant to SP800-90B by accepting the assessment in chapter 4 and by applying the testing of the entropy source as outlined in this chapter on the target system.

- The kernel must be compiled with the kernel configuration option of `CONFIG_LRNG_CPU_ENTROPY_RATE=0` or unset the option `CONFIG_LRNG_CPU` unless the CPU-based entropy source (e.g. RDSEED on Intel) have an SP800-90B compliant entropy assessment and comply with all requirements from SP800-90B.

- The kernel must be compiled with the kernel configuration option of `CONFIG_LRNG_JENT_ENTROPY_RATE=0` or unset the option `CONFIG_LRNG_JENT` unless the Jitter RNG entropy source has an SP800-90B compliant entropy assessment and comply with all requirements from SP800-90B[19].

The following requirements apply to the runtime configuration:

- The kernel must be booted in FIPS mode to ensure the oversampling for the conditioning and the initial seeding of the DRBG is applied. This is achieved with the kernel command line option of `fips=1`.

- If the `lrng_drng_drbg` kernel module is compiled (i.e. `CONFIG_LRNG_DRBG` was set to module), the administrator must ensure the `lrng_drng_drbg` kernel module is loaded into the kernel before the first user of the LRNG accessing the SP800-90C compliant interfaces appears. Note: it is permissible to implement a separate kernel module that provides the DRBG and the hash operation following the API defined with the lrng.h header file. However, in this case the following conditions must hold:

  - The DRBG must be SP800-90A compliant and provide 256 bits of security strength. It is permissible that the invocation of this DRBG may cause the caller to sleep. The DRBG is never called in atomic contexts.
  - If the CTR DRBG is used, it must provide a derivation function.
  - The hash that is specified with the kernel module must provide at least 384 bits of output size and security strength (i.e. either one of the following SHA2-384, SHA2-512, SHA3-384 or SHA3-512). The implementation of the hashes *must not* sleep or use a mutex. The hash is called in atomic contexts.

- 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.

- 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 entropy source is SP800-90B compliant with its own entropy assessment.

[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.

To verify that the SP800-90C compliance is achieved, the file /proc/lrng_type provides an appropriate status indicator. The SP800-90C compliant is ensured only for the respectively marked LRNG interfaces specified in section 3.5. All other interfaces are not providing SP800-90C compliant random numbers.

5.2 FIPS 140-3 IG D.K

To comply with the FIPS 140-3 IG D.K, requirements, the considerations of section 5.1 are applicable.

In addition, IG D.K mandates that in order to seed a DRBG from a primary DRBG, the primary DRBG must behave like a conditioning functions - only as much data is allowed to be produced as entropy was received. This implies that the primary DRBG must be reseeded before delivering data to the caller. The LRNG's DRBG can be operated in this manner with the following invocation methods:

- When opening /dev/random with the `O_SYNC` flag, the DRBG behaves FIPS 140-3 IG D.K compliant.
- When invoking the `getrandom` system call with the flag `GRND_RANDOM`, the DRBG behaves FIPS 140-3 IG D.K compliant.

Please note that using the interfaces in the mentioned manner may imply short reads the caller *must* handle. For example, if the caller wants 384 bits of data from the DRBG, the interfaces invoked with the mentioned flags will at most return 256 bits as this is the maximum amount of entropy that can be processed. The LRNG interfaces will report the amount of data returned. The caller must process the returned information and re-invoke the LRNG interface again until sufficient data is returned.FIPS 140-3 Considerations

For the following entropy sources, the LRNG contains the SP800-90B compliant detection of a permanent failure as outlined in section 3.3.37:

- Jitter RNG
- Interrupt entropy source
- Scheduler-based entropy source

If one of those entropy sources delivers entropy and the LRNG is part of a FIPS module, the permanent health failure must trigger the ceasing of all operations of the FIPS module. To ensure that, the LRNG can be booted with the kernel command line option documented in section 3.3.37 which enforces a panic upon reaching a permanent health error.

5.3 AIS 20 / 31

The German BSI defines construction methods of RNGs with AIS 20/31 with the revision from 2022 available as draft at the time of writing. In particular, this document defines different classes of RNGs where the LRNG is capable of meeting class NTG.1.

The NTG.1 requirements are met as follows:

- NTG.1.1: The LRNG offers interfaces to collect raw unconditioned entropy as outlined in section 2.17.

- NTG.1.2: The LRNG maintains an entropy estimator for the auxiliary pool as well as each entropy pool. The credit operation of the LRNG is documented in section 2.1. To outline the debit operation section 3.3.6 provides several examples showing the approach. For each of the fast entropy sources, the LRNG obtains the entropy estimate when fetching data from them and applies this estimate when seeding the DRNG.

- NTG.1.3: The LRNG provides a switchable DRNG support using a ChaCha20-based DRNG which is documented in section 2.13.2. Furthermore, the LRNG allows using an SP800-90A compliant DRBG (all three types of CTR-DRBG, Hash DRBG and HMAC DRBG are supported). All those DRNGs are characterized as DRG.3 since they all support backward secrecy and use contemporary cryptographic primitives for the state transition as well as output generation.

- When compiling the LRNG with the option `CONFIG_LRNG_AIS2031_NTG1_SEEDING_STRATEGY` enabled and booting the kernel with the kernel command line option of `lrng_es_mgr.ntg1=1`, the initial seeding strategy of the LRNG with the 32/128/256 bit reseeds is altered to 32/128 bits followed by two entropy sources required to deliver 220 bits of entropy. Only when the last step is completed, the blocking output interfaces listed in section 3.5 are released to produce random numbers. This implies that those interfaces operate NTG.1-compliant.

- NTG.1.5: The output functions of the offered DRNGs are all based on contemporary cryptographic primitives. Therefore, the output is assumed that it cannot be distinguished from random number output sequences of an ideal RNG. This is supported by the fact that the SP800-90A DRBGs are successfully passing the NIST CAVP test suite. The ChaCha20 DRNG can be tested using the ChaCha20 DRNG stand-alone implementation which is derived from the LRNG implementation. The tester can verify that both operate identical because the power-on self test in the stand-alone ChaCha20 DRNG implementation is identical to the LRNG power-up self test found in lrng_lrng_selftest.c.

- NTG.1.6: Using the entropy analysis tools such as the NIST SP800-90B test tools, results shown in appendix C can be obtained for the interrupt entropy source. For the second entropy source, such assessment must be performed as well. For example, the Jitter RNG is provided with a complete independent entropy assessment. When comparing the results with the heuristic entropy estimate, it is clear that the LRNG collects more entropy than it credits. Further, considering the mathematical operations processing the raw entropy data outlined in section 3.3.5, the processing is considered to not destroy entropy.

Note, the LRNG also complies with the AIS 20/31 version 2011 where all statements from above comply with the exception of NTG.1.4. This requirement is met as follows:

- NTG.1.4: Due to the blocking behavior and only returning as many bits of data as entropy is available via /dev/random opened with `O_SYNC`, or using `getrandom(2)` with `GRND_RANDOM` the LRNG meets the requirement.

5.3.1 NTG.1 Compliant Configuration - Version 2022

To achieve NTG.1 compliant operation, the following configuration must be applied. It is permissible to use the configuration in parallel with the SP800-90B and SP800-90C configurations.

The following compile-time settings must be observed:

- The LRNG compile time option of `CONFIG_LRNG_AIS2031_NTG1_SEEDING_STRATEGY` must be enabled.

- The LRNG must have access to two entropy sources which are capable of delivering 220 bits of entropy each. It is not necessary that the entropy sources deliver the 220 bits in one call, but may provide the required amount throughout multiple calls. The LRNG performs the invocation as often as needed to collect the 220 bits. For example, the interrupt entropy source can deliver 220 bits of entropy In addition, the Jitter RNG entropy source is capable of delivering 220 bits of entropy when booting the kernel with the kernel compile-time option of `CONFIG_LRNG_JENT_ENTROPY_RATE` is set to any value larger than zero. See section 2.9.1 for details on the Jitter RNG entropy rate.

The following requirements apply to the runtime configuration:

- None.

To verify that the NTG.1 compliance is achieved, the file /proc/lrng_type provides an appropriate status indicator. The NTG.1 (2022) compliant is ensured only for the respectively marked LRNG interfaces specified in section 3.5. All other interfaces are not providing SP800-90C compliant random numbers.

5.3.2 NTG.1 Compliant Configuration - Version 2011

To achieve NTG.1 compliant operation, the following configuration must be applied. It is permissible to use the configuration in parallel with the SP800-90B and SP800-90C configurations.

The following compile-time settings must be observed:

- None

The following requirements apply to the runtime configuration:

- The caller must use /dev/random and open it with `O_SYNC`, `getrandom(2)` with `GRND_RANDOM`.

To verify that the NTG.1 compliance is achieved, the file /proc/lrng_type provides an appropriate status indicator. The NTG.1 (2011) compliant is ensured only for the respectively marked LRNG interfaces specified in section 3.5. All other interfaces are not providing SP800-90C compliant random numbers.

5.3.3 DRG.4 / PTG.3 Compliant Configuration

The LRNG allows a configuration that would make the LRNG compliant to the requirements of a DRG.4 as well as a PTG.3. This is achieved by deconfiguring all entropy sources except one that provides the data from a PTG.2 entropy

source. For example, an rngd can be created having access to a smart card that is provides a PTG.2 entropy source. In this case, all other entropy sources must be deactivated using the following runtime configuration.

The PTG.3 naturally can only be claimed if the PTG.2 entropy source is part of the validation. If the PTG.2 entropy source is not part of the validation, a DRG.4 must be claimed instead.

For achieving such compliance claim, the following compile-time configuration must be applied:

- The kernel command line must contain the following setting to ensure the interrupt entropy source is not credited with entropy: `CONFIG_LRNG_IRQ_ENTROPY_RATE=4294967295`.
- Configure the Jitter RNG entropy source to not credit entropy: boot the kernel with the kernel compile-time option of `CONFIG_LRNG_JENT=0`. If the value is set to 0, the entropy source is disabled.
- Adjust the entropy rate of the CPU entropy source to zero: boot the kernel with the kernel compile-time option of `CONFIG_LRNG_CPU_ENTROPY_RATE=0`. If the value is set to 0, the entropy source is disabled.

The following requirements apply to the runtime configuration:

- The PTG.2 entropy source delivers its entropy to the LRNG via either the `add_hwgenerator_randomness` or via the IOCTL `RNDADDENTROPY`.

6 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.

Note, this section applies to the legacy /dev/random implementation before the Linux kernel version 5.18 was released. After the release of 5.18 with its re-architecture, older LTS-kernels received the newer implementation as well.

6.1 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 entropy sources where only the interrupt entropy source will always be present. Thus the test is first performed with all entropy sources enabled followed by disabling the fast entropy sources of CPU entropy source.

Listing 1: Time until fully initialized -- LRNG using all entropy sources

```
$ dmesg | grep "LRNG minimally seeded"
[  1.718705] lrng_es_mgr: LRNG minimally seeded with 128 bits of entropy
--- 0 sm@x86-64 ~ ------------------------------------------------------------
$ dmesg | grep "LRNG fully seeded"
[  2.056685] lrng_es_mgr: 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_es_cpu.cpu_entropy=0` and `lrng_es_jent.jent_entropy=0` to disable the fast entropy sources.

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

```
$ cat /sys/module/lrng_es_cpu/parameters/cpu_entropy
0
--- 0 sm@x86-64 ~ ------------------------------------------------------------
$ dmesg | grep "LRNG minimally seeded"
[  1.683981] lrng_es_mgr: LRNG minimally seeded with 128 bits of entropy
--- 0 sm@x86-64 ~ ------------------------------------------------------------
$ dmesg | grep "LRNG fully seeded"
[  2.110482] lrng_es_mgr: 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 entropy 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.

6.2 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	42
LRNG with health tests enabled, but no SP800-90B compliance and using 8 LSB of time stamp	78
LRNG with SP800-90B compliant health tests	138
Legacy /dev/random implementation	97

Table 2: 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 collection pool. The table above shows the measured number for the default collection size of 1,024 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.

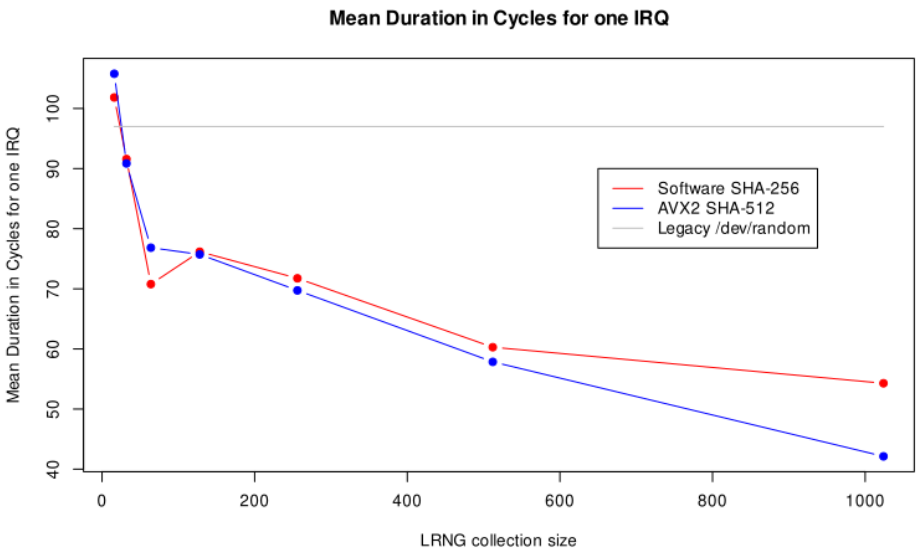

Figure 6.1: Average Cycle Count To Process One Interrupt Depending on Collection Size

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 dur-

ing 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 collection pool is completely filled. When the collection pool 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.

Note, the interrupt handler performance can be even more increased by disabling the continuous compression support. By either setting `CONFIG_LRNG_CONTINUOUS_COMPRESSION_DISABLED` to hard-code disabling the continuous compression support or by configuring `CONFIG_LRNG_SWITCHABLE_CONTINUOUS_COMPRESSION` and to disable it at boot-time, the hash operation during the interrupt handling can be disabled entirely. This implies that the occasional hash operation in the interrupt handler is not executed implying that the interrupt handler of the LRNG only concatenates the received data into an array improving the performance even further.

6.3 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 3. 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.

[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 3: LRNG performance on 64-bit

In addition, table 4 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 4: 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 2.13.1.

6.4 ChaCha20 Random Number Generator

The ChaCha20 DRNG is analyzed to verify the following properties:

- whether the self-feeding RNG ensures backward secrecy, and
- whether the absence of the CPU entropy 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 entropy sources have been disabled to

clearly demonstrate that the backward secrecy 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 entropy 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 backward secrecy 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.

6.5 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 [2] 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 [5] sections 4.4 and 4.5. This includes neither a start-up health test nor a run-time health test.

As mandated in [5] 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 [5] section 3.1.6.

When using multiple noise sources such as `add_disk_randomness`, `add_input_randomness` or `add_interrupt_randomness`, [5] 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, [5] 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 [5] 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 [5] 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 [4].
- 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 [5] section 2.2.

6.6 Use of Legacy RNG with LRNG

As outlined in section 2.11 the entropy source of the legacy RNG can be enabled with the LRNG. However, this implies that the LRNG automatically disables:

- the IRQ entropy source,
- the common interface implementations, such as the getrandom system call, /dev/random, /dev/urandom, or get_random_bytes.

Figure 6.2 outlines the disabled functions with the shaded areas.

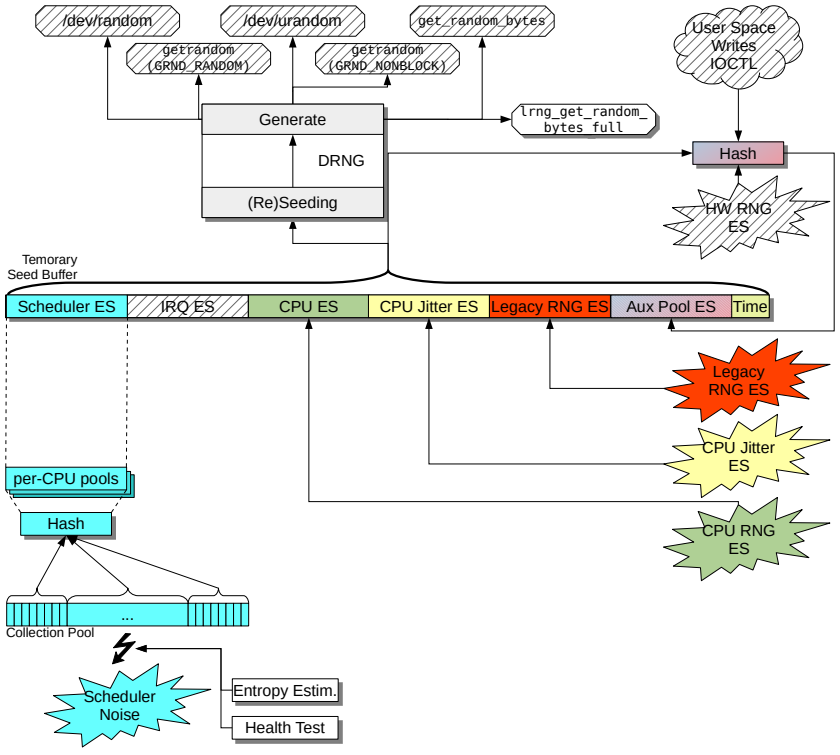

Figure 6.2: LRNG Functions Disabled with Legacy RNG

On the other hand, when the IRQ entropy source is enabled, the legacy RNG must by definition be disabled. Again, the shaded areas marked with figure 6.3 outlines the LRNG mechanisms.

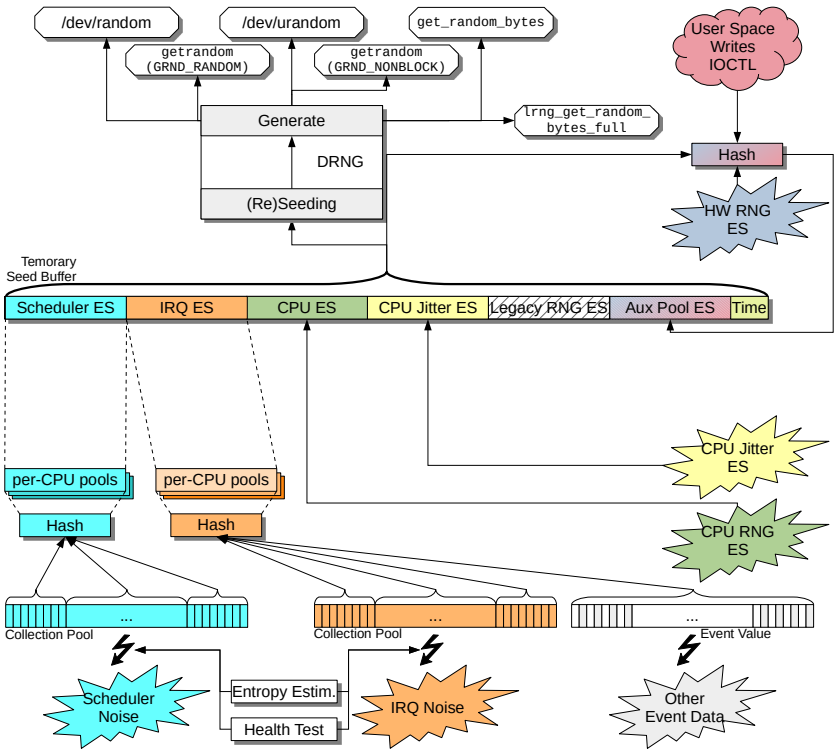

Figure 6.3: LRNG Functions Disabled with IRQ ES

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.

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 3.3.5. 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 3.3.6 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 3.3.6, 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 3.3.3.

Test System	Entropy of 1,000,000 Traces	Sufficient Entropy
AMD Ryzen 5950X - 64-bit KVM environment	4.531023	Y
AMD EPYC Milan 7713 2 sockets 128 cores 8-way NUMA	7.007947	Y
ARMv7 rev 5	1.9344	Y
ARMv7 rev 5 (Freescale i.MX53)[22]	7.07088	Y
ARMv7 rev 5 (Freescale i.MX6 Ultralite)[23]	6.638399	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 E7 4870 8 sockets 160 CPUs 8-way NUMA	7.287790	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[24]	7.157064	Y
MIPS Lantiq 34Kc V5.6[25]	7.032740	Y
Qualcomm IPQ4019 ARMv7[26]	6.638405	Y
SiFive HiFive Unmatched RISC-V U74	2.387470	Y

Table 6: 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`.

[22] USBArmory MK I
[23] USBArmory MK II
[24] Ubiquiti Nanostation M5 (xm)
[25] AVM Fritz Box 7490
[26] AVM Fritz Box 7520

The effect of the application of the GCD can be clearly demonstrated with the Intel Atom Z530 listed in the above table. The table shows the measurement of without dividing the time stamp by the GCD. The GCD measurement during boot detects that all time stamps have a GCD of 4 which means that the low 2 bits are always unset. Re-running the entropy measurements again on the time stamp that is already divided by 4, the resulting entropy rate is 7.299 bits of entropy per the 8 LSB of the time stamp. This clearly shows that the now considered 2 additional bits of the 8 LSB time stamp after the division with the GCD provides additional entropy which again demonstrates that the LRNG heuristic entropy estimation is safe. It may be noted that by considering 2 additional bits that are now considered for the entropy rate seemingly provide more than 2 bits of entropy (before the GCD, the entropy rate was measured at 3.38 which would imply that by adding 2 bits that may provide full entropy, the rate cannot be higher than 5.38). This seeming inconsistency is due to the fact that a new test run was conducted to get new entropy data. The ping flood used to trigger the IRQ events may have been affected by network congestion adding some delays to the interrupts caused by the ping flood.

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 continuous read operation on /proc/lrng_type is started. While the read operations are performed, 5,000 insmod / rmmod operations of the `lrng_drng_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.

The following additional tests were conducted:

- use of automated test harness validating the different LRNG configurations
- use of the kernel-internal KASAN test framework,

- use of the kernel-internal UBSAN test framework,
- use of the kernel-internal lockdep test framework,
- use of the kernel-internal Sparse test framework,
- use of the kernel-internal memory leak detector test framework,
- use of the LRNG without compiling the kernel crypto API,
- syscall validation testing,
- test of the LRNG in atomic contexts,
- the regression test suite provided with the LRNG showing that the LRNG operation with different configurations is as expected.

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

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

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

[5] 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

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

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

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

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

G Change Log

Date	LRNG Version	Change
2022-05-14	v45	New: Scheduler ES New: Legacy RNG ES New: `lrng_get_random_bytes_full` API New: Kernel crypto API interface New: HW_RAND interface New: /dev/lrng interface Small editorial changes
2022-05-30	v45	Fix typos in section 5.1
2022-07-31	v46	Add prediction resistance support and DRBG chaining support with O_SYNC /proc/sys/kernel/random/[poolsize|entropy_avail] now only shows auxiliary pool Editorial changes Update to SP800-90C compliance specification Update NTG.1 documentation

Date	LRNG Version	Change
2022-10-08	v47	Add GRND_RANDOM for getrandom(2) Addition of GRND_SEED to getrandom(2) AIS 20/31 (2011) compliance Add AIS 20/31 (2022) compliance
2023-01-08	v48	Add forced seeding operation discussion Add FIPS IG D.K documentation with 5.2
2023-02-22	v49	Add FIPS 140-3 discussion
2023-04-26	v50	Replace the notion of FIPS 140-3 degraded mode with discussion of permanent vs intermittent health failures
2023-09-04	v51	Add asynchronous Jitter RNG processing
2023-11-27	v52	Editorial changes
2024-01-22	v53	No changes for release v53
2024-03-21	v54	Update URLs to chronox.de