Federal Office
for Information Security

Documentation and Analysis of the Linux Random Number Generator

Version: 5.0

Document history

Version	Date	Editor	Description
5.0	2022-09-01	Stephan Müller	Kernel version 5.18.1

This analysis was prepared for BSI by atsec information security GmbH.

Federal Office for Information Security
Post Box 20 03 63
D-53133 Bonn
Internet: https://www.bsi.bund.de

Table of Contents

Figures

Tables

1 Introduction

The evaluation of the suitability and quality of cryptographic mechanisms is tasked to the BSI (Bundesamt für Sicherheit in der Informationstechnik – Federal Office for Information Security) in Germany. The BSI therefore initiated this study of the Linux Random Number Generator (Linux-RNG). Linux is used not only in numerous server and desktop systems but also in mobile IT devices, covering sensitive areas in enterprises as well as in government. Good random numbers are a prerequisite for the secure processing of data in governmental as well as enterprise and end user systems.

The Linux operating system kernel offers via the device files /dev/random and /dev/urandom as well as the `getrandom` system call access to its random number generator for user space applications. In addition, the Linux-RNG offers in-kernel interfaces to allow other Linux kernel components to obtain random numbers. The functionality, properties and usage of the Linux-RNG are subject to assessment in this document. This assessment covers the collection of entropy and discussion of the noise sources, the post-processing of the collected true random data and the generation of random numbers that are provided to the calling applications or in-kernel service functions.

One focal point of this study in addition to the assessment of the algorithmic part of the Linux-RNG is the estimation of the entropy of the raw data that is provided to the Linux-RNG by the noise sources. The goal of the assessment is to determine whether the Linux-RNG is able to provide 100 bits, the threshold defined by [TR021021], of entropy early after a system boot.

The entire implementation of the Linux-RNG is explained in detail to allow a full understanding of the flow of information, starting at the point where the entropy is gathered up to the point where random numbers are returned to either in-kernel or user space callers. Each of the noise sources providing entropy to the Linux-RNG is described, detailing why the obtained data is unpredictable. The design description is complemented with functional verification and statistical tests covering the different noise sources and all stages of data processing. The primary goal is to analyze whether the entropy obtained from the noise sources is appropriately collected, compressed, processed without losing entropy, and delivered to the caller.

Besides, this study is intended to analyze whether the design of the Linux-RNG complies with the NTG.1 or DRG.3 requirements for RNGs specified by AIS 20/31 [AIS2031]. AIS 20/31 is a specification issued by the BSI to design and analyze deterministic as well as non-deterministic random number generators. This document provides support for an analysis of RNGs by defining different classes of RNGs where NTG.1 specifies requirements for “non-physical true random number generators” and DRG.3 specifies requirements for “deterministic random number generators”.

The tests conducted for this study are fully explained to the extent that users can reproduce them. Further, the tests are documented with a rationale for why they are appropriate to observe the intended behavior of the Linux-RNG. For each test, the obtained results are discussed with a conclusion as to whether the observed behavior supports the generation and maintenance of entropy. The source code of the tests are made available to the BSI to allow fellow-researchers to verify the testing and its conclusions.

The tests are all performed on an Intel x86 hardware system, as well as a virtual machine executed on Intel x86, using the virtualization extension of VT-x. The majority of the tests are applicable to other architectures as the code implementing the Linux-RNG is independent of the hardware architecture. One exception is the assessment of the noise sources, which is only applicable to the tested architecture because the majority of the entropy is derived using a high-resolution time stamp. Albeit all major hardware architectures including ARM, MIPS, IBM System Z, POWER, and Sparc have high-resolution timers used by the Linux-RNG, about one half of all hardware architectures supported by Linux do not provide such a high-resolution timer. Even if an architecture provides a high-resolution timer, the resolution may still vary and thus the amount of entropy derived from this timer.

The entropy is derived from events triggered by hardware devices. The number and type of devices vary greatly between architectures. Thus, the amount of entropy available to the Linux-RNG varies too. However, the quality of the entropy and the amount of entropy per device event is very consistent for one hardware

architecture. Therefore, the test results obtained on one particular Intel x86 hardware system can be applied to other Intel x86 hardware systems.

Based on the design and test results, recommendations about using the Linux-RNG are given to allow vendors an appropriate employment of the Linux-RNG into their systems.

1.1 Authors

Stephan Müller, atsec information security GmbH

Sebastian Mayer, atsec information security GmbH

Dr. Caroline Holz auf der Heide, atsec information security GmbH

Dr. Andreas Hohenegger, atsec information security GmbH

1.2 Copyright

2 Architecture of Non-Deterministic Random Number Generators (NDRNGs)

The analysis of the Linux-RNG shall answer the question whether it is a complete standalone NDRNG that has no further dependencies on other software. To draw such conclusions, this section describes a general architectural model for NDRNGs. During the design description of the Linux-RNG, it will be compared to the general architectural model to understand whether all components of a NDRNG are present within the Linux-RNG.

2.1 Terminology

Before starting with the technical aspects of RNGs, the terminology used in the subsequent sections and chapters is defined.

Term	Definition
ChaCha20 DRNG	The ChaCha20 deterministic random number generator (DRNG) referred to in this document is conceptually similar to an entropy pool: a memory segment holds ideal random data. The cryptographic function of ChaCha20 is used as a state-transition function as well as an output function. The ChaCha20 implementation is derived from [RFC7539] sections 2.1 to 2.3 where the random number is the key stream generated by the ChaCha20 block operation.
Conditioning	Conditioning is the process where input data is processed such that the resulting data will not allow an observer to derive the original input data. In addition, conditioning is also the process to reduce statistical weaknesses exhibited in the raw data stream. Such conditioning operations can be performed using cryptographic or non-cryptographic operations. An example for cryptographic conditioning is the application of a hash function. A linear feedback shift register (LFSR) is an example for a non-cryptographic conditioning operation.
Deterministic Random Number Generator (DRNG)	A deterministic random number generator is an algorithm for generating sequences of data with properties approximating those of random numbers. The output of a DRNG is determined by its initial seed data. When initialized with the same seed, it will produce the same sequence of data. See also “Random Number Generator”.
Entropy Pool	The term entropy pool in this document refers to a memory area holding true random data which is processed with a deterministic input and state-transition function as well as output function based on the Blake2s hash / keyed hash function. Considering the state-transition and output function, the processing of the data maintained by an entropy pool is fully deterministic in nature.
Human Interface Devices (HID)	The term human interface device collectively refers to all hardware devices that a human user can use to interact with a computer, such as a keyboard, a mouse, a tablet and similar.
Ideal Random Number Generator	An ideal random number generator generates random numbers which are independently and identically distributed (IID as defined by [SP800-90B]), and follow an equidistribution. These requirements imply that the generated data does not exhibit any statistical patterns, e.g. are serially uncorrelated samples.
Jiffies	The Linux kernel maintains a monotonically increasing counter called Jiffies. This counter is incremented by one at a fixed time interval. This time interval is specified

Term	Definition
	during compile time of the kernel. The default on Intel x86 platforms is 1000 Hz, i.e. the Jiffies counter is incremented once every millisecond. Other common values are 100 Hz and 250 Hz.
Most Significant Bits (MSB)	The processing of a bit-stream may operate only on a subset of it. To reference the location of that subset, the term “most significant bits” refers to the left-most bits of the bit-stream. They are called most significant bits as they denominate large integer numbers when viewing the bit-stream as an integer. See also Least Significant Bits.
Least Significant Bits (LSB)	The processing of a bit-stream may operate only on a subset of it. To reference the location of that subset, the term “least significant bits” refers to the right-most bits of the bit-stream. They are called least significant bits as they denominate small integer numbers when viewing the bit-stream as an integer. See also Most Significant Bits.
Linear Feedback Shift Register (LFSR)	A linear feedback shift register is a special case of a shift register where the input data is a linear function of the previous state of the LFSR. This implies that an LFSR is a circular application of a shift register. All LFSRs discussed in this document use the linear function of XOR to combine parts of the previous state with input data. The LFSRs discussed in this study are all Fibonacci LFSR where the parts of the previous state that are selected are based on taps defined by a polynomial.
Linux Random Number Generator (Linux-RNG)	The Linux Random Number Generator is the software component in the Linux kernel that implements the logic to provide random numbers via the /dev/random, /dev/urandom device files and the `getrandom` system call to user space. In addition, the Linux-RNG provides random numbers to in-kernel users via the `get_random_bytes` application programming interface (API). The Linux-RNG is completely implemented in the Linux kernel source code file drivers/char/random.c.
Noise Source	A noise source provides true random data. In case of the Linux-RNG, a noise source is the software component that monitors hardware events to derive entropy from these events.
Non-deterministic Random Number Generator (NDRNG)	A non-deterministic random number generator generates a sequence of data that cannot be predicted better than using random chance.
Non-Uniform Memory Access (NUMA)	Hardware systems with many CPUs may not place all CPUs on one motherboard, but use several individual motherboards with CPUs which communicate with a high-speed interconnect. Each individual motherboard is called a node. Access to memory present on the same motherboard as the requesting CPU (i.e. “NUMA-node local access”) is faster than CPUs requesting access to memory on a different NUMA-node.
Random Number Generator (RNG)	See also “Non-deterministic Random Number Generator”.
SHA-1	SHA-1, short for Secure Hash Algorithm, is a cryptographic one-way function where an input bit stream of arbitrary length is turned into an output bit string of 160 bits. SHA-1 exhibits various cryptographic properties to convert arbitrary input data to output data that shows the characteristics of an ideal random number generator.
True Random	True Random Data is a data stream of arbitrary size that is believed to contain entropy.

Term	Definition
Data	The amount of entropy contained in the true random data is not defined.

Table 1: Terminology

2.2 General Architecture

NDRNGs can be found in many forms, including:

- RNGs and noise sources designed for the sole purpose of providing entropy bits. Such noise sources can be found on physical devices like smart cards, special circuitry, hardware security modules (HSMs), etc.
- RNGs and noise sources that observe the behavior of events of regular hardware. These would include observing the timing of events obtained from human interface devices (HID) (e.g. mouse movements or typing on a keyboard), block devices (e.g. spinning hard disks) or interrupts.
- Noise sources that include a RNG utilizing capabilities of the CPU, including timer-based noise sources, CPU instructions using hardware noise sources like RDRAND on Intel processors (see [INTELDRNG]), etc.

Irrespective of the nature of the non-deterministic random number generator, all forms follow a general design pattern outlined in figure 1. This illustration closely resembles the specification outlined in [SP800-90B], chapter 2, regarding the noise source and [SP800-90C], section 5.1, for the interlink between a noise source and a DRNG. In addition, this figure also relates to the description of a noise source given in [AIS2031] with the difference that the health tests are not as pronounced in figure 1.

The document [SP800-90B] covers the design requirements as well as quantitative assessments of noise sources. The description is complemented by [SP800-90C] outlining principles on the architecture of NDRNGs where one or more noise sources are combined with deterministic post-processing to deliver cryptographically strong random numbers. Both documents are provided by the US governmental body, NIST.

The document [AIS2031] is similar in nature to the aforementioned documents by outlining the architecture of noise sources, their combination with deterministic post-processing logic to deliver cryptographically strong random numbers, and the discussion of how such designs are assessed. [AIS2031] is published and mandated by the German governmental body, BSI.

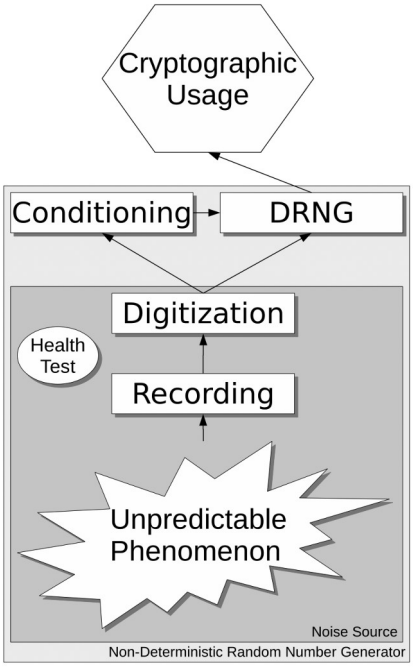

Figure 1: Non-deterministic random number generator architecture

Figure 1 shows the entire logic flow for generating random numbers. The origin of any random number is the noise source marked as a dark gray field in figure 1. The output of a noise source is fed into a DRNG which generates the output for cryptographic use cases. In some systems, a conditioner is applied to the output of the noise source where the output of the conditioner is then used as input for a DRNG. The combination of the noise source and the DRNG, possibly supported by a conditioner, is a non-deterministic random number generator. Figure 1 denotes it with a light gray box.

It is possible, and even often seen in real-life environments, that multiple DRNGs are chained. Such a chain of DRNGs is fed by the noise source or conditioned noise source data. For example, user space cryptographic daemons using the OpenSSL cryptographic library obtain their seed from /dev/random or the `getrandom` system call (depending on the used OpenSSL version) and its deterministic component to seed the OpenSSL deterministic random number generator.

The architecture of a non-deterministic random number generator together with its noise source as shown in figure 1 contains the following major parts:

- A phenomenon is measured that exhibits an unpredictable or partially unpredictable pattern to the observer. It is key to understand that the unpredictability always relates to the observer and may vary depending on the type and skills of the observer – i.e. the unpredictability and therefore the resulting entropy is relative to the observer. For a lot of noise sources, the observed phenomenon may be completely deterministic if all parameters are known that affect the phenomenon. Such noise sources depend on the fact that one or more of these parameters cannot be predicted by an observer with the required accuracy. This unpredictable phenomenon can either be:
  - a microscopic property of a physical system that shows chaotic or quantum behavior, including thermodynamic systems. Examples are measurements of thermal noise, shot noise, metastability in bi-stable circuits or even radioactive decay[1]; or

1 Albeit radioactive decay is a good example of an unpredictable physical phenomenon with a proven physical theory behind it, the author is well aware that radioactive decay is highly impractical in normal computing environments. Therefore, it shall serve as an example for discussion only.

    - an unpredictable phenomenon triggered by the interaction between the computer hardware and its environment (for example, human interaction, or the receipt of interrupts triggered from external devices recording some externally triggered events would fall into this category).
- A recording logic is required that is capable of measuring the events generated by the unpredictable phenomenon. The recording logic does not necessarily need to store the measured data.
- Using the recorded events, the digitization logic turns the recorded data into a digital data stream which is then provided to either a post-processing conditioner or directly into a DRNG. The use of a DRNG at this stage is not intended to stretch the entropy over a large amount of output, but its purpose is the same as that of the conditioner discussed in the following. Commonly, only one of the mentioned mechanisms is used to post-process the data from a noise source. Albeit it may be possible to use the output of the digitization logic directly as input into cryptographic use cases, such a course of action is commonly disregarded. Conditioners or DRNGs will counter statistical anomalies in temporary or even permanent skews of recorded events. The conditioner as well as the DRNG perform an operation to transform the recorded data such that it is indistinguishable from an ideal random number generator where the operation does not reduce the collected entropy. The key value of those components is to increase the entropy per bit by performing a compression operation. The problem of a compression operation, however, is to find one that does not result in an entropy loss. In addition, the conditioner may be used to hide skews in the raw data by applying, for example, a Linear Feedback Shift Register (LFSR) or using cryptographic mechanisms like a message digest.
- For noise sources, it is commonly suggested – and it is required for noise sources to be accepted by BSI according to the requirements set forth in [AIS2031] – to employ some form of health check to guard against total breakdown of the event recording or the operation of the measured phenomenon. Naturally, the health check cannot detect changes in the entropy rate delivered by the recording logic, for example, due to aging or negative influences from the environment. However, small statistical tests tailored to the entropy source can detect non-tolerable defects in the stochastic behavior of the noise source in a reasonable time window. An example of such a test is the Chi-Squared test.
- Finally, the output of the noise source is fed into a cryptographically secure DRNG that uses cryptographic primitives to generate data indistinguishable from an ideal random number generator. The following variations may be visible in that last stage for different implementations:
    - The DRNG produces only data when an equal amount of true random data from the noise source is injected into the DRNG.
    - The DRNG generates output even when not reseeded by the noise source for a period of time. When sufficient entropy is collected by the noise source, the DRNG is reseeded again.

With the general architecture description in mind, the Linux-RNG design is described in the following chapter.

3 Design of the Linux-RNG

3.1 Historical Background

The initial implementation of the Linux-RNG was provided by Theodore Ts'o in 1994. The original design of the Linux-RNG is based on the US export restrictions on cryptography that were in place at that time.

Theodore Ts'o explained in a response ([T06]) to the work from Gutterman et al. ([GPR06]) that due to the US export restrictions enforced back then, the use of encryption mechanisms were discarded in favor of using the SHA-1 hash function (which is now replaced by a Blake2s message digest algorithm). Also, the Linux-RNG was constructed so that in case of a break of the collision resistance of SHA-1 the Linux-RNG would not be compromised.

With the introduction of the ChaCha20 block operation to generate random numbers in the Linux kernel version 4.8, a departure from the long-standing design concept of using SHA-1 is evident. Finally, SHA-1 is removed completely with version 5.17 which uses a more modern Blake2s message digest algorithm instead.

Starting with version 5.18, the entire post-processing of the entropy data has been completely re-implemented and all legacy concepts have been replaced. Any previous analyses on the deterministic processing are not applicable any more. Additional significant changes were incorporated with 5.18.1[2].

3.2 Linux-RNG Architecture

The Linux-RNG is a random number generator that uses hardware events detected by the Linux kernel as noise sources to feed a deterministic random number generator. A brief characterization of the operation of the Linux-RNG is provided in the following description.

The Linux-RNG uses one entropy pool, the input pool. Its purpose is to collect, and compress, and thus accumulate the entropy provided by the different noise sources. A ChaCha20 DRNG, the “base CRNG”, is seeded from the input pool. A set of secondary ChaCha20 DRNGs, one for each online CPU, are seeded from the base CRNG and provide the random bits for callers at the interfaces. User space interfaces of /dev/urandom, /dev/random, the system call `getrandom` and the in-kernel application programming interface (API) of `get_random_bytes` are available to pull random bits from the secondary DRNGs.

The term “entropy pool” in this document refers to Blake2s message digest instance as specified in [RFC7693]. The input function is a Blake2s hash update operation. The use of the hash update function mathematically implies that all data that is “inserted” into the entropy pool is concatenated into a long string which is then hashed. The output operation is a Blake2s hash final operation. Considering the state-transition and output function, the processing of the data maintained by an entropy pool is fully deterministic in nature.

The ChaCha20 DRNG as used in this document is conceptually similar to an entropy pool: a memory buffer of 32 bytes in size which is the ChaCha20 key used to initialize a transient ChaCha20 instance to generate random bits. The cryptographic function ChaCha20 is used as a state-transition function as well as an output function. Its implementation is derived from [RFC7539], sections 2.1 to 2.3, where the random number is the key stream generated by the ChaCha20 block operation.

The Linux-RNG operation can be characterized as follows. After the occurrence of a hardware event, such as an interrupt, the event is awarded an entropy estimation by the Linux-RNG. The event time and the event value are hashed into the entropy pool, the Blake2s state. This entropy pool is called input pool.

2 This is a stable kernel release. It is very unusual that non-bug fixes are added to stable releases. The entire new architecture is also backported to older “stable” kernels such as 5.15.44 even though the changes do not address security-critical issues.

Upon request, this entropy pool feeds the DRNG based on the ChaCha20 block operation.

Albeit multiple secondary ChaCha20 DRNGs are used, they all behave identically regarding how random bits are generated for a caller. Yet, there are differences in the behavior of the different interfaces accessing the ChaCha20 DRNG instances. The key difference is the timing when random numbers are generated in relationship to the then current entropy level of the Linux-RNG:

- Unrestricted generation of random numbers is available with /dev/urandom and the in-kernel function `get_random_bytes`. This implies that irrespective whether the entropy pool or the ChaCha20 DRNG received sufficient entropy, random data is generated.
- When accessing /dev/random, random numbers are only generated if the entropy pool or the ChaCha20 DRNG received at least 256 bits of initial entropy. After reaching that threshold of 256 bits of entropy once, /dev/random will operate non-blocking for the lifetime of the system and thus operate identically to /dev/urandom.

In addition, the Linux kernel offers the `getrandom` system call documented by its respective man page which provides access to the Linux-RNG as follows[3]:

- When invoking `getrandom` where the flag field is zero, the system call accesses the ChaCha20 DRNG identically to /dev/random.
- When invoking `getrandom` with a flag of `GRND_INSECURE`, the system call behaves like /dev/urandom.
- The flag `GRND_RANDOM` is currently unused.

When generating data from the input pool, a Blake2s hash final operation is performed. This implies that all data since the initialization of the pool is mathematically concatenated and then hashed. This resulting data is now processed by a construct which is similar to the [RFC5869] extract and expand mechanism, but with Blake2s instead of a HMAC-SHA2. In a first step, the extract phase, a new key is generated from the concatenated output from the input pool and the output of 256 bits from an optionally existing CPU-based random number generator like Intel's RDSEED. The resulting message digest is used to re-initialize the Blake2s hash context for gathering the next data from the noise sources – the Blake2s state forming the input pool is re-initialized with a state derived from the previous state. The result of the extract phase is also used to perform an expand phase to generate random bits 32 bytes in size which are used to seed the ChaCha20 DRNG key with.

The ChaCha20 DRNG operates by invoking the ChaCha20 DRNG block operation repeatedly until the requested number of bytes are generated. Hence, the output function of this DRNG is based on the ChaCha20 block operation. The application of the ChaCha20 block operation changes the state of the DRNG as defined for ChaCha20 in [RFC7539], section 2.4: the counter value of the state is incremented by one after each ChaCha20 block operation. To ensure backtracking resistance, the fast-key-erasure approach specified in [FKE] is applied.

3.2.1 Linux-RNG Internal Design

The Linux-RNG maintains one entropy pool and a set of ChaCha20 DRNGs to collect, compress and maintain entropy. Figure 2 depicts the relationship between the entropy pool, the ChaCha20 DRNGs and the entropy sources. The arrows in this figure explain the flow of information.

3 At the time of writing the man page does not fully contain all details about GRND_INSECURE and GRND_RANDOM flags as their meaning were changed with kernel 5.6. The explanation in this document is consistent with the Linux kernel source code.

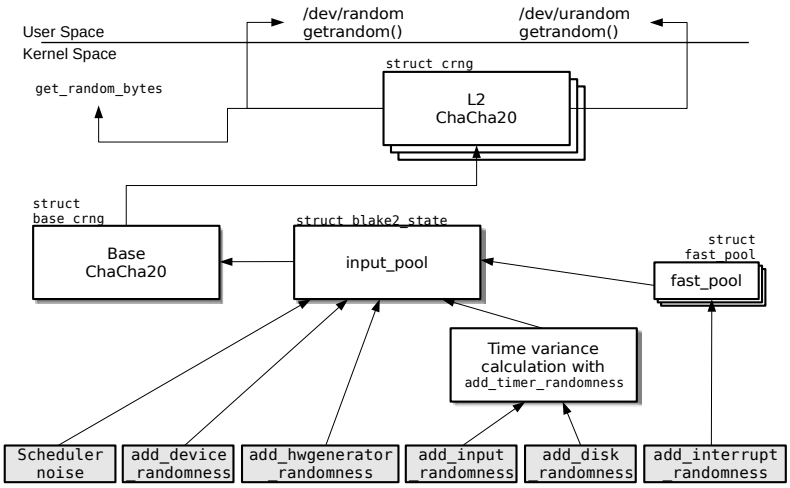

Figure 2: Relationship of entropy pool, ChaCha20-DRNG and entropy sources

The following relationships are evident in figure 2:

- The input_pool is the entropy pool that collects and compresses the entropy from hardware events. That entropy pool is the Blake2s internal state and has a size of 256 bits. The purpose of the input pool is to collect and compress entropy from the noise sources and provide it to the deterministic random number generator discussed in the following bullet point.
- The base ChaCha20 DRNG obtains its seed data from the input pool with the purpose to provide a seed source to the second-level ChaCha20 DRNGs.
- The Linux RNG instantiates a second level ChaCha20 DRNG one for each online CPU in the system to have a CPU-local ChaCha20 DRNG instance. The second-level ChaCha20 DRNGs are seeded from the base ChaCha20 DRNG. When a caller requests random bits, the ChaCha20 DRNG instance for the CPU the caller currently executes on is used. Those are accessible through the following interfaces:
  - from user space via /dev/random, /dev/urandom or the `getrandom` system call, and
  - from kernel space via the `get_random_bytes` function.

The ChaCha20 DRNG has an internal state of 512 bits. However, only 256 bits, the key part of the ChaCha20 state, are filled with random data. Further details about the maintenance of the ChaCha20 state are given in section 3.3.2.

The noise sources depicted by the gray boxes in figure 2 feed the input pool. According to this approach, the input pool collects the entropy from the noise source and compresses it.

The noise sources can be characterized as follows:

- Device drivers may provide data that the device driver author believes to contain some randomness via the `add_device_randomness` API. Discussions in later sections will explain that the Linux-RNG will use the data from this noise source, but treats it as having no entropy. Thus, the data is used to stir the internal state only.
- The Linux kernel implements device drivers for hardware random number generators. They may provide true random data via the `add_hwgenerator_randomness` API. Such hardware random number generators are available in specialized hardware only.
- HID such as keyboards or mice form the next noise source used by the Linux-RNG and may provide entropy via the `add_input_randomness` API. The data obtained by HID events such as a pressed key

or mouse movement is supplemented with a time stamp that the Linux-RNG obtains when an event arrives using the `add_timer_randomness` function.

- Hardware events pertaining to any kind of block devices such as hard disks are obtained by the Linux-RNG with the `add_disk_randomness` API forming another noise source. Events cover read and write operations of a hard disk. Similarly to the HID noise source, the Linux-RNG adds a time stamp to each disk event by invoking the `add_timer_randomness` function. More details are provided in section 3.5.2.3 about the collection of data from block devices. At this point, however, it shall be noted that not all block devices will contribute as a noise source. For example, solid-state-drives (SSD) are not used as noise sources whereas hard disks with spinning disks are used as such.
- When an interrupt arrives, the Linux-RNG is triggered with the `add_interrupt_randomness` API. For each received interrupt, the Linux-RNG obtains a time stamp and supplemental data which is fed into a fast_pool instance that is local to the CPU on which the interrupt is processed. The use of fast_pool instead of injecting the data directly into the input pool is required to maintain performance. Receiving and processing an interrupt is a very performance-critical code path. Normal work loads trigger hundreds to thousands of interrupts each second where a complex operation would simply decrease the system performance significantly. Throughout this document, the fast_pool is considered to belong to the interrupt noise source. The discussion of the fast_pool indicated in figure 2 will be given in section 3.5.2.2 as it is tightly integrated with the gathering of raw entropy from interrupts. Therefore, fast_pool is not considered as a stand-alone entropy pool or random number generator like the ones mentioned before.
- During boot time when a user space caller requests data from /dev/random or the `getrandom` system call and the Linux-RNG has not yet obtained 256 bits of entropy, the Linux-RNG tries to generate entropy from the interaction of a high-resolution timer and the Linux kernel scheduler. For this, the kernel first verifies whether it has a high-resolution time stamp. If so, it kicks off the entropy generation logic which runs in parallel with the remainder of the Linux-RNG operation. If the entropy generation fails, the entropy pool will not gain any additional data and the entropy estimator remains unchanged.

The input pool together with the noise sources form a NDRNG in its own right. The ChaCha20 DRNG is a separate random number generator in the Linux-RNG which is seeded by the input pool. The input pool will exclusively deliver data to the ChaCha20 DRNG which implies that a caller will never obtain data from the input pool directly.

3.3 Deterministic Random Number Generators (DRNGs)

The Linux-RNG entropy pool of the input pool can be considered as a DRNG when disregarding the noise sources. This section discusses the state maintenance of the deterministic operation of the entropy pool as well as the ChaCha20 DRNG.

3.3.1 Entropy Pool input pool

The random number generator implementation maintains a state memory block for the input pool which is technically equal to a Blake2s state h as defined by the Blake2s algorithm. The pool is governed by a structure which contains the following important information:

```
static struct {
          struct blake2s_state hash;
          spinlock_t lock;
          unsigned int init_bits;
} input_pool = {
```

```
.hash.h = { BLAKE2S_IV0 ^ (0x01010000 | BLAKE2S_HASH_SIZE),
            BLAKE2S_IV1, BLAKE2S_IV2, BLAKE2S_IV3, BLAKE2S_IV4,
            BLAKE2S_IV5, BLAKE2S_IV6, BLAKE2S_IV7 },
.hash.outlen = BLAKE2S_HASH_SIZE,
```

The member variables' relevance is discussed in the following bulleted list:

The member variable `hash` holds the Blake2s state. Besides containing the actual message digest state `h`, it contains some auxiliary variables used to implement the actual Blake2s algorithm. These auxiliary member variables are of no relevance to the operation of the Linux-RNG and thus are not further considered in the analysis. Yet, [RFC7693] outlines the auxiliary information required to implement a Blake2s message digest. The message digest state `h` is initialized at compile time with the initialization vectors as defined by [RFC7693] section 2.6 and the XORing of the 0x01010000 defining the first parameter block as defined in [RFC7693] section 2.5 for unkeyed hashing as well as section 3.2.

- The `lock` member variable is a spinlock which is used to serialize operations on the Blake2s state. These operations include adding of data, i.e. performing a hash update operation, or extracting data from the Blake2s state by performing a hash final operation followed by a hash init operation.
- The member variable `init_bits` is used to track the amount of entropy in bits inserted into the entropy pool. As outlined in section 3.3.1.2, this variable is only used during initialization until 256 bits of entropy initially have been collected. Afterwards entropy is not tracked any more.

3.3.1.1 Entropy Pool State Transition Function

The state transition function, i.e. the function used when new data is inserted into the entropy pool, is the Blake2s update function as outlined in [RFC7693] section 3.2 with the compression function F which also uses the mixing function G specified in [RFC7693] section 3.1 as part of its operation.

The state transition function can be characterized with the left part of figure 3. This illustration shows that the entropy pool maintenance mathematically calculates a Blake2s message digest of all data that is inserted into the entropy pool since its initialization.

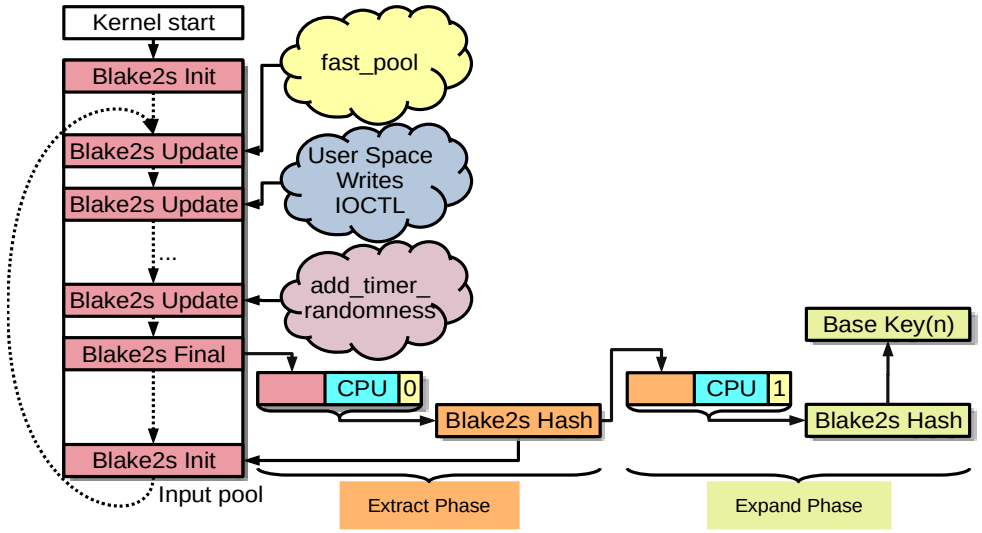

Figure 3: Entropy Pool Maintenance

Considering figure 3, the insertion of data into the pool implemented by the function `_mix_pool_bytes` is simply a Blake2s update operation:

```
static void _mix_pool_bytes(const void *buf, size_t len)
{
          blake2s_update(&input_pool.hash, buf, len);
}
```

The lower and right part of figure 3 illustrates the output function which is discussed in section 3.3.1.3.

3.3.1.2 Entropy Estimator

Before discussing the data generation, the aspect of entropy estimation must be discussed at this point. The Linux-RNG maintains a separate integer value, the entropy estimator. This integer value is intended to hold the amount of entropy present in the entropy pool during initialization time until the entropy pool is defined to be fully seeded with 256 bits of entropy.

The entropy estimator is an integer value which is incremented by the amount of bits of entropy the different noise sources report when they add entropy data. Yet, it is never decremented. Thus, its purpose is only to detect how much entropy was collected during boot time until reaching the threshold of 256 bits of entropy.

During incrementing the entropy estimator its value is checked and the following states are defined in the function `_credit_init_bits`:

- `CRNG_EMPTY`: The entropy pool received none or little entropy. All interfaces that may block as specified in section 3.4 are gated and continue to block. The base ChaCha20 DRNG is reseeded from the entropy pool during each request for random bits.
- `CRNG_EARLY`: The entropy pool collectively received 128 bits of entropy from all noise sources. All interfaces that may block as specified in section 3.4 are gated and continue to block. Yet, the constant reseeding of the base ChaCha20 DRNG is stopped.
- `CRNG_READY`: The entropy pool collectively received 256 bits of entropy from all noise sources. Upon reaching this threshold the base ChaCha20 DRNG is forced to be reseeded. Also all blocking interfaces are now released. Callers that registered to be informed when the Linux RNG becomes fully seeded are notified.

After reaching the `CRNG_READY` state, the entropy estimator is not used any more. Even if noise sources increment it, its value is never processed any more.

3.3.1.3 Entropy Pool Output Function

The extraction of random numbers from the entropy pool is implemented with the function `extract_entropy`. This function extracts data from the input pool.

The extraction operation is illustrated in figure 3 with the lower and right parts. Mathematically the output function follows the extract and expand function concept that is defined in [RFC5869] for HMAC-SHA-based keyed message digests. However, instead of using HMAC-SHA, Blake2s is used. As Blake2s is also a keyed message digest, it can follow the concept outlined in the mentioned RFC.

The following steps are performed as part of the output function:

1. The message digest of the entropy pool is calculated by invoking the Blake2s final operation.
2. 256 bits of random data is attempted to be collected from CPU-based entropy sources like RDSEED. On Intel systems, if RDSEED is not available, RDRAND is used. If no CPU-based entropy source is present the CPU's high-resolution timer is used to fill the buffer.

3. Depending on the operating system being 32 bit or 64 bit, a 32 bit or 64 bit integer operating as a counter set to 0 is initialized.
4. Now, the extract phase is performed by calculating a Blake2s keyed message digest using the data from step 1 as a key and the concatenated data from steps 2 and 3 as message data.
5. The message digest from step 4 is used to re-initialize the Blake2s state forming the entropy pool. The re-initialization uses the message digest as a key which implies that from this point on the entropy pool is managed as keyed message digest. This means that the first generation of the entropy pool immediately after power-on is an unkeyed version of Blake2s whereas all subsequent generations of the entropy pool operate as keyed Blake2s message digests.
6. The same message digest obtained in step 4 is also used to perform the expand phase. This expand phase is initializing a new Blake2s keyed message digest with the output from step 4 as key and the data obtained from step 2 and the counter from step 3 incremented by one as data.
7. The message digest generated in step 6 is now the output data. Although technically the expand phase is able to iterate step 6 as often as needed by incrementing the counter by one during each loop to generate arbitrary numbers of bits, all callers of the output function request exactly 256 bits of data, i.e. one Blake2s output block.

3.3.1.4 Initialization

When the Linux-RNG is initialized, the entropy pool and the ChaCha20 DRNG are initialized to prevent them from being empty. The initialization is performed during boot time of the kernel.

When the kernel initializes the driver for the random number generator, it calls the function `random_init`.

```
static void init_std_data(void)
{
          ktime_t now = ktime_get_real();
...
#if defined(LATENT_ENTROPY_PLUGIN)
          static const u8 compiletime_seed[BLAKE2S_BLOCK_SIZE]
__initconst __latent_entropy;
          _mix_pool_bytes(compiletime_seed, sizeof(compiletime_seed));
#endif

          for (i = 0, arch_bytes = BLAKE2S_BLOCK_SIZE;
               i < BLAKE2S_BLOCK_SIZE; i += sizeof(entropy)) {
                    if (!arch_get_random_seed_long_early(&entropy) &&
                        !arch_get_random_long_early(&entropy)) {
                              entropy = random_get_entropy();
                              arch_bytes -= sizeof(entropy);
                    }
                    _mix_pool_bytes(&entropy, sizeof(entropy));
```

```
        }
        _mix_pool_bytes(&now, sizeof(now));
        _mix_pool_bytes(utsname(), sizeof(*(utsname())));
        _mix_pool_bytes(command_line, strlen(command_line));
        add_latent_entropy();
}
```

The function `random_init` performs the following initialization steps.

As a first step, the function obtains the current time and mixes it into the entropy pool. The entropy pool is not empty, but contains the contents of the memory allocated for the pool. As the pool is statically allocated and the memory is occupied during early boot process, it is likely that it contains zeros. The resolution of that time value is discussed in the kernel code:

```
/*
 * ktime_t:
 *
 * On 64-bit CPUs a single 64-bit variable is used to store the hrtimers
 * internal representation of time values in scalar nanoseconds. The
 * design plays out best on 64-bit CPUs, where most conversions are
 * NOPs and most arithmetic ktime_t operations are plain arithmetic
 * operations.
 *
 * On 32-bit CPUs an optimized representation of the timespec structure
 * is used to avoid expensive conversions from and to timespecs. The
 * endian-aware order of the tv struct members is choosen to allow
 * mathematical operations on the tv64 member of the union too, which
 * for certain operations produces better code.
 *
 * For architectures with efficient support for 64/32-bit conversions the
 * plain scalar nanosecond based representation can be selected by the
 * config switch CONFIG_KTIME_SCALAR.
 */
```

In case a CPU random number generator is known to the Linux-RNG, data from that hardware RNG is mixed into the entropy pool in a second step.

In a next step, the initialization operation obtains the system-specific information and mixes the collected data into the entropy pool. The collected data contains the following information which is also explained in the man page uname(2):

- Operating system name (e.g. “Linux” – this is a compile time variable)
- Name within “some implementation-defined network” (such as the DNS hostname – at the time of initialization of the Linux-RNG, this variable is not set)

- Operating system release (e.g. 5.18.1 for the kernel version of 5.18.1 – this is a compile-time variable)
- Operating system version (this is a compile-time variable)
- Hardware identifier (such as “x86_64” – this is a compile time variable)
- Domainname when the operating system is part of a NIS or Yellow-Pages network infrastructure (at the time of initialization of the Linux-RNG, this variable is not set)

Subsequently, the kernel command line string is inserted into the entropy pool to stir it a bit more.

Finally the data from the latent entropy GCC plugin is inserted into the entropy pool. See section 3.5.2.9 for more details on this data.

3.3.2 ChaCha20 DRNG

The ChaCha20 DRNG is based on the identically named stream cipher developed by Daniel Bernstein [CHACHA20]. The ChaCha20 DRNG uses a data structure that complies with the definition of [RFC7539], section 2.3.

To maintain a ChaCha20 DRNG, the Linux RNG only stores the ChaCha20 key with the following data structure used for the base ChaCha20 DRNG:

```
static struct {
          u8 key[CHACHA_KEY_SIZE] __aligned(__alignof__(long));
          unsigned long birth;
          unsigned long generation;
          spinlock_t lock;
} base_crng
```

The member variables are used to store the following information:

- The `key` buffer of 256 bits holds the ChaCha20 key used to initialize a transient ChaCha20 block operation state.
- The `birth` variable contains the time when the ChaCha20 DRNG was seeded last. The ChaCha20 DRNG is automatically reseeded after 60 seconds irrespective of the amount of data produced by the DRNG.
- The `generation` variable is used to determine the reseed time of the secondary ChaCha20 DRNGs. Its purpose is discussed below. This variable is incremented by one every time the base ChaCha20 DRNG is (re)seeded.
- Finally, a spinlock is used with the `lock` variable to serialize the operations on this state.

The secondary ChaCha20 DRNGs are maintained with an even smaller state:

```
struct crng {
          u8 key[CHACHA_KEY_SIZE];
          unsigned long generation;
          local_lock_t lock;
};
```

The member variables are identical in their purpose as outlined for the base ChaCha20 DRNG state. The difference is that the last reseed time is not maintained. This is due to the following link concept between the base ChaCha20 DRNG and each secondary ChaCha20 DRNG. Every ChaCha20 DRNG seeds from the

base ChaCha20 DRNG which implies that 256 bits are generated by the base DRNG. After the seeding the secondary ChaCha20 DRNG operates independently. However, every time the transient ChaCha20 block operation instance is formed using the ChaCha20 DRNG key, the Linux RNG checks whether the `generation` of the current secondary ChaCha20 DRNG is equal to the `generation` of the base ChaCha20 DRNG. If not, the secondary ChaCha20 DRNGs are (re)seeded from the base ChaCha20 DRNG. This implies always when the base ChaCha20 DRNG is seeded, all secondary ChaCha20 DRNGs are reseeded at the time the next random bits are to be generated.

One important exception to the mentioned rule is present: until the base ChaCha20 DRNG is fully seeded, any request for random numbers is solely satisfied by the base ChaCha20 DRNG. At this time, the secondary ChaCha20 DRNG instances are not available.

Considering the state information above maintained for each DRNG, it is now required to explain how this state is used to generate random bits. The function _get_random_bytes defines a stack variable that holds the ChaCha20 block operation state as defined in [RFC7539] section 2.3. The state is filled as follows:

- using the constants as defined in section 2.3 [RFC7539],
- using the value of the aforementioned ChaCha20 DRNG state variable `key`, and
- initializing the counter as well as the nonce values to zero.

Using this transient ChaCha20 block operation state, the ChaCha20 block operation as defined in section 2.3.1 [RFC7539] is invoked as often as needed to generate the requested amount of output bits.

The relationship of the base ChaCha20 DRNG with one instance of the secondary DRNGs as well as how random bits are produced is outlined with figure 4.

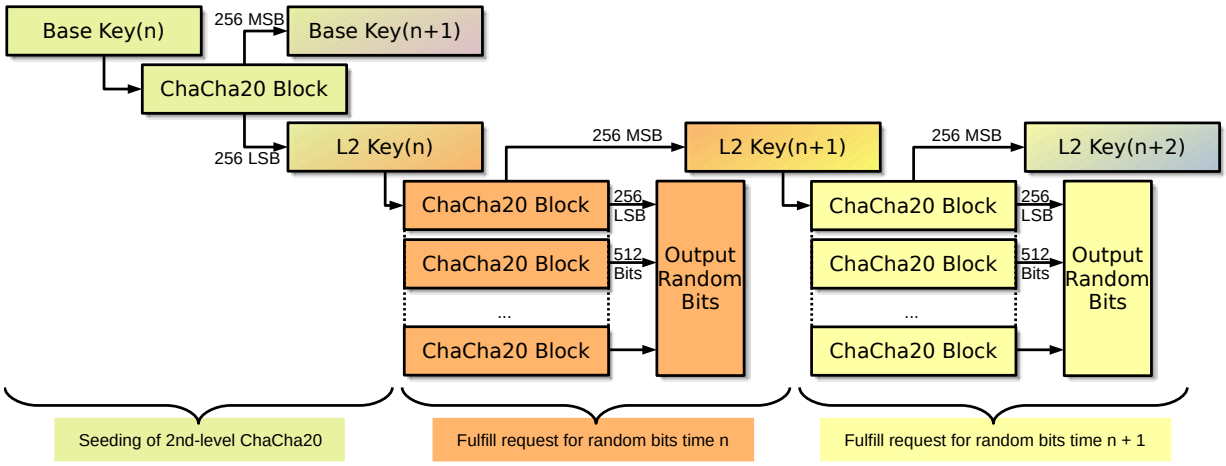

Figure 4: ChaCha20 DRNG Operation and Base DRNG to Secondary DRNG Relationship

The illustration in figure 4 shows three phases:

1. The light green left-most part depicts the seeding of the secondary ChaCha20. This seeding operation instantiates a ChaCha20 block operation from the base ChaCha20 DRNG key via the fast-key-erasure approach outlined in section 3.3.2.2.
2. The orange middle part illustrates the instantiation of a ChaCha20 block operation from a secondary ChaCha20 DRNG key via the fast-key-erasure approach to fulfill a request of random bits. Each output from a ChaCha20 block round is concatenated. The ChaCha20 block operation is repeated as often as needed to produce the requested amount of bits.
3. The light yellow right-most part illustrates a second request for random bits fulfilled by the secondary ChaCha20 DRNG. This request is handled identically to step 2.

3.3.2.1 (Re)Seeding of ChaCha20 DRNG

Irrespective of an initial seeding or reseeding the same operation is performed as follows:

- Base ChaCha20 DRNG: 256 bits are obtained from the entropy pool using the operation outlined in section 3.3.1.3. This value is used to overwrite the key value in the `base_crng` data structure.
- Secondary ChaCha20 DRNG: 256 bits are obtained using the fast-key-erasure instantiation of the base ChaCha20 DRNG which is used to overwrite the key value in the respective `crng` data structure instance of the secondary DRNG to be reseeded.

The question arises whether overwriting the old key value is appropriate. The entropy pool management ensures a link of the different generations of the entropy pool by generating a Blake2s key from the previous entropy pool state to initialize the new entropy pool state. Thus the seed data for the base ChaCha20 DRNG contains also the information of historic events from the entropy pool output function. Similarly, when seeding a secondary ChaCha20 DRNG, the implicit event history present in the base ChaCha20 DRNG based on its seed is inserted into the secondary DRNG.

Therefore, the overwriting of the key is not considered to destroy historic information which may be used to smooth over temporary deficiencies in the noise sources.

The function `crng_has_old_seed` forces the reseed of the base ChaCha20 DRNG. The reseeding of the base ChaCha20 DRNG is forced at runtime during the first request for random numbers after the elapse of 60 seconds since the last reseed. During the first 2 minutes after the kernel was booted, the reseeding is performed much more frequently. The check for seeding is again performed at the time random bits are to be generated. The reseeding is performed proportionally to the numbers of seconds since boot: every (n + n/2) seconds after reboot the reseeding is triggered where n is the number of seconds since boot.

3.3.2.2 Fast-Key-Erasure ChaCha20 DRNG

The fast-key-erasure approach implemented by the Linux RNG is based on the approach outlined in [FKE]. The mechanism is applied when a ChaCha20 block operation instance shall be created from a given ChaCha20 DRNG key:

1. The ChaCha20 block operation instance is initialized using the key of the ChaCha20 DRNG state.
2. Invoke the ChaCha20 block function to generate a 512 bit output block from that state.
3. The 256 most significant bits (MSB) of the block obtained in step 2 are used to replace the key in the used ChaCha20 DRNG state. This new key is used for a next DRNG generate request. This also implies that once the transient ChaCha20 block operation state is erased, backtracking resistance is guaranteed as it is impossible for an attacker to deduce the generated bits from the existing memory content.
4. The 256 least significant bits (LSB) are returned to the caller. These bits are either used to seed the secondary ChaCha20 DRNG in case the current DRNG is the base ChaCha20 DRNG, or these are already the first 256 output bits that can be returned to a caller in case of a secondary ChaCha20 DRNG.

The fast-key-erasure approach is illustrated in figure 4 with the colors of the newly generated keys for n+1 or n+2. The L2 Key(n) in figure 4 has the color green to orange illustrating that it was generated during the green stage of seeding the secondary DRNG but is used to instantiate the orange ChaCha20 block operation instance used to fulfill the first request for random bits.

3.3.2.3 Generation of Random Numbers

When random numbers are to be generated, the following function is used:

```
static void _get_random_bytes(void *buf, size_t len)
```

```
{
        u32 chacha_state[CHACHA_STATE_WORDS];
        u8 tmp[CHACHA_BLOCK_SIZE];
        size_t first_block_len;

        if (!len)
                return;

        first_block_len = min_t(size_t, 32, len);
        crng_make_state(chacha_state, buf, first_block_len);
        len -= first_block_len;
        buf += first_block_len;

        while (len) {
                if (len < CHACHA_BLOCK_SIZE) {
                        chacha20_block(chacha_state, tmp);
                        memcpy(buf, tmp, len);
                        memzero_explicit(tmp, sizeof(tmp));
                        break;
                }

                chacha20_block(chacha_state, buf);
                if (unlikely(chacha_state[12] == 0))
                        ++chacha_state[13];
                len -= CHACHA_BLOCK_SIZE;
                buf += CHACHA_BLOCK_SIZE;
        }

        memzero_explicit(chacha_state, sizeof(chacha_state));
}
```

The function performs the following steps:

1. Perform the fast-key-erasure operation to instantiate a ChaCha20 block operation algorithm, generate a new key for the next ChaCha20 block operation instantiation and return 256 bits of random data. Details are documented in section 3.3.2.2.
2. Return the random data that are obtained during the fast-key-erasure operation in step 1.

3. If the amount of random data is not yet sufficient to cover the caller's request a loop is started to invoke the ChaCha20 block generation operation repeatedly with the ChaCha20 block operation instance to generate as many random bits as required.
4. Zeroize the ChaCha20 block operation state.

This function is the service function to handle in-kernel users. The interface function `get_random_bytes_user` performs the exact same operation with the addition that the generated random bits are copied to the user space buffer pointed to by the user space caller.

3.4 Interfaces to Linux-RNG

3.4.1 Character Device Files

The devices /dev/random and /dev/urandom are registered by providing file operations data structures linking the system call operations with the service functions. The data structures contain pointers to the respective call-back functions implemented by the Linux-RNG which are made known to the system call handler functions. Both devices are linked with the kernel-internal functions handling read, write and other types of requests on these character device files with the following code:

```
static const struct memdev {
        const char *name;
        mode_t mode;
        const struct file_operations *fops;
        struct backing_dev_info *dev_info;
} devlist[] = {
...
         [8] = { "random", 0666, &random_fops, NULL },
         [9] = { "urandom", 0666, &urandom_fops, NULL },
...
};
```

The code shows that for the /dev/random device file, a function pointer data structure `random_fops` is registered. This function pointer data structure contains the handler functions implementing the kernel-side read and write operations that are triggered when user space performs a read or write on /dev/random. The device file of /dev/random is defined to be created with world-read/writable Unix permission bits. The same is done for the /dev/urandom device where the function pointer data structure of `urandom_fops` is registered.

The devices stored in `devlist` are all registered during kernel boot with the `chr_dev_init` function.

The callback functions registered for /dev/random are:

```
const struct file_operations random_fops = {
        .read  = random_read_iter,
        .write = random_write_iter,
        .poll  = random_poll,
```

```
        .unlocked_ioctl = random_ioctl,
...
        .fasync = random_fasync,
        .llseek = noop_llseek,
...
};
```

Similarly, the callback functions for /dev/urandom are:

```
const struct file_operations urandom_fops = {
        .read  = urandom_read_iter,
        .write = random_write_iter,
        .unlocked_ioctl = random_ioctl,
...
        .fasync = random_fasync,
        .llseek = noop_llseek,
...
};
```

These functions referenced in the `random_fops` and `urandom_fops` are all implemented as part of the Linux-RNG and are discussed in the following subsections.

3.4.1.1 random_poll

The `random_poll` function registered in the function pointer data structures is invoked when user space uses the `poll` system call with /dev/random.

The poll system call implementation allows processes to be triggered on two occasions, depending on the `poll` system call request type invoked by user space as follows:

- read: When sufficient entropy is available indicated by the fact that the ChaCha20 DRNG is fully seeded, the kernel wakes up the polling processes to allow them obtaining data with a separate call. This allows user space to asynchronously poll /dev/random to avoid the blocking behavior when reading /dev/random in case the entropy is not fully seeded with 256 bits. After the ChaCha20 DRNG is fully seeded, the read poll will always return successfully until the next reboot. The read-like poll is applied when the caller uses the POLLIN option as discussed in the `poll` man page. This operation is also used when user space uses the select system call to sleep until the fully seeded event occurs.
- write: When insufficient entropy is available during initialization time until reaching 256 bits of entropy, the kernel wakes up the waiting processes with a write poll. Any subsequent write poll operations after 256 bits of initial entropy have been received will never wake up the caller. This implies that the Linux-RNG interface of /proc/sys/kernel/random/write_wakeup_threshold is available, but not enforced any more. The write-like `poll` is applied when the caller uses the POLLOUT option as discussed in the `poll` man page.

3.4.1.2 Read and Write Operation

For entropy extraction via the device files, the kernel implements the following methods. These methods are referenced by the aforementioned function pointer data structures.

/dev/random: When accessing the random number generator using this device file, the read function of `random_read_iter` is called. This function blocks the caller until the ChaCha20 DRNG is fully seeded. Once that happened, random numbers are generated identically to /dev/urandom outlined below.

/dev/urandom: When random data is extracted via /dev/urandom, the ChaCha20 output function `get_random_bytes_user` discussed in section 3.3.2.3 is invoked.

The write service function is identical for both devices. The random number device driver allows writing of data into the /dev/random and /dev/urandom device files. Both devices use the same function to implement the write method: `random_write_iter`. `random_write_iter` calls the `write_pool_user` service function which mixes the data provided by user space with the input pool. The entropy estimator is not changed when mixing data into the entropy pool using the write operation.

```
static ssize_t write_pool_user(struct iov_iter *iter)
{
          u8 block[BLAKE2S_BLOCK_SIZE];
...
          for (;;) {
                    copied = copy_from_iter(block, sizeof(block), iter);
                    ret += copied;
                    mix_pool_bytes(block, copied);
...
          memzero_explicit(block, sizeof(block));
...
```

The code listing shows that the user space data is mixed into the pool in 32 byte chunks using the functionality outlined in section 3.3.1.1.

3.4.1.3 Input/Output Controls (IOCTLs) Usable With Device Files

Both device files implement the following IOCTL commands which are usable with the `ioctl` system call:

- `RNDGETENTCNT`: Extraction of the entropy estimator value for the input pool. This IOCTL is identical to the contents of /proc/sys/kernel/random/entropy_avail.
- `RNDADDTOENTCNT`: Add a user space supplied integer value to the entropy estimator for the input pool discussed in section 3.3.1.2. This IOCTL is restricted to the capability of `CAP_SYS_ADMIN`, which is only given to administrative processes.
- `RNDADDENTROPY`: Mix in random user space supplied data into the input pool using the same logic as outlined in section 3.4.1.2. In addition, add a user space supplied integer value to the entropy estimator for the input pool in section 3.3.1.2. This IOCTL is restricted to the capability of `CAP_SYS_ADMIN`.
- `RNDZAPENTCNT`: This IOCTL has no effect any more.
- `RNDCLEARPOOL`: This IOCTL has no effect any more.

- RNDRESEEDCRNG: If the caller possesses the capability of CAP_SYS_ADMIN, the base ChaCha20 DRNG is reseeded. In addition, all secondary ChaCha20 DRNG instances will be reseeded the next time they are used as forced by the management of the DRNG generation outlined in section 3.3.2.

3.4.2 System Call

In addition to the character device files of /dev/random and /dev/urandom, the Linux-RNG offers the getrandom system call to user space for obtaining random data. This system call uses three parameters: the first and second parameter allow the caller to supply the buffer pointer and the size of the buffer that shall receive the random data. The third parameter flags allow the caller to define:

- GRND_RANDOM – This flag is currently unused.
- GRND_NONBLOCK – By default getrandom blocks if the ChaCha20 DRNG is not fully seeded. If the GRND_NONBLOCK flag is set, then getrandom does not block in these cases, but instead immediately returns -1 with errno set to EAGAIN. In case of this error code, no random bits are returned.
- GRND_INSECURE – When using this flag, the getrandom system call behaves like /dev/urandom by delivering data irrespective whether the ChaCha20 DRNG is fully seeded.

When invoking the getrandom system call without any flags, it behaves identically to /dev/random: it blocks the caller and does not return random data until the ChaCha20 DRNG is considered to be fully seeded. After reaching this state, getrandom will not block any more and generate unlimited amounts of random data.

The advantage of using the getrandom system call over accessing the character device files is the exclusion of the Linux kernel virtual file system (VFS) layer. That layer adds huge complexity which may be the cause of errors returned to users. These errors may have no relationship to the Linux-RNG operation. Thus, the system call allows bypassing the VFS which is not of relevance to the Linux-RNG.

3.4.3 In-Kernel Interfaces

To supply in-kernel consumers such as the kernel crypto API or the networking stack with entropy, the Linux-RNG offers the function get_random_bytes. This function behaves exactly like /dev/urandom for user space as it delivers the requested amount of random data irrespective of the seed status of the entropy pool. The function get_random_bytes requires the following arguments: a pointer to the buffer and the size of the buffer to be filled with random data. The call to this function will always succeed.

In addition, functions filling an unsigned int variable, i.e. a variable with 4 bytes, and an unsigned long long variable, i.e. an 8-byte variable, with random bytes efficiently is provided with the API calls of get_random_u32 and get_random_u64, respectively. The kernel maintains one memory block of 1.5 times the block size of ChaCha20 (768 bits) on each CPU – the reason for this value is that 256 bits from the fast-key-erasure operation specified in section 3.3.2.2 plus one full ChaCha20 block can be maintained. The CPU-local buffer allows a lock-less access of the memory. When using these APIs, the ChaCha20 DRNG is used to fill the respective CPU-local buffer. After filling the CPU-local buffer, the needed 4 or 8 random bytes are copied from that buffer to the caller. The kernel remembers which bytes of the CPU-local buffer have already been given to callers. In a next call of the API, the kernel returns the next unused 4 or 8 bytes to the caller. This is continued until all random data in the CPU-local buffer is used which will trigger a new invocation to the ChaCha20 DRNG to overwrite the respective CPU-local buffer.

During boot time, a number of in-kernel callers request random numbers from the Linux-RNG. The author of this study performed some measurements on how many bytes are requested by in-kernel users during boot time before even user space is booted. Depending on the kernel functions and hardware support present in the underlying system, the number of bytes can be up to 1,000 bytes. Considering the work of this

study, quantitative testing shows that the Linux-RNG will not be seeded with sufficient data at that point, which implies that these callers receive random data with questionable entropy. Luckily, none of the callers that were identified use the random numbers for cryptographic purposes. Often, these random numbers are used to initialize hash maps, universally unique identifies (UUIDs), initial values for networking-related operations and similar items.

Although the kernel does not offer an equivalent to /dev/random inside the kernel, it offers an interface that is conceptually similar to the `getrandom` system call where the caller only receives random data after the primary ChaCha20 DRNG has been fully seeded. The difference, however, is that an in-kernel caller may not be blocked like user space processes. The concept rests on the function `register_random_ready_notifier` offered by the Linux-RNG to in-kernel users. This function allows callers to register a callback function that is invoked when the primary ChaCha20 reaches the fully seeded state.

In addition, the kernel offers the service functions of `get_random_XXX_wait`, where XXX refers to either u32, u64, int or long to generate random numbers and put it into the provided buffers with the respective data type. The waiting operation is identical to the waiting operation defined for /dev/random. However, at the time of writing these functions are not used in the kernel code base.

The reader should note that using `get_random_bytes` without any precautions does not guarantee that sufficient entropy has been collected to generate cryptographically strong random numbers.

The behavior of `register_random_ready_notifier` is asynchronous in nature. A synchronous waiting until the ChaCha20 is initially seeded is provided with the API call of `wait_for_random_bytes`. This function will put the caller to sleep as long as the ChaCha20 DRNG is not initially seeded. Once the initially seeding threshold is reached, the caller is woken up. At that point, the caller can now invoke the `get_random_bytes` API call to obtain random data from the initially seeded ChaCha20 DRNG. When this function is called, the scheduler-based entropy generation is started.

3.4.4 /proc Files

The following /proc files are provided by the Linux-RNG to allow all users to read status information and to allow administrators to alter the behavior of the Linux-RNG. More information can be obtained with the random man page.

- /proc/sys/kernel/random/poolsize: size of the input pool in bits, i.e. 256 bits.
- /proc/sys/kernel/random/entropy_avail: current state of the entropy estimator of the input pool. The entropy estimator is adjusted to show the entropy content in bits.
- /proc/sys/kernel/random/write_wakeup_threshold: This interface returns the number of bits when the Linux-RNG is considered to be fully seeded, i.e. 256 bits. It can be modified from user space, but this modification does not alter the Linux-RNG behavior.
- /proc/sys/kernel/random/boot_id: UUID generated during boot.
- /proc/sys/kernel/random/uuid: UUID that is re-generated during each request.
- /proc/sys/kernel/random/urandom_min_reseed_secs: This interface returns the number of seconds when the Linux-RNG is reseeded during runtime, i.e. 60 seconds. It can be modified by user space, but this modification does not alter the Linux-RNG behavior.

Most of the proc files are read-only: the file permission settings do not allow a write operation and the kernel does not implement a write-handler. The files containing the threshold values are writable by the root-user only.

3.5 Entropy Sources

The purpose of the Linux random number generator is to:

- collect entropy from various sources,
- mix gathered input values into the input pool, and
- estimate the obtained entropy during boot time.

The following sections discuss these aspects.

Data that is believed to contain entropy and contribute to the entropy collection of the Linux-RNG is specifically marked such that the reader can immediately identify such data.

3.5.1 Timer State Maintenance for Entropy Sources

Each hardware entropy source maintains a timer state. That state is used to store the time deltas as well as the time of the last hardware event occurrence.

The timer state keeps the following information:

```
struct timer_rand_state {
        cycles_t last_time;
        long last_delta, last_delta2;
};
```

The variables `last_time`, `last_delta` and `last_delta2` are used for the entropy calculation to support the calculation of time deltas discussed in section 3.6.

According to figure 2 the kernel maintains several classes of entropy sources. For three of these classes, the kernel instantiates a timer state data structure.

HID, i.e. devices that are defined as being the ”console” of the system: the kernel maintains one instance of the timer state data structure for the collection of all HID devices. Therefore, the time deltas of events of all HID devices are stored together which implies that the collection of all HID devices are used as one entropy source. The instance is defined in the code with the static variable:

```
static struct timer_rand_state input_timer_state = { INITIAL_JIFFIES };
```

Disk devices: the kernel maintains one time state data structure instance per physical disk. Therefore, the time deltas of events triggered by one disk are maintained separately. This implies that one physical disk represents one independent entropy source[4]. The following code listing shows how the timer state data structure is instantiated per disk by allocating the required amount of memory and registering it with the per-disk data structure maintained by the kernel for each disk instance:

```
void __cold rand_initialize_disk(struct gendisk *disk)
{
        struct timer_rand_state *state;

        /*
         * If kzalloc returns null, we just won't use that entropy
```

4 The allocation of the time state data structure is performed irrespective of whether a block device is considered to contribute entropy or not as discussed in section 3.5.2.3.

```
         * source.
         */
        state = kzalloc(sizeof(struct timer_rand_state), GFP_KERNEL);
        if (state) {
                state->last_time = INITIAL_JIFFIES;
                disk->random = state;
        }
}
```

Interrupts: the kernel sets up one fast_pool instance per CPU accessible in a per-CPU variable `irq_randomness`. The idea is that any operation on the fast_pool instance can be performed without holding a lock. The fast pool is defined as follows:

```
struct fast_pool {
          struct work_struct mix;
          unsigned long pool[4];
          unsigned long last;
          unsigned int count;
};


static DEFINE_PER_CPU(struct fast_pool, irq_randomness) = {
#ifdef CONFIG_64BIT
#define FASTMIX_PERM SIPHASH_PERMUTATION
          .pool = { SIPHASH_CONST_0, SIPHASH_CONST_1, SIPHASH_CONST_2,
SIPHASH_CONST_3 }
#else
#define FASTMIX_PERM HSIPHASH_PERMUTATION
          .pool = { HSIPHASH_CONST_0, HSIPHASH_CONST_1, HSIPHASH_CONST_2,
HSIPHASH_CONST_3 }
#endif
};
```

The `pool` array holds the actual entropy data and constitutes the pool. The `last` variable holds the time when the fast_pool was read last to inject its data into the input pool. The content of the fast_pool is not transferred to the input pool if the last operation is less than one second ago. The `count` variable counts the number of interrupts processed by this fast_pool to ensure that when receiving at least 64 interrupts since the last transfer the fast_pool can be transferred to the input pool. The macro `DEFINE_PER_CPU` implies that an instance of the fast_pool data structure is allocated for each CPU.

The initialization already indicates how the fast pool is maintained: it is processed as a SipHash as defined in [SIPHASH]. Yet, on 32 bit systems, the fast pool state is defined to consist of four 32 bit words whereas on 64 bit systems, the fast pool consists of four 64 bit words. This implies that on 32 bit systems at most 128 bits of data can be processed.

3.5.2 Entropy Collection

The random number generator exports service functions which are placed at well-defined locations in the kernel code to obtain hardware-related events. These events and the time stamp when these events occur are used to stir the input pool and to potentially increase its entropy estimator.

As depicted in figure 2, various entropy collection functions are defined for different classes of hardware events. The following sections discuss the individual entropy collection functions.

The specially marked values identified in the subsequent subsections identify the raw entropy which is added to the input pool. The term raw entropy references the entropy content of events. Per definition, entropy cannot be measured, yet the Linux-RNG wants to quantify the amount of entropy that it receives from its noise sources. The quantification of entropy can only be performed using a heuristic approach which attributes an entropy estimate to the data received from the noise sources. The quality of this raw entropy relative to the heuristically assumed entropy for each event defines the strength of this RNG. When the heuristic entropy value is smaller than the raw entropy, the available entropy is underestimated, i.e. the Linux-RNG would be considered conservative and thus would certainly have the cryptographic strength identified by the entropy estimator. On the other hand, if the heuristic entropy value is larger than the raw entropy, the Linux-RNG would overestimate the available entropy. In this case, the random numbers produced by the Linux-RNG would not be as cryptographically strong as indicated by the entropy estimator.

3.5.2.1 add_input_randomness

The input layer of the kernel that handles all input devices like keyboards or mice, calls this service function every time an input event is handled by the kernel. Such events are key presses, mouse movements, mouse button presses and similar events. To ensure that auto-repeat events are detected and properly discarded, the service function of `add_input_randomness` only stirs the random pool if the event value is different from the previous value[5].

Every event has a value that is processed with `add_input_randomness`. For example, the key strokes from a keyboard are associated with a key code. When a mouse is moved, the dimensions such as left or right, forward or backward of the mouse is recorded.

The function `add_input_randomness` compares the event value of the current event with the one of the previous event. If both event values are identical (for example, a mouse is moved in one direction by two steps or the same key is pressed twice) the event is discarded. Otherwise, the event is added to the input pool. The following code shows the important steps:

```
void add_input_randomness(unsigned int type, unsigned int code,
                                  unsigned int value)
{
        static unsigned char last_value;

        /* ignore autorepeat and the like */
        if (value == last_value)
                return;
...
```

5 Each event is assigned a value, such as the key code of the keyboard key that was pressed. Therefore, if repeatedly the same key is pressed, the service function would obtain the same key value and therefore discard this value.

```
        last_value = value;
        add_timer_randomness(&input_timer_state,
                             (type << 4) ^ code ^ (code >> 4) ^ value);
```

The listed code does not contain any locks which protect the comparison with the previous value against simultaneous events on other CPUs. This is considered acceptable because in the worst-case one event that should be discarded is not. When events occur simultaneously it is not really possible to state in which order these events are to be processed. Therefore, a missing lock is uncritical.

The `add_input_randomness` function uses the following values as event value that will eventually be mixed into the input pool:

low 4 bits of the event type ⊕ event code ⊕ high 4 bits of the event code ⊕ event value

The interpretation of the event type, event code and event value varies greatly, depending on the type of HID. As the quantitative analysis will show, the HID event information contains very little entropy. Therefore, further explanation of the kind of data related to the event information is not considered relevant.

The time variances used to mix the random values into the input pool compare all HID which means that one global `input_timer_state` static variable is used as discussed in section 3.5.1. This means that one time state variable is maintained for all input device events.

The event value is statistically analyzed in section 6.1.3.

3.5.2.2 add_interrupt_randomness

As the name of the service function already suggests, interrupts are used as a source of entropy. This service function is placed inside the standard Linux interrupt handler and invoked every time an interrupt is received by the kernel. In addition, this function is called inside the VMBus interrupt handler, because when Linux executes as guest on Microsoft Hyper-V, all interrupts from the hypervisor are exclusively processed by the VMBus interrupt handler.

Before discussing the code and data structures involved in the gathering of interrupts to be mixed into the entropy pool, the concept of the handling of interrupts must be clarified. After the discussion of the concept, the code analysis is presented.

Considering figure 2, the interrupts are not directly fed into the input pool but rather into a “baby entropy pool” called fast_pool. This fast_pool is four C-language long words which makes it 128 bits in size on 32 bit system and 256 bits in size on 64 bit systems.

One instance of the fast_pool is created per CPU. Depending on which CPU an interrupt is received, the fast_pool of that CPU is used. The instruction pointer as well as the time stamp of each interrupt are mixed into the respective fast_pool. The entire content of that fast_pool is mixed into the input pool after the either of the following requirements is met:

- the count variable of the current fast_pool is at least 64, or
- the last mix-in of that fast_pool was more than a second ago, which is tracked with the last variable of the fast_pool data structure.

This implies that the function `add_interrupt_randomness` does not use the function `add_timer_randomness` to add time stamps as used by other entropy collection functions.

The fast_pool is managed as a SipHash 1-0 / HalfSipHash 1-0 using an empty key, i.e. the internal state of the SipHash is initialized only with its constants as defined in the specification which mathematically implies that the 64 bit (32 bit architectures) or 128 bit (64 bit architectures) key is zero. The authors of SipHash describe in chapter 3 [SIPHASH] the strength of the algorithm and outline that SipHash has an appropriate strength for the 2-4 construction. Smaller constructions are not awarded a security strength.

The SipHash construction follows the specification in figure 2.1 [SIPHASH] to the extend that:

- Figure 2.1 m_0 is the high-resolution time stamp.
- Figure 2.1 m_1 is the instruction pointer.
- One SipRound is performed after the insertion step of m_0.
- One SipRound is performed after the insertion of m_0 and m_1.
- The final insertion of m_1 is not followed by any subsequent operation.

This implies that the Linux-RNG does not insert the 0xff byte into the SipHash state as mandated by [SIPHASH].

The first step is the fetching of the fast_pool for the CPU processing the interrupt. When looking at the file /proc/interrupts, the CPU executing the interrupt handler of a specific interrupt number is presented. In most cases, CPU0 is used to serve the interrupt which is the first CPU.

After obtaining the reference to the fast_pool, the interrupt event data is added to the content of the fast_pool as discussed above. The addition of the data to the fast_pool is followed by a stirring of the fast_pool using the `fast_mix` function.

If both conditions listed above about the number of interrupts and the expired time since last read-out are met, the current fast_pool content is injected into the input pool.

```
static void mix_interrupt_randomness(struct work_struct *work)
{
...
     fast_pool->count = 0;
     fast_pool->last = jiffies;
...
     mix_pool_bytes(pool, sizeof(pool));
     credit_init_bits(max(1u, (count & U16_MAX) / 64));
...
void add_interrupt_randomness(int irq)
{
...
     if (new_count < 64 && !time_is_before_jiffies(fast_pool->last + HZ))
          return;
...
     queue_work_on(raw_smp_processor_id(), system_highpri_wq, &fast_pool-
>mix);
```

The code snippet shows:

1 If either condition, the number of interrupts and the expired time is reached, the data from the fast pool is inserted into the input pool.

2 The time stamp of the last read-out of the fast_pool is set to the current time.

3 The content of the fast_pool is mixed into the input pool. This mixing operation is performed in a separate thread and therefore, performed asynchronously with the processing of interrupts. The Linux-RNG contains a precaution that only one asynchronous invocation is active at one given time for the respective fast_pool.

4 Reset the number of received interrupts to zero for the initial condition check in step 1.

5 Increase the entropy estimator of the input pool by 1. Note, the calculation of `(count & U16_MAX) / 64` relates to the `add_timer_randomness` handling: in case the `add_timer_randomness` was invoked during an interrupt, instead of adding the entropy directly to the entropy estimator, it is added to the fast_pool of the current CPU. In this case, the entropy estimate for the fast pool is larger than 1. The concept is that the entropy estimated by `add_timer_randomness` is inserted into the entropy pool via the fast_pool calculation. Mathematically this does not change the entropy estimation approach: data handled with the `add_timer_randomness` are estimated. Thus, this special case is not further considered during the assessment. Yet, this change now implies that there is no double-accounting of entropy for one event any more as it used to be the case in the Linux-RNG before.

With these steps, it is evident that

all four data words of the fast_pool and the fast_pool meta data

are used to update the input pool.

That means that between 1 and 64 interrupts are assumed to represent one bit of entropy. This implies that in the one extreme case the Linux-RNG awards each interrupt 1 bit of entropy in case the interrupt is not related to HID devices. For HID devices, in the extreme case, the Linux-RNG awards an interrupt up to 11 bits of entropy. The other extreme is that the Linux-RNG awards 1/64th bit of entropy to each interrupt.

The event value is statistically analyzed in section 6.1.1.

3.5.2.3 add_disk_randomness

The last entropy gathering service function which adds data into the input pool and can increase its entropy estimator is `add_disk_randomness`, which is called by the scsi_lib subsystem, function `scsi_end_request`, which handles the accesses to ATA, SATA and SCSI mass storage devices attached to the system.

When a disk event occurs, the device number forming the major and minor device number plus `0x100` is used to add entropy to the input pool:

```
void add_disk_randomness(struct gendisk *disk)
{
        if (!disk || !disk->random)
                return;
...
        add_timer_randomness(disk->random, 0x100 + disk_devt(disk));
}
```

The function `disk_devt(disk)` obtains the member variable `device->devt` from the device data structure registered with the disk device structure which holds the device definition of the disk device.

In addition to the device number, the timer state variable `disk->random` is used to add entropy to the pool. The kernel maintains one timer state variable per disk device.

The timer state variable for a disk device is initialized with `rand_initialize_disk` that allocates zeroized memory and registers it with `disk->random`. This service function is called unconditionally when a new disk device is allocated by the block device layer.

```
struct gendisk *alloc_disk_node(int minors, int node_id)
{
...
                rand_initialize_disk(disk);
...
```

The function `add_disk_randomness` is only invoked if the following constraint is met for the given block device.

```
static bool scsi_end_request(struct request *req, blk_status_t error,
                unsigned int bytes)
{
...
        if (blk_queue_add_random(rq->q))
                add_disk_randomness(rq->rq_disk);
...
```

The code shows that only if the wait queue of the respective block device holds the flag `QUEUE_FLAG_ADD_RANDOM` (which is obtained with the `blk_queue_add_random` macro), the handler function of the random number generator is triggered. Per default, that flag is set for each block device:

```
#define QUEUE_FLAG_DEFAULT      ((1 << QUEUE_FLAG_IO_STAT) |            \
                                 (1 << QUEUE_FLAG_STACKABLE)    |       \
                                 (1 << QUEUE_FLAG_SAME_COMP)    |       \
                                 (1 << QUEUE_FLAG_ADD_RANDOM))
```

However, using the SysFS file of `add_random` found for each block device, that flag can be toggled. If the file contains a 1, the flag is set for the respective block device wait queue. This toggling can be used together with the contents of the SysFS file `rotational`, which identifies a block device based on rotating disks.

In addition, the kernel unsets the flag for disks that are known to not have rotational disks. Such unsetting of the flag is done for:

- RAM-backed block devices,
- SSDs,
- MMCs,
- Network block Devices,
- ZRAM block devices,

- Multiple Devices (MD – software RAID) block devices,
- Memory Technology Device (MTD) block devices,
- Device Mapper block devices,
- S390 Support for Storage Class Memory (SCM) block devices,
- S390 XPRAM block devices, and
- the IBM PCIe SSD storage device: Flash Adapter 900GB Full Height.

The reason why disk devices are used as an entropy source is based on the nature of the disk devices and the resolution of the timer maintained by the kernel. The timer is very precise so that time variances in reading sectors from disks can be measured. Such time variances occur when the disk is spinning. For example, when sector 0x100 is to be read and the disk has to spin a quarter turn before reaching the start of this sector, the waiting time for the kernel is smaller than when the kernel would read that sector again and the disk would need to spin, say, three quarters. Moreover, the time to position the reading head also affects the timer.

However, a drawback must be considered when using disks that have no spinning disk. As the discussed time variances when reading only depend on moving parts, the entropy gathered by disks without spinning disks must be considered minimal.

The data obtained by the entropy collection value is the

block device number + 0x100, and

High-resolution time stamp.

The high-resolution time stamp is added to the input pool via the `add_timer_randomness` function. The time stamp event value is statistically analyzed in section 6.1.2.

3.5.2.4 add_device_randomness

Contrary to the aforementioned entropy collection functions, the goal of `add_device_randomness` is to feed the input pool during initialization time of device drivers. The function of `add_device_randomness` is intended to be invoked only once by device drivers with device-specific data.

The device-specific data is usually data that contains some uncertainty. This device-specific data together with the time stamp of the invocation of the function is mixed into the entropy pool.

The entropy estimator of the input pool is left unchanged which implies that the device-specific data is not assumed to contain entropy. Therefore, the device-specific data is only used to further stir the entropy pool.

The data to be mixed into the entropy pool is:

Device driver specific value, and

High-resolution time stamp.

3.5.2.5 add_hwgenerator_randomness

The Linux kernel contains an additional entropy collection mechanism for in-kernel hardware-RNG device drivers. Before the advent of the `add_hwgenerator_randomness` function, the user space rngd daemon was required to transport random bits from /dev/hwrng – the interface to the hardware-RNG framework – to /dev/random. With the functionality described in the following, this detour via user space is no longer needed.

Contrary to the aforementioned interface functions which use the `add_timer_randomness` function to feed the entropy into the input pool[6], the interface for the hardware-RNG driver framework mixes the obtained entropy directly into the input pool, by using the function `mix_pool_bytes`. Therefore, this interface establishes another seed source for the input pool in addition to those listed in figure 2.

Until the base ChaCha20 DRNG is fully seeded, the Linux-RNG allows all data received through this API call to be added to the entropy pool. After the fully seeded state is reached, the caller of the API is put to sleep and woken up once every 60 seconds.

The interface first mixes the random data into the input pool using the standard function `mix_pool_bytes` discussed in section 3.3.1.1. After stirring the pool, the entropy estimator for the input pool is updated with the entropy value that the caller of the interface specifies – i.e. the random number generator does not apply any heuristics to estimate the entropy from the obtained values using time variances. The interface is intended to bypass the entropy estimation heuristics implemented with the standard function of `add_timer_randomness` and therefore does not use that function.

The entropy collection function `add_hwgenerator_randomness` is exclusively used for mixing random data into the Linux-RNG that is derived from hardware random number generators. Per default, hardware random number generators are used as noise source for the Linux-RNG, if they are defined with a positive entropy "quality" value. At the time of writing, the following hardware random number generator drivers define a quality value:

- The driver for the Cavium random number generator (drivers/char/hw_random/cavium-rng-vf.c) defines a quality of 1,000 which is translated by the framework into 1000/1024=0.977 bits of entropy per data bit.
- The ARM CryptoCell True Random Number Generator driver (drivers/char/hw_random/cctrng.c) uses the quality setting of 1024 which implies that this driver claims one bit of entropy per data bit.
- The Marvell CN10K Random Number Generator driver (drivers/char/hw_random/cn10k-rng.c) applies the quality value of 1000 implying 0.977 bits of entropy per data bit.
- The Freescale i.MX RNGC Random Number Generator implemented with drivers/char/hw_random/imx-rngc.c uses a quality of value of 19. This is translated into 19/1024=0.019 bits of entropy per data bit.
- The Mediatek hardware random number generator driver of drivers/char/hw_random/mtk-rng.c uses a quality value of 900 which translates into 900/1024=0.879 bits of entropy per data bit.
- The Nuvoton NPCM random number generator driver of drivers/char/hw_random/npcm-rng.c uses a quality value of 1,000 and thus uses the same entropy rate as outlined for the Cavium driver.
- The OMAP and OMAP3 hardware random number generator implemented with drivers/char/hw_random/omap-rng.c and omap3-rom-rng.c use a quality value of 900. See the Mediatek driver for the entropy rate.
- The OP-TEE based Random Number Generator support driver (drivers/char/hw_random/optee-rng.c) applies the quality value of 1000 implying 0.977 bits of entropy per data bit.

6 See also figure 2 which shows how the seed sources are linked into the random number generator.

- The IBM System Z / S390 TRNG available via CPACF extension MSA 7 is accessible via drivers/char/hw_random/s390-trng.c and uses a quality value of 1024 with an entropy rate of 1024/1024=1 bit of entropy per data bit.
- The STMicorelectronics STM32 random number generator driver implemented in drivers/char/hw_random/stm32-rng.c uses a quality rate of 900. See the Mediatek driver for the entropy rate.
- The virtio-rng driver (drivers/char/hw_random/virtio-rng.c) defines a quality of 1,000 which implies that its data is treated with the same entropy content as described for the Cavium RNG.
- The Xiphera FPGA based True Random Number Generator support driver (drivers/char/hw_random/xiphera-trng.c) applies the quality value of 900 implying 0.879 bits of entropy per data bit.

Please note that the CPU hardware RNGs like the Intel RDRAND or RDSEED instructions are not processed with the `add_hwgenerator_randomness` function.

The developers of the respective device drivers are responsible to define an entropy content delivered by the respective hardware random number generator. The Linux-RNG does not implement any heuristics to estimate the entropy content of data obtained from these hardware random number generators. Further details about hardware RNGs are provided in section 3.9.2.

The data to be mixed into the entropy pool of the Linux-RNG is the

random number produced by the hardware random number generator.

3.5.2.6 add_bootloader_randomness

The boot loader may hold data that is considered to contain entropy. This data can be inserted into the Linux-RNG using the `add_bootloader_randomness`. If the kernel is compiled with `CONFIG_RANDOM_TRUST_BOOTLOADER` or by specifying the kernel command line option `trandom.trust_bootloader=1` this data inserted by the bootloader is credited full entropy.

The data to be mixed into the entropy pool of the Linux-RNG is the

random number provided by the hardware random number generator.

3.5.2.7 add_vmfork_randomness

When Linux executes within a virtual machine, there is a problem that most hypervisors can snapshot the state of the machine and later rewind the VM state to the saved state. This results in the machine running a second time with the exact same RNG state, which leads to serious security problems. Some hypervisors provide a new VM ID via ACPI when the VM is woken up or otherwise started. This value is mixed into the input pool.

When the Linux-RNG is fully seeded at the time a new VM ID is inserted, the base ChaCha20 DRNG is reseeded.

The data to be mixed into the entropy pool of the Linux-RNG is the

new virtual machine identifier.

3.5.2.8 Scheduler-based Entropy Collection

During boot time when the base ChaCha20 DRNG is not fully seeded and user space requests random numbers via /dev/urandom are received or an in-kernel caller is made to wait for reaching the fully seeded level, the kernel tries to obtain entropy by measuring the uncertainty from the scheduling operation.

This measurement only works if a high-resolution time stamp is present. If the kernel detects that such high-resolution time stamp is not available, the scheduler-based entropy collection is not used.

The entropy collection is based on the following simple concept: A high-resolution time stamp with a size of 64 bits is mixed into the entropy pool after a schedule operation is completed. The scheduling operation implies that the current execution thread is stopped and all other pending execution threads are processed until their time slice expires or they surrender the CPU. After all threads are processed, the thread of the Linux-RNG trying to generate entropy is started again by injecting a new high-resolution time stamp into the entropy pool. This sequence is repeated until the ChaCha20 DRNG is marked as fully seeded.

The functionality discussed so far only mixes data into the entropy pool. In addition to the mixing operation, the Linux-RNG arms a timer which expires after a time of one Jiffy – i.e. depending on the HZ kernel compilation option the timer expires after 1 ms or 4 ms. The timer is re-armed after expiry for another Jiffy period. During each expiry the entropy estimator of the Linux-RNG is increased by one bit. The timer is rearmed for as long as the high-resolution time stamp collection loop mentioned above executes.

Once the ChaCha20 DRNG is fully seeded, all operations of the scheduler-based entropy collection ceases and the entropy estimator is not modified any more by this entropy collector.

The data to be mixed into the entropy pool of the Linux-RNG is the

continuous stream of 64 bit high-resolution time stamps.

3.5.2.9 Latent Entropy GCC Plugin

Starting with kernel 4.9, a GNU Compiler Collection (GCC) plugin named “latent_entropy” is added to the kernel source code tree. As the name indicates, it is a plugin to the C compiler used to generate the binary code out of the kernel source code. This GCC plugin can be used to alter the binary code behavior compared to the “assumed” behavior visible with the C code. However, the GCC plugin code will not end up as part of the Linux kernel binary.

The latent entropy GCC plugin is designed to extract as much uncertainty from a running system at boot time as possible, hoping to capitalize on any possible variation in CPU operation (due to runtime data differences, hardware differences, SMP ordering, thermal timing variation, cache behavior, etc).

The concept of the GCC plugin is the permutation of a global variable based on any variation in code execution. During the boot process, the Linux kernel uses all available CPUs for the initialization of the kernel. Depending on the state of the CPU, sometimes a function on one CPU will complete earlier than a function on another CPU. In a subsequent boot process, this may be reversed. These variances are picked up by mixing a function-specific value into the global variable. The variable may therefore be different depending on the particular order of functions that were executed. The GCC plugin inserts a local variable in every marked function at compile time. The GCC plugin also adds logic so that the value of this variable is modified by randomly chosen operations (ADD, XOR and left-rotation) and random values (GCC generates separate static values for each location at compile time and also injects the stack pointer at runtime). The resulting value depends on the control flow path (e.g., loops and branches taken).

Before the modified function returns, the plugin mixes this local variable into the latent_entropy global variable. The value of this global variable is added to the input pool during initialization of the kernel when the function `do_one_initcall` is invoked, and during the creation of a new process when invoking the `_do_fork function`. In both cases, the `add_device_randomness` Linux-RNG API function is

invoked with the current content of the `latent_entropy` global variable. As discussed in section 3.5.2.4, the injected data is considered to have no entropy.

Additionally, the plugin can pre-initialize arrays with build-time random contents, so that two different kernel builds running on identical hardware will not have the same starting values.

3.5.2.10 Mixing Entropy Source Data Into Entropy Pool

When entropy from the HID and block device entropy sources discussed above is mixed into the random number state, the function `add_timer_randomness` is used. This function always mixes the gathered entropy into the input pool. Hardware entropy is never mixed into the ChaCha20 DRNGs.

When mixing the hardware data into the input pool, the function `add_timer_randomness` adds not just the hardware-related data, but also timing data:

```
static void add_timer_randomness(struct timer_rand_state *state, unsigned
int num,
                                 unsigned int hid)
{
        unsigned long entropy = random_get_entropy(), now = jiffies,
flags;
...
                  _mix_pool_bytes(&entropy, sizeof(entropy));
                  _mix_pool_bytes(&num, sizeof(num));
...
```

The above code snippet shows a data structure that is filled with:

- a high-resolution time stamp at the time of invocation of this code using the `random_get_entropy` function where the 32 low bits of the time stamp are used, and
- the value `num` which is the hardware-related data provided by the hardware entropy gathering functions like `add_input_randomness`.

The platform dependent function `random_get_entropy` is used to read the hardware timer. This function uses the following processor functions on the individual platforms to extract the timer value:

- The `RDTSC` (Read Time Stamp Counter) instruction on Intel x86 and AMD-Opteron
- The `STCK` (Store Clock) instruction on the zArchitecture
- The `MFTB` (Move From Time Base) instruction on Power
- On ARM 32-bit systems, the register value from the internal co-processor P15 is read using the opcode 0 and `CRm = c14`. This operation is implemented in the function `arch_counter_get_cntpct` with the invocation of the following assembler code:
  `asm volatile("mrrc p15, 0, %Q0, %R0, c14" : "=r" (cval));`
- On ARM 64-bit systems, the register CNTVCT_EL0 is read by the function `arch_counter_get_cntvct` which in turn uses the invocation `arch_timer_reg_read_stable(cntvct_el0)` that uses the following assembler helper code:
  `asm volatile("mrs %0, " __stringify(r) : "=r" (__val));`

In all cases those instructions return a 64-bit value of the current hardware time counter irrespective of the word size of the underlying CPU.

In the case of the Intel x86 and the Opteron the value of the clock is incremented every processor cycle, even when the processor is halted. This results in about 1 Billion increments per second on a 1 GHz processor. The value of the hardware time counter is reset to zero when the processor receives a reset signal.

In the case of IBM System Z there are 2^{31} increments of the hardware time counter every 1.048576 seconds. The value stored is the Time Of Day (TOD) clock, which is initialized when the kernel is started.

In the case of the Power architecture, the hardware time counter is incremented every 32 cycles of the processor. This results in 31,250,000 increments per second on a 1 GHz CPU. This hardware time counter is reset to zero when the CPU is reset.

Direct access to the hardware timer eliminates potential effects of a software maintained timer and the influence of any software running on the kernel on the value of the timer. It also provides the highest resolution possible for the given hardware platform.

In addition to the mixing of data into a given pool, `add_timer_randomness` also triggers the calculation of the entropy estimation for the processed data. This entropy estimation is discussed in section 3.6.

Using the data structure sample, the following data is mixed into the entropy pool:

high-resolution time stamp || Jiffies || event value[7]

This value is subjected to quantitative analyses in section 6.2.

3.6 Entropy Estimation

In the preceding section it was shown how the entropy pool is updated. As noted there, the update of the entropy pool does not imply an update of the entropy that is estimated to exist in the given pool.

The entropy estimation is only carried out in the function `credit_init_bits`. The name of the function is indicative of the concept: it only credits entropy while the base ChaCha20 DRNG is not fully seeded yet. After reaching this state, the entropy estimator becomes irrelevant, it is not used any further to regulate the behavior of the Linux-RNG. The entropy estimator may be updated, e.g. by the IOCTL `RNDADDENTROPY` but this update is not going to trigger any operation once the base ChaCha20 DRNG is fully seeded.

If data is mixed into the random pools by the noise sources that credit entropy, the entropy estimator is increased by the heuristically applied entropy value. The following entropy sources provide entropy:

- Injection of data from the interrupt noise source: `add_interrupt_randomness` applies one bit of entropy per fast_pool to input pool transferal as specified in section 3.5.2.2.
- Injection of data with the add_hwgenerator_randomness function discussed in section 3.5.2.5.
- Generation of entropy with the scheduler-based entropy source described in section 3.5.2.8.

The Linux-RNG estimates the entropy of a hardware event and modifies the entropy estimator of the input pool accordingly. The ChaCha20 DRNG instances do not have any entropy estimator as they operate as a DRNG which is seeded once every 60 seconds. The base ChaCha20 DRNG is seeded with the available entropy from the input pool of up to 256 bits. Yet, even if no new entropy was added to the input pool, the Blake2s message digest as discussed in section 3.3.1.3 is calculated and injected into the ChaCha20 DRNG state.

The idea of the entropy estimation is that each hardware event is awarded a heuristically estimated entropy value which then increments the entropy estimator of the entropy pool. The process of estimating the entropy is performed for each received event.

7 The symbol "||" marks a concatenation of data.

The entropy estimator is an integer value stored in the data structure of the entropy pool. The integer value holds the entropy in bits. The entropy estimator value `init_bits` is updated after the values are stirred into the input pool.

For the noise sources using the add_timer_randomness service function, the heuristic entropy estimate is obtained as follows. The Jiffies time of the hardware event is t_n . The Jiffies time stamps referring to prior hardware events of the respective hardware component are denoted with $t_{(n-1)}$ through $t_{(n-3)}$. Section 3.5.1 illustrates which hardware components are tracked individually or jointly.

The following values are calculated in `add_timer_randomness`:

- `delta` = $\left|(t_n - t_{(n-1)})\right|$
- `delta2` = $\left|((t_n - t_{(n-1)}) - (t_{(n-1)} - t_{(n-2)}))\right|$
- `delta3` = $\left|(((t_n - t_{(n-1)}) - (t_{(n-1)} - t_{(n-2)})) - ((t_{(n-1)} - t_{(n-2)}) - (t_{(n-2)} - t_{(n-3)})))\right|$

These values can be interpreted as the first, second and third discrete derivative of the event time for the hardware component.

The entropy of one event is now heuristically determined as follows:

1 Compute the minimum delta value `min(delta, delta2, delta3)`.

2 Calculate the $\log_2$ of the minimum delta as an integer operation – i.e. value after the decimal point is discarded. The implementation of the logarithmic operation is achieved by dividing the minimum delta value by 2 and obtain the index of the highest bit of the value.

3 Use the minimum of the bit index and 11.

The method used is heuristic and assumes that the lower bits of the time of a hardware entropy event are unpredictable. Even two identical instruction sequences in a system with no network interrupt would result in very different interrupt timings. Yet, there is no rationale or test specification that conclusively demonstrates this assumption.

The use of Jiffies for the entropy calculation is historic: in the old days, the kernel only had the Jiffies time stamp available. With the advent of high-resolution timers, the majority of entropy is derived from this time stamp. Yet, the heuristic entropy estimation logic is not updated.

The heuristic entropy estimation value for the given hardware event is now used to increase the entropy estimator of the input pool by the given value.

Interrupts increase the entropy estimator by one (bit) every time a copy of the fast_pool is inserted into the entropy pool unless it is an interrupt that was also processed as a HID event. In this case, the interrupt is awarded with up to 11 bits of entropy as outlined for the `add_timer_randomness` entropy discussion.

The entropy estimator is not reduced when obtaining random bits from the entropy pool. Thus the entropy estimator is only used to determine when overall 256 bits of entropy have been received.

The `credit_entropy_bits` function also triggers the wakeup of read-like polling processes if the base ChaCha20 DRNG is considered to be fully seeded.

3.7 Generic Architecture and Linux-RNG

With chapter 2, a general architecture of NDRNGs is given. This section now maps the general architecture to the Linux-RNG to analyze whether all components that are expected to be present with a NDRNG are in fact present to consider the Linux-RNG as a stand-alone system.

Figure 5 provides a mapping of the Linux-RNG with the theoretical discussion about NDRNG architecture. Using the mapping, the noise sources as well as the DRNG can be clearly identified and separated from the remainder of the Linux-RNG processing.

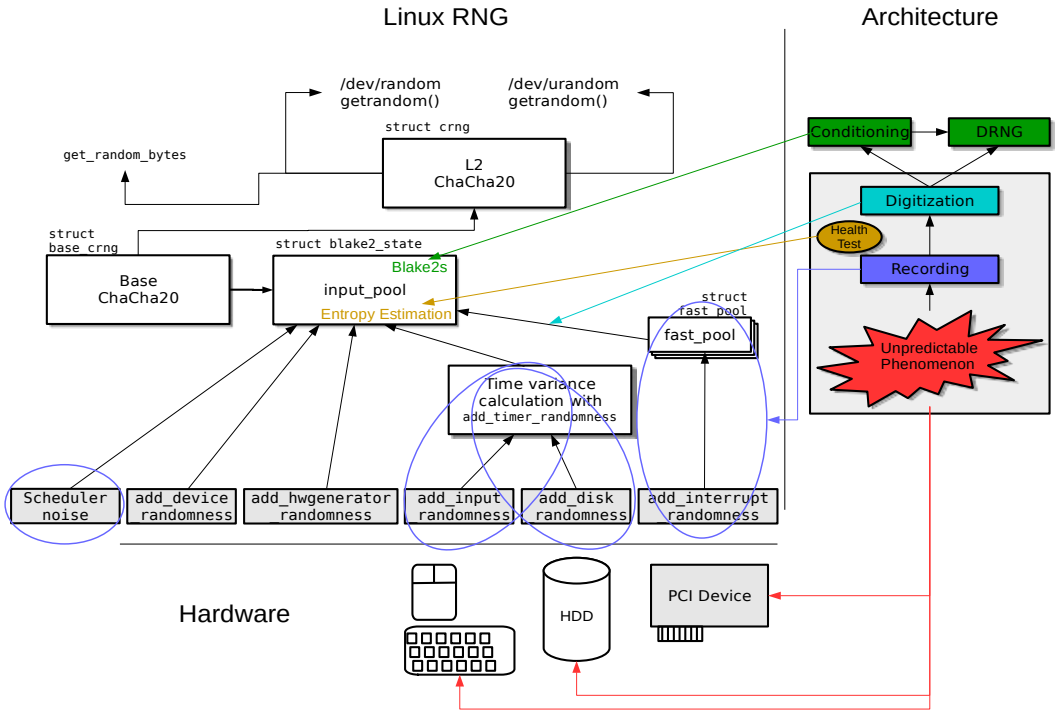

Figure 5: Linux-RNG architecture compared with the generic architecture

With figure 5, three areas are illustrated which are separated by a dotted line:

- The upper left part contains the Linux-RNG illustration shown in the preceding section.
- The Linux-RNG observes and records events from various hardware devices. These hardware devices are illustrated in the lower left part of figure 5. Each of the gray boxes of the Linux-RNG containing "`add_*_randomness`" maps to a device type that is monitored by the Linux-RNG. The Linux-RNG boxes of `add_device_randomness` and `add_hwgenerator_randomness` are not further mapped and discussed, as they either do not deliver any entropy or access highly specialized hardware that is not commonly present in standard systems. To keep the entire discussion concise, these two boxes are therefore disregarded. In addition, the scheduler-based noise source is specified with the gray box in the lower left corner. Further details about these functions are given in section 3.5.
- The right part of figure 5 contains the architecture illustration from figure 1. As the discussion is about NDRNG, figure 5 does not further show the box about the cryptographic usage of data obtained from the noise source via the DRNG.

The right side of figure 5 shows the theoretical noise source concept from figure 1. Figure 5 uses different colors for the different components and uses equally colored arrows that point to the respective components of the Linux-RNG. To be precise:

- The unpredictable phenomenon identified with the red arrows in the Linux-RNG are the events triggered by the monitored hardware components. Section 3.5 provides details about these sources of unpredictability.

- The recording of the unpredictable phenomenon, i.e. the events and their precise timing triggered by the aforementioned hardware components, is performed by the blue-marked components of the Linux-RNG, namely the `add_*_randomness` functions as well as the scheduler-based noise source.
- The digitization of the data obtained with the recording components is implemented by injecting the recorded data into the input pool. Digitization is performed in a very simple fashion in the Linux-RNG as follows:
  - For HID and block devices, the recorded event type and time stamp are stored in a data structure which then is simply treated as a byte-stream that is injected into the input pool. The interpretation of the recorded data as a byte stream is the digitization of that data.
  - For interrupts, regular snapshots of the fast_pool are injected into the input pool. Similarly to the HID and block devices, the contents of the fast_pool is treated as a byte-stream when it is mixed into the input pool. Again, the interpretation of the fast_pool as a byte-stream implements the digitization aspect.
  - For the scheduler-based noise source, the 64-bit time stamp value is mixed into the input pool. This value is treated as a 64-bit data stream.
- When considering the health test, the Linux-RNG implements one mechanism that serves as a common health test for all data that will be mixed into the input pool: the entropy estimation heuristic. The entropy estimation calculates the first, second and third derivative of the Jiffies time stamp of each event. Now, when this entropy estimation is zero, the base ChaCha20 DRNG may not reach its fully seeded state during boot time. In effect, this implies that the mixed in event data is treated as poor data where the noise source failed to deliver entropy. Yet, this health test effect is only used until the Linux-RNG reaches the fully seeded state.
- The function inserting data into the input pool using a Blake2s message digest can be considered a conditioning logic. An additional conditioning logic is evident with the fast_pool maintenance.
- The DRNG part is implemented by the output function to read the input pool: the output function calculates a Blake2s hash over the entire input pool which is used as the random number.

The description illustrates that the NDRNG solely is provided by the input pool and the functions feeding it with data. This allows the following interpretation of the Linux-RNG architecture:

The input pool together with its feeding functions is the NDRNG as already mentioned.

The ChaCha20 DRNG is a standalone, independent DRNG seeded from the input pool.

3.8 Use of the Linux-RNG

To ensure that the data read from the Linux-RNG contains sufficient entropy, a number of precautions must be taken. If one of these measures is not carried out, the quality of the data read from the Linux-RNG can be reduced significantly.

The quantitative analysis on the entropy of interrupts given in section 6.2.1 outlines that significant entropy is provided on systems with a high-resolution time stamp such as Intel x86 systems. On such systems, the following precautions may not need to be fulfilled in their strictest form.

During the shutdown of Linux, a number of bytes is suggested to be read from /dev/random and stored in non-volatile storage. The data is not treated to have entropy, but it helps to stir the input pool during boot time when the available entropy is still low. When storing the data, the storage must ensure that the data is inaccessible to untrusted entities. For example, the permissions of a file holding the data shall be 600 and the directory holding the file shall not be writable by untrusted users. Moreover, the number of bytes to be read is defined by the contents of /proc/sys/kernel/random/poolsize. The following code may be used to generate such a seed file:

```
umask 077
rm -f /var/lib/random-seed
dd if=/dev/urandom \
   of=/var/lib/random-seed \
   bs=$(cat /proc/sys/kernel/random/poolsize) \
   count=1
```

Note, the location of the seed file is arbitrary, as long as the file is accessible during boot and it is protected from read/write access by any user other than root.

If the system does not have a defined shutdown cycle (for example it is an embedded device), the generation of the seed file should be performed during run time at either given intervals or once after boot. For example, the seed file can be generated every hour or one hour after boot.

During the startup of the user space the seed generated during the last run must be mixed into the state of the RNG by simply writing the seed into /dev/random or /dev/urandom. When writing into these files, the entropy pool is further mixed, ensuring that the state and therefore the entropy of the previous instance of the Linux-RNG is used to update the current state.

Considering regular Linux distributions, the initial installation writes a large amount of data to disk resulting in a large quantity of entropy. In addition, the installation process may require human interaction which leads to additional entropy being added via the HID of mouse or keyboard. That entropy should be saved similarly to saving the seed for a regular reboot discussed for step 1.

In the case of a full disk encryption configuration, the volume master key used for encrypting all data is generated very early in the initial installation cycle using a random number generator that is seeded by /dev/urandom. Considering the worst-case scenario of having an automated installation process with only limited administrator interaction, the entropy in the Linux kernel is very low. Therefore, the volume master key also will not have much entropy. In such a scenario, it is mandatory that additional entropy is gathered before the key is generated. For example, the installer may require a number of keyboard interactions before /dev/urandom is accessed and read from. As a conservative rule of thumb, one key stroke may be assumed to have one bit of entropy. On the other side, the developer may use the `getrandom` system call without any flags or /dev/random to ensure a fully seeded Linux-RNG.

In the case of LiveCDs, the boot sequence should be interrupted to require the user to provide entropy using the HIDs such as mouse or keyboard before any cryptographically strong key is to be generated. For example, when starting the OpenSSH daemon, the entropy inside the Linux kernel should be topped off. The reason for this requirement is that such LiveCDs do not implement the reseed maintenance. The subsequent mix-in into /dev/random during the next boot cycle therefore lacks significant entropy before disks are accessed or Human Interface Device devices are utilized. Again, a solution would be also to use the `getrandom` system call without any flags or /dev/random to ensure a fully seeded Linux-RNG.

3.9 Hardware-based Random Number Generators

3.9.1 CPU Hardware Random Number Generators

The driver for the Linux kernel random number generator uses the following hooks to request random numbers from a hardware noise source in the source code file include/linux/random.h:

```
#ifdef CONFIG_ARCH_RANDOM
# include <asm/archrandom.h>
```

```
#else
static inline bool __must_check arch_get_random_long(unsigned long *v)
{
        return false;
}
static inline bool __must_check arch_get_random_int(unsigned int *v)
{
        return false;
}
static inline bool __must_check arch_get_random_seed_long(unsigned long
*v)
{
        return false;
}
static inline bool __must_check arch_get_random_seed_int(unsigned int *v)
{
        return false;
}
#endif
/*
 * Called from the boot CPU during startup; not valid to call once
 * secondary CPUs are up and preemption is possible.
 */
#ifndef arch_get_random_seed_long_early
static inline bool __init arch_get_random_seed_long_early(unsigned long
*v)
{
        WARN_ON(system_state != SYSTEM_BOOTING);
        return arch_get_random_seed_long(v);
}
#endif

#ifndef arch_get_random_long_early
static inline bool __init arch_get_random_long_early(unsigned long *v)
{
        WARN_ON(system_state != SYSTEM_BOOTING);
        return arch_get_random_long(v);
```

```
}
#endif
```

The functions have the following meaning:

- `arch_get_random_long`: returns a 64-bit value (64-bit architectures) or a 32-bit value (32-bit architectures) from the DRNG output interface .
- `arch_get_random_int`: returns a 32-bit value from the DRNG output interface.
- `arch_get_seed_long`: returns a 64-bit value (64-bit architectures) or a 32-bit value (32-bit architectures) from the seeding output interface.
- `arch_get_seed_int`: returns a 32-bit value from the seeding output interface.
- The functions `arch_get_random_seed_long_early` and `arch_get_random_seed_long_early` are to be used for the first call to the CPU-based entropy source to allow potential initialization.

The mentioned functions are set to return false which implies that the Linux-RNG would perform its operation completely without the help of a hardware RNG. However, if the kernel is compiled with hardware support, the file asm/archrandom.h contains replacements for the given functions.

The following sections discuss the currently implemented hardware RNG support.

3.9.1.1 Intel RDRAND and RDSEED Instructions

Starting with the IvyBridge x86_64 processor, Intel implements the `RDRAND` instruction. That instruction provides access to a hardware noise source that is processed by a deterministic SP800-90A compliant DRBG based on AES in CTR mode[8].

Starting with the Broadwell Intel x86_64 CPU release, the `RDSEED` instruction is offered in addition. The `RDSEED` instruction allows access to the output of the AES CBC-MAC conditioned noise data which is also used to seed the aforementioned CTR DRBG.

The Linux kernel implements the support for the `RDRAND` and `RDSEED` instructions by implementing the above mentioned architecture-specific callback functions to return random numbers with a different size.

The implementation is based on assembler code provided in the file arch/x86/include/asm/archrandom.h. The assembler code makes sure that the instruction is only invoked if the CPU implements the requested `RDRAND` or `RDSEED` instruction by checking the CPUID feature of `X86_FEATURE_RDRAND` or `X86_FEATURE_RDSEED`, respectively.

3.9.2 Hardware Random Number Generator Framework

The Linux kernel implements a framework for hardware RNGs which are provided with dedicated hardware components such as PCI cards or auxiliary hardware components which are not commonly present for the majority of users. This framework exports a character device file to user space, /dev/hwrng, that allows user space to read data from a device driver that registered with the framework and found the associated hardware. The hardware random number generator framework is unrelated to the Linux-RNG and to the CPU hardware RNG support mentioned above.

8 For more details, see http://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide/.

The Linux-RNG offers the function `add_hwgenerator_randomness` to the hardware RNG framework. Using this interface function, the hardware random number generator framework can inject entropy into the input pool directly without requiring user space support.

The following subsection illustrates the different use cases of the hardware random number generator support using the example of the IBM POWER system.

3.9.2.1 IBM POWER Random Number Generator

The IBM POWER CPU implements a hardware noise source based on ring oscillators. This noise source is only accessible from software executing in supervisor state, i.e. a driver in an operating system kernel.

The IBM POWER system is offered with two hypervisors that are mutually exclusive: the IBM proprietary PowerVM, and PowerKVM based on Linux with KVM support. When using hypervisors, the noise source can only be accessed by the hypervisor. Guest operating systems must interact with the virtual machine monitor to access the data from the hypervisor.

Figure 6 illustrates the data flow of the random numbers from the noise source to the Linux random number generator when Linux executes as a guest operating system in a PowerVM environment:

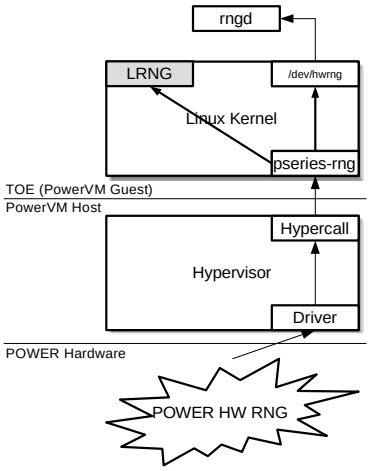

Figure 6: Flow of random numbers in a PowerVM environment

The figure for PowerVM shows the following information flow when new random numbers are requested in the Linux guest operating system:

1 New data is obtained from the noise source by a PowerVM proprietary device driver.

2 PowerVM makes this data available via a hypercall. That hypercall is used by the Linux guest kernel driver pseries-rng.

3 The driver pseries-rng is registered with the Linux kernel hardware RNG framework that makes the data available to the Linux guest operating system user space via the /dev/hwrng device file.

4 The rngd daemon may pull the data from /dev/hwrng.

5 Using the `add_hwgenerator_randomness` interface, the hardware random number generator framework injects the data into the Linux-RNG of the Linux guest operating system.

Figure 7 illustrates the data flow of the random numbers from the noise source to the Linux-RNG when Linux executes as a guest operating system in a PowerKVM environment.

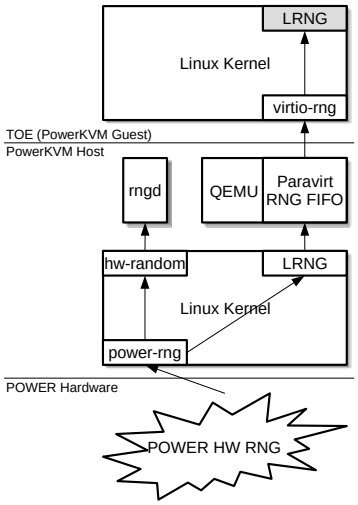

Figure 7: Flow of random numbers in a PowerKVM environment

The figure for PowerKVM shows the following information flow when new random numbers are requested in the Linux guest operating system:

1 New data is obtained from the noise source by the Linux power-rng device driver in the Linux host. This device driver is connected with the Linux kernel hardware RNG framework that makes the data available to the Linux host operating system user space via the /dev/hwrng device file.

2 The rngd daemon in the PowerKVM host may pull the data from /dev/hwrng.

3 Using the `add_hwgenerator_randomness` interface, the hardware random number generator framework injects the data into the Linux-RNG on the Linux host operating system.

4 The QEMU virtual motherboard application in the PowerKVM host provides a para-virtualized device which acts as a first-in, first-out (FIFO) between the guest OS and the PowerKVM host /dev/random[9].

5 The QEMU para-virtualized device is accessed by the Linux guest operating system device driver virtio-rng.

6 The driver virtio-rng is registered with the Linux kernel hardware random number generator framework that makes the data available to the Linux guest operating system user space via the /dev/hwrng device file. In addition, that particular device driver uses the hardware random number generator framework to establish a dedicated link to the Linux random number generator of the Linux guest operating system and can provide data to the input pool. The framework establishes a kernel thread named “hwrng” that pulls 32 bytes from the virtio-rng device (and thus from the PowerKVM host /dev/random device) and adds it to the Linux guest operating system input pool as discussed in section 3.5.2.5. When the thread is spawned at the time the virtio-rng driver is loaded and initialized, the first 32 bytes are provided to the input pool.

3.10 Support Functions for Other Kernel Parts

The source code file drivers/char/random.c implementing the Linux-RNG offers service functions to other kernel parts that are not related to the Linux-RNG. All interface functions eventually will invoke the standard get_random_bytes function and thus use the Linux-RNG as analyzed in the remainder of this document.

9 The administrator of the host system can configure QEMU to access also /dev/urandom.

In addition, the Linux-RNG provides the function `get_random_bytes_arch` which allows callers to obtain random numbers from the CPU hardware RNG. If the current CPU does not provide such support, the service function fails and returns that failure to the caller.

3.11 Time Line of Entropy Requirements

To give the reader a general impression when random numbers are required in a Linux system, this section describes the boot process of a common Linux environment. The reader should consider this section as guidance only, since the precise random number requirements highly depend on the structure of the Linux system, including whether it uses an initramfs, is a Live-CD, how user space is booted, etc.

This section uses the common Linux distributions such as Fedora, openSUSE's Leap or Debian as examples and assumes the use of systemd as user space initialization framework and the use of an initramfs.

The description also provides an indication of event times since boot when certain events happen. These event times naturally may vary widely depending on the CPU speed, used hardware components that need initialization and similar factors. Therefore, these event times should be used by the reader only as an indication where the relative sequence of events in a large number of cases remains the same.

3.11.1 Installation Time

The installation of a Linux system is commonly started by booting a Live-CD or a USB thumb-drive with an ISO image. The booted Linux environment is solely started from the boot media, such as a DVD or USB drive.

The installation environment is not considered a general-purpose computing environment and thus is not intended to be used to process user data with cryptographically secure mechanisms. In special circumstances, such "installation-like" Linux environments are used for active use including cryptographic purposes, like Live-CDs. Yet, such use cases are rare and require additional consideration regarding entropy.

So, why is the installation time still of interest? The answer lies in more and more common full-disk encryption installations. As of now, Linux with its dm-crypt full disk encryption solution does not support encrypting the disk at runtime. Therefore, when the full disk encryption support is enabled, it is mandatory that the partitions subjected to encryption are prepared accordingly before any data is copied onto them. For the root partition, such a setup can consequently only be performed before it is created and data is copied to it.

The installation tools commonly ask all installation-relevant questions before any operation including disk accesses is performed. This also covers the request of the user's password during the installation process that will guard the master volume key. One of the first steps during the installation will be the preparation of the hard disk as a dm-crypt container. This preparation includes the generation of the master volume key that is a random number commonly obtained from libgcrypt's DRNG that is seeded from /dev/urandom. That implies that this master volume key, which remains unchanged for the lifetime of the file system, is created before hardly any disk operations have been performed. In some cases, the installation tool is even ASCII-based or even completely unattented which requires the user to press a few keys only without the use of a mouse. Thus, the two noise sources for block devices and HID hardly collect entropy.

After the installation is completed, some installers already create the seed file which is injected into the Linux-RNG during the first boot from the newly installed hard disk. Since by the end of the installation time many disk accesses have been performed without much random data having been extracted from the Linux-RNG, its entropy pool can be considered to be full of entropy and thus the seed data to be entropic, too.

3.11.2 First Reboot After Installation

After the installation of the system, the first following reboot is important regarding its cryptographic security. During that first reboot, the SSH host keys are created at the time the SSH daemon is started. The start of the SSH daemon commonly happens during the start phase of the network daemons commonly between 2 and 10 seconds after boot, depending on the CPU speed and other properties of the system.

Other cryptographic keys may be automatically generated almost at the same time, such as keys and certificates for TLS servers and others.

3.11.3 Regular Usage

The following description outlines the sequence of events with respect to the Linux-RNG and the use of random numbers that may be commonly observed during regular boot sequences:

1 The system is powered on, and the kernel is loaded into memory and boots.

2 After ending the initramfs phase which mounted the root file system, the user space initialization starts. During the early phase of this initialization, the seed file is written into /dev/random.

3 Cryptographic daemons such as the SSH server daemon, web servers with TLS support, and the IKE daemon start. During their startup, the used cryptographic libraries seed their DRNG from /dev/urandom. Those daemons are accessible from remote entities and are intended to grant secure cryptographic operation. Note, that those user space DRNGs either do not reseed automatically at all (like it is the case with OpenSSL's SSLeay DRNG before OpenSSL 1.1.1) or only after a large reseed interval (in cases like SP800-90A DRBGs, or libgcrypt's CSPRNG). This means that by the time the daemons start and initialize their DRNG, sufficient entropy must be present in the Linux-RNG as these daemons must be considered to be cryptographically insecure otherwise.

4 In case the scheduler-based entropy source is unavailable, far later than the completion of the user space initialization, the input pool is filled with 256 bits of entropy for the first time. To give readers an impression about the delay in a worst-case, the author installed a system in a virtual machine with hardly any devices. Although the ChaCha20 DRNG initial seeding step was reached after about 0.9 to 1 seconds after boot, the fully seeded stage that marks the receipt of 256 bits of entropy in the input pool was reached up to 90 seconds after boot. The boot process of user space with an SSH daemon was fully completed after 2.5 seconds. This implied that the systemd initialization function would time-out and the SSH-daemon would never be started automatically.

3.12 Security Domain Protecting the Linux-RNG

Based on the architecture description of the preceding sections it is evident that the Linux-RNG keeps a state which collects and maintains the entropy from the noise sources. Furthermore, the Linux-RNG reads data from the noise sources which contain the raw entropy. All entropy will be immediately lost if either the state of the Linux-RNG or the behavior of the noise sources can be observed by an untrusted entity.

Moreover, the processing logic is vital to ensure that the entropy is maintained and proper random numbers are generated. Thus, besides maintaining the internal state of the RNG, the processing logic must be protected against modification by an untrusted entity.

The protection of the Linux-RNG state, the noise sources and the processing logic of the Linux-RNG can only be achieved by requiring the hosting execution platform to provide a security domain for the Linux-RNG. Such security domain is available with the Linux kernel which hosts the Linux-RNG. The protection requirements and assurance level of the Linux-RNG are at least as high as those of any other kernel functionality and data.

Any violation of the security domain of the Linux kernel by an untrusted entity, including either read and/or write access to the Linux kernel data or processing logic, implies that the entropy of the random numbers generated by the Linux-RNG must be considered compromised. It would mean that their cryptographic strength is diminished.

Such violations of the security domain include:

- Execution of untrusted code as part of the Linux kernel security domain: This would be the case if that untrusted code is loaded into the kernel and executed with kernel privileges. This can either happen because of Linux kernel bugs allowing the insertion of untrusted code via broken kernel interfaces, or if a privileged user space application is compromised to permit loading untrusted code. The execution of untrusted code allows read and write access to the Linux-RNG, its state and its noise sources.
- Read access to the state of the Linux kernel security domain: If an untrusted entity gains read access to the state data maintained within the Linux kernel the security domain is violated. Such read access may either be direct by exploiting bugs in the Linux kernel allowing such read operations or by using side effects of either the Linux kernel behavior or the underlying environment. As an example of undesired side channels are all attacks abusing cache behavior (L1, L2, L3 caches, TLB), branch-prediction and similar mechanisms. In addition, read access to the Linux kernel security domain may be possible by a virtual machine monitor if the Linux kernel executes as a guest or by more privileged software components. The latter includes the BIOS, the System Management Mode (SMM) or the Management Element (ME) found in contemporary x86 hardware.
- Write access to the state or the processing logic of the Linux kernel security domain may allow an untrusted entity to alter either the behavior of the Linux-RNG or its state. Such write accesses may also be either using direct means by exploiting Linux kernel bugs or by indirect means of side channels.

The software of the Linux kernel cannot defend against attackers with physical access to the execution environment. Thus, the proper operation of the Linux-RNG depends on the security of the Linux kernel and its execution environment where the administrator must ensure the following by virtue of operational procedures:

- the physical security of the execution environment,
- a proper patch management to ensure that the Linux kernel receives timely security updates, and
- by using a trustworthy execution environment including trustworthy hardware for the Linux-RNG.

4 Conducted Analyses of the Linux-RNG

In the past, the Linux-RNG received attention by different reviewers: An analysis of the Linux-RNG implemented in the Linux kernel version 2.6.10 has been published in 2006 by Gutterman et al. In [GPR06]. A study by Lacharme et al. from the year 2012 [LRSV12] has been carried out for kernel version 2.6.30.7 and some newer versions show that some attacks described in [GPR06] are no longer possible with newer kernel versions.

However, due to the complete re-architecture of the Linux-RNG with kernel version 5.18, none of the analyses are applicable any more.

4.1 Considerations by Müller

Over the last years, one of the author of this study has developed a drop-in replacement of the Linux-RNG available at LRNG web site[10] implemented in the kernel as well as a full user space implementation provided at the ESDM web site[11]. Both implementations are equipped with a complete design description [LRNGREPLACEMENT] (the design of both with respect to the entropy sources and DRNG management are identical, thus only the LRNG kernel implementation is referenced henceforth), qualitative and quantitative entropy assessment as well as compliance assessment with [SP800-90B] and [AIS2031].

As part of the documentation and based on the knowledge gained during the implementation of that replacement implementation, the following considerations about the Linux-RNG are raised.

In addition, the LRNG design description (section 2.2.5) hints to another issue that is outlined with the following description: Depending on the assumed entropy in the data provided by one block device event or human interface device event, zero to 11 bits[12] of entropy may be stirred into the entropy pool per injection. One injection of a fast_pool into the input pool is credited with one bit of entropy. At the same time the entropy pool is seeded with the input containing entropy, callers may read random data from it. For an insufficiently seeded random number generator, this leads to a loss in entropy that is visualized with the following worst-case analogy: when an RNG receives one bit of entropy which is followed by a generation of one or more random numbers, the caller is required to guess one bit to break the state of the RNG. When one new bit of entropy is received after the attacker's gathering of random data, the new state of the RNG will again only have one bit of entropy and not two bits (the addition of the first and second seed). Hence, in a pathological case, the entropy pool may receive 128 bits of entropy in 128 separate seeding steps where an attacker can request random data from the entropy pool between each seeding operation. An attacker has to guess $2^1 \cdot 128$ different states and not 2^{128} – i.e. the amount of guessing to deduce the RNG state is reduced to a manageable level. However, as the issue implies that (a) an event has only as much entropy as specified by the heuristically awarded entropy and (b) an attacker knows the internal state of the Linux-RNG at some point – which both are unlikely to be the case – this illustrated issue is not considered to lead to an attack. This issue is mitigated to some extent by blocking /dev/random or getrandom until the base ChaCha20 DRNG is fully seeded. Yet, the same base ChaCha20 DRNG is already available via /dev/urandom. The issue is aggravated by the fact that many reseeds take place during early boot as outlined in section 3.3.2.1. Yet, the increasing delay in reseed events mitigates the issue again. Even during runtime this issue is present because at any stage, the input pool generates 256 bits of data when the ChaCha20 DRNG is supposed to be seeded once every 60 seconds. Due to all of these mitigating and aggravating factors, a conclusive determination whether the issue is a real-life problem cannot be answered in this document. However, the entire discussion around this matter could easily be silenced if the input pool generates only data when, say, at least 128 bits of entropy have been received since the last generation attempt.

10 https://www.chronox.de/lrng.html
11 https://www.chronox.de/esdm.html
12 The explanation for the limit of 11 bits is given in section 3.6.

The following issue was identified during the development of the LRNG. Though it has been addressed in the LRNG implementation, it is still present in the Linux-RNG. When injecting new seed data from user space by either the IOCTL or by writing into either /dev/random or /dev/urandom, the seed data is added to the input pool. It remains unused there until the base ChaCha20 DRNG decides it is time to reseed. This means that new seed data provided by user space will not be put to use in the ChaCha20 DRNG accessible via the Linux-RNG interfaces for up to 60 seconds. This issue, however, can be alleviated when user space actively calls the IOCTL `RNDRESEEDCRNG` which requires root-privileges though.

Considering the nature of a random number generator as the foundation of cryptography (with the exception of hashes), any issue in the random number generator can lead to a break of the entire cryptography resting on it. Thus, particular care must be taken that the random number generator operates as expected. It is commonly the case that random number generators implement a self-test during boot time or even during runtime. This self-test should verify that the deterministic processing steps still work as intended. With the more complex post-processing operation implemented in the Linux-RNG, such self-tests are hard to add. The following self-tests could be considered:

- The operation of the ChaCha20 DRNG could be verified with a test vector. The ChaCha20 DRNG implemented in the Linux-RNG, however has internal interface functions which do not lend itself for an easy add-on of a self-test.
- The Blake2s hash operation providing the state transition function as well as the output function of the input pool is another deterministic operation which is vital to the entropy maintenance as it is the core function that compresses the input data. A self-test would be straight forward to implement with the existing Linux-RNG code base.
- The SipHash logic handling the fast_pool is yet another vital post-processing function that is deterministic. It should be subject to a self test.

With the LRNG, all of these self-tests are implemented and automatically invoked during boot time. In addition, these self-tests can be triggered at runtime at any time. This implementation could be mined to add such self-tests to the Linux-RNG.

The entropy estimation of the Linux-RNG is based on calculating the first, second and third discrete derivative of the measured time stamps delivered by the Jiffies timer. The Jiffies timer is a coarse timer which ticks, depending on the kernel compilation configuration, at a rate of 1ms or 4ms. This timer value is even not being used as raw entropy source. Yet, the Jiffies timer data is used to calculate the heuristic entropy value awarded to an entropy event for human interface devices or block devices. This statement already allows the conclusion that the heuristic entropy estimation based on the Jiffies timer has little relationship with the actual entropy delivered with the recorded events. The calculations provided in chapter 6 outlines that the Linux-RNG significantly underestimates entropy. However, as the heuristic entropy estimation has hardly any relationship with the actual entropy, it is considered to be coincidental that the Linux-RNG underestimates entropy. As the entropy sources whose heuristic entropy value is calculated based on Jiffies are all a “derivative” of interrupts, the solution to this issue is to simply discard these events as noise and re-architect the interrupt noise source such that its entropy estimation has more resemblance with reality. Such approach is taken with the handling of the event data in the LRNG – see the processing of the raw noise data outlined in [LRNGREPLACEMENT] figure 2.1.

The extraction function `extract_entropy` is unnecessarily complex because it implements a pseudo-random function (PRF) to extract arbitrary long bit sequences from the input pool. Conceptually it works like HKDF with an expand and extract function. Yet, all callers of the function only request 32 bytes, exactly one Blake2s block. Thus, the extract operation will always be invoked only once. Thus the PRF behavior of extracting arbitrary bit sequences is never used.

`add_timer_randomness` uses the function `_credit_init_bits` which performs entropy accounting even after the entropy pool reached its fully seeded state. The entropy accounting is not used after reaching the fully seeded state which implies that this call performs operations that are not needed.

The use of SipHash/HalfSipHash 1-0 for the fast_pool has an unspecified security strength. Yet, it is estimated that its security behavior is better compared to the old fast_pool. The author of this study acknowledges that neither the input nor the output of the SipHash function is visible to attackers. Thus, the security strength is irrelevant to the discussion. However, it is most important that the SipHash operation does not destroy entropy in its compression of the input data. Yet, it is unclear which impact SipHash has on entropy of the input data. Considering that at least sixty four 64-bit time stamps whose entropy is in the least-significant bits are compressed and that the resulting data is awarded one bit of entropy, it is assumed that SipHash does not destroy entropy to the extent that eventually less than one bit of entropy of the input data remains. Nonetheless, this is an assumption only, that is not verified by this analysis. Furthermore, crediting only one bit of entropy for at least 64 time stamps which each has more than one bit of entropy indicates the very conservative approach of the Linux-RNG.

The changes to the handling the entropy estimator implies that the entire concept of waking up user-space entropy providers when entropy runs low is now completely obsolete. Except during early boot time, it makes no sense for user space entropy providers to select(2) on /dev/random and expect to be woken up when a reseed is needed. All that such entropy-providers can now do is to periodically wake up and insert data without relying on the kernel to inform them. This is a conceptual departure that may not be known to all such entropy daemons and thus those entropy daemons may now behave differently than intended by their developers.

The Linux-RNG does not have any effective entropy source health test any further. Before the entropy heuristic for the HID and block devices could be considered as a form of health test because the input pool did not provide any data if the input data was identified to be poor and thus not having any entropy. With the current implementation, there is no effective detection of a temporary or permanent break in the noise source any more.

Considering the mentioned concerns it is not clear why the re-implementation of the Linux-RNG partially appears in stable kernels like 5.18.1 or is backported to stable kernels of 5.15.44 and other long-term-support (LTS) kernels. None of the changes address a security or safety critical aspect of the Linux kernel. This marks a significant departure of how stable kernels are defined: they only receive patches that address weaknesses or bugs.

5 Coverage of BSI Requirements NTG.1 and DRG.3

The functionality classes of NTG.1 and DRG.3 are defined in [AIS2031], sections 4.10 and 4.8, respectively. The current chapter lists all requirements of the respective functionality classes and compares them with the implementations found in the Linux-RNG.

The analysis demonstrates the following: The implementation of the ChaCha20 DRNG feeding /dev/random, /dev/urandom, the `getrandom` system call, and the in-kernel `get_random_bytes` API complies with the requirements of DRG.3 if the constraints outlined in section 5.2.1 apply. When a suitable replacement for the seed source discussed in section 5.2.1 is found, this conclusion applies to other hardware architectures as DRG.3 defines procedural requirements only.

5.1 NTG.1 Compliance

Starting with kernel version 5.18, the input pool of the Linux RNG does not comply with NTG.1 any more because the input pool does not apply any limitation when insufficient entropy is present. This verdict applies also to kernel versions to which these changes released with 5.18 are backported to. For example, 5.15.44 and later also lost its NTG.1 compliance.

5.2 ChaCha20 DRNG: DRG.3

The ChaCha20 DRNG backing /dev/random, /dev/urandom, the `get_random_bytes` API and the `getrandom` system call is analyzed to whether they comply with DRG.3 in this section.

This section analyzes the primary and the secondary ChaCha20 DRNGs separately as both have different characteristics.

Starting with 5.18, the Linux RNG does not comply with DRG.3 any more except for an edge use case because the secondary ChaCha20 DRNGs seed from a primary ChaCha20 DRNG that is not guaranteed to receive fresh entropy when such reseed operations are performed. This means a chaining of DRNG instances is applied without ensuring proper reseeding of the initial DRNG.

5.2.1 DRG.3.1

The requirement of DRG.3.1 is defined as:

“If initialized with a random seed [selection: using a PTRNG of class PTG.2 as random source, using a PTRNG of class PTG.3 as random source, using an NPTRNG of class NTG.1 [assignment: other requirements for seeding]], the internal state of the RNG shall [selection: have [assignment: amount of entropy], have [assignment: work factor], require [assignment: guess work]].”

The seeding of the ChaCha20 DRNG allows the use of CPU-based noise sources which contribute entropy. These noise source, however, cannot be analyzed or tested and thus must be assumed to have zero bits of entropy for this discussion. Therefore, to ensure that no entropy is assigned to these noise sources, the following kernel command line options must be set:

- `random.trust_cpu=0` or the kernel compile-time option of `CONFIG_RANDOM_TRUST_CPU` must be disabled, and
- `random.bootloader=0` or the kernel compile-time option of `CONFIG_RANDOM_TRUST_BOOTLOADER` must be disabled.

To seed the base ChaCha20 DRNG from the input pool after reaching the fully seeded state (which marks the unblocking of the blocking interfaces), 256 bits of entropy are collected by the latter and then used to seed the ChaCha20 DRNG. Using the SP800-90B min-entropy estimation measured for the interrupt events in sections 6.3.1 and 6.3.3, each interrupt event delivers more than 2 bits of entropy (using the lowest SP800-90B value from the referred sections). This implies that when receiving 256 interrupts, the Linux-RNG has received at least 512 bits of entropy. Thus for the base ChaCha20 DRNG, the following selection of DRG.3.1 above can be applied: "the internal state of the RNG shall have almost 256 bits of entropy".

The in-kernel API call of `register_random_ready_notifier` allows registering callback functions by kernel subsystems. When using this API, the registered kernel subsystem's callback function is invoked after the ChaCha20 DRNG is initially seeded with 256 bits of entropy. Thus, when using the API call to register a callback, the aforementioned statements about the amount of entropy present in the ChaCha20 DRNG equally apply after the callback functions have been triggered.

Similarly, the use of the in-kernel service function of `wait_for_random_bytes` ensures the caller is suspended until the ChaCha20 DRNG is fully seeded. It therefore is identical to the `register_random_ready_notifier` function behavior with the exception that it provides a synchronous waiting.

Contrary, due to the lack of initial seeding enforcement, the following methods of using the ChaCha20 DRNG are not DRG.3.1 compliant:

- using /dev/urandom,
- using `get_random_bytes` either before the callback functions registered with the API of `register_random_ready_notifier` has been triggered or using `get_random_bytes` without registering a callback at all, and
- using `get_random_bytes` before the service function of `wait_for_random_bytes` returns.

In case only one CPU is present in the system the base ChaCha20 only provides data to one secondary ChaCha20. The primary ChaCha20 is not otherwise usable. In this case, the data extracted from the secondary ChaCha20 DRNG via the output interfaces like /dev/random or the `getrandom` system call are considered data from a DRG.3.

In case more than one CPU is present in the system (which common on contemporary systems), the base ChaCha20 DRNG reseeds multiple secondary ChaCha20 instances – one per CPU – without guaranteeing a fresh reseed from the input pool. The output from the multiple secondary ChaCha20 DRNGs is not considered to be DRG.3 compliant. As the base ChaCha20 DRNG is not accessible by any caller, the conclusion is that the Linux-RNG executing on a multi-CPU system is not DRG.3 compliant.

Thus the final conclusion is that on common contemporary systems, the Linux-RNG is not meeting DRG.3.1 requirements as the secondary ChaCha20 DRNG is not guaranteed to be initially seeded with 256 bits of entropy.

Considering that the secondary ChaCha20 DRNG is always seeded from a fully seeded primary DRNG, this non-conformity, however, is not considered to be a cryptographic weakness.

5.2.2 DRG.3.2

The requirement of DRG.3.2 is defined as:

"The RNG provides forward secrecy."

The forward secrecy is guaranteed by the ChaCha20 DRNG as follows: The ChaCha20 DRNG maintains an internal state which holds a key that is unknown to the caller. Furthermore, the ChaCha20 DRNG increments the counter by one after each generated block. Assuming that the ChaCha20 block function is

irreversible for an observer that does not have access to the ChaCha20 state with its key, a caller cannot deduce subsequent random numbers from his obtained random number.

Furthermore, ChaCha20 is resistant against determining the used key by assessing the already generated random numbers.

These properties therefore guarantee forward secrecy.

Therefore, the ChaCha20 DRNG complies with the requirements of DRG.3.2.

5.2.3 DRG.3.3

The requirement of DRG.3.3 is defined as:

"The RNG provides backward secrecy even if the current internal state is known."

The first block generated by an instantiated ChaCha20 block operation is used as a new key for the next ChaCha20 block operation instance. The currently used key is destroyed which means that the current key is not present any more except in the instantiated ChaCha20 block operation which is also securely erased from memory once the request for random numbers is satified. Assuming that the ChaCha20 block function is irreversible without the key, an attacker cannot deduce the previous state used to generate previous random numbers via the ChaCha20 block operation even when the current ChaCha20 state is known to the attacker.

Therefore, the ChaCha20 DRNG complies with the requirements of DRG.3.3.

5.2.4 DRG.3.4

The requirement of DRG.3.4 is defined as:

"The RNG, initialized with a random seed [assignment: requirements for seeding], generates output for which [assignment: number of strings] strings of bit length 128 are mutually different with probability [assignment: probability]."

The ChaCha20 DRNG state has a size of 256 bits equal to the key size of ChaCha20. In a worst-case scenario when the state has just been fully seeded with fresh entropy it can generate random bits depending on the performance of the CPU within 60 seconds before it reseeds with fresh entropy if present which allows the generation of large amounts of random data backed by little entropy only in a worst case.

To generate a bit string of 128 bits, the read operation of the ChaCha20 DRNG performs one ChaCha20 block operation.

In the ideal case the generated bit strings exhibit an equidistribution. Considering the birthday paradox, this implies that after 2^{64} blocks of 128 bits each have been generated, probably the following collision is present:

$$P(Collision\ after\ 2^{64}\ blocks) \approx 0.3935.$$

The probability that after generating n 128 bit blocks no collisions are present can be calculated as follows. The number of possibilities for the output of n pairwise different bit strings of length 128 bits is:

$$A = 2^{128} \cdot (2^{128} - 1) \cdot \ldots \cdot (2^{128} - n + 1)$$

Therefore, the probability that there are no collisions after the generation of n blocks results in

$$P(n) = \frac{A}{(2^{128})^n}$$

Instead of using the Stirling formula, an easier estimation of the lower boundary for the probability P is provided as follows. This rough estimation can be used due to the presence of large numbers:

$$A = 2^{128} \cdot (2^{128} - 1) \cdot \ldots \cdot (2^{128} - n + 1) > (2^{128} - n + 1)^n$$

Using this rough estimation formula, a lower boundary for the probability P can be obtained using the following easy to process formula:

$$P(n) > \left(\frac{2^{128} - n + 1}{2^{128}}\right)^n$$

Using this formula, a probability can be calculated that 2^{55} successive bit strings of size 128 bits are pairwise different with a probability of P > 0.999996.

Stating the obtained results differently, $k > 2^{55}$ bit strings of size 128 bits can be generated where no collisions occur with a probability of $P > 1 - \varepsilon$, with $\varepsilon = 3.8e\text{-}6$. This means that with the given probability, the bit strings are pairwise different. These values should be put into perspective with the requirement of [AIS2031] for AVA_VAN.5 with $k > 2^{34}$ and $\varepsilon < 2^{-16}$.

Using the formula with $n = 2^{64}$ the following estimate can be obtained for the probability of having no collisions (i.e. the bit strings are pairwise different):

$$P(\text{no collisions after } 2^{64} \text{ blocks}) > \frac{1}{e} \approx 0.3678.$$

Comparing this value with the precise probability using the initially stated probability for collisions of 0.3935, the probability for having no collisions is 1 − 0.3935 = 0.6065. Comparing this value with the estimated value using the estimation formula it can be concluded that the probabilities P(n) in reality are significantly higher than the ones calculated with that formula. This means that significantly more than 2^{55} bit strings with a length of 128 bits will be pairwise different with a probability of $P > 1 - 2^{-16}$. Therefore, it can be concluded that random bits obtained from the input pool are resistant against an attacker with high attack potential.

To apply the findings to the Linux-RNG ChaCha20 state it can be concluded that the behavior of the that state comes close to the ideal case:

- The ChaCha20 state has a size of 256 bits. The pre-image for each ChaCha20 value is therefore large, which implies that the assumption of an equidistribution must be considered appropriate. However after each ChaCha20 block generation in the worst case only one bit in the instantiated ChaCha20 block operation is modified (the counter is incremented by one) in order to generate the next block.
- ChaCha20 was subject to significant assessments considering it became [RFC7539] that have shown that generated ChaCha20 blocks are a close approximation of an equidistribution supported by the avalanche effect.
- Between the reseeds, the quality of the random numbers rests on the quality of the ChaCha20 block operation. Given that the block operation follows an equidistribution, this resting foundation is considered satisfied.

Therefore, the ChaCha20 DRNG complies with the requirements of DRG.3.4.

5.2.5 DRG.3.5

The requirement of DRG.3.5 is defined as:

“Statistical test suites cannot practically distinguish the random numbers from output sequences of an ideal RNG. The random numbers must pass test procedure A [assignment: additional test suites].”

Section 8.1 provides a rationale for the execution of the Test Procedure A defined in [AIS2031].

Additional statistical tests are applied as covered in section 8.1, which documents the statistical methods applied to the output of the ChaCha20 DRNG. In addition, all tests conducted in chapter 6 and following can be considered to support the stated requirement.

Considering section 8.1, the assignment of the requirement can be specified as: "... as well as the dieharder test suite, the Chi-Squared test and the test of compressing the generated data with gzip, bzip2, xz and lzma".

This allows the conclusion that the ChaCha20 DRNG complies with the requirements of DRG.3.5.

6 Test Series: Raw Entropy

The test series documented in this chapter cover the analysis of the output of the noise sources depicted on the lower part of figure 2. The tests are devised so that the unprocessed data recorded from the noise sources are measured and obtained for this analysis.

The noise sources with their generated data are described in section 3.5.2. This section also outlined that only a subset of the noise sources provide data which is assigned an entropy estimate. The following sections only perform an assessment of the noise sources with an entropy estimate. All other noise sources mix the entropy pools but do not affect any conclusions drawn in chapter 5 regarding the NTG.1 or DRG.3 assessment of the Linux-RNG. One exception is to be noted: although hardware random number generators can contribute entropy, they are considered specialized hardware which is not present in common hardware systems. Furthermore, any assessment requires further analysis of the design of these hardware random number generators. As they are commonly proprietary, such information is not publicly available preventing a full analysis.

This study attempts to deliver a conservative analysis that should be applicable to a large array of systems and use cases. Therefore, if data received from a noise source has questionable entropy content, this study assumes a worst-case scenario where the data is assumed to contribute no entropy to the Linux-RNG.

The entropy analysis including the reboot testing is also conducted in compliance with SP800-90B. The reason for this approach is that the raw entropy data is not identically and independently distributed (non-IID) which implies that the Shannon entropy plug-in estimator and the min-entropy plug-in estimator allow only limited conclusions to be drawn. The mathematical test analysis provided by SP800-90B together with the test tool take the non-IID property into consideration and therefore is considered to be an appropriate fit for this testing.

6.1 Analyzed Noise Source Data

Before the analyses of the data from the noise sources are conducted, the noise sources are again discussed regarding their produced data and the relevance of that data concerning entropy.

6.1.1 Interrupt Noise Source

As outlined in section 3.5.2.2, the noise source of Interrupts collects different data for each event. Based on the following considerations, the implied entropy in the data parts varies significantly:

- The Jiffies time stamp recorded for one interrupt commonly has a resolution of 1000 Hz. Interrupt occurrence can be observed by monitoring /proc/interrupts which contains the number of interrupts received for each interrupt in real time. The corresponding number is incremented as soon as a new interrupt is processed. Considering that an attacker is able to monitor that file and that the increment of the numbers in that file happens as soon as an interrupt arrives, it is assumed for this study that the Jiffies value awarded for a respective interrupt by the Linux-RNG can be obtained with full accuracy by an attacker. This implies that for a worst-case scenario no entropy would be delivered with the Jiffies value. Therefore, this Jiffies value will not be further analyzed and is considered to deliver no entropy by this study.
- In addition to the Jiffies value, the Linux-RNG records the instruction pointer and the content of one of the registers. This data varies depending on the type of interrupt. Yet, for one given interrupt it is assumed that these values are predictable. The instruction pointer is constant for a given interrupt. The registers may change depending on the recorded data by the hardware device. As the hardware device may store data that can be deducted by an attacker, such as memory addresses where hardware event

information is found, the study is conservative and treats the data obtained from the registers and the instruction pointer as having no entropy. Consequently, such data will not be analyzed.

- Finally, the interrupt noise source records the 32 LSB of the high-resolution time stamp. Albeit the issue discussed for Jiffies affects also the high-resolution time stamp, it is of no concern due to the following. The high-resolution time stamp has a resolution of nanoseconds. When observing hardware events or /proc/interrupts, an attacker must be able to deduce the nanosecond value obtained by the Linux-RNG for a given interrupt with a high degree of precision. The degree of precision the attacker must apply to deduce the time stamp value must be higher than the entropy awarded to the event by the Linux-RNG. In other words, if an attacker can deduce the used time stamp with a precision of, say, 2 bits (i.e. the attacker's uncertainty is only 2 bits), but the Linux-RNG would award this event more than 2 bits, the Linux-RNG would overestimate the available entropy. As the Linux-RNG awards between 1 and 64 interrupts one bit of entropy, a single interrupt is implied to have between 1 and 1/64th bit of entropy. Thus, the attacker must deduct the high-resolution time stamp with full accuracy if he wants to undermine the entropy estimation of the Linux-RNG. Even when he cannot deduct the last bit with a precision better than the random chance of a half, the best attack against the noise source of the interrupts is brute force. Therefore, the high-resolution time stamp is considered for further entropy analysis.

6.1.2 Block Device Noise Source

Sections 3.5.2.3 and 3.5.2.10 outlines the data obtained by the Linux-RNG for one block device event. Just as for the interrupt noise source, the following list discusses each data component regarding its entropy contribution:

- With the function `add_disk_randomness`, the block device number that triggered the event is recorded. Hardware commonly has one block device attached, i.e. one hard disk is attached. Therefore, this value will always be the same for each event. Even with two or more hard disks, an attacker can trigger block device events on each disk separately. Hence, no or hardly any entropy must be considered present with the block device number. Thus, the study will disregard this value for the entropy analysis.
- The function `add_timer_randomness` adds the high-resolution time stamp to each block device event. Albeit an attacker can cause block device events, with the high resolution of the time stamp of nanoseconds, it is considered to be impossible to deduct the precise timing of the block device event at this resolution, i.e. the attacker would not be able to deduce the LSBs of the time stamp with a precision higher than the entropy awarded to the event by the Linux-RNG. Hence, this study will focus on the assessment of the high-resolution time stamp for block device events.

6.1.3 HID Noise Source

The HID noise source delivers data as discussed in sections 3.5.2.1 and 3.5.2.10. Again, the following list provides a rationale why data components are included or excluded from the entropy assessment:

- The function `add_input_randomness` records the event number processed by the HID. For example, a keyboard records the key number and whether the key was pressed or released. For a mouse, commonly two coordinates for the two-dimensional movement are recorded. All these values are considered observable by an attacker. This is particularly the case when using the graphical interface of X11. As long as an attacking process can interact with the X11 server by having the X11 cookie, the X11 input facility can be misused to enable a perfect key logger without the need to possess any privilege[13]. A

13 To invoke such perfect key logger, the following command can be used:
```
xinput list | grep -Po 'id=\K\d+(?=.*slave\s*keyboard)' | xargs -P0 -
n1 xinput test
```

similar command can be used to obtain mouse movement data. This implies that the HID event data must be assumed to have no entropy in the worst-case. Thus, no analysis is performed for this data.

- Like for `add_disk_randomness`, `add_input_randomness` invokes `add_timer_randomness` to add the high-resolution time stamp for each HID event. The same considerations as outlined in section 6.1.2 apply to HID events. Therefore, the high-resolution time stamp is subject for further analysis.

6.1.4 Scheduler-Based Noise Source

The noise source may deliver entropy during boot time as discussed in section 3.5.2.8. The description of the scheduler-based noise source given in section 3.5.2.8 outlines that it may be used during boot time but never during runtime. The gathering of raw entropy data takes this use case into account and only performs the analysis of the entropy obtained during boot time including the testing of the data during reboots as provided with section 6.3.

The entropy is obtained by reading the high-resolution time stamp during a tight loop that is interrupted by a re-scheduling event. Thus, this time stamp is the raw noise data to be analyzed. The extraction operation gathers the 32 LSB of that time stamp to be in line with the data gathered for the other noise sources.

6.2 Min-Entropy Estimation as per SP800-90B

The discussions of the noise sources in section 6.1 concludes that solely the high-resolution time stamp used for each event is of relevance to the entropy analysis.

The high-resolution time stamp is recorded using a kernel patch exporting the kernel-collected time stamp.

To extract the raw noise data from the kernel during run time, a new kernel facility is added that creates one DebugFS file per noise source. In addition, for each noise source, that facility maintains one ring buffer of 1024 32-bit integer values for each noise source. The ring buffer handling is performed with a reader and writer function. The writer function ensures that the caller-provided 32-bit integer is written sequentially into the ring buffer guaranteeing that any existing data is not overwritten while the gathering operation is in progress. The reader operation reads the ring buffer sequentially guaranteeing that at most it reads the data up to the point where the writer operation stopped.

The writer of the ring buffer is invoked at well-defined places from the Linux-RNG as described below. The reader operation is linked with the DebugFS files.

Only when a read operation is in progress, the writer operation stores data into the ring buffer. This guarantees that only current raw entropy data is obtained via the DebugFS files.

In addition, the raw noise data gathering facility can also handle early boot data with the same implementation. The only difference is that the writer function will store any data it obtains until the ring buffer is full. This implies that the writer function stores even the first entropy event data during boot. The reader function allows reading of the ring buffer that was filled during boot time but not altered afterwards.

Specifically, this test framework records the following data:

- HID measurement: to measure the high-resolution time stamp of HID events, the test framework instruments add_timer_randomness to read out the high-resolution time stamp from the `sample` data structure (see section 3.5.2.10 for details about this data structure). It takes the 24 LSB of that time stamp. These 24 bits are concatenated with an 8 bit integer of the heuristic entropy value awarded to this event. The resulting data is a 32 bit integer that is given to the entropy recording facility for pick-up by user space.

- Block device measurement: the test framework is used to record the same data for block devices as outlined for HID devices above.
- Interrupt measurement: the test framework instruments `add_interrupt_randomness`. It obtains the high-resolution time stamp for each received interrupt. The full 32 LSB of the time stamp is extracted for user space without recording any heuristic entropy information as the Linux-RNG applies a fixed estimate of one bit per injection of a fast_pool content into the input pool.
- Scheduler-based entropy measurement: The testing is implemented by modifying the entropy harvesting function of `try_to_generate_entropy` as follows: The loop that performs the harvesting of entropy is executed exactly 1024 times instead until the ChaCha20 DRNG is fully seeded. A global variable is introduced that is incremented each time the timer fires and would increase the heuristic entropy estimator of the Linux-RNG by one. Each time stamp injected into the Linux-RNG is extracted for user space to be picked up. To start the testing, an automatic trigger is added to initiate the scheduler-based noise source during the late stage of the kernel boot, but before user space starts. Note, this trigger is required as the scheduler-based noise source is only initiated if user space queries `getrandom` or /dev/random before the ChaCha20 DRNG is fully seeded.

The recorded data set is simply a set of 32 bit integer values holding the high-resolution time stamps for each recorded interrupt. A script is used that triggers the testing to obtain data for 1,000,000 noise source events.

The resulting data for the high-resolution time stamp is analyzed for its min-entropy estimation as defined in [SP800-90B]. In order to perform the calculations, the type of data to be processed must be determined, i.e. whether the input data is IID or non-IID. With a time stamp value, even when it is fast moving and thus wrapping within some seconds, it is still a monotonically increasing counter. Therefore, this data set is always considered to be non-IID. This determination implies that the following types of min-entropy estimations are calculated defined by [SP800-90B]:

- Most Common Value Estimate
- Collision Estimate
- Markov Estimate
- Compression Estimate
- t-Tuple Estimate
- Longest Repeated Substring (LRS) Estimate
- Multi Most Common in Window Prediction Estimate
- Lag Prediction Estimate
- MultiMMC Prediction Estimate
- LZ78Y Prediction Estimate

As documented in [SP800-90B] almost all of these min-entropy estimations can only be calculated for input data that has a small width. A high-resolution time stamp has a width of 32 bits (interrupts) or 24 bits (HID / block devices), respectively. To allow processing the time stamps with the aforementioned min-entropy estimation calculations, the test tools obtain the 8 least significant bits of the time stamp and concatenates those 8 bits of all time stamps into a bit stream. This means that the input data width is now 8 bits instead of 32 bits. The calculation of the SP800-90B min-entropy estimations using 8 bits instead of 32 bits is considered to support the conservative assessment of this study. The following tables therefore provide the entropy estimation for 8 bit input data widths. The tool used to calculate the SP800-90B min-entropy estimation is available at NIST GitHub repository[14].

14 https://github.com/usnistgov/SP800-90B_EntropyAssessment

For comparison, plug-in estimates for the min-entropy and Shannon entropy based on the empirical distribution are calculated as well. The used formulas are provided e.g. in section 2.3.2 of [AIS2031] and are not re-iterated here. The time stamp is a monotonically increasing integer which implies that the entropy lies in the deltas of the time stamps and the distribution of those deltas. This means that to perform the calculation for the nin-entropy and Shannon entropy, the time stamp deltas are used as a basis for the calculation. The time stamp deltas are calculated from the adjacent time stamps from the absolute time stamps recorded by the measurements.

The calculation of the min-entropy and the Shannon entropy is also performed for the 8 LSB of the time stamps to allow an immediate comparison of all values. However, the following additional consideration is applied: neither the min-entropy nor the Shannon entropy estimates are applicable to non-IID data. Time stamps as delivered by the different noise sources are a monotonically increasing counter value which wraps when reaching its maximum. This monotonically increase is a dependency that can be removed by calculating the first discrete derivation, i.e. the time delta between adjacent values. Thus, before applying the formulas for the Min-Entropy and the Shannon entropy, the time stamps are processed as follows:

1. Calculate the delta between two adjacent time stamps.
2. Obtain the 8 LSB from the time deltas.

The resulting 8 LSB are used to obtain the Shannon entropy and the min-entropy plug-in estimates.

6.2.1 Interrupt Noise Source Entropy Estimation

The collection of data for interrupts was conducted with a worst-case approach. Considering that the entropy estimate is fixed irrespective of the raw noise data, the worst case testing is intended to analyze that the entropy estimate is always appropriate, i.e. underestimating the available entropy. The worst-case covered the approach where a system in close network proximity, i.e. the network switch sent a ping flood to the test system. Each received ICMP request and response triggered an interrupt that was recorded.

The worst-case test execution returned the following data.

Entropy Estimation Type	Entropy Estimate
Most Common Value Estimate	7.256857
Collision Estimate	7.330672
Markov Estimate	7.93132
Compression Estimate	5.590176
t-Tuple Estimate	6.520281
LRS Estimate	7.522373
Multi Most Common in Window Prediction Estimate	6.638383
Lag Prediction Estimate	6.638405
MultiMMC Prediction Estimate	5.306248
LZ78Y Prediction Estimate	5.306244

Table 2: Interrupts: SP800-90B Min-Entropy Estimates Worst Case

The associated Shannon entropy plug-in estimate is 7.97 bits per interrupt event. The nin-entropy plug-in estimate is 6.56 bits per interrupt event.

The conclusions that can be drawn from the numbers are the following. The high-resolution time stamp of each interrupt will return more than three bits of entropy.

The Linux-RNG requires the data of at least 1 interrupt to be collected and mixed into the input pool. The entire data from between 1 to 64 interrupt is credited with one bit of entropy during boot time. This implies that significantly more entropy is collected than the Linux-RNG will credit.

Even when the fast_pool operation will not retain all entropy delivered by the interrupt noise source data, the massive underestimation of entropy by the Linux-RNG is assumed to counter such a potential effect.

As the Linux-RNG massively underestimates the entropy present in the interrupt noise source event data, the Linux-RNG acts conservatively and thus upholds the cryptographic strength it reports with its entropy estimation.

During runtime of the Linux kernel after the Linux-RNG became fully seeded until a new reboot, however, the kernel does not credit entropy to the interrupt data. Therefore, no conclusions can be drawn from this data at runtime.

6.2.2 Block Device Noise Source Entropy Estimation

On contemporary hardware with a lot of RAM, a normal usage of block devices will cause insignificant block device events. This is due to the fact that the entire unused portion of RAM is used as a buffer cache to prevent repeating disk accesses. To obtain sufficient data, a worst-case has been measured. This worst-case has been implemented by constantly writing 10 MB of data onto a block device where the file is opened with O_SYNC causing the bypassing of the buffer cache. The worst-case produced the following data:

Entropy Estimation Type	Entropy Estimate
Most Common Value Estimate	7.890384
Collision Estimate	7.544968
Markov Estimate	7.9964
Compression Estimate	7.208808
t-Tuple Estimate	7.890384
Longest Repeated Substring (LRS) Estimate	7.928188
Multi Most Common in Window Prediction Estimate	7.949105
Lag Prediction Estimate	7.955759
MultiMMC Prediction Estimate	7.977489
LZ78Y Prediction Estimate	7.976726

Table 3: Block Devices: SP800-90B Min-Entropy Estimates – Worst Case

Using the Shannon entropy plug-in estimate, 8 bits per block device event is calculated. A value of 7.941 bits per block device event is calculated with the min-entropy plug-in estimate.

During runtime of the Linux kernel after the Linux-RNG became fully seeded until a new reboot, however, the kernel does not credit entropy to the block device data.

Similar to the interrupt noise source, a “normal use case” test is performed with the block device noise source. This normal case consisted of several Linux kernel compilation runs, some wait time inbetween, starting of various user applications like a browser or word editor.

The following data for the normal use case is collected:

Entropy Estimation Type	Entropy Estimate
Most Common Value Estimate	7.835243
Collision Estimate	8
Markov Estimate	7.974104
Compression Estimate	6.914048
t-Tuple Estimate	7.333766
Longest Repeated Substring (LRS) Estimate	7.718814
Multi Most Common in Window Prediction Estimate	7.863313
Lag Prediction Estimate	7.912553
MultiMMC Prediction Estimate	7.900896
LZ78Y Prediction Estimate	7.901578

Table 4: Block Devices: SP800-90B Min-Entropy Estimates – Normal Use Case

The Shannon entropy plug-in estimate shows 7.999 bits per block device event. 7.888 bits per block device event are calculated with the min-entropy plug-in estimate.

The measured values show that sufficient entropy is present. However, as the Linux-RNG does not perform any entropy heuristics at runtime, no conclusions can be drawn from them.

6.2.3 HID Noise Source Entropy Estimation

The entropy measurements for HID is only performed for regular use cases. No worst-case scenario can be devised for HID.

To perform testing of the HID noise source within a reasonable time, a small but effective test system was devised: a mouse was placed on a loop-sided surface. The cable of the mouse was connected to a mobile office fan which swings its fan. Due to the movement of the fan, the mouse was moved as well in a regular fashion. The quite regular movement can be considered as a form of worst-case since a normal user would not move a mouse in a regular fashion for long hours. The entropy estimates for the high-resolution time stamp applied to those events are listed in the table below.

Entropy Estimation Type	Entropy Estimate
Most Common Value Estimate	7.883070
Collision Estimate	7.451912
Markov Estimate	7.982736
Compression Estimate	7.200336
t-Tuple Estimate	7.295390
LRS Estimate	7.892103
Multi Most Common in Window Prediction Estimate	7.961163
Lag Prediction Estimate	7.740675
MultiMMC Prediction Estimate	7.688495
LZ78Y Prediction Estimate	7.688474

Table 5: HID: SP800-90B Min-Entropy Estimates

The Shannon entropy plug-in estimate applied on the data set results in 7.979 bits per HID event. 7.627 bits per HID event are calculated when using the min-entropy formula.

Just like the disk noise source, the HID noise source is not credited with entropy at runtime of the Linux kernel after the Linux-RNG became fully seeded until a new reboot. The measured values show that sufficient entropy is present. However, as the Linux-RNG does not perform any entropy heuristics at runtime, no conclusions can be drawn from them.

6.2.4 Conclusion of SP800-90B Measurements

The conclusions given for each noise source regarding the SP800-90B measurements are collectively summarized as follows.

For all noise sources that contribute entropy to the Linux-RNG, the Linux-RNG applies no heuristic entropy estimation during runtime of the Linux kernel after the Linux-RNG became fully seeded until a new reboot. It simply reseeds the DRNG from the available data, irrespective whether it is good or bad or non-existent data.

Therefore, the Linux-RNG cannot overestimate the available entropy.

During boot time, the Linux-RNG applies 1 bit of entropy per interrupt. This is a significant underestimation of available entropy which implies that when reaching the fully seeded level, the input pool received significantly more than 256 bits of entropy. The measurements show that on the fully seeded level was reached before block devices or HID devices became available. Thus, the entropy contribution from those devices with their entropy heuristic based on Jiffies was not applied by the Linux-RNG.

However, this verdict leads to the conclusion that no real statements about the entropy behavior of the Linux-RNG can be given. If entropy events are received, they are used to seed the DRNG. If no or very little events were received, the DRNG still reseeds from the input pool that received little to no entropy.

With the lack of any health test, the Linux-RNG cannot detect a degradation of is entropy source. This issue aggravates the conclusion that the entropy behavior during runtime cannot be determined.

6.3 Entropy During Early Boot

The measurements of the raw noise source data shows that at runtime, the Linux-RNG entropy estimator maintained for an entropy pool indicates at least the cryptographic strength of the data present in that entropy pool.

At runtime, when sufficient data is added to the entropy pool, the Linux-RNG state is always considered to be sufficiently strong.

However, the following question must be raised: are the noise source data received by the Linux-RNG during early kernel boot time equally entropic to support cryptographically strong random numbers to be produced by the Linux-RNG during boot time? This question is of particular importance to system services requiring seed data from /dev/random or /dev/urandom during system boot time.

The following test has been devised to measure the entropy during early boot. This test considers that during early boot, only interrupts are triggered and received. No block device is yet set up, and no HID is initialized to allow users to interact with the system. Therefore, testing is limited to measure interrupt event data as well as the scheduler-based noise source. As outlined in section 6.1.1, only the high-resolution time stamp recorded for interrupts is of interest to entropy measurements. In addition, the scheduler-based noise source is relevant during boot time as well.

The Linux kernel has been modified with the test framework discussed in section 6.2. The explanation of that test framework shows that it is equally applicable to measure the boot time raw unconditioned noise data. To be in line with [SP800-90B] section 3.1.4, the test framework obtains the data from the first 1000 interrupt events as well as the first 1000 scheduler-based noise source events.

The test is performed for 1,000 reboot cycles for the virtual environment as well as for the bare-metal environment.

The first analysis performs a row-wise and column-wise SP800-90B min-entropy estimate calculation discussed in [SP800-90B] section 3.1.4. In addition, the sanity test outlined in [SP800-90B] section 3.1.4 is calculated as well.

The testing of the early boot entropy is conducted twice due to its importance. The first test is performed in a virtualized environment. This environment has very few devices that can trigger interrupts. This means that the time until 1000 interrupts are received is longer relative to the boot time of the Linux kernel. Yet, more variations must be expected as the virtual machine monitor may reschedule the virtual machine guest that is tested. Such rescheduling operations may introduce delays which would be visible with more variations in the time stamps. The second early boot entropy test is executed with a Linux kernel executing directly on hardware. This hardware has more devices that can deliver interrupts. Yet this test environment is not affected by virtual machine monitor rescheduling events.

The calculation of the 8 LSBs from the time delta for the Shannon entropy and min-entropy as discussed in section 6.2 is applied to the row-wise time stamp data. For the column-wise , the 8 LSB of the time stamps without the delta calculation are used with the formulas for the Shannon entropy and min-entropy. The reason is that there is no monotonically increasing timer dependency between, say, the first time stamp of an event from one reboot compared to the first time stamp of another reboot.

6.3.1 Early Boot Interrupt Entropy Testing in a Virtual Environment

As outlined in [SP800-90B] section 3.1.4, the data obtained from the reboot tests shall be placed into a matrix with 1000 columns referencing the 1000 consecutive event values obtained from one reboot. Each row in that matrix references one reboot.

The generated matrix is to be processed row-wise and column-wise. Thus, the analysis results in separate conclusions for the row-wise and column-wise assessment.

The SP800-90B entropy estimates are calculated for each time stamp row and column out of the generated matrix.

6.3.1.1 Column-Wise Reboot Data Assessment

When processing the raw data column-wise, for each column, the SP800-90B min-entropy estimates, the Min-Entropy as well as the Shannon-Entropy plug-in estimates are calculated. Instead of listing 1000 values for each aspect, the following table provides the minimum values calculated over all columns.

	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	**Min-Entropy Plug-in Estimate of 8 Bits Width Time Stamp**	**Shannon Entropy Plug-in Estimate of 8 Bits Width Time Stamp**
Minimum entropy estimation	7.719	6.0589	7.737

Table 6: Interrupts: Early Boot SP800-90B Min-Entropy Estimates in Virtual Environment – Column-Wise

The table shows that the high-resolution time stamp of each of the first 1000 interrupts has an SP800-90B min-entropy estimate of 7.7 bits of entropy per interrupt event. Considering the min-entropy plug-in estimate applied to the time deltas (i.e. the difference of two adjacent time stamps), the value 6.0 bits per interrupt event. The Shannon entropy values for the time deltas are even higher and are close to the maximum of 8 bits per interrupt event.

To allow the reader to get a graphical view of the time stamp distribution, figure 8 is provided. Considering the statement above regarding time deltas, such time deltas are used as a basis for the distribution graph instead of absolute time stamps. Therefore, figure 8 shows the time delta distribution of the time stamps recorded for the first and second interrupt – the X-axis presents the number of ticks of the time delta.

The histogram shows that the time delta is widely distributed over the entire continuum of possible time delta values. It shows some concentration of time deltas in the low end of the possible range of time delta values ranging from zero to 2^{32}. The two green bars show the 25% and 75% quartile of the data set.

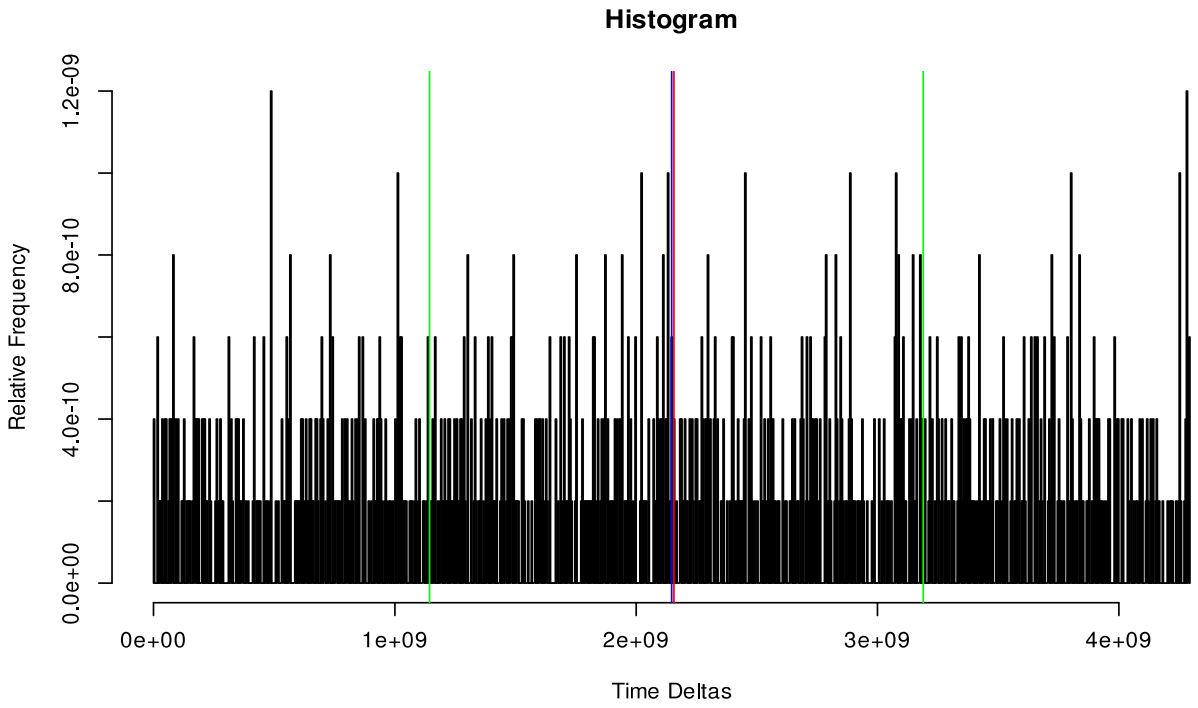

Figure 8: Histogram of Time Deltas for First and Second Interrupt in a Virtual Environment – Column-Wise

The measured time stamps of the first 1000 interrupts visualized in figure 8 allows the conclusion that the entropy present in the time stamps is already sufficiently large for achieving a commonly required security strength of 128 bits. As these first 1000 interrupts are not obtained from block device or HID events, the correlation issue outlined in section 6.2.4 is not applicable. Therefore, the Linux-RNG massively underestimates the boot-time entropy present with the interrupt time stamps.

Finally, [SP800-90B] section 3.1.4 mandates that the minimum value of the column-wise calculated SP800-90B min-entropy estimations must not be less than half of the one obtained during runtime discussed in section 6.2.1. This requirement is verified by the aforementioned NIST tool.

6.3.1.2 Row-Wise Reboot Data Assessment

In addition to the column-wise assessment, [SP800-90B] section 3.1.4 requires a row-wise assessment. With the collection of data from 1,000 reboots, the data set encompasses 1,000 entropy values. Like for the column-wise data assessment, minimum values are provided in the following table.

	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	**Min-Entropy Plug-in Estimate of 8 Bits of Time Delta**	**Shannon Entropy Plug-in Estimate of 8 Bits of Time Delta**
Minimum entropy estimation	6.638	5.964	7.703

Table 7: Interrupts: Early Boot SP800-90B Min-Entropy Estimates in Virtual Environment – Row-Wise

Similarly to the column-wise assessment, figure 9 shows the row-wise time deltas for the first and second event. To make the graphic more readable, only the 90% quantile of the time delta is depicted. The remaining 10% cover such a large value span with so little probability of occurrence that they would render the graphic unreadable.

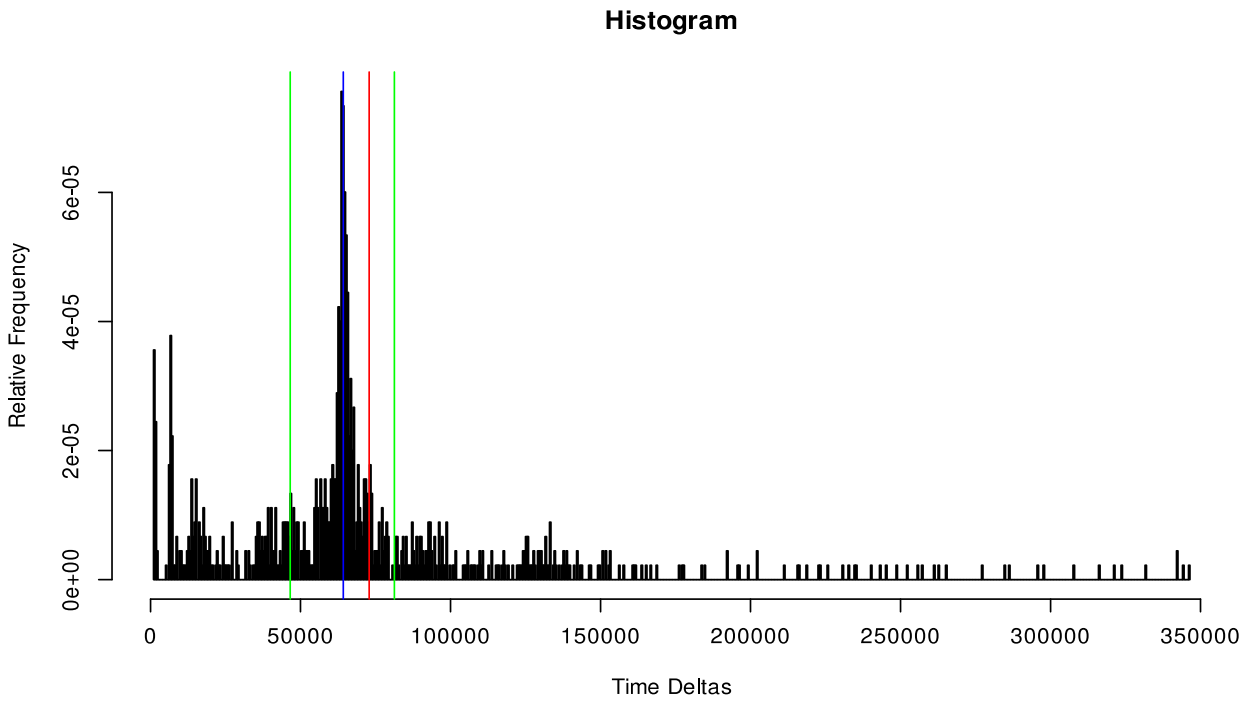

Figure 9: Histogram of Time Deltas for First and Second Interrupt in a Virtual Environment – Row-Wise

Just like for the column-wise entropy values, the minimum SP800-90B entropy estimate for the row-wise analysis must not be less than half of the runtime entropy rate. This again is verified by the NIST SP800-90B entropy assessment tool.

6.3.1.3 SP800-90B Sanity Test

In addition to the row-wise and column-wise entropy assessment, [SP800-90B] section 3.1.4 also mandates a sanity test. To calculate the sanity test, the entire time stamp is used as this sanity test is not intended to provide a lower boundary for the entropy estimate but rather shall verify that the collected data in general is usable.

In addition, the sanity test requires that an anticipated entropy rate is provided for the calculation. The entropy rate expected to be present is at least 1 bit of entropy per time stamp as at least 64 interrupts are required to be gathered before injecting the data into the input pool and increase the entropy estimator by one during boot.

The following data is obtained from the sanity test:

- Maximum number of occurrences of a value: 2
- By using an anticipated entropy of at least 1 bit per time stamp, the sanity test passes.

6.3.2 Early Boot Scheduler-Based Entropy Testing in a Virtual Environment

The scheduler-based noise source that may be active during boot time contributes to the initial seeding of the ChaCha20 DRNG as outlined in section 3.5.2.8. This implies that this noise source is to be assessed in the same way as the interrupt noise source in section 6.3.1. The following subsections therefore apply the same test concepts.

Before showing the measured numbers, an interesting detail must be mentioned that was detected during the data collection: The scheduler-based entropy event data was started in the late kernel boot stage and its entropy collection duration reached into the early user space boot. During the data collection, the Linux kernel never awarded any entropy using its entropy heuristic to the scheduler-based noise source data.

6.3.2.1 Column-Wise Reboot Data Assessment

When processing the raw data column-wise, for each column, the SP800-90B min-entropy estimates, the min-entropy as well as the Shannon-Entropy values are calculated. Instead of listing 1000 values for each aspect, the following table provides the minimum values calculated over all columns.

	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	**Min-Entropy Plug-in Estimate of 8 Bits Width Time Stamp**	**Shannon Entropy Plug-in Estimate of 8 Bits Width Time Stamp**
Minimum entropy estimation	7.870	6.060	7.738

Table 8: Scheduler-Based Noise Source: Early Boot SP800-90B Min-Entropy Estimates in Virtual Environment – Column-Wise

The table shows that the high-resolution time stamp of each of the first 1000 scheduler events has an SP800-90B min-entropy estimate of 7.8 bits of entropy per event. Considering the min-entropy plug-in estimate applied to the time deltas (i.e. the difference of two adjacent time stamps), the values is 6 bits per scheduler event. The Shannon entropy plug-in estimate values for the time deltas are even higher and is close to the maximum of 8 bits per interrupt event.

A graphical view of the time stamp distribution for the first time delta is provided with, figure 10.

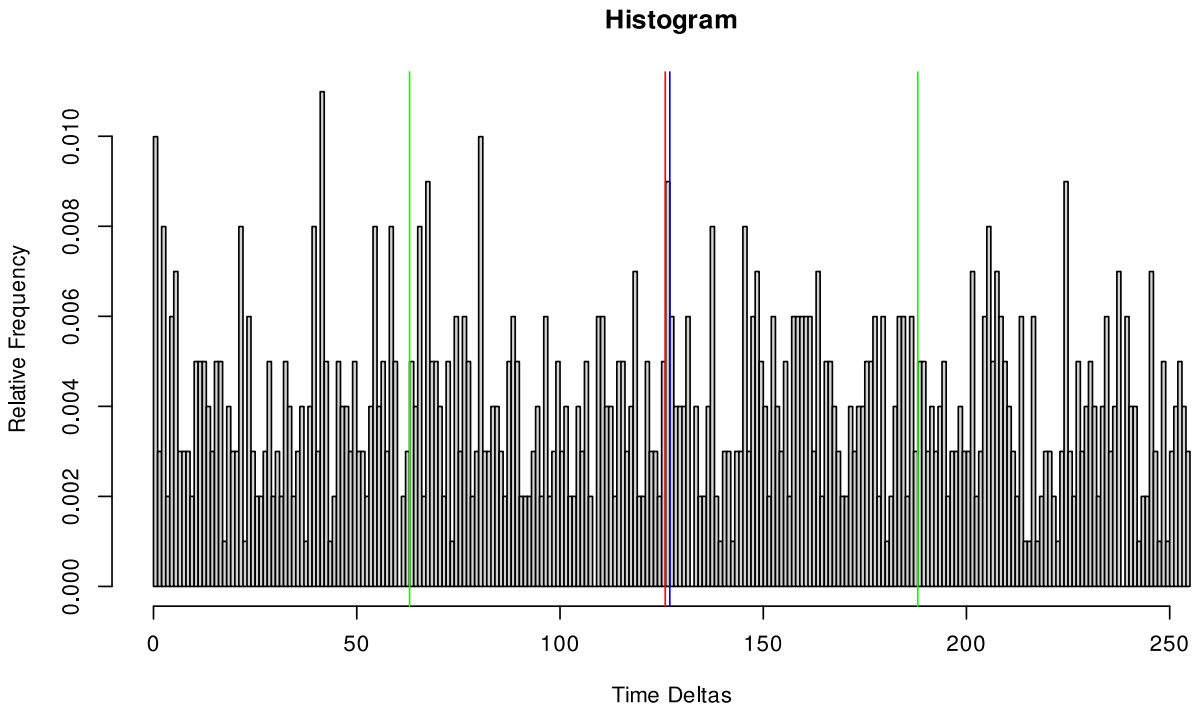

Figure 10: Histogram of Time Deltas for First and Second Scheduler-Based Noise Source Event in a Virtual Environment – Column-Wise

The measuredtime stamps of the first 1000 scheduler-based noise source events visualized in figure 10 and the consideration that in the worst case at most one bit of entropy is harvested from one time stamp allow the conclusion that the entropy present in the time stamps is already sufficiently large for achieving a commonly required security strength of 128 bits. The Linux-RNG massively underestimates the boot-time entropy present with the scheduler-based noise source event time stamps.

6.3.2.2 Row-Wise Reboot Data Assessment

The minimum values for the different entropy estimations are provided in the following table.

	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	**Min-Entropy Plug-in Estimate of 8 Bits of Time Delta**	**Shannon Entropy Plug-in Estimate of 8 Bits of Time Delta**
Minimum entropy estimation	2.530	1.936	3.492

Table 9: Scheduler-Based Noise Source: Early Boot SP800-90B Min-Entropy Estimates in Virtual Environment – Row-Wise

Similar to the column-wise assessment, figure 11 shows the row-wise time deltas for the first and second event.

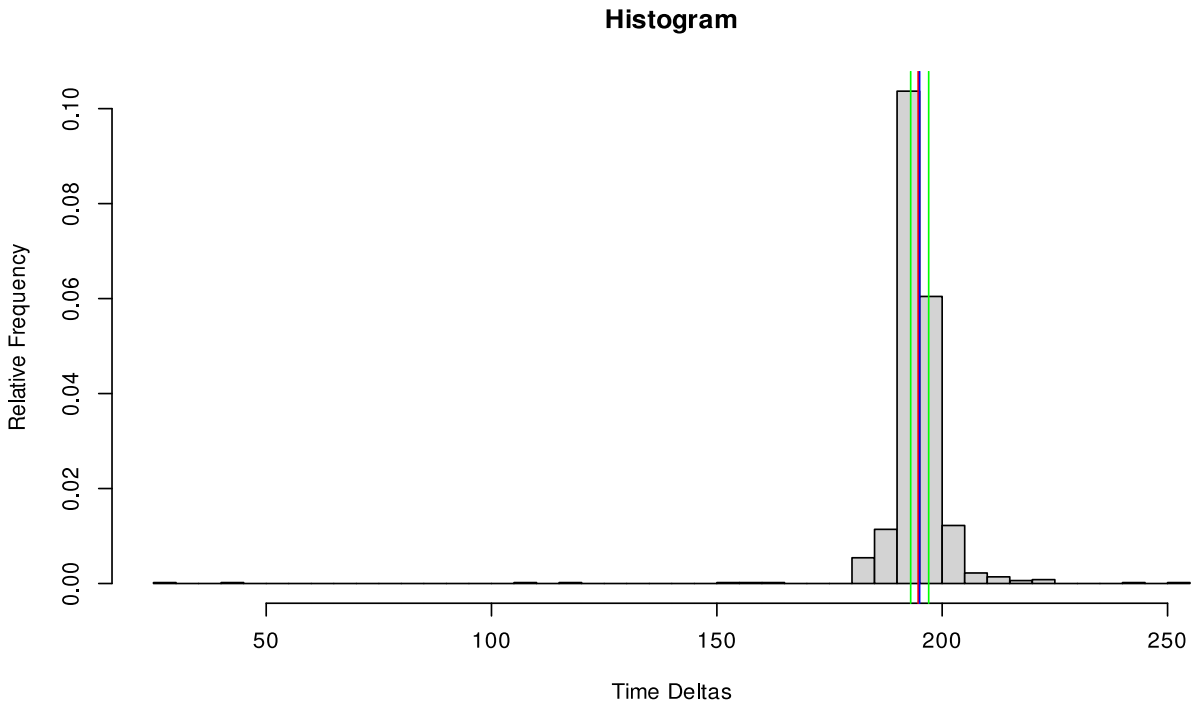

Figure 11: Histogram of Time Deltas for First and Second Interrupt in a Virtual Environment – Row-Wise

The graph for the row-wise entropy assessment, i.e. the entropy provided by the scheduler-based noise source during one boot operation, shows a very large concentration of the time deltas either on the absolute high-side. This figure supports the lower values for the different min-entropy estimates.

Yet, the Linux-RNG at most awards one bit of entropy per time stamp. Thus, the measured numbers show that the Linux-RNG underestimates the available entropy in that worst-time assumption. Please note that during the collection of the 1000 scheduler-based noise source events, the kernel never awarded any entropy using its entropy heuristic. This shows that the kernel massively underestimates the available entropy.

6.3.2.3 SP800-90B Sanity Test

The anticipated entropy rate to be applied for the scheduler-based noise source SP800-90B sanity test is assumed to be 1 bit. This is a worst case assumption due to the following: In a worst case, the timer increasing the entropy estimator by one bit fires after each obtained time stamp. In this worst case, each time stamp is assumed to have at least one bit of entropy.

The following data is obtained from the sanity test:

- Maximum number of occurrences of a value: 72
- By using an anticipated entropy of 1 bit per time stamp, the sanity test passes.

6.3.3 Early Boot Interrupt Entropy Testing on Native Hardware

The test to obtain early boot data used as input to the Linux-RNG is re-performed with the Linux kernel executing on native hardware. This re-testing is provided to allow a comparison between a virtual and a native environment. The virtual environment has fewer devices compared to native hardware and thus generates fewer interrupts during boot as fewer devices need to be initialized and interacted with. It is

expected that this property reduces the amount of entropy present in the measurements for virtual environments. Conversely, virtual environments are subject to frequent re-scheduling events performed by the host. Such rescheduling events increase the variations of the interrupt event time stamps which can be interpreted as entropy. A Linux kernel executing on native hardware is not subject to scheduling events enforced by external entities. Thus, the time stamps picked up by the Linux-RNG interrupt noise source executing on native hardware should have fewer variations.

Both described effects oppose each other, i.e., the one effect is expected to increase the entropy on native hardware whereas the other is expected to decrease the entropy. To obtain a better understanding of the magnitude of the effects, the early boot interrupt event time stamps are obtained for a Linux-RNG executing on native hardware.

6.3.3.1 Column-Wise Reboot Data Assessment

The following SP800-90B min-entropy estimates, the Min-Entropy as well as the Shannon-Entropy values are calculated

	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	**Min-Entropy Plug-in Estimate of 8 Bits Width Time Stamp**	**Shannon Entropy Plug-in Estimate of 8 Bits Width Time Stamp**
Minimum entropy estimation	6.638383	5.878	7.738

Table 10: Interrupts: Early Boot SP800-90B Min-Entropy Estimates in a Native Environment – Column-Wise

The interpretation of the table is identical to the table presented for the virtual environment boot time measurements.

The different statistical entropy values calculated from the measurements of the first interrupt event time stamps obtained by the Linux-RNG after boot on native hardware do not deviate significantly from the same values obtained on a virtual environment. Thus, the mentioned contrary effects are concluded to cancel each other out or are insignificant to the overall entropy present in the Linux kernel boot process.

A graphical representation of the values presented in the table is given in figure 12. It shows the histogram of the delta between the first and the second interrupt event time stamp of each boot cycle recorded by the Linux-RNG where the X-axis represents the number of ticks of the high-resolution time stamp between the occurrence of both interrupts.

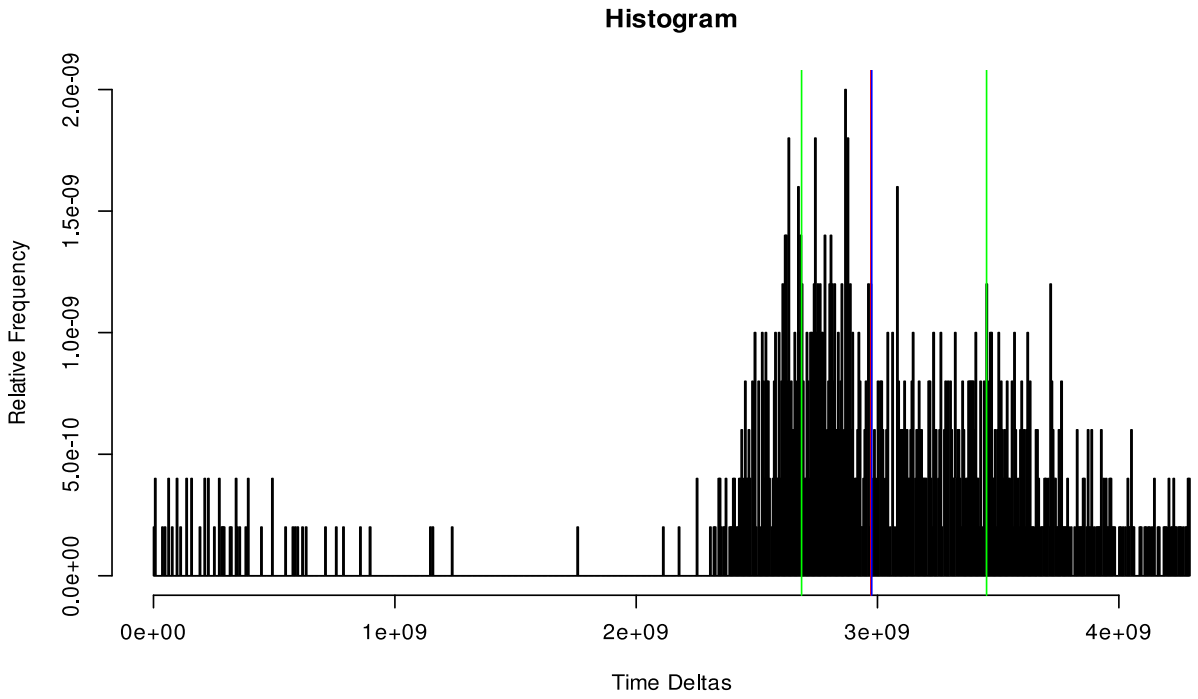

Figure 12: Histogram of Time Deltas for First and Second Interrupt in a Native Environment – Column Wise

Starting with the second time delta depicted in figure 13, the distribution of the time delta values exhibits more distinct spikes. Yet, considering the scales of the X and Y axis, the distribution is sufficiently large to support the conclusion of the presence of sufficient entropy.

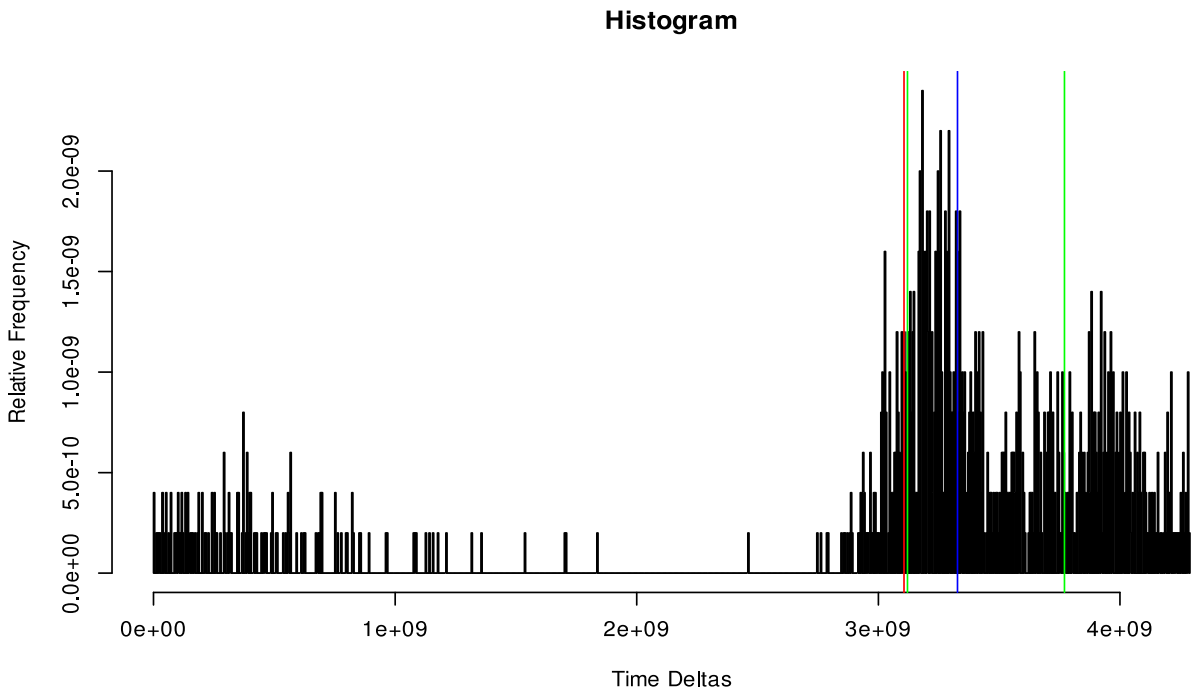

Figure 13: Histogram of Time Deltas for Second and Third Interrupt in a Native Environment – Column-Wise

With the obtained results, the same conclusions for the measurements in virtual environments given in section 6.3.1 can be drawn. Disregarding the correlation problem due to the presence of only the interrupts as discussed above, and considering that the Linux-RNG awards the time stamps between 1 to 64 interrupts only one bit of entropy, the Linux-RNG is considered to massively underestimate the entropy present in the interrupt time stamps during early boot.

Just like for the measurements and results of the testing in virtual environments, the NIST tool verified that the column-wise entropy is not less than half of the runtime entropy value.

6.3.3.2 Row-Wise Reboot Data Assessment

In addition to the column-wise assessment, [SP800-90B] section 3.1.4 requires a row-wise assessment. With the collection of data from 1,000 reboots, the data set encompasses 1,000 entropy values. Like for the column-wise data assessment, the minimum values are provided in the following table.

	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	**Min-Entropy Plug-in Estimate of 8 Bits of Time Delta**	**Shannon Entropy Plug-in Estimate of 8 Bits of Time Delta**
Minimum entropy estimation	7.276950	5.964	7.739

Table 11: Interrupts: Early Boot SP800-90B Min-Entropy Estimates in a Native Environment – Row-Wise

Similar to the column-wise assessment, figure 14 shows the row-wise time deltas for the first and second event.

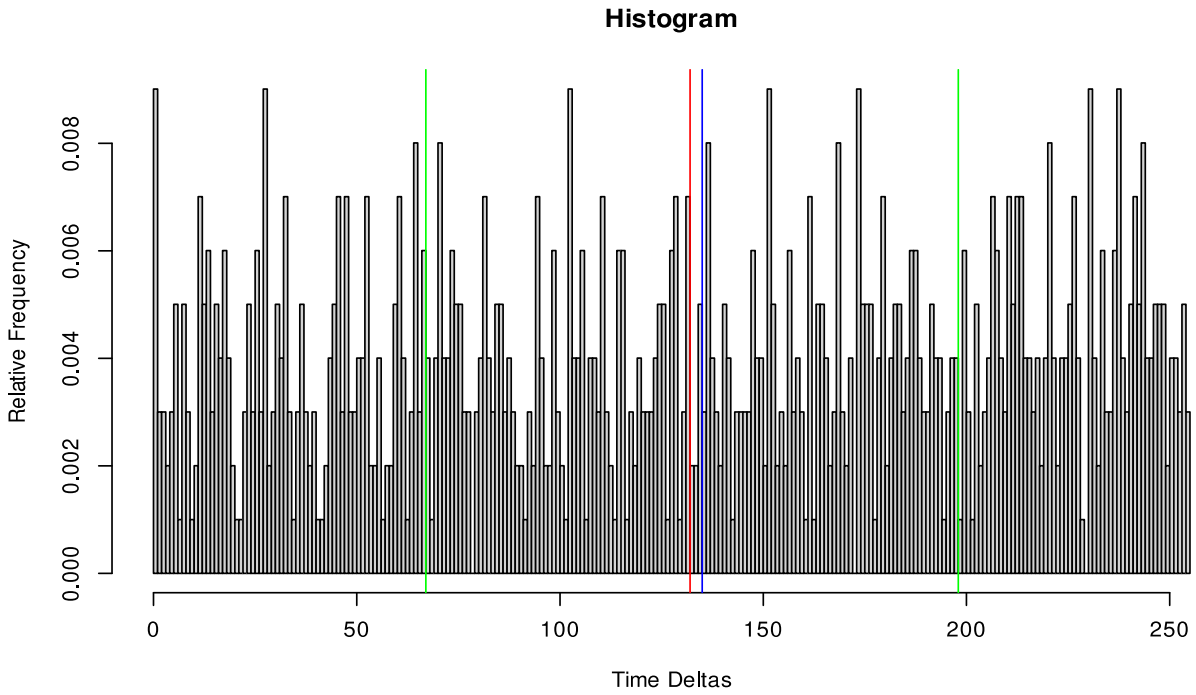

Figure 14: Histogram of Time Deltas for First and Second Interrupt in a Native Environment – Row-Wise

Just like for the column-wise entropy values, the minimum SP800-90B entropy estimate for the row-wise analysis must not be less than half of the runtime entropy rate. This again is verified by the NIST SP800-90B entropy assessment tool.

6.3.3.3 SP800-90B Sanity Test

The following data is obtained from the sanity test with the same considerations as outlined in section 6.3.1.3:

- Maximum number of occurrences of a value: 2
- By using an anticipated entropy of 1 bit per time stamp, the sanity test passes.

6.3.4 Early Boot Scheduler-Based Entropy Testing in a Native Environment

The scheduler-based noise source behavior on native hardware is shown in this section.

6.3.4.1 Column-Wise Reboot Data Assessment

When processing the raw data column-wise, for each column, the SP800-90B min-entropy estimates, the Min-Entropy as well as the Shannon-Entropy values are calculated. Instead of listing 1000 values for each aspect, the following table provides the minimum values calculated over all columns.

	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-Entropy Plug-in of 8 Bits Width Time Stamp	Shannon Entropy Plug-in Estimate of 8 Bits Width Time Stamp
Minimum entropy estimation	4.302038	6.059	7.741

Table 12: Scheduler-Based Noise Source: Early Boot SP800-90B Min-Entropy Estimates in a Native Environment – Column-Wise

The table shows that the high-resolution time stamp of each of the first 1000 scheduler-based noise source events has an SP800-90B min-entropy estimate of 4.3 bits of entropy. Considering the Min-Entropy plug-in estimator applied to the time deltas (i.e. the difference of two adjacent time stamps), a value of 6.0 bits per event is measured. The Shannon entropy plug-in estimate values for the time deltas is 7.7 bits per event.

A graphical view of the time stamp distribution for the first time delta is provided with figure 15.

The table with the Min-Entropy for the time stamps of the first 1000 events visualized in figure 15 and the consideration that in the worst case at most one bit of entropy is harvested from one time stamp allow the conclusion that the entropy present in the time stamps is already sufficiently large for achieving a commonly required security strength of 128 bits. The Linux-RNG underestimates the boot-time entropy present with the scheduler-based noise source event time stamps.

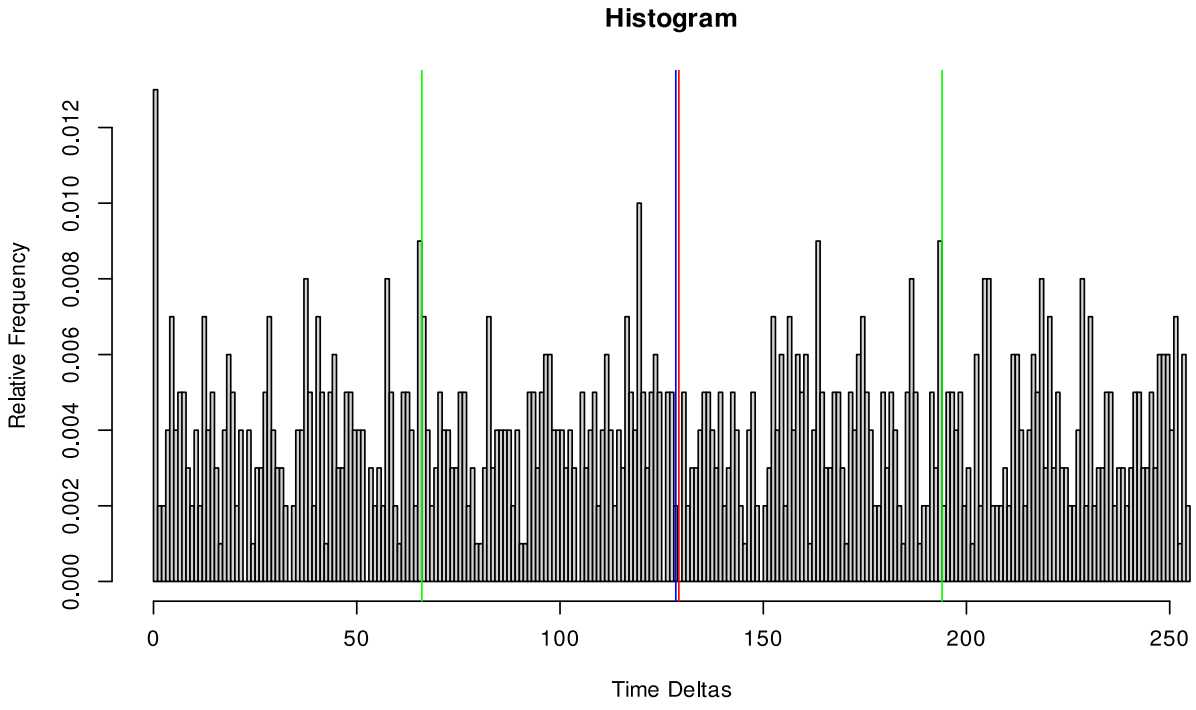

Figure 15: Histogram of Time Deltas for First and Second Scheduler-Based Noise Source Event in a Native Environment – Column-Wise

6.3.4.2 Row-Wise Reboot Data Assessment

The high and low values for the different entropy estimations are provided in the following table.

	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-Entropy Plug-in Estimate of 8 Bits of Time Delta	Shannon Entropy Plug-in Estimate of 8 Bits of Time Delta
Minimum entropy estimation	0.052736	0.917	2.124

Table 13: Scheduler-Based Noise Source: Early Boot SP800-90B Min-Entropy Estimates in a Native Environment – Row-Wise

Similar to the column-wise assessment, figure 16 shows the row-wise time deltas for the first and second event.

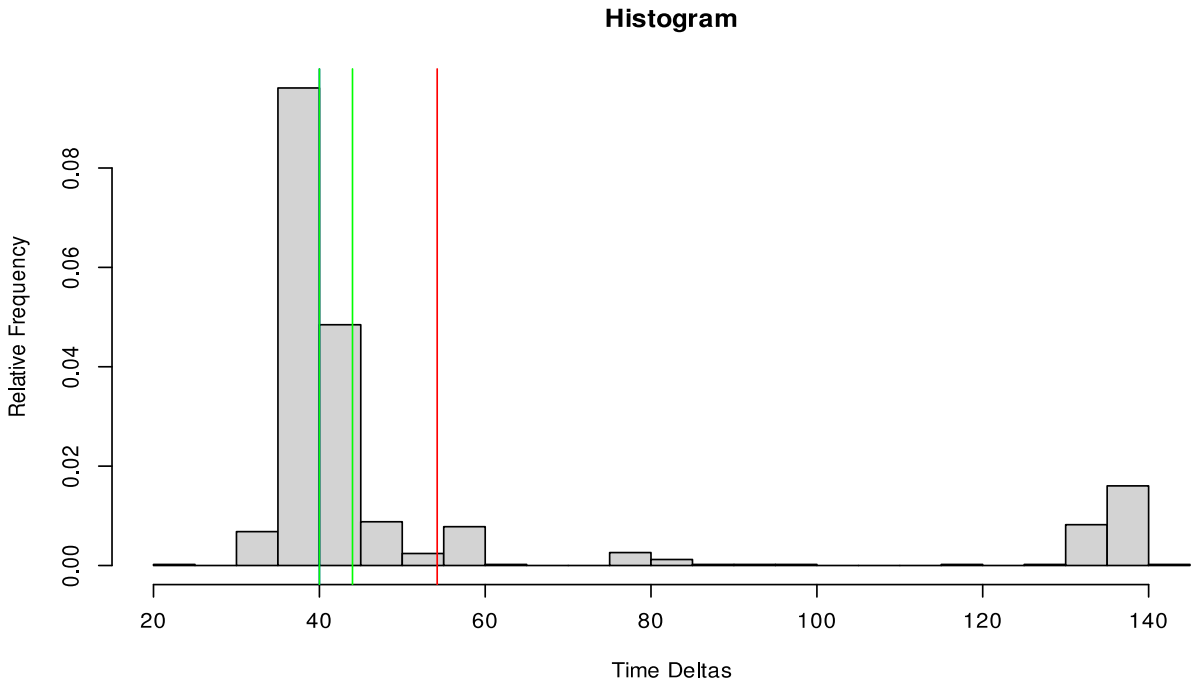

Figure 16: Histogram of Time Deltas for First and Second Scheduler-Based Noise Source Event in a Native Environment – Row-Wise

Similarly, to the data for the scheduler-based noise source in virtual environments, the graph for the row-wise entropy assessment, i.e. the entropy provided by the scheduler-based noise source during one boot operation, shows a significant concentration of the time deltas on the low side supporting the low SP800-90B entropy estimates.

The Linux-RNG at most awards one bit of entropy per time stamp. In this worst case, the Linux kernel would overestimate the entropy significantly. However, note that during the collection of the 1000 scheduler-based noise source events, the kernel never awarded any entropy using its entropy heuristic. This shows that the kernel underestimates the available entropy during the measurement.

The SP800-90B requirement that the row/column-wise entropy assessment should not be less than half of the runtime entropy is not met. Thus, SP800-90B considers this noise source as inappropriate which should be treated with zero bits of entropy.

6.3.4.3 SP800-90B Sanity Test

The anticipated entropy rate to be applied for the scheduler-based noise source SP800-90B sanity test is assumed to be 1 bit. This is a worst case assumption due to the following: In a worst case, the timer increasing the entropy estimator by one bit fires after each obtained time stamp. In this worst case, each time stamp is assumed to have at least one bit of entropy.

The following data is obtained from the sanity test:

- Maximum number of occurrences of a value: 100
- By using an anticipated entropy of 1 bit per time stamp, the sanity test passes.

6.3.5 Conclusions of Early Boot Entropy Measurements

The measurements of the entropy contained in the interrupt event time stamps recorded by the Linux-RNG for the first 256 interrupts show that it amounts to significant values. The entropy per time stamp recorded for one interrupt considerably exceeds one bit. On the other hand, the scheduler-based noise source shows mixed results: the reboot tests in virtual environments show it is usable, but on bare-metal those test indicate the noise source should not be used. To be on the safe side, it should be generally treated with zero bits of entropy.

When interpreting the entropy measurements with a safety margin to assume worst-case scenarios by cutting the measured values in half, the entropy values are still more than one bit of entropy per time stamp. For the following discussion, one bit of entropy per time stamp is assumed. Thus, the measurements show that collecting 128 interrupt event time stamps while booting is sufficient to cover the initial seeding requirements set forth by the German BSI with [TR021021] as well as [SP800-131A] specified by the US NIST.

Applying the general Linux-RNG entropy heuristics, the Linux-RNG significantly underestimates the available entropy. This finding is supported by the fact that the correlation problem between interrupts on one side and HID / block device noise sources on the other side as discussed above is not in full effect during early boot. Based on the aforementioned measurements and applying the discussed safety margin where each time stamp is considered to contain one bit of entropy, 256 bits of entropy are injected into the ChaCha20 DRNG state before the Linux-RNG is defined to be fully seeded and the blocking interfaces are released. This allows the conclusion that when the two interfaces unblock, sufficient entropy has been accumulated available for use cases with strong cryptographic requirements.

The measurements of the available entropy during boot for virtual environments and native hardware hardly differ. Thus, the conclusion is equally applicable to both environments.

It is important to note that this conclusion is only applicable to environments with a high-resolution time stamp. Hardware architectures with a low-resolution time stamp will not have significant amounts of entropy after boot.

Even the `getrandom` system call always provides data from a sufficiently seeded DRNG. This finding is not applicable to /dev/urandom or even the `get_random_bytes` in-kernel API, as explained by the following observations:

- On the test system executed within a virtual environment, the kernel boot process completes after around one second after the start of the boot process. At that time, the user space from the initramfs is started. The first 128 interrupts are received at around this time when user space starts. Interrupts are collected in per-CPU fast_pools. A copy of a fast_pool is injected into the input pool after the fast_pool received between 1 and 64 interrupts. This implies that 256 fast_pools with at least one interrupt are required to be injected into the input pool to reach the seeding level of 256 bits of entropy. This implies that a large number of interrupts are required during boot time to fulfill that requirement.

- Executing the Linux-RNG on native hardware shows that the kernel boot process is finished some two seconds after boot. By that time it is likely but not guaranteed that 256 interrupts are received. Thus, the outlined consideration for /dev/urandom is still relevant for native hardware, though with a lesser probability.

7 Test Series: State Transition Function of DRNG

With chapter 6, the analysis of the unprocessed data obtained from the noise sources was conducted. The Linux-RNG receives that data and mixes it into the input pool using the Blake2s operation. When data is injected into the Blake2s state, it is mixed with the residual data including data obtained prior the last hash-final operation. This is the basis for the generation of ChaCha20 DRNG key.

This chapter analyzes the state transition function used to process input data and to update the internal state used for the deterministic processing.

This chapter is separated into two main components:

- The first set of tests performs an analysis of the state transition function without using any data from the noise sources. This is done by extracting the state transition function of the Linux-RNG into standalone code. This standalone code can now be invoked with arbitrary input data to study the behavior of the function. To allow the reader to reproduce the results of this test, the extracted code for the state transition function is identical to the corresponding code in the random.c Linux kernel code. The functions that deliver the input are changed such that a counter starting at one is increased by one with each request and provides the input data. The state after the completion of the state transition function is dumped as a hexadecimal string for the analysis.
- In a second set of tests, the state transition function of an operational Linux-RNG is monitored. Snapshots of the state content after the state transition function has processed the entire state are taken and analyzed once to see whether they exhibit characteristics of an ideal random number generator.

7.1 Standalone Operation of State Transition Functions

The code that is extracted from random.c is marked as such in the C code used for the following tests. The extracted code is identical to the Linux kernel code to allow an immediate confirmation that the state transition functions used by the Linux-RNG are analyzed.

To utilize the state transition functions, the following additional code is added:

- The code from the state transition function is part of a user space application. This means that a `main` function is present as the entry function used during startup of the application.
- The state transition function requires input data. In the Linux kernel code, the data from the noise sources is mixed into the input pool. The ChaCha20 DRNG uses the output data from the input pool. The data is replaced by a data generating function which maintains an 8 bit variable, i.e. C character data type. That variable is used as a counter which is incremented by one each time new data is requested. When the variable reaches 255, it will wrap back to zero upon the next increment. This allows a byte-wise analysis of the behavior of the state transition function.
- The state transition function may require helper code which is added. The following types of helper code are added:
  - For the ChaCha20 operation, the ChaCha20 block function is implemented. To ensure that this block function operates correctly, a self-test is added using the test vectors from [RFC7539] section 2.3.2.
  - Converter code from binary into hexadecimal representation is added.
- Particularly for the ChaCha20 code extracted from random.c, code fragments in the extracted functions had to be commented out as it covered aspects not applicable to the test code. The original code is still left in the test code, but commented out to allow reviewers to verify that the applied changes are appropriate. These changes include:

- Limiting the number of secondary ChaCha20 DRNGs to one. This can be justified because all secondary DRNGs are processed identically.
- Disabling the reseed timer enforcement. As mentioned in the ChaCha20 DRNG design, the ChaCha20 DRNG is reseeded every 60 seconds. As this is irrelevant for testing, the respective trigger code is disabled.
- Removing the locking code because the testing executes single-threaded.

7.1.1 Blake2s State Transition Function

The processing of the input pool is equivalent to concatenating all input data and calculate a Blake2s message digest and use the result as a key to the new Blake2s state.

7.1.2 ChaCha20 State Transition

To demonstrate the ChaCha20 state transition behavior, the test code exports the kernel implementation into user space for analysis.

The code provides a snapshot of the ChaCha20 state after each operation is applied as part of the state transition operation. Thus, the code allows the assessment of the following aspects:

- The ChaCha20 DRNG initialization fills the key part of the state as well as the counter and the nonce part. The initial fixed values are filled with the known ASCII string.
- ChaCha20 DRNG reseed provides well-defined seed information to allow studying whether the application of the cryptographic operation has any weaknesses.
- The generation of data results in data showing the characteristics of an ideal random number generator even though the seed is deterministic. The seed fills the buffer byte-wise with an increasing integer.

The test assumes the presence of only one secondary ChaCha20 DRNG. If more are present, they will adhere to the same behavior outlined in the following

After initialization before the ChaCha20 DRNGs are used for anything, the base ChaCha20 DRNG state is:

```
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

The secondary ChaCha20 DRNG state is:

```
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

That ChaCha20 state shows the expected content: the ChaCha20 key is zero

After requesting one random bit stream to be generated, the base ChaCha20 state is:

```
023f3720 3a2476c4 2566a61c c55c3ca8 75dbb4cc 41c0deb7 89f8e7bf 88183638
```

and the secondary ChaCha20 DRNG state is:

```
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

The base ChaCha20 DRNG was seeded with 0x01 followed by a ChaCha20 block operation whose 256 most significant bits are the new ChaCha20 DRNG state. The secondary ChaCha20 DRNG is not used as the test specifies that the Linux-RNG is not yet initialized.

A second request for random bits where the Linux-RNG is assumed to be early seeded, which implies that the secondary ChaCha20 DRNG is still not used, but a separate seeding of the base ChaCha20 DRNG from the input pool is not performed, looks similar: After fulfilling the request, the base ChaCha20 DRNG state is:

```
b066b2f2 c8b84002 dd7319a2 c023597b cebd2d2f 63eca4c4 3a9d2db4 7cf71740
```

and the secondary ChaCha20 DRNG:

```
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Before the third request, the Linux-RNG is assumed to receive sufficient entropy to be fully seeded. This triggers a reseed of the base ChaCha20 DRNG. In the testing, the reseed returns the bytes 0x02. The base ChaCha20 DRNG state looks as follows:

```
02020202 02020202 02020202 02020202 02020202 02020202 02020202 02020202
```

This indicates that the data from the input pool overwrites any existing key material.

From this point on, the base ChaCha20 only seeds the secondary ChaCha20 DRNG and is otherwise dormant. The request for random bits is satisfied by the secondary ChaCha20 DRNG: The base ChaCha20 DRNG state after the generation is:

```
f6a12ca8 ffc30a66 ca140ccc 72763361 15819361 186d3f53 5dd99f8e aaca8fce
```

and the secondary ChaCha20 DRNG:

```
66312b5d 6bef4b5e d7b2fc8f e2ec3fec abfccae2 b187eb62 c3e7ee70 4bc5d1dd
```

Processing the next request for random numbers leaves the following states: The base ChaCha20 DRNG state after the generation is:

```
f6a12ca8 ffc30a66 ca140ccc 72763361 15819361 186d3f53 5dd99f8e aaca8fce
```

and the secondary ChaCha20 DRNG:

```
a12591fe 04f43ee2 87c4002f b9fe260b 608293fa 7828076f b0ec8d67 5259cb2d
```

This indicates that the base ChaCha20 DRNG is left unchanged and unused when satisfying the request for random bits. When now obtaining 100,000 blocks from the secondary ChaCha20 DRNG without reseeding, the resulting data shows the following statistical properties:

- The Chi-Squared value when treating the data stream as bit-wise is 46.94 which indicates the characteristics of an ideal random number generator.
- The Chi-Squared value for a byte-wise processing of the data stream is 61.59 which also indicates the characteristics of an ideal random number generator.
- All compression algorithms deliver a “compressed” data whose size is larger than the original data stream. This implies that no structures are found by the compression algorithms, which is an indication that the data stream exhibits the characteristics of an ideal random number generator.
- All tests defined by the test procedure A are passed.

7.2 AIS 20/31 Test Procedure A for Entropy Pool

Considering that the entropy pool is a self-feeding Blake2s message digest loop and Blake2s is considered to be cryptographically strong, the test procedure A has not been applied to this output assuming it will not show any deviations as otherwise Blake2s must be considered to be broken.

8 Test Series: DRNG Output Functions

The test series in this chapter is not so much about entropy and its maintenance, but it rather focuses on the correctness of the different DRNG output functions. The goal is to identify problems in such output functions highlighted with issues like CVE:2013-4345 which indicates an off-by one issue in the Linux kernel ANSI X9.31 DRNG output function. Or even the error introduced by the author of this study to the SP800-90A DRBG present in the Linux kernel crypto API causing truncated outputs, which he fixed with the patch 8ff4c191d1123ea1ba610dbc25e93568d9e7756c contained in the upstream Linux kernel Git tree. These bugs are caused when random data shall be produced that are not equal to the block size of the deterministic random number generation process, i.e. the block size of the used cryptographic function in the random number generator output function.

The testing is intended to obtain data from the output functions which generate random numbers. The output is then processed by statistical testing to analyze whether deviations from the expected ideal random number generator behavior are present.

The testing is conducted on the output received by callers via the /dev/random device which delivers data generated by the ChaCha20 DRNG.

The conducted testing can be summarized as follows. The device file /dev/random is accessed such that 1000 blocks of data are created. The testing covers all block sizes ranging from 1 byte to 4096 bytes. The use of different block sizes shall verify that the code producing the random numbers can handle every request of any length correctly. It is assumed that when the test result for the block sizes up to 4096 bytes shows no deficiencies, larger block sizes are handled correctly as well by the deterministic random number generation process.

To validate the output, the generated data is subjected to the following analyses:

- The generated data is processed with the `ent` tool to obtain the Chi-Squared test result. If the Chi-Squared test result is below 0.10 or above 99.9, the result is flagged for further analysis. This test is considered to be a search for a “smoking gun” as to whether the generated data does not exhibit the characteristics of an ideal random number generator. The calculation of the Chi-Squared value is considered an easy approach to identify data that exhibits the characteristics of an ideal random number generator due to the following: if the Chi-Squared test fails, then the data does not show the characteristics of an ideal random number generator. However, there could be false positives in the sense that the Chi-Squared result indicates the data is from an ideal random number generator where in fact a pattern is present. This applies in particular to the types of errors this set of tests wants to detect: programming errors leading to a pattern present in the output data. Thus, the Chi-Squared testing is deemed sufficient to find a “smoking gun” for further analysis.
- The generated data is compressed with the `gzip2`, `bzip2`, `xz` and `lzma` compression tools. These tools cover contemporary as well as state-of-the-art compression algorithms with high compression factors. The size of the original binary data is then compared with the size of the “compressed” file. The test would mark an error if the “compressed” file is smaller than the original size. If the compressed file is smaller, then patterns are present that can be detected with the compression algorithms. If a file is not compressible it is deduced that no pattern detectable by the compression algorithms is present. In this case, the “compressed” file must be larger because the compression algorithms add extra data to their output. If at least one of the compression operations is able to create a file with smaller size than the original file, the processed data does not follow the characteristics of an ideal random number generator, signaling a failure in the deterministic random number generating functions of the Linux-RNG.
- The tool `dieharder` is used to process the random data extracted from /dev/random. To apply all tests implemented in the `dieharder` statistical tool to the output of /dev/random, the following call is executed:
  ```
  cat /dev/random | dieharder -a -g 200
  ```

- After obtaining 5MB of data, the Test Procedure A defined in [AIS2031] is applied to the binary data produced by /dev/random. This Test Procedure A covers the Monobit test, the Poker test, the Runs test, the Long Runs test, and the Autocorrelation test. The test procedure A is implemented with the test tool test_proc_A.pl provided as part of the test suite.

8.1 Output of ChaCha20 DRNG

The output data from the ChaCha20 DRNG that backs /dev/random shows 8 out of 4096 data sets with Chi-Squared values outside the allowed range. Again, when re-running the testing for the affected block size, the observed Chi-Squared value is back in the expected range, confirming that the initial outliers are false positives. Thus, the Chi-Squared test results do not indicate any programming errors in the ChaCha20-DRNG feeding /dev/random.

The file compression test showed that all “compressed” files for all compression algorithms and all block sizes are larger than the original files. This result confirms the Chi-Squared testing result that no implementation error in the ChaCha-20 DRNG random number generation function can be detected.

All `dieharder` tests results are marked as “passed” except for 2 “weak” results. It is generally accepted that few “weak” results are present in the dieharder output as it runs many sensitive statistical tests. The nature of random numbers is that once in a while they may be flagged as weak by sensitive tests. No failed test result is present. This type of result is expected for data from an ideal random number generator. Thus, the `dieharder` test result confirms the initial test results.

All tests pass the test procedure A, indicating data exhibiting the characteristics of an ideal random number generator confirming the results of the previous tests.

8.2 Conclusion of the Output Function Testing

The testing has shown that the output function generating random numbers for /dev/random and dev/urandom produce data exhibiting the characteristics of an ideal random number generator. Thus, no implementation errors that would diminish the entropy in the random numbers were identified.

9 Guidance For Using the Linux-RNG

Throughout the design description and the assessment whether the Linux-RNG conforms to NTG.1 or DRG.3 requirements, different constraints on either the use of the Linux-RNG or the compilation of the Linux kernel code are outlined. This section consolidates these assumptions and requirements to give a user a check list to verify whether the conclusions given in this document can be applied to his kernel.

- The kernel configuration option `CONFIG_RANDOM_TRUST_BOOTLOADER` must not be present. This ensures that any data provided by the boot loader during kernel initialization time is not assumed to have any entropy.
- Either the kernel command line option `random.trust_cpu=0` must be set or the kernel compile-time option of `CONFIG_RANDOM_TRUST_CPU` must be unset. This ensures that the data from CPU-based noise sources like Intel RDRAND/RDSEED is not assumed to provide trustworthy entropy.
- Any caller of the `add_hwgenerator_randomness` interface function offered by the Linux kernel must be separately analyzed with an independent entropy analysis to show that the amount of entropy delivered via this interface is indeed present. This function is only invoked by device drivers for specialized cryptographic hardware which either is assumed to be not present or has its own entropy assessment demonstrating that the expected entropy rate is actually provided. However, there is the following exception: the network driver ATH9K WLAN uses `add_hwgenerator_randomness` if the kernel compilation option `CONFIG_ATH9K_HWRNG` is set to inject entropy into the Linux-RNG from the WLAN RNG. In case the Atheros WLAN hardware is present, this option is only allowed to be set if the Atheros WLAN hardware has its own entropy assessment.

All other kernel configuration options or kernel command line parameters do not affect the operation of the Linux-RNG or the entropy rate it delivers.

As of kernel version 5.18, only the base ChaCha20 DRNG can achieve all DRG.3 requirements. If more than one CPU is available (which is usually the case in contemporary hardware), the Linux-RNG interfaces are served by secondary ChaCha20 DRNGs that are seeded by the base DRNG. Although this does not imply a cryptographic weakness, the secondary ChaCha20 DRNGs do not fulfill the DRG.3.1 requirement. The following Linux-RNG interfaces assure seeding by a DRG.3-compliant base DRNG in case secondary DRNGs are used:

-
- /dev/random,
- `getrandom` system call with the flags field being zero,
- invoking the in-kernel `get_random_bytes` API call when the callback registered with `register_random_ready_notifier` was invoked,
- invoking the in-kernel `get_random_bytes` API call after the `wait_for_random_bytes` API call returns – note, service functions like the `get_random_XXX_wait` API call family where XXX is either u32, u64, int or long fall into this category.

In case the Linux kernel source code is to be modified, the following files must remain unchanged if the conclusions given in this document shall remain applicable:

- drivers/char/random.c must remain unchanged,
- lib/crypto/chacha.c must remain unchanged as it provides the ChaCha20 block operational, and
- include/crypto/blake2s.h together with lib/crypto/blake2s.c must remain unchanged as it provides the Blake2s implementation used for the entropy pool output function.

Code invoking functions providing data with an entropy estimate to the Linux-RNG, such as via `add_hwgenerator_randomness`, must have their own entropy assessment backing the entropy rate used to feed the Linux-RNG.

10 New Developments in Linux-RNG

The current document analyzes one particular version of the Linux kernel with its Linux-RNG implementation. The document always applies to the Linux kernel versions found at http://www.kernel.org.

For each new Linux kernel version, the current document is subject to review analyzing the following possible differences to the assessed newer Linux kernel version:

- All changes performed to the following files of drivers/char/random.c, include/linux/random.h, include/uapi/linux/random.h, arch/x86/include/asm/archrandom.h.
- Changes to the invocation of the entropy gathering functions documented in sections 3.5.2.1, 3.5.2.2, 3.5.2.3, and 3.5.2.5. This assessment shall include new conditions applied to the invocation of these entropy gathering functions.
- Functions marked with either EXPORT_SYMBOL or EXPORT_SYMBOL_GPL implemented in random.c shall be assessed whether their invocation in the remainder of the Linux kernel has changed. These functions are interfaces exported by the Linux-RNG to other kernel parts.

Any changes identified for the aforementioned items are assessed in the following sections regarding their impact to the documented Linux-RNG functionality. The preceding sections are updated as necessary.

10.1 Linux Kernel 5.18.1

Previously assessed Linux kernel version: 5.17 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.17.tar.xz

Currently assessed Linux kernel version: 5.18.1 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.18.1.tar.xz

Assessment of changes: Compliance with NTG.1 and DRG.3 is lost. Only one very special use case of the Linux-RNG is DRG.3 compliant as outlined in section 5.2.1. The configuration requirements given in chapter 9 apply unchanged.

10.1.1 Changes to the Linux-RNG Implementation

10.1.1.1 File drivers/char/random.c

Complete re-implementation. The changes are too many to list them individually.

10.1.1.2 File include/linux/random.h

Editorial changes without affecting the functionality are applied.

In addition, the new notification interfaces functions of random_prepare_cpu, random_online_cpu are added.

The asynchronous signal functions for achieving the fully seeded state are renamed: register_random_ready_notifier, unregister_random_ready_notifier.

10.1.1.3 File include/uapi/linux/random.h

No changes.

10.1.1.4 File arch/x86/include/asm/archrandom.h

No changes.

10.1.2 Changes to Invocation of Entropy Gathering Functions

10.1.2.1 add_input_randomness

No change to the invocation in `input_handle_event` and thus no effect on the Linux-RNG.

10.1.2.2 add_interrupt_randomness

No change to the invocation in `handle_irq_event_percpu`, `vmbus_isr` and `sysvec_hyperv_stimer0` and thus no effect on the Linux-RNG.

10.1.2.3 add_disk_randomness

No change to the invocation in `scsi_end_request` and thus no effect on the Linux-RNG.

10.1.2.4 add_hwgenerator_randomness

No change to the invocation in `hwrng_fillfn` and thus no effect on the Linux-RNG.

This function is still invoked in the ATH9K WLAN driver if the Linux kernel configuration option `CONFIG_ATH9K_HWRNG` is set.

10.1.3 Definition and Use of new Interfaces

The Linux-RNG adds the following interfaces:

- random_prepare_cpu, random_online_cpu: Those functions are invoked by the kernel when new CPUs are made available to allow the Linux-RNG to re-initialize the batched entropy management.
- The functions register_random_ready_notifier, unregister_random_ready_notifier replace the register_random_ready_notifier, del_random_ready_callback without changing the properties of the functions.
- add_vmfork_randomness is added to trigger a reseed of the base ChaCha20 DRNG in case a spawning of a new virtual machine is performed by the kernel.

Appendix A: Testing Aspects and Implementation

To reach conclusions about the quality of the random numbers produced by the Linux-RNG its behavior and its operation had to be monitored at runtime. For such monitoring the Linux kernel must be instrumented to allow for the reading of various parameters and state information without significantly affecting this data by the test approach itself.

The Linux kernel implements several tracing mechanisms which can be used during runtime. The following tracing mechanisms are available:

- SystemTap
- Ftrace
- Kernel debugger
- Manual instrumentation of the source code
- `ptrace` system call for analyzing system calls

In the following sections, the used tracing method for measuring the Linux-RNG is described. The rationale discusses also the impact of the tracing mechanism on the obtained results.

All tracing mechanisms have an impact on the timing behavior of the Linux kernel in general and the Linux-RNG in particular. As the Linux-RNG uses timing variations as the raw noise, all tracing mechanisms impact the Linux-RNG operation. Since that impact, however, is applicable to each measurement this impact is akin to the Linux-RNG operating on a slower CPU. Since the Linux-RNG is expected to deliver consistent results irrespective of the CPU execution speed, it can be concluded that the timing impact of the tracing mechanisms is visible in the measurements but its impact on the conclusions drawn from the measurements is negligible.

There are many possibilities to implement a tracing mechanism in the Linux kernel. Even when the Linux kernel would not provide any tracing mechanism, it is still possible to modify the kernel, compile it and start the measurements. Such an approach, however, has significant drawbacks due to the following base requirements for selecting a suitable tracing mechanism:

- The impact of the tracing mechanism on the measurements must be negligible.
- Measurements should be generated at runtime of a stock kernel such as delivered by Linux distributions. This means that the application of kernel patches which requires a re-compilation and reboot of the kernel would be detrimental. For example, kernels installed on target systems can readily be tested and measurements can be obtained using SystemTap.
- The measurements should be repeatable on newer kernel versions without much effort.

Kernel Instrumentation

For the testing performed for this study, a kernel extension has been developed that hooks at well-defined code locations of the Linux-RNG. This approach has been chosen as this mechanism covers all aforementioned concerns, is easy to use and is automatable.

The kernel patch maintains a ring-buffer for each value that is intended to be extracted from the kernel. The ring buffer contains 1,024 32-bit words. A 32-bit data word is stored in the ring buffer by selecting the next unused word. If the ring buffer is full, the oldest value is overwritten.

The instrumentation patch provides one DebugFS file per ring buffer allowing user space to read the contents of the ring buffer.

Data collection is performed when either user space requests data by reading the mentioned DebugFS files or by setting a kernel command line option. With the kernel command line option, all data is stored in the ring buffer that is received right from the very start of the kernel. When this kernel command line option is set, the data is collected in the ring buffer but when the buffer is full, collection ceases. With this approach, the collection of first event data during boot time can be collected.

The instrumentation exports callback functions which:

- Collect one 32-bit word, and
- Return an indication whether the data was collected.

These hooks are inserted at the following locations in the Linux-RNG code base:

- The function add_timer_randomness exports a 32-bit value that contains the 24-bit time stamp of the event and 8 bits holding the entropy estimation applied by the Linux-RNG.
- The function add_interrupt_randomness exports the 32-bit value of the CPU time stamp.
- The function try_to_generate_entropy exports the 1,024 time stamps that are generated by the operation.

The kernel instrumentation is provided as a kernel patch that is to be applied before compiling the kernel. Yet, its only dependency is to have the DebugFS file system support compiled. This file system is commonly compiled.

The ring buffer data can be directly read from the DebugFS files exported by the kernel instrumentation patch. Yet, to format the data nicely, a small user space application is provided. This tool reads the DebugFS files with a data size that is a multiple of 32 bits to prevent data truncation. The received buffer is written to STDOUT with one 32-bit word per line as decimal integer value.

The kernel instrumentation tool has been derived from the LRNG test framework.

For more details about the usage of the kernel instrumentation, see section 6.2.

Impact of Measurement on Test Results

The raw entropy gathering framework only have an impact on the timing behavior of the Linux kernel. The functionality of the entire kernel remains unchanged. Thus, only the aforementioned consideration regarding the timing impact is applicable.

Test Execution

The tests specified in chapters 6 and following use the test code instrumenting the Linux-RNG to collect the relevant data.

Besides following the instructions in the different sections regarding the test invocation, no additional operations are needed.

Listing of Used Hardware and Software

The testing was executed on the following hardware:

- Thinkpad T530 used for the native hardware early boot entropy tests documented in 6.3.3:
  - 2 core CPU with two hyperthreads per core
  - Intel(R) Core(TM) i7-3520M CPU @ 2.90GHz
- QEMU 6.1.0 used for the virtual environment early boot entropy tests documented in 6.3.1:

- 19 virtual CPUs corresponding with the 10 cores and their 20 hyperthreads provided by each core of the host system
- Intel Core Processor (Comet Lake, no TSX)

Reference Documentation

TR021021	BSI: BSI - Technical Guideline Cryptographic Mechanisms: Recommendations and Key Lengths
AIS2031	Wolfgang Killmann, Werner Schindler: A proposal for: Functionality classes for random number generators
RFC7539	Y. Nir, A. Langley: RFC 7539: ChaCha20 and Poly1305 for IETF Protocols
SP800-90B	Meltem Sönmez Turan, Elaine Barker, John Kelsey, Kerry McKay: NIST Special Publication 800-90B Recommendation for the Entropy Sources Used For Random Bit Generation
INTELDRNG	Intel: Intel Digital Random Number Generator (DRNG) Software Implementation Guide
SP800-90C	Elaine Barker, John Kelsey: NIST Special Publication 800-90C Recommendations for Random Bit Generator (RBG) Constructions
T06	Theodore Ts'o: Re: /dev/random on Linux http://lkml.org/lkml/2006/5/16/300
GPR06	Zvi Gutterman, Benny Pinkas, Tzachy Reinmann: Analysis of the Linux Random Number Generator http://eprint.iacr.org/2006/086
FIPS180-4	NIST: FIPS PUB 180-4 Secure Hash Standard (SHS)
CHACHA20	Daniel J. Bernstein: ChaCha, a variant of Salsa20
SP800-38A	NIST: Special Publication 800-38A Recommendation for Block Cipher Modes of Operation
SP800-90A	Elaine Barker, John Kelsey: NIST Special Publication 800-90A Recommendation for Random Number Generation Using Deterministic Random Bit Generators
LRSV12	Patric Lacharme, Andrea Röck, Vincent Strubel, Marion Videau: The Linux Pseudorandom Number Generator Revisited http://eprint.iacr.org/2012/251
P12	Benjamin Pousse: Short communication: An interpretation of the Linux entropy estimator http://eprint.iacr.org/2012/487
LRNGREPLACEMENT	Stephan Müller: Linux Random Number Generator - A New Approach
LRNGVIRT	Stephan Müller: Analysis of Random Number Generation in Virtual Environments
SP800-131A	Elaine Barker, Allen Roginsky: NIST Special Publication 800-131A Revision 1 Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths

Keywords and Abbreviations

Abbreviation	Description
AES	Advanced Encryption Standard (FIPS 197)
API	Application Programming Interface
BSI	Bundesamt für Sicherheit in der Informationstechnik (Federal Office for Information Security)

Abbreviation	Description
CTR	Counter mode as defined in SP800-38A
DRBG	Deterministic Random Bit Generator (see SP800-90A)
DRNG	Deterministic Random Number Generator
FIFO	First-In First-Out
FIPS	Federal Information Processing Standard
GCC	GNU Compiler Collection – When referenced in this document, the C compiler component is referred to
HID	Human Interface Devices
HSM	Hardware Security Module
IID	Independent and identically distributed
IOCTL	Input / Output Control (Linux kernel system call)
LFSR	Linear Feedback Shift Register
LSR	Longest Repeated Substring (as defined in SP800-90B)
LSB	Least Significant Bit(s)
MSB	Most Significant Bit(s)
NDRNG	Non-deterministic Random Number Generator
NUMA	Non-Uniform Memory Access
RNG	Random Number Generator
UUID	Universally Unique Identifier