Federal Office
for Information Security

Documentation and Analysis of the Linux Random Number Generator

Version: 4.11

Document history

Version	Date	Editor	Description
4.0	2020-07-15	Stephan Müller	Kernel version 5.6
4.1	2020-07-15	Stephan Müller	Kernel version 5.7
4.2	2020-08-03	Stephan Müller	Kernel version 5.8
4.3	2020-10-12	Stephan Müller	Kernel version 5.9
4.4	2021-01-05	Stephan Müller	Kernel version 5.10
4.5	2021-02-19	Stephan Müller	Kernel version 5.11
4.6	2021-06-18	Stephan Müller	Full test cycle of kernel version 5.12
4.6.1	2021-06-29	Stephan Müller	Incorporate clarifications
4.7	2021-06-29	Stephan Müller	Kernel version 5.13
4.8	2021-09-03	Stephan Müller	Kernel version 5.14
4.9	2021-11-08	Stephan Müller	Kernel version 5.15
4.10	2022-02-04	Stephan Müller	Kernel version 5.16
4.11	2022-04-27	Stephan Müller	Kernel version 5.17

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] and that they are fully met by /dev/random. Also, the study will evaluate the rationale why /dev/urandom, the `getrandom` system call and its in-kernel equivalent of the get_random_bytes function complies with the DRG.3 requirements defined 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

1.3 BSI-Reference

BSI Title: Analysis of the Linux Random Number Generator

BSI Project Number: 449

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 holdsideal 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. 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 based on an LFSR. The output function of the Linux entropy pool is based on the Blake2s 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.

Term	Definition
True Random Data	True Random Data is a data stream of arbitrary size that is believed to contain entropy. 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.

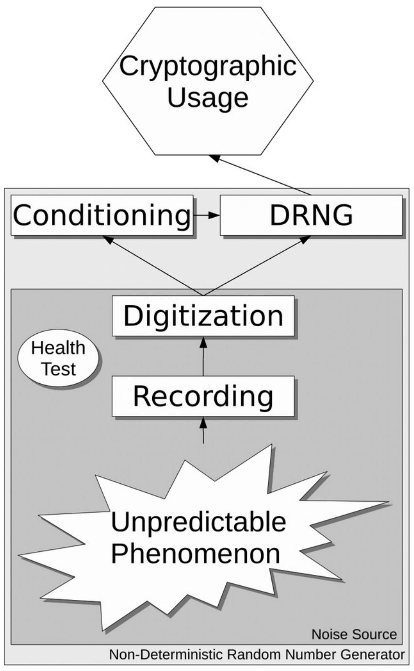

Figure 1: Non-deterministic random number generator architecture

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/urandom 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).
- 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 stream cipher 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.

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. The purpose of the entropy pool called input_pool is to collect, and compress, and thus accumulate the entropy provided by the different noise sources. A ChaCha20 DRNG is seeded from the input_pool to produce random numbers for the user space interface of /dev/urandom, /dev/random, the system call `getrandom` and the in-kernel application programming interface (API) of `get_random_bytes`.

The term “entropy pool” in this document refers to a memory area holding random data which is processed with a deterministic input and state-transition function based on an LFSR. The output function of an entropy pool is based on the Blake2s hash function. Considering the state-transition and output function, the processing of the data maintained by an entropy pool is fully deterministic in nature.

The reference to a ChaCha20 DRNG in this document is conceptually similar to an entropy pool: a memory buffer that holds random data. 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 mixed into the entropy pool that has a size of 4096 bits. This entropy pool is called input_pool.

Upon request, this entropy pool feeds the DRNG based on the ChaCha20 stream cipher.

So far, the description outlined that all interfaces obtain random numbers from the same DRNG, the ChaCha20 DRNG. This begs the question of the difference between these interfaces. The key difference is the timing when random numbers are generated in relationship to the 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 128 bits of initial entropy. After reaching that threshold of 128 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[2]:

- 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, the entire entropy content is hashed using Blake2s. Therefore, the Blake2s hash operation is the output function used for the input_pool, also referenced as entropy pool throughout the remainder of this document. The calculated Blake2s hash value is truncated to return only the 128 most significant bits to seed the ChaCha20 DRNG. The resulting 128 bits are the random number that is injected into the ChaCha20 DRNG To ensure backtracking resistance, the entire 256 bits of the Blake2s message digest are also mixed into the input_pool. In case the caller requested more data, the process of generating the Blake2s hash, truncating it, providing it to the caller, and mixing it back into the entropy pool is repeated until the requested number of bytes are generated or insufficient entropy is detected. The function which mixes data into the entropy pool is based on an LFSR. This implies that the LFSR is the state-transition function of the entropy pool.

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 stream cipher. The application of the ChaCha20 stream cipher 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. After a caller's request is satisfied, 256 bits of unused ChaCha20 block function data is XORed with the key part of the ChaCha20 state defined in [RFC7539], chapter 2, to ensure enhanced backward secrecy. The ChaCha20 DRNG is seeded by the input_pool by XORing the input_pool data with the key part of the ChaCha20 state.

3.2.1 Linux-RNG Internal Design

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

2 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.

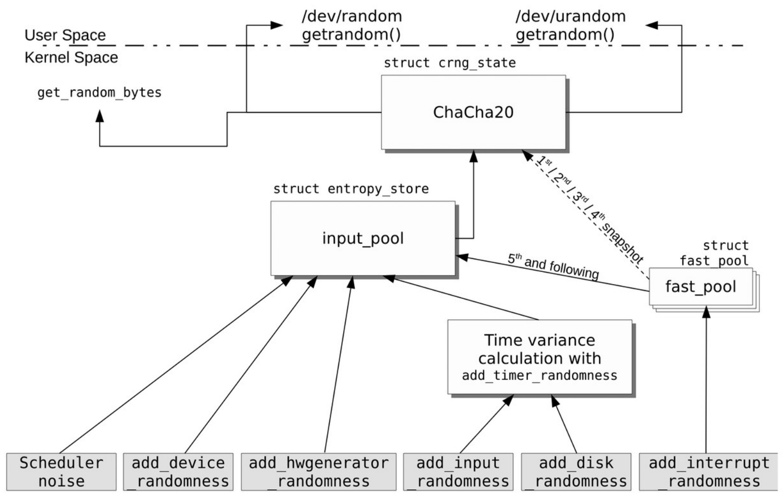

Figure 2: Relationship of entropy pool, ChaCha20-DRNG and entropy sources

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 has a size of 4,096 bits. The purpose of the input_pool is to collect entropy from the noise sources and provide it to the deterministic random number generator discussed in the following bullet point.
- The ChaCha20 DRNG obtains its seed data from the input_pool and is accessible through the following interfaces:
  - from user space via /dev/urandom, /dev/random 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 keyboard 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 128 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. The pool is governed by a structure which contains the following important information:

```
static u32 input_pool_data[POOL_WORDS] __latent_entropy;


static struct {

...

        u16 add_ptr;

        u16 input_rotate;

        int entropy_count;

}
```

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

- The variable `input_pool_data` contains the pointer to the static data block that holds the actual random pool. The trailing keyword of __latent_entropy refers to the latent entropy GNU Compiler Collection (GCC) plugin which will be discussed in section 3.5.2.7.
- The variable `add_ptr` holds the index to the current pool word that was accessed when mixing data into the random pool. For more information about the pool index, see section 3.3.1.1
- The variable `entropy_count` holds the entropy estimator value used to determine how much entropy is currently stored in the entropy pool.
- The variable `input_rotate` contains the input data rotation value that is used when mixing new input value into the random pool as explained in section 3.3.1.1.

The state data structure is instantiated for each entropy pool during compile time.

As discussed, an LFSR is used to insert new values into the entropy pool. The following code describes the polynomial used for the LFSR. See section 3.3.1.1 for the description on how the polynomials are applied.

```
enum poolinfo {
        POOL_WORDS = 128,
        POOL_WORDMASK = POOL_WORDS - 1,
        POOL_BYTES = POOL_WORDS * sizeof(u32),
        POOL_BITS = POOL_BYTES * 8,
        POOL_BITSHIFT = ilog2(POOL_BITS),
...
        /* x^128 + x^104 + x^76 + x^51 +x^25 + x + 1 */
        POOL_TAP1 = 104,
        POOL_TAP2 = 76,
        POOL_TAP3 = 51,
        POOL_TAP4 = 25,
        POOL_TAP5 = 1,
...
}
```

3.3.1.1 Entropy Pool State Transition Function

The Linux-RNG maintains the entropy pool by using an LFSR discussed in this section.

When data is received that is to be inserted into the entropy pool, the data and the existing state of the entropy pool are processed using an LFSR. That data is mixed into the entropy pool using the `mix_pool_bytes` function. The function uses a reference to the entropy pool the given data shall be mixed into, the data to be mixed in and the size of the data buffer to be mixed in. The actual work for mixing data into the pools is done by `_mix_pool_bytes`. Its logic follows that of a linear shift register with a twist as discussed below.

```
static u32 const twist_table[8] = {
        0x00000000, 0x3b6e20c8, 0x76dc4190, 0x4db26158,
        0xedb88320, 0xd6d6a3e8, 0x9b64c2b0, 0xa00ae278 };
```

```
...
 * The pool is stirred with a primitive polynomial of the appropriate
 * degree, and then twisted.  We twist by three bits at a time because
 * it's cheap to do so and helps slightly in the expected case where
 * the entropy is concentrated in the low-order bits.
 */
static void _mix_pool_bytes(struct entropy_store *r, const void *in,
                                   int nbytes)
{
...
        input_rotate = input_pool.input_rotate;
        i = input_pool.add_ptr;

        /* mix one byte at a time to simplify size handling and churn
faster */
        while (nbytes--) {
                w = rol32(*bytes++, input_rotate);
                i = (i - 1) & POOL_WORDMASK;

                /* XOR in the various taps */
                w ^= input_pool_data[i];
                w ^= input_pool_data[(i + POOL_TAP1) & POOL_WORDMASK];
                w ^= input_pool_data[(i + POOL_TAP2) & POOL_WORDMASK];
                w ^= input_pool_data[(i + POOL_TAP3) & POOL_WORDMASK];
                w ^= input_pool_data[(i + POOL_TAP4) & POOL_WORDMASK];
                w ^= input_pool_data[(i + POOL_TAP5) & POOL_WORDMASK];

                /* Mix the result back in with a twist */
                input_pool_data[i] = (w >> 3) ^ twist_table[w & 7];

                /*
                 * Normally, we add 7 bits of rotation to the pool.
                 * At the beginning of the pool, add an extra 7 bits
                 * rotation, so that successive passes spread the
                 * input bits across the pool evenly.
                 */
                input_rotate = (input_rotate + (i ? 7 : 14)) & 31;
        }


        input_pool.input_rotate = input_rotate;
        input_pool.add_ptr = i;
}
```

The mixing function operates on the entropy pool and performs the following steps to mix the input data into the random pool state of `input_pool_data`:

Fetch one byte of the input data starting at the offset of the number of loops (i.e. first loop iteration implies 1 byte offset, second loop iteration implies 2 bytes offset), and cast the one byte into a four-byte variable. The casting operation fills the leading 3 bytes with zeros. After the casting, the logic left-rotates the bit-representation of these 4 bytes by the value resulting from the mathematical operation `input_rotate & 31` that discards the high 27 bits and only leaves the low 5 bits. This operation “slides” the one input byte within the 4 byte buffer such that over time each bit position of the input byte will be hit each byte of the 4 byte buffer with an equal probability. The variable `input_rotate` is incremented by 7 unless the word 0 is processed where the variable is incremented by 14 at the end of processing one input byte. This approach to increment the rotation pointer ensures that “successive passes spread the input bits across the pool evenly”.

1 Pick the index which points to the entropy pool word that is to be updated. This implies that one processed byte from the input updates one 32-bit word in the entropy pool. The entropy pool words are accessed sequentially.

2 XOR the 4 bytes from step 1 with:

   a the current word of the entropy pool pointed to by the at the index value obtained in step 1, and

   b the current value of the pool word pointed to by the index plus the first tap (i.e. exponent value) of the LFSR polynomial. All other pool words pointed to by the index plus the respective other taps of the LFSR are also obtained for the XOR operation. Note that this index is wrapped if needed. All selected words using the offsets defined by the polynomial implements the LFSR.

3 The u32 value calculated by XORing the input with the 5 different entropy pool words pointed to by the taps of the polynomial from step 2 is further stirred by XORing it with one value of the `twist_table`. This operation permutates the first three bits of the word using a bijective operation. The idea is that these three bits are mixed using the “twist”.

4 The value calculated in step 3 replaces the previously existing value as the new value of the pool word pointed to by the index in step 1.

5 Repeat with step 1 until all input bytes are mixed into the pool value.

The result of the entire operation is that data with an arbitrary length can be mixed (the Linux-RNG source code uses the term “stir”) into the entropy pool.

Figure 3 illustrates the update of one pool value of the `input_pool_data` state variable associated with the input_pool. The following figure uses the values specified with the polynomial defined for the input_pool. The figure assumes that the index of the pool member variable to be changed is 40 (i.e. i==40).

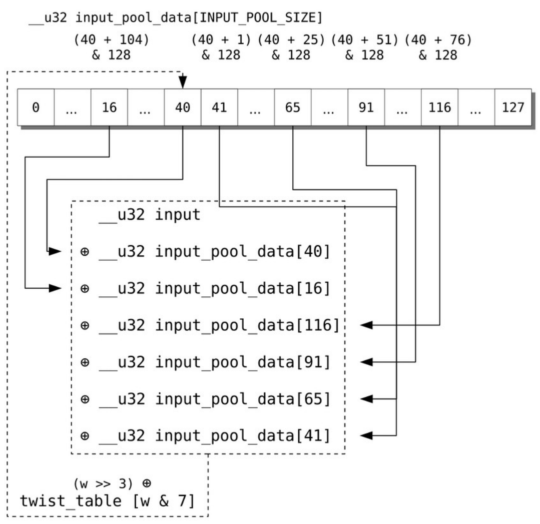
__u32 input_pool_data[INPUT_POOL_SIZE]
(40 + 104) & 128
(40 + 1) & 128
(40 + 25) & 128
(40 + 51) & 128
(40 + 76) & 128
0
...
16
...
40
41
...
65
...
91
...
116
...
127
__u32 input
⊕ __u32 input_pool_data[40]
⊕ __u32 input_pool_data[16]
⊕ __u32 input_pool_data[116]
⊕ __u32 input_pool_data[91]
⊕ __u32 input_pool_data[65]
⊕ __u32 input_pool_data[41]
(w >> 3) ⊕ twist_table [w & 7]

Figure 3: Update of pool value for input_pool

The entropy pool word 40 is replaced with the following 32 bit values XORed:

- the one input byte that is currently being processed and expanded to 32 bits,
- the content of the entropy pool word 40,
- the content of the entropy pool word 41 – i.e. 40 + first LFSR tap value,
- the content of the entropy pool word 65 – i.e. 40 + second LFSR tap value,
- the content of the entropy pool word 91 – i.e. 40 + third LFSR tap value,
- the content of the entropy pool word 116 – i.e. 40 + fourth LFSR tap value,
- the content of the entropy pool word 16 – i.e. 40 + fifth LFSR tap value wrapped at 127 (note: the wrapping occurs at 127 which is the last of the 128 entropy pool words considering the words are counted starting with zero).

The resulting 32 bit value is right-shifted by 3 bits. The "lost" 3 bits from the right-shift operation is used as an index into the array of `twist_table` with 8 different entries. The selected twist_table entry is finally XORed with the right-shifted 32 bit value.

Note that the function to mix data into the pool does not update the entropy estimator which will be discussed in the subsequent sections.

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 denominate the amount of entropy present in the entropy pool.

The entropy estimator value will never be larger than the corresponding entropy pool is in size, because an entropy pool can hold at most as much entropy as it is in size. The entropy estimator on the other side cannot fall below zero. It is key that the entropy estimator must be processed with the same dimension as the value it is compared or processed with. When considering the size of the entropy pool in bytes, the entropy estimator must be processed in bytes. If for example the newly provided input entropy is measured in bits, the entropy estimator must be processed in bits. Thus, the Linux-RNG applies an appropriate conversion logic to the value of the entropy estimator depending on the processed data as discussed below.

The basic concept of maintaining the entropy estimator can be summarized with the following bulleted list:

- If new data is mixed into the entropy pool, the entropy estimator is increased by the heuristically determined entropy content associated with the mixed-in data. The entropy estimation heuristic is outlined in section 3.6. This section also explains how the estimated entropy content value is used to increase the entropy estimator of the respective entropy pool.
- When random data is obtained from the entropy pool, the number of generated random bytes is simply subtracted from the entropy estimator. The process of extracting random numbers and the implicit decrease of the entropy estimator is outlined in section 3.3.1.3.

A subtle detail is important for the debiting as well as crediting entropy to the entropy estimator. The entropy estimator integer value denominates the entropy in fraction of bits. Considering the use of a 32-bit integer value, the question must be raised how fractions can be processed. Fractions are maintained by declaring the three LSB of the integer value as values right of the decimal point. All remaining 29 MSB of the integer value denominate values left of the decimal point. Figure 4 illustrates the entropy estimator integer value.

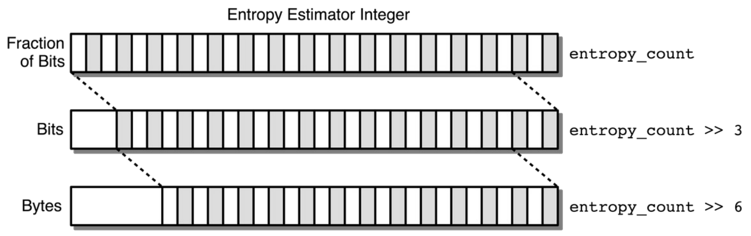

Figure 4: Entropy estimator integer value conversion

The code that credits and debits or simply reads the entropy estimator ensures that the proper integer value is used that corresponds with one of the following use cases:

- When operating on the integer value with fractions of bits, the integer value is used unchanged.
- In case the Linux-RNG code requires obtaining the amount of entropy in whole numbers of bits present in an entropy pool, it uses the entropy estimator and shifts its value right by 3 bits.
- When the Linux-RNG operation requires processing the entropy content of an entropy pool in bytes, the entropy estimator is shifted right by 6 bits.

For example: The entropy estimator contains a value of 132. This value has the third and eighth bit set. Thus, the integer value is to be interpreted as:

- 132 one-eighth of a bit,
- 132 / 8 = 132 / 2^3 = 132 >> 3 = 16 bits (note, due to the use of integer arithmetic, the fractions of a value are discarded),
- 16 / 8 = 16 / 2^3 = 16 >> 3 = 132 / (2^3*2^3) = 132 / 2^6 = 132 >> 6 = 2 bytes.

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. Also, it contains the FIPS 140-2 related continuous self-test to verify that the previously calculated random value is different from the current random value.

With respect to the extraction of random numbers, the function `_extract_buf` is invoked which implements the output function for the entropy pool. Before the `extract_buf` function is called to transfer the actual data, an update of the entropy estimator of the entropy pool producing the requested number of random bytes is performed using the `account` function. That function is given the number of bytes to be obtained from that entropy pool requested by the caller. First, the function verifies that sufficient entropy is available. If not, it reduces the number of random bytes to be obtained from the entropy pool to the available entropy which implies that the caller only receives the available amount of entropy. Secondly, it reduces the entropy estimator by the number of bytes determined by the previous check before extracting and transferring the actual random bytes.

The code outlined below must always consider that the entropy estimator integer value denominates the entropy content in fractions of bits where the value for requested random number is in bytes. Therefore, the function implements the proper conversion from fraction of bits into bytes. The service function returns the number of bytes that are deemed available.

```
static size_t account(size_t nbytes, int min)
{
...
        entropy_count = orig = READ_ONCE(r->entropy_count);
        /* never pull more than available */
        ibytes = min_t(size_t, nbytes, entropy_count >>
(POOL_ENTROPY_SHIFT + 3));

                if (ibytes < min)
                ibytes = 0;
        nfrac = ibytes << (ENTROPY_SHIFT + 3);
        if ((size_t) entropy_count > nfrac)
                entropy_count -= nfrac;
        else
                entropy_count = 0;

        if (cmpxchg(&r->entropy_count, orig, entropy_count) != orig)
...
        return ibytes;
}
```

In the first step of the `account` function, the entropy estimator for the processed entropy pool is read. As the `account` function is invoked when data is obtained from the entropy pool, the `account` function reduces the entropy estimator by the requested amount of bytes, if sufficient entropy is available. Hence, the debit operation must verify that the entropy estimator does not become negative. Therefore, after reading the entropy estimator, the code compares the amount of bytes requested by the caller with the entropy estimator value and obtains the smaller integer of both. This obtained value is now subtracted from the entropy estimator value of the entropy pool. The code converts the values as the entropy estimator maintains the entropy in fractions of bits whereas initially bytes were processed. The conversion is performed with the bit shift operation – the code replaces the macro value of `ENTROPY_SHIFT` with 3. Thus, the conversion operation shifts the integer values by 6 to achieve the conversion between fraction of bits to bytes. After completing this debit operation, the calculated entropy estimator value is stored back into the variable field of the entropy pool.

The `account` function also wakes up any process that polls the /dev/random device file when it falls below the threshold stored in `random_write_wakeup_thresh`. The idea of this wake-up is to allow user space application to provide more entropy to the Linux-RNG once the entropy estimator signals a low entropy content. As mentioned in section 3.4.1.1, that threshold can be modified at runtime with a file having the same name present under the /proc directory structure.

The number of bytes that were debited is returned by `account` and is now used to perform the extraction of the random value using the function `extract_buf` with the following steps:

1 When extracting a random number, a Blake2s hash is calculated of the entire entropy pool. But before the Blake2s hash is calculated, the Blake2s hash is initialized. The memory used for the Blake2s operation is initialized with the Blake2s constants defined in the algorithm specification. However, if the CPU provides a hardware random number generator, the memory holding the 4th through 7th Blake2s state constants are XORed with the output of that random number generator. This means that in this case, the official Blake2s operation is not used, but a variant with the identical operation, but with a different initialization. Nonetheless, the subsequent description still refers to the operation as Blake2s.

2 The entire entropy pool is processed by the Blake2s transformation which implies that the Blake2s hash is calculated across the entire entropy pool content.

3 The resulting Blake2s hash is mixed into the entropy pool with the process discussed in section 3.3.1.1.

4 The calculated Blake2s hash is truncated to return only the 128 most significant bits.

5 The 16 bytes calculated in step (4) are returned.

The truncation of the Blake2s hash is implemented with the following code snippet:

```
static void extract_buf(, u8 *out)
{
...
        /* Note that EXTRACT_SIZE is half of hash size here, because
above
         * we've dumped the full length back into mixer. By reducing the
         * amount that we emit, we retain a level of forward secrecy.
         */
        memcpy(out, hash, EXTRACT_SIZE);
        memzero_explicit(hash, sizeof(hash));
...
```

The function `extract_buf` is intended to be invoked repeatedly to generate as many random bytes as requested by the caller. If the caller requires data that is not a multiple of 128 bits, the final data block is truncated to the most significant bits to satisfy the data request.

3.3.1.4 Initialization

When the Linux-RNG is initialized, the entropypool 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 `rand_initialize`. This function calls `init_std_data` for the entropy pool.

```
static void init_std_data(void)
```

```
{
...
        ktime_t now = ktime_get_real();
...
         mix_pool_bytes(&now, sizeof(now));
        for (i = POOL_BYTES; i > 0; i -= sizeof(rv)) {
                if (!arch_get_random_seed_long(&rv) &&
                    !arch_get_random_long(&rv))
                        rv = random_get_entropy();
                mix_pool_bytes(r, &rv, sizeof(rv));
        }
        mix_pool_bytes(utsname(), sizeof(*(utsname())));
}
```

The function `init_std_data` 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 last 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. 5.6.0 for the kernel version of 5.6.0 – 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)

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. The ChaCha20 DRNG is therefore maintained with the following data structure:

```
struct crng_state {
        u32           state[16];
        unsigned long   init_time;
...
};
```

The member variables are used to store the following information:

- The `state` array of 16 32-bit words holds the ChaCha20 stream cipher state according to [RFC7539] section 2.3. This section defines the state of a ChaCha20 operation which is identical to the state used by the Linux-RNG in the discussed array denominated by the `state` variable. This array can therefore be segmented into the following parts:
  - The first four words hold the constants used by the ChaCha20 block operation.
  - The next eight words hold the key information used by the ChaCha20 block operation.
  - The thirteenth word is the counter.
  - The fourteenth to sixteenth words are the nonce.
- The `init_time` variable contains the time when the ChaCha20 DRNG was seeded last. The ChaCha20 DRNG is automatically reseeded after 5 minutes irrespective of the amount of data produced by the DRNG.

The DRNG initial state and state update is depicted in figure 5.

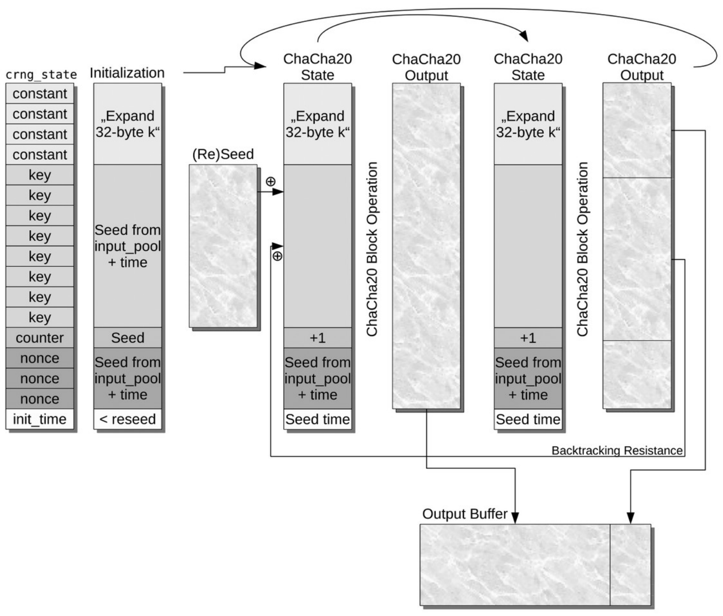

Figure 5: ChaCha20 DRNG state and state transition

The left-most column in figure 5 shows the ChaCha20 DRNG state with its various u32 words as described in the previous bulleted list.

The second left column indicates the ChaCha20 DRNG state after its initialization operation:

- The constants are filled with the 16-bytes string "Expand 32-byte k". This string is derived from the reference implementation of ChaCha20 as provided by Daniel Bernstein and found at the URL http://cr.yp.to/chacha.html.
- The key part, the counter, and the nonce are filled with random data extracted from the current content of the input_pool (which is considered to have hardly any entropy at the time the ChaCha20 DRNG initializes).
- The init_time value is set such that the reseed is triggered with the next request to the ChaCha20 DRNG to generate random numbers.

3.3.2.1 ChaCha20 DRNG State Transition Function and Output Function

The ChaCha20 DRNG operation can be explained when considering how the ChaCha20 stream cipher operates. ChaCha20 is conceptually very similar to the counter block chaining mode defined in [SP800-38A]. Using the ChaCha20 state, the ChaCha20 block operation generates a 64-byte output block from the state. After the generation of the output block, the counter value in that state is increased by one. With the updated state which differs only slightly from the previous state based on the increment operation, a new ChaCha20 block operation can be performed resulting in another output block, and so on. The generated blocks are concatenated to form a bit-stream.

The ChaCha20 stream cipher uses the generated sequence of output blocks as a key bit-stream. That key bit-stream is XORed with the plaintext or ciphertext to perform the ChaCha20 encryption or decryption operation.

The ChaCha20 DRNG uses the aforementioned key bit-stream as a stream of random numbers (the term random bits would be a more appropriate reference in this case). The DRNG therefore operates using the following steps that are implemented with the invocation of the function `extract_crng` and its repeated invocation:

1 If a CPU hardware random number generator is present, generate a 32-bit data block and XOR it with the second nonce value. This operation is intended to stir the ChaCha20 state.

2 Generate one 64-byte output block from the ChaCha20 DRNG state with the ChaCha20 block operation defined in [RFC7539], sections 2.2 and 2.3. Both sections explain in detail how the ChaCha20 Quarter Round operation is applied to the state and how the Quarter Rounds are formed into one complete ChaCha20 block operation.

3 Increment the counter variable by one.

4 Invoke steps 1 through 3 again as often as needed in order to produce sufficient output blocks to satisfy the requested number of bytes. The number of generated ChaCha20 output blocks can be defined as: $\lceil \frac{(requested\, bytes)}{(64\, bytes)} \rceil$. The number of generated blocks are concatenated to form the random number. If the caller requested random numbers that are not divisible to the block size of ChaCha20, the required most significant bits of the last ChaCha20 block are used to completely satisfy the requested random number size. For example, if the caller requests 80 bytes of random numbers, two ChaCha20 output blocks are generated. The first block forms the first 64 bytes. The 16 most significant bytes from the second block are used to satisfy the remaining part of the requested number size.

5 After a request for random numbers is satisfied, 256 bits from a yet unused ChaCha20 output block (i.e. data not given to a user) are XORed with the key part of the ChaCha20 DRNG. This operation implies a non-revocable change of the ChaCha20 state to support enhanced backward secrecy. The required 256 bits are obtained as follows: if the last output block generated to satisfy the caller's request has at least 256 bits left that were not returned to the caller, the leftmost 256 unused bits are used for the state update. If less than 256 unused bits are present, another ChaCha20 block operation is performed to generate 512 new bits. The 256 most significant bits are used to update the state. Any remaining unused bits are discarded. Continuing the previous example with the generation of 80 bytes, the second output block has yet 48 unused bytes remaining. From those 48 bytes, the 256 most significant bits are used. The remaining 16 bytes are discarded.

With this description, the following state transition functions are present:

- As long as one request for random numbers is not satisfied: increment the counter by one after each ChaCha20 block operation.
- After one request for random numbers is satisfied: XOR the key part of the ChaCha20 DRNG state with 256 unused bits from the last ChaCha20 block operation.

The output function of the ChaCha20 DRNG is identical with the ChaCha20 block operation.

Considering steps 2 through 5, the ChaCha20 DRNG has a very close relationship to the CTR DRBG without a derivation function and without prediction resistance defined in [SP800-90A]. But instead of using the Advanced Encryption Standard (AES) as cipher core, ChaCha20 is used. Another difference is evident with step 5: the CTR DRBG unconditionally generates a new AES block for updating its internal state whereas the ChaCha20 DRNG may use “left-over” bytes from the last output block that were not used by the caller.

The reseed operation of the ChaCha20 DRNG is implemented with the following steps:

1. Obtain 32 bytes from the input_pool. It may be possible that the input_pool returns less than the requested 32 bytes in case insufficient entropy is present. In case the underlying hardware system is a NUMA system and the ChaCha20 DRNG instance to be reseeded is a secondary ChaCha20 DRNG (see section 3.3.2.2), the required 32 bytes are obtained from the primary ChaCha20 DRNG. The primary ChaCha20 DRNG will always deliver the requested amount of bytes.
2. If RDSEED is available and delivers data on the current CPU, the 32 bytes obtained in step 1 are XORed with output from RDSEED. If RDSEED is not present or cannot deliver data and RDRAND is present, 32 bytes from RDRAND are obtained and XORed with the data obtained in step 1. If neither RDSEED nor

RDRAND are present and can deliver data, 8 high-resolution time stamps are XORed with the data from step 1.

3. The data from steps 1 and 2 are XORed with the key component of the ChaCha20 DRNG state. Figure 5 illustrates the reseed operation.

The ChaCha20 DRNG as used in the Linux-RNG produces unlimited amounts of random numbers between reseeds. It is reseeded after 5 minutes. The seeding operation of the ChaCha20 DRNG ensures that the entropy present in the seed data is evenly distributed over the ChaCha20 key based on the properties of ChaCha20.

3.3.2.2 ChaCha20 on Non-Uniform Memory Access (NUMA) Systems

Commonly, the Linux-RNG maintains one instance of the ChaCha20 DRNG. If the kernel is compiled with support for non-uniform memory access (NUMA), one secondary ChaCha20 DRNG instance is allocated per online NUMA node.

- One ChaCha20 DRNG instance is designated as primary ChaCha20 DRNG. This primary ChaCha20 DRNG is seeded with data from the input_pool. In a non-NUMA system, the one present ChaCha20 DRNG is identical to the primary ChaCha20 DRNG in a NUMA environment.
- For each online NUMA node, a secondary ChaCha20 DRNG is created whose memory that is used for its state is NUMA-node-local. When callers request data with one of the available interfaces, the kernel first identifies the NUMA node the caller operates on. The request for random numbers is processed by the ChaCha20 DRNG instance that is allocated for that NUMA node. Each secondary ChaCha20 DRNG obtains the seed data from the primary ChaCha20 DRNG instance.

The secondary ChaCha20 DRNGs are initialized identically to the primary ChaCha20 DRNG stated before with one small difference: instead of obtaining random numbers from the input_pool, the secondary ChaCha20 DRNG seed from the primary ChaCha20 DRNG.

The Linux-RNG ensures that the secondary per-NUMA-node-DRNGs are initialized after the primary DRNG is fully seeded. Prior to the availability of the per-NUMA-node-DRNGs, any request for random numbers is served with the primary DRNG.

3.3.2.3 ChaCha20: Initially or Fully Seeded

The Linux-RNG maintains the concept of an initially[3] or fully[4] seeded primary ChaCha20 DRNG. Initially versus fully seeded is defined as follows:

- During initialization time of the kernel, the kernel inject a fast_pool content into the primary ChaCha20 DRNG upon receipt of 64 interrupts by that fast_pool. When four of these injection operations are completed, the primary ChaCha20 DRNG is considered initially seeded. Note, the content of the fast_pool that was injected into the primary ChaCha20 DRNG is not used for injection into the input_pool. In addition, if data is received via the `add_hwgenerator_randomness` interface and the DRNG is not yet seeded, it is treated in the same way as the fast_pool: it is sent to the ChaCha20 DRNG immediately to initially seed the DRNG.
- During initialization time of the kernel, after the entropy estimator of the input_pool reaches 128 for the first time, the primary ChaCha20 DRNG is reseeded as documented in section 3.3.2.1 from the input_pool. Reaching this state indicates that the ChaCha20 DRNG is fully seeded.

Commonly, the initially seeded state is reached much earlier than the fully seeded state. In some rare cases it may be possible that the fully seeded state is reached earlier than the initially seeded state – in this case, the

3 The kernel prints the log message “random: fast init done” in this case.
4 The kernel prints the log message “random: crng init done”.

initially seeded state is skipped. However, the step to reach the fully seeded state, i.e. the reseed from the input_pool, is always executed.

If the ChaCha20 DRNG state initialization successfully used the RDRAND or RDSEED instruction, the ChaCha20 DRNG is treated as fully seeded already at initialization time. With this behavior, the Linux-RNG "trusts" the CPU-based noise sources to deliver data with sufficient entropy. This default behavior can be adjusted as follows:

- During compile time of the kernel, if the configuration option of CONFIG_RANDOM_TRUST_CPU is *not* set, the default behavior is that RDRAND and RDSEED are not considered trustworthy and thus the ChaCha20 DRNG is not considered fully seeded after initialization. This applies even if RDRAND/RDSEED successfully delivered random numbers. Note, if the kernel configuration variable is set, the ChaCha20 DRNG is seeded with 256 bits of data from the CPU random number generator and treated as fully seeded afterwards.
- At boot time of the Linux kernel, the kernel command line argument "random.trust_cpu" can be used to toggle the trusting of the RDRAND/RDSEED instructions. The toggling of the parameter changes the default behavior outlined above. If the kernel command line argument is set to true, the CPU-based noise sources are "trusted" and thus a successful reading of data from RDRAND/RDSEED at initialization time will cause the ChaCha20 DRNG to be treated as fully seeded. Conversely, if the kernel command line argument is set to false, the CPU-based noise sources are "not trusted". If the kernel command line argument is not set, the aforementioned default behavior applies.

The discussion in the subsequent section will explain when the notion of an initially or fully seeded primary ChaCha20 DRNG is relevant.

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,
        .write = random_write,
        .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,
        .write = random_write,
        .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 initial seeding operation of the ChaCha20 DRNG with 128 bits of entropy did not occur. 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.

- write: When insufficient entropy is available indicated by the fact that the entropy estimator falls below a given threshold, the kernel wakes up the processes that waited with a write poll. Although the poll system call backend is only implemented for /dev/random, the kernel wakes the polling process up when the entropy estimator of either /dev/random or /dev/urandom falls below the threshold. The threshold can be modified at runtime by writing a positive integer value into /proc/sys/kernel/random/ write_wakeup_threshold. That value specifies the threshold in bytes. 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` 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 `extract_crng` discussed in section 3.3.2.1 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`. `random_write` calls the `write_pool` 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 int
write_pool(const char __user *buffer, size_t count)
{
...
        __u32 buf[16];
...

        while (count > 0) {
                bytes = min(count, sizeof(buf));
...
                count -= bytes;
...
                mix_pool_bytes(buf, bytes);
...
```

The code listing shows that the user space data is mixed into the pool in 16 byte chunks.

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 using the logic discussed in section 3.6. 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 using the logic discussed in section 3.6. This IOCTL is restricted to the capability of `CAP_SYS_ADMIN`.
- `RNDZAPENTCNT`: Initialize the entropy pool and reset its content as discussed in section 3.3.1.4. This IOCTL is restricted to the capability of `CAP_SYS_ADMIN`.
- `RNDCLEARPOOL`: See `RNDZAPENTCNT`.
- `RNDRESEEDCRNG`: If the caller possesses the capability of `CAP_SYS_ADMIN`, the primary ChaCha20 DRNG is reseeded. In addition, all NUMA-node-local ChaCha20 DRNG instances will be reseeded next time they are invoked.

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 with the block size of ChaCha20 (512 bits) on each CPU. 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 `add_random_ready_callback` 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.

An example of the use case of this callback mechanism is given with the in-kernel SP800-90A DRBG implemented by the Linux kernel crypto API and used for generating the GCM IV for IPSec seeding operation. This DRBG is seeded with the following steps:

1 During initialization of the DRBG, the code tries to register a callback function with `add_random_ready_callback`. If that operation fails with the error code `-EALREADY`, the DRBG code knows that the Linux-RNG is fully seeded. In this case, it pulls the required amount of seed data from `get_random_bytes` and considers its state fully seeded. Otherwise, seed data from another noise source separate from the Linux-RNG is used and the DRBG applies the following steps.

2 Pull the required amount of data defined by the DRBG seed size from `get_random_bytes`. In addition, the DRBG pulls an equal amount of random data from the separate noise source (the Jitter RNG noise source present in the kernel crypto API). Both data blocks are used to initialize and seed the DRBG.

3 The reseeding threshold of the DRBG is set to 50 which means that the DRBG will reseed itself with step 2 after 50 requests of random numbers.

4 At some point the Linux-RNG considers itself fully seeded and triggers the registered callback function. That callback function now performs the following steps:

a Pull the required amount of data defined by the DRBG seed size from `get_random_bytes` and reseed the DRBG.

b Set the reseed threshold to the regular value compliant with SP800-90A.

The behavior of `add_random_ready_callback` 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.
- /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: when entropy_avail falls below that threshold, user space suppliers of entropy are woken up with the goal to inject entropy into the Linux-RNG.
- /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: currently unused.

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.

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;
};

#define INIT_TIMER_RAND_STATE { INITIAL_JIFFIES, };
```

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.

The macro `INIT_TIMER_RAND_STATE` can be used to initialize the `last_time` member variable to a value that causes a wrapping of the value 5 minutes after boot. The idea here is that potential bugs hiding in the kernel code can be found faster whereas the debugging setting itself is irrelevant to the operation of the Linux-RNG.

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 = INIT_TIMER_RAND_STATE;
```

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[5]. 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 rand_initialize_disk(struct gendisk *disk)
{
        struct timer_rand_state *state;

        /*
         * If kzalloc returns null, we just won't use that entropy
         * source.
         */
        state = kzalloc(sizeof(struct timer_rand_state), GFP_KERNEL);
        if (state) {
                state->last_time = INITIAL_JIFFIES;
                disk->random = state;
        }
}
```

5 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.

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 {
        u32            pool[4];
        unsigned long   last;
        unsigned short  reg_idx;
        unsigned char   count;
};
...
static DEFINE_PER_CPU(struct fast_pool, irq_randomness);
```

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. Except to seed the primary ChaCha20 DRNG during early boot, 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 at least 64 interrupts have been received before the fast_pool can be transferred to the input_pool. Finally, the `reg_idx` variable is the index which CPU register shall be mixed into the fast_pool. This index is incremented by one each time an interrupt is received for the given fast_pool. The macro `DEFINE_PER_CPU` implies that an instance of the fast_pool data structure is allocated for each CPU.

The entropy calculation discussed in section 3.6 requires the information from the timer state.

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[6].

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;
...
        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 discussed in section 3.5.1. This means that one time state variable is maintained for all input device events.

6 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.

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 u32 words in size and therefore contains 128 bits. In addition, the fast_pool maintains a variable called `reg_idx` which may be used to sequentially point to a CPU register whose content may be used to be mixed into the entropy pool as discussed below.

One instance of the fast_pool is created per CPU. Depending on which CPU an interrupt is received from, the fast_pool of that CPU is used. The event values as well as the time stamps 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 following requirements are all met:

- the receipt of at least 64 interrupts that are mixed into the current fast_pool – this case is tracked with the count variable of the fast_pool, and
- 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.

Figure 6 illustrates the occurrence of an interrupt on a particular CPU, the mixing into the applicable fast_pool and the final mixing with the input_pool.

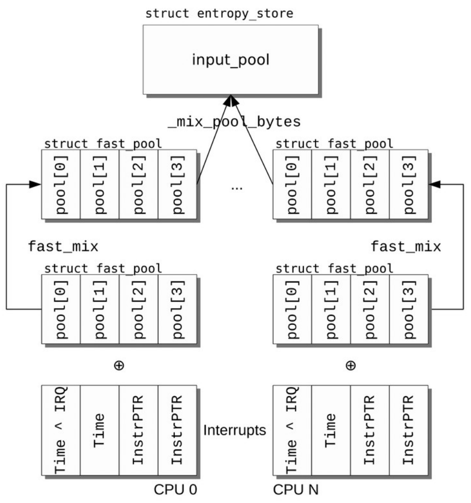

Figure 6: Processing of interrupts by fast_pools and connection to input_pool

For every interrupt that is received by the Linux kernel, the four u32 words of the fast pool are updated as depicted in figure 6 at the bottom. These words are updated with the following data:

- `fast_pool->pool[0]` – the first fast_pool word: This word is the XOR-combination of the low 32 bits of the high-resolution time stamp (processor cycles), the 32 high bits of the coarse Jiffies time stamp and the interrupt number together with the existing value in the word. Details of the time stamps are given in section 3.5.2.8. However, one key difference to section 3.5.2.8 is evident: the fast_pool operation uses absolute time stamps instead of time variances.
- `fast_pool->pool[1]` – the second fast_pool word: the lower 32 bits of the Jiffies time stamp is combined with the high 32 bits of the high-resolution time stamp together with the existing value in that word using XOR.
- `fast_pool->pool[2]` and `fast_pool->pool[3]` – the third and fourth fast_pool word: the upper and lower 32 bit of the 64 bit value of the CPU instruction pointer is XORed with the existing values in those words. If this value is not available, the return address of the add_interrupt_randomness function is used, which is static for the given kernel binary.

The following code shows the mixing of the interrupt into the random number state.

```
void add_interrupt_randomness(int irq)
{
...
        struct fast_pool *fast_pool = this_cpu_ptr(&irq_randomness);
        struct pt_regs *regs = get_irq_regs();
        unsigned long now = jiffies;
        cycles_t cycles = random_get_entropy();
...
        c_high = (sizeof(cycles) > 4) ? cycles >> 32 : 0;
        j_high = (sizeof(now) > 4) ? now >> 32 : 0;
        fast_pool->pool[0] ^= cycles ^ j_high ^ irq;
        fast_pool->pool[1] ^= now ^ c_high;
        ip = regs ? instruction_pointer(regs) : _RET_IP_;
        fast_pool->pool[2] ^= ip;
        fast_pool->pool[3] ^= (sizeof(ip) > 4) ? ip >> 32 :
                get_reg(fast_pool, regs);

        fast_mix(fast_pool);
...
```

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. This stir operation tries to combine the content of all four words with the goal of distributing the information of each of the words evenly among the words.

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.

```
void add_interrupt_randomness(int irq)
{
...
        if ((fast_pool->count < 64) &&
            !time_after(now, fast_pool->last + HZ))
                return;


...
        fast_pool->last = now;
        __mix_pool_bytes(&fast_pool->pool, sizeof(fast_pool->pool));


...


        fast_pool->count = 0;


        /* award one bit for the contents of the fast pool */
        credit_entropy_bits(1);
```

The code snippet shows:

1 Both conditions, the number of interrupts and the expired time must be met.
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.
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 – if the CPU hardware random number generator was present, the entropy estimator is increased by 2.

With these steps, it is evident that

all four u32 words of the fast_pool

are used to update the input_pool.

That means that the 64 interrupts[7] are assumed to represent one bit of entropy.

The conservative estimation of the entropy is warranted considering the relationship of the interrupt noise source and other noise sources. The interrupt handler function implementing the Top-Half interrupt handler of an interrupt executes `add_interrupt_randomness` function. However, if the interrupt was from a HID device or a block device event, the Bottom-Half interrupt handler will invoke the respective other entropy gathering function. That means that the interrupt collection function must be considered to have a correlation with the HID/block device collection function. As any correlation diminishes entropy, a conservative estimation of the implied entropy is warranted and implemented in the Linux-RNG.

7 It is also possible that more interrupts were processed if the interrupts came in at a rate faster than 64 interrupts per HZ time.

During boot time of the kernel, the following code in `add_interrupt_randomness` is important:

```
#define crng_ready() (likely(crng_init > 1))
#define CRNG_INIT_CNT_THRESH (2*CHACHA20_KEY_SIZE)
...
void add_interrupt_randomness(int irq, int irq_flags)
{
...
        if (unlikely(crng_init == 0)) {
                if ((fast_pool->count >= 64) &&
                    crng_fast_load((char *) fast_pool->pool,
                                   sizeof(fast_pool->pool))) {
                        fast_pool->count = 0;
                        fast_pool->last = now;
                }
                return;
        }
...
static int crng_fast_load(const char *cp, size_t len)
...
      if (crng_init_cnt >= CRNG_INIT_CNT_THRESH) {
...
               crng_init = 1;
...
```

This code shows the initialization of the primary ChaCha20 DRNG. The first four sets of 64 interrupts received by a fast_pool will be used to seed the primary ChaCha20 DRNG. The requirement that only four sets of 64 interrupts are used is enforced with the value of crng_init which is set to one after the receipt of 256 interrupts.

Note that in case the ChaCha20 is not fully initialized, the provided data is also mixed into the input_pool but without crediting it any entropy.

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 device,
- 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 in 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.

The 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.

In case the primary ChaCha20 DRNG is not yet considered to be initially seeded, the data is also mixed into the ChaCha20 DRNG state using a small LFSR with a period of 255 to ensure that each part of the ChaCha20 DRNG key maintained in the state buffer is modified.

The data to be mixed into the entropy pool is:

Device driver specific value, and

High-resolution time stamp ⊕ Jiffies.

During boot time, before the ChaCha20 DRNG is considered to be initially seeded, all data that is received by `add_device_randomness` is injected into the ChaCha20 DRNG instead of the input_pool. This shall guarantee that the data stream originating from /dev/urandom during boot benefits from the input data.

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[8], 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.

Figure 7 illustrates the interface and how it links with the input_pool.

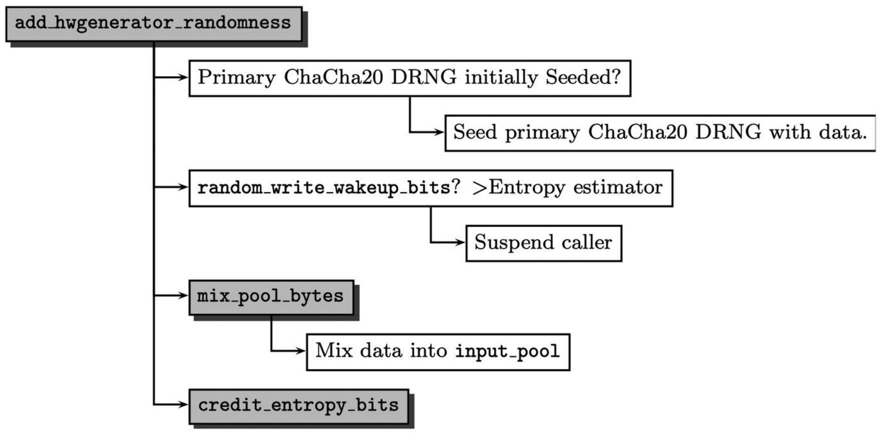
add_hwgenerator_randomness
Primary ChaCha20 DRNG initially Seeded?
Seed primary ChaCha20 DRNG with data.
random_write_wakeup_bits? >Entropy estimator
Suspend caller
mix_pool_bytes
Mix data into input_pool
credit_entropy_bits

Figure 7: Linux-RNG interface for hardware RNG drivers

If the interface detects that the primary ChaCha20 DRNG is not yet initially seeded, the data received via the API call is used to seed the primary ChaCha20 DRNG.

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 hardware-RNG driver framework implements a kernel thread which continuously reads data from the registered hardware RNG devices and invokes the `add_hwgenerator_randomness` interface function with the data obtained from the hardware device. The interface function implements a throttling

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

mechanism. The caller is allowed to feed data into the Linux-RNG if the value in the entropy estimator of the input_pool falls below a threshold set by the variable `random_write_wakeup_bits`.

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 $\frac{1000}{1024}=0.977$ bits of entropy per data bit.
- The HiSilicon TRNG v2 driver implemented with drivers/char/hw_dandom/hisi_trng-v2.c uses a quality of value of 512. This is translated into $\frac{512}{1024}=0.5$ 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 $\frac{900}{1024}=0.879$ 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 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 999 with an entropy rate of $\frac{999}{1024}=0.976$ bits 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.

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 Scheduler-based Entropy Collection

During boot time when the ChaCha20 DRNG is not fully seeded and user space requests random numbers via either the `getrandom` system call or /dev/random, 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.

3.5.2.7 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.

3.5.2.8 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 DRNG with the exception of the ChaCha20 DRNG initial seeding as illustrated in figure 2.

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
num)
{
...
        struct {
                long jiffies;
                unsigned cycles;
                unsigned num;
        } sample;
...
        sample.jiffies = jiffies;
        sample.cycles = random_get_entropy();
        sample.num = num;
        mix_pool_bytes(&sample, sizeof(sample));
...
```

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

- the current jiffies value which is a 64-bit value on a 64-bit CPU and a 32 bit value on a 32-bit CPU,
- 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`.

Please note that the compiler performs a padding of the data structure where the separate variables all occupy 8 bytes on a 64-bit CPU. Therefore, `sizeof(sample)` will be 24 bytes on a 64-bit system.

After filling the data structure with the mentioned values, the input_pool is selected as destination for the data. The final invocation of the function `mix_pool_bytes` mixes the data into the input_pool as illustrated in section 3.3.1.1

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[9]

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 `add_timer_randomness`. Therefore, if data is mixed into the random pools by any source other than the hardware events (like when user space writes data into the device files), the entropy estimator is not updated. The following exceptions to this rule are evident:

- Injection of data from the interrupt noise source: `add_interrupt_randomness` applies onebit of entropy per fast_pool to input_pool transferal as specified in section 3.5.2.2.
- Update of the entropy estimator from the scheduler-based entropy collection specified in section 3.5.2.6.

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

The Linux-RNG estimates the entropy of a hardware event and modifies the entropy estimator of the input_pool accordingly. The ChaCha20 DRNG does not have any entropy estimator as it operates as a DRNG which is seeded once every 5 minutes with the available entropy of up to 256 bits. In the worst case, e.g. if no entropy is present in the Linux-RNG, the ChaCha20 DRNG is not reseeded.

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.

The entropy estimator is an integer value that is stored in the data structure for each entropy pool. The integer value denominates the entropy in 1/8th of a bit. This allows the tracking of fractions of bits. To handle these fractions, the following translation code is used every time the entropy held in the entropy pool in bits shall be obtained:

```
        /* To allow fractional bits to be tracked, the entropy_count
field is
         * denominated in units of 1/8th bits. */
        POOL_ENTROPY_SHIFT = 3,
#define POOL_ENTROPY_BITS() (input_pool.entropy_count >>
POOL_ENTROPY_SHIFT)
        POOL_FRACBITS = POOL_BITS << POOL_ENTROPY_SHIFT,
```

The entropy estimator value `entropy_count` is updated after the values are stirred into the input_pool 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` = $|(t_n - t_{(n-1)})|$
- `delta2` = $|((t_n - t_{(n-1)}) - (t_{(n-1)} - t_{(n-2)}))|$
- `delta3` = $|(((t_n - t_{(n-1)}) - (t_{(n-1)} - t_{(n-2)})) - ((t_{(n-1)} - t_{(n-2)}) - (t_{(n-2)} - t_{(n-3)})))|$

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 highest bit of the value.

3 Mask out the bits higher than bit 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, since the time, for example, of a disk I/O interrupt depends on the status of the disk such as the position of the read/write head and the position of the sector relative to the read/write head (this applies only to spinning hard disks and not to solid state disks). Therefore, the lower bits of the timer value are highly unpredictable even in the case where the same instruction sequences are executed.

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. The increase operation is explained in the following code comment:

```
static void credit_entropy_bits(int nbits)
{
...
                /*
                 * Credit: we have to account for the possibility of
                 * overwriting already present entropy.  Even in the
                 * ideal case of pure Shannon entropy, new contributions
                 * approach the full value asymptotically:
                 *
                 * entropy <- entropy + (pool_size - entropy) *
                 *      (1 - exp(-add_entropy/pool_size))
                 *
                 * For add_entropy <= pool_size/2 then
                 * (1 - exp(-add_entropy/pool_size)) >=
                 *    (add_entropy/pool_size)*0.7869...
                 * so we can approximate the exponential with
                 * 3/4*add_entropy/pool_size and still be on the
                 * safe side by adding at most pool_size/2 at a time.
                 *
                 * The use of pool_size-2 in the while statement is to
                 * prevent rounding artifacts from making the loop
                 * arbitrarily long; this limits the loop to
log2(pool_size)*2
                 * turns no matter how large nbits is.
                 */
...
```

With that description, it is clear that the entropy estimate for a given hardware event is not simply added to the entropy estimator up to the ceiling of the size of the entropy pool. Hence, the estimated value of the pool asymptotically approaches the maximum possible value defined by the size of the entropy pool. The fuller the entropy pool gets with entropy – i.e. the entropy estimator gets larger – the less of the entropy estimate increases the entropy estimator. That is, if the entropy estimator is close to its ceiling (the size of the entropy pool) the entropy value of a hardware event that gets added is only a minuscule fraction of the value that was heuristically attributed to the event.

On the other hand, when extracting data from of the input_pool, the amount of produced bits is simply subtracted from the entropy estimator.

The `credit_entropy_bits` function also triggers the wakeup of read-like polling processes if the entropy estimator rises above a set threshold.

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 8 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.

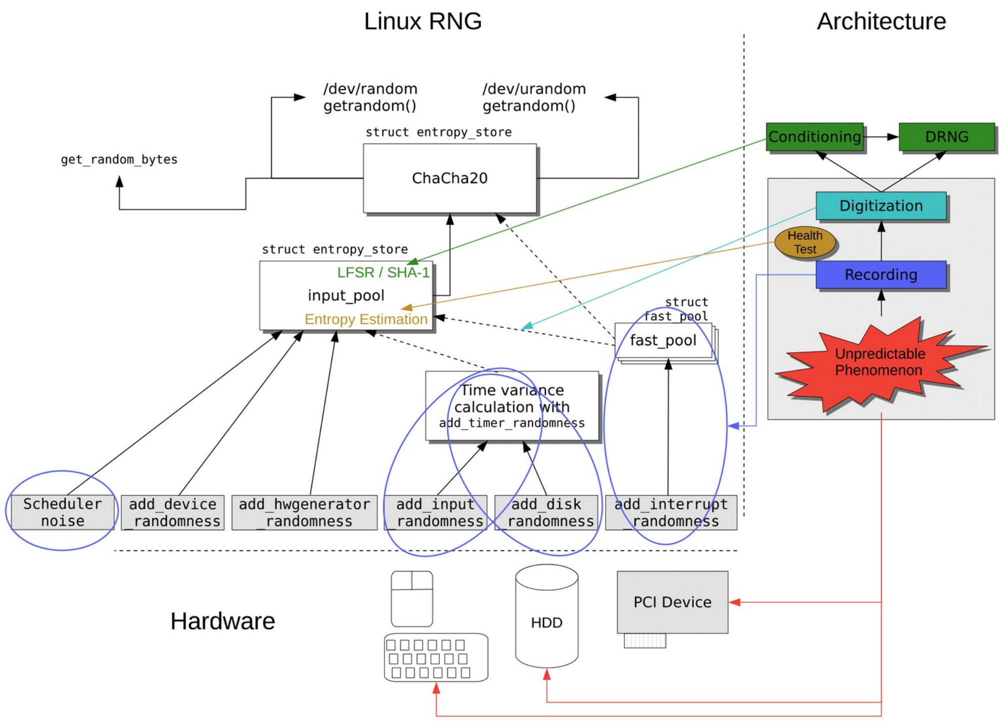

Figure 8: Linux-RNG architecture compared with the generic architecture

With figure 8, 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 8. 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 8 contains the architecture illustration from figure 1. As the discussion is about NDRNG, figure 8 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 8 shows the theoretical noise source concept from figure 1. Figure 8 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 input_pool does not provide any data to the ChaCha20 DRNG. Therefore, if an event is received where one of these derivatives is zero and hence indicates a pattern that should be considered to have very little or no entropy, the event data is mixed into the input_pool without allowing the input_pool being read by that amount of data. In effect, this implies that the mixed in event data is treated as poor data where the noise source failed to deliver entropy.
- The function inserting data into the input_pool using an LFSR can be considered a conditioning logic. An additional conditioning logic is evident with:
  - the fast_pool maintenance, and
  - loading of data from the `add_device_randomness` function before the ChaCha20 DRNG is fully seeded.
- 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 must be read from /dev/urandom and stored in non-volatile storage. 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
```

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 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[10].

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.

The common use case of the hardware RNG framework is to install a user space program, such as the rngd, which:

1 Opens /dev/random and waits on this device with poll or select until insufficient entropy is present in the input_pool.

2 When the Linux-RNG identifies that the entropy is running low, the rngd process is woken up as described above.

3 The rngd process now reads some data from /dev/hwrng and injects that data using the IOCTL `RNDADDENTROPY` into the input_pool. This injection increases the entropy estimator by the value provided by rngd.

To avoid such a detour through user space, 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.

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

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 9 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:

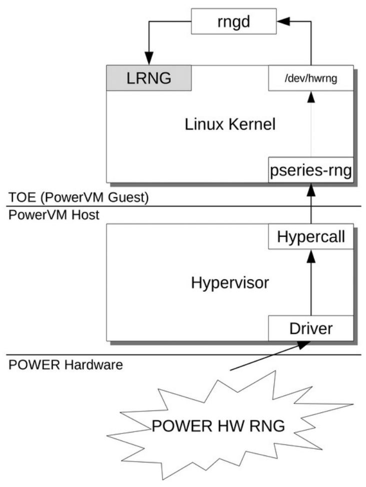

Figure 9: Flow of random numbers in a PowerVM environment

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 pulls the data from /dev/hwrng.

5 The rngd injects the data into /dev/random on the Linux guest operating system using the RNDADDENTROPY IOCTL to mix it into the input_pool.

Figure 10 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.

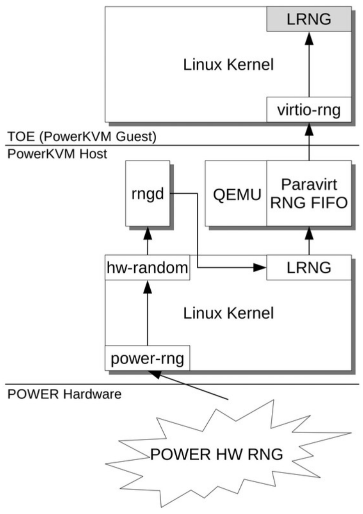

Figure 10: Flow of random numbers in a PowerKVM environment

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 pulls the data from /dev/hwrng.

3 The rngd injects the data into /dev/random on the Linux host operating system using the `RNDADDENTROPY` IOCTL to mix it into the input_pool.

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[11].

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.

11 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 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 Either in the very late stages of the kernel boot or during the first steps of the initramfs user space boot operation, the first four fast_pools are injected into the ChaCha20 DRNG to bring it into an initially seeded state.

3 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.

4 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.

5 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 128 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 128 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.

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. Latter include 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

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 of the attacks described in [GPR06] are no longer possible with newer kernel versions.

It should be noted that some attacks require access to system resources which allow full control over the system itself. If an attacker can read the state of the ChaCha20 DRNG hosted in the Linux kernel, he has much more dangerous power than subverting the random number generator. In this case, the attacker can subvert every operation of software on the system by reading, changing, and replaying any data. Thus, an attacker can reach his ultimate goal of controlling user data much easier than using the detour of controlling a random number generator.

4.1 Attacks of Gutterman et al. and its Relevance

The following section gives a brief overview of the different attacks which have been discussed by Gutterman et al. Potentially applied countermeasures already mentioned by [LRSV12] are outlined as well.

Please note that this section discusses the mentioned document that covers an older version of the Linux-RNG. Therefore, references to older concepts such as the nonblocking_pool are still found in the discussion.

4.1.1 Denial of Service Attacks

Two different denial of service attacks are presented in [GPR06]. The first consists of a continuous request of random numbers from the blocking_pool. It is suggested that a set of quotas may be used, which is considered impractical.

A continuous reading from /dev/urandom, i.e. the nonblocking_pool that was part of the discussed kernel version could reduce the entropy present in the input_pool faster than it was replenished by the noise sources. This also implies an implicit denial of service attack to the blocking_pool and thus /dev/random. In kernels starting with version 4.8 this issue has been completely removed by using the ChaCha20 DRNG instead of the nonblocking_pool. This DRNG performs a reseed after 5 minutes irrespective of the amount of random numbers that were generated.

4.1.2 Use of Diskless Systems

As discussed in section 3.8, the Linux-RNG is commonly used such that during shutdown, a seed is generated from /dev/urandom which is stored on disk. This seed is written back into /dev/random during system boot to stir the entropy pools. Such an approach should cover scenarios where insufficient entropy is available during boot time.

The writing or reading of the seed data is not possible on diskless systems like Live-CDs or router-style use cases like OpenWRT. The author of the Linux-RNG Ted Ts'o responded to the case that he does not consider this issue as an error of the Linux-RNG but rather a usage error.

In the current implementations of the Linux-RNG, this issue is mitigated by injecting four sets of 64 interrupts into the ChaCha20 DRNG to ensure that it is seeded as soon as possible. Furthermore, the introduction of the `getrandom` system call with its blocking nature also counters the problems of diskless systems.

4.1.3 Enhanced Backward Secrecy

In the considered scenario the attacker knows the content of the entropy pool and wants to deduce the previous state. In the kernel version analyzed in [GPR06] a sub-optimal use of the feedback function implied that the enhanced backward secrecy property was not fully present. Changes to the Linux-RNG feedback implementation documented in [LRSV12] prevented the attacks outlined by Gutterman et al.

4.2 Lacharme's Analysis

Further conclusions from [LRSV12] are presented in the following sub-sections.

Note, this assessment applies to an older version of the Linux-RNG which had several entropy pools. As the statement still applies to the one existing entropy pool today, Lacharme's analysis is referenced here still as is.

Please note that this section discusses the mentioned document that covers an older version of the Linux-RNG. Therefore, references to older concepts such as the SHA-1 input_pool extraction hash are still found in the discussion.

4.2.1 Linux-RNG Without Input to the Entropy Pools

In case no entropy is collected from the noise sources, the input_pool and the blocking_pool will stop producing random numbers. The nonblocking_pool that served /dev/urandom for the discussed kernel versions started to operate as a deterministic random number generator using an LFSR as state transition function and SHA-1 as output function.

The LFSR state transition function implies that the internal state will become cyclic eventually. At the time of writing of [LRSV12], non-irreducible polynomials were used for the LFSR which implied that the period length was less than possible. This issue is fixed for the current kernels, as the LFSRs now use irreducible, primitive polynomials. In addition, the nonblocking_pool is replaced by a ChaCha20 DRNG that uses the ChaCha20 block function for its state transition which is not affected by polynomials.

4.2.2 Attacks on the Input

An important conclusion which is mathematically proven by Lacharme et al. is that the mix-in of new data into the entropy pools will never lead to a reduction of the entropy already contained in the entropy pools. Thus, input-based attacks are not possible which means that an attacker not knowing the state of an entropy pool cannot adversely affect the entropy of the entropy pools.

4.2.3 Assessment of the Entropy Estimation

The Linux-RNG heuristic to estimate the entropy of events obtained by the noise sources must guarantee that at least as much "real" entropy is present as estimated. Measurements which support this requirement are provided in [LRSV12].

The measurements given in chapter 6 also support this conclusion and even determine that the Linux-RNG significantly underestimates the gathered entropy.

4.3 Conclusions from [LRSV12] and [GPR06]

Both studies give hints for further development of the Linux-RNG. These hints mostly refer to direct countermeasures of the discussed weaknesses. In addition, Gutterman et al. challenged the unnecessary complexity of the Linux-RNG design including the large pool sizes and the invocation of SHA-1 that happens too often. Instead a Barak-Halevi-construction is suggested.

As already outlined, the concerns raised in [GPR06] have been addressed in newer kernel versions.

Lacharme et al. confirm that the Linux-RNG reaches an appropriate security level. The authors also point out that replacing the used SHA-1 hash with more modern hash functions like SHA-3 would require a complete redesign.

Lacharme et al. complain that no theoretical basis for the entropy estimation is present. Such theoretical basis is presented by Benjamin Pousse with [P12]. Pousse uses the so-called Kolmogorov-complexity which is defined by the shortest possible description of a message. In other words, the Kolmogorov-complexity of a bit string is the bit length of an optimal compression of the considered bit string.

4.4 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[12]. This implementation is furnished with a complete design description [LRNGREPLACEMENT], 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.

With the LRNG design description, section 1.1, the issue of double accounting of entropy is outlined. In short, one entropy event from a human device interface or block device is credited entropy twice, because each event is detected by

1. `add_interrupt_randomness` since the event triggers an interrupt, and
2. `add_disk_randomness` or `add_input_randomness` since the event is a block device event / human interface device event, respectively.

All block device and human interface device events are a “derivative” of an interrupt. In both detection functions, the event is credited with entropy. As seen in the subsequent section, the entropy credited to interrupt events is minimal compared to the actual available entropy. Thus, the Linux-RNG is not believed to overestimate entropy due to the double-accounting. Nonetheless, this issue is considered an architecture deficiency. As a side note, this very issue is the core problem that prevents the Linux-RNG from becoming SP800-90B compliant even after receiving the required SP800-90B noise source health tests.

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[13] 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

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

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 the massive underestimation of entropy present in interrupts which implied that the several injections of the fast_pool into the ChaCha20 DRNG during boot time ensures that the ChaCha20 DRNG has sufficient initial seed. Subsequent reseeding operations of the ChaCha20 DRNG, however, are affected by this consideration. Thus, when applying an information theoretical entropy assessment, the outlined issue leads to an attacker-controllable insufficient entropy level when using the Linux-RNG interfaces without further countermeasures.

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 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 5 minutes (on systems without NUMA support) or for up to 10 minutes (on systems with NUMA due to the use primary and secondary ChaCha20 DRNGs). This issue, however, can be alleviated when user space actively calls the IOCTL `RNDRESEEDCRNG` which requires root-privileges though.

When performing an entropy analysis, it is always important to assess the post-processing operation and how it impacts the entropy of the processed data. The more post-processing is applied, the more complex the assessment of the resulting entropy rate will get. The Linux-RNG performs post-processing in several places which all individually would need to be assessed. It is partially unclear why the different post-processing steps are performed at the following different stages compared to just use the input_pool and ChaCha20 DRNG post-processing as a clean design approach:

- The fast_pool operation is a post-processing operation. It behaves like an LFSR operating on the fast_pool. It is unclear how much impact the operation has on the collected entropy. Based on the massive underestimation of entropy provided with interrupts, the entropy behavior of the fast_pool post-processing operation is considered to not affect the final conclusion that the Linux-RNG entropy estimation does not overestimate the existing entropy. The reason for having the fast_pool is the fact that the interrupt event code of the Linux-RNG executes as part of the interrupt handler which is a highly performance-critical code path. Thus, injecting the data into the input_pool directly would be too costly. Yet, instead of using such fast_pool with its LFSR, a simple concatenation of the input data could be implemented, which would even be faster and has a very clear and fully understood impact on the entropy rate. An example of such approach is given in with the LRNG, function `lrng_time_process`.
- During boot time, data received by add_device_randomness is processed with some form of an LFSR before injecting the data into the ChaCha20 DRNG. It is not clear why such LFSR is used instead of injecting the data directly into the ChaCha20 DRNG and using the ChaCha20 DRNG backtracking operation to mix the state in case of seed data that is longer than the key size of the ChaCha20 cipher. Thus, with a different use of the ChaCha20 DRNG, the LFSR operation could have been avoided. An example on how the ChaCha20 DRNG could process seed data with arbitrary length is provided by the LRNG, specifically the function `lrng_cc20_drng_seed_helper`.
- When requesting the ChaCha20 DRNG to generate random numbers, a 64 bit data from RDRAND is XORed with the second nonce value of the ChaCha20 DRNG state. This is effectively some form of unconditional reseeding of the ChaCha20 DRNG in addition to the regular reseeding operation.

The listed post-processing is in addition to the aforementioned post-processing of data implemented with the input_pool and ChaCha20 DRNG management.

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 LFSR operation providing the state transition 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 reading of the entropy pool using Blake2s, i.e. the output function for the input_pool, is yet another deterministic operation which should be subject to self-test. In particular, the operation to generate data streams that are longer than one (half) Blake2s block is of key interest. Yet, the output function is tightly integrated with other code which would make an addition of a self-test more complex.
- Finally, the management of the fast_pool is another deterministic operation which is vital to the entropy conservation. It also could 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. As shown in the quantitative entropy assessment in chapter 6, almost all entropy is derived from the high-resolution time stamp. Hardly any entropy is provided with the Jiffies timer data. 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. Based on that finding, the asymptotic reduction of the entropy value during increasing of the entropy estimator explained in section 3.6 seems to be difficult to understand. 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.

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 applying the constraints outlined in section 5.3.1. When a suitable replacement for the seed source discussed in DRG.3.1 is found, this conclusion can be applied to other hardware architectures as DRG.3 defines procedural requirements only.

Although NTG.1 cannot be readily claimed for the Linux-RNG starting with kernel version 5.6, the approach oulined in section 5.1 comes very close to NTG.1.

5.1 Approach to NTG.1

This section provides a suggestion how the NTG.1 properties can be (almost) achieved. It outlines the code required to be applied in user space by a caller and outlines the considerations where the approach is not fully compliant to NTG.1.

Section 5.3 shows that the Linux-RNG is a DRG.3. The question is now, what is the conceptual difference between a DRG.3 and an NTG.1? The difference is the following: contrary to a DRG.3, an NTG.1 mandates that when it generates a given amount of random bits, it must be seeded with an equal amount of entropy beforehand. For example, if a caller requests 10 bytes from a random number generator, the RNG must contain at least 10 bytes of information-theoretical entropy. This can be achieved by mandating a reseed that delivers that amount of entropy from its noise sources before the requested 10 random bytes are provided to the caller.

This concept can be achieved with the Linux-RNG from user space using the following steps. By using these steps, user space is able to implement an NTG.1 based on the Linux-RNG. Unfortunately there is one issue discussed below that prevents this approach from being fully NTG.1 compliant.

User space would need to implement these steps to create an (almost) NTG.1 compliant approach:

1. User space must ensure that the maximum amount of random numbers requested at once does not exceed 32 bytes. The reason is that the ChaCha20 DRNG tries to reseed itself with 32 bytes from the input_pool.
2. User space check the amount of available entropy in the Linux-RNG by reading the file /proc/sys/kernel/random/entropy_avail. This value provides the available entropy in bits. To ensure a safety margin, user space should ensure that at least twice as much entropy is available in the Linux-RNG as data is requested.
3. User space requires a reseeding of the Linux-RNG by issuing the `RNDRESEEDCRNG` IOCTL.
4. User space reads the requested amount of random numbers via the `getrandom` system call.

If another random number with NTG.1 properties should be generated, all mentioned steps need to be invoked again.

An example implementation is provided with the function `generate_ntg1` provided with the LRNG test code.

As mentioned, however, there is one issue that prevents this approach from being fully NTG.1 compliant: a race condition. Although the steps 2 through 4 are executed within a tiny amount of time – roughly 50µs to 100µs – these steps are not atomic in nature, meaning that code executing on either another CPU or code that is scheduled by the Linux operating system in that time may have the ability to obtain the "first" random bits from the `getrandom` system call after the reseed operation was triggered by the NTG.1 user space code. This time window may be enlarged by an attacker when attacking the Linux scheduling policy. Although it is believed that only a small amount of random numbers can be generated by an attacker in that race condition time window, the pure fact that this race condition exists renders the approach to be not exactly NTG.1 compliant.

5.2 input_pool: NTG.1

As discussed in section 3.7, the input_pool can be regarded as a separate random number generator. Although this random number generator is not directly accessible by any caller, the assessment of its properties supports the analysis of the ChaCha20 DRNG.

This section analyzes how the input_pool is NTG.1 compliant.

5.2.1 NTG.1.1

The requirement of NTG.1.1 is defined as:

"The RNG shall test the external input data provided by a non-physical entropy source in order to estimate the entropy and to detect non-tolerable statistical defects under the condition [assignment: requirements for NPTRNG operation]."

NPTRNG refers to "non-physical true random number generator".

This requirement is implemented with the functionality maintained for the input_pool as follows. The following statements have to be added to the assignment operation of the NTG.1.1 requirement:

- For each noise source, the entropy collection functions implement checks as to whether the event value is "appropriate". Using these mechanisms, statistically significant skews are prevented. Specifically the following checks are implemented:
  - HID: The function `add_input_randomness` discards event values that are identical to the preceding event value.
  - Block devices: The function `add_disk_randomness` discards an event if either no data structure is allocated for a particular block device to maintain noise source specific information or if the block device is considered inappropriate as a source for entropy.
  - Interrupts: The interrupt noise source is stimulated by interrupts received from devices. If the interrupt data is not correct, the interrupt cannot be processed by the Linux kernel and the device will be inoperable and thus cease to function. If multiple devices were to suffer from such conditions, it is likely that either the kernel will crash or the entire system will cease to function.
- HID / block device noise source: The Linux-RNG function of `add_timer_randomness` calculates an entropy estimate for each received HID or block device event as discussed in section 3.6. Using the minimum of the first, second and third discrete derivative of the time stamp, statistical defects in the monotonically increasing counter are detected. If such defects are detected, the event is credited with zero bits of entropy.
- Interrupts: The Linux-RNG awards a set of 64 or more interrupts exactly one bit of entropy as discussed in section 3.5.2.2. Statistical defects are caught by the aforementioned requirement that interrupts must operate benignly, i.e. in a way that supports the operating system execution, as otherwise the system may cease to function. Furthermore, due to the massive underestimation of entropy in the interrupts, potential statistical abnormalities or the absence of any entropy delivered by RDSEED are countered.

- The read function of the input_pool takes the entropy estimation into consideration and only delivers as many bits as entropy is available. For each read bit, the entropy estimate is reduced by one bit.

These mechanisms implemented by the Linux-RNG imply that the processing of the data from the noise sources complies with the requirements of NTG.1.1.

5.2.2 NTG.1.2

The requirement of NTG.1.2 is defined as:

“The internal state of the RNG shall have at least [assignment: Min-entropy]. The RNG shall prevent any output of random numbers until the conditions for seeding are fulfilled.”

Based on the design and the statistical analysis, the assignment for the Min-entropy can be specified as follows: “... as many bits of theoretical min-entropy as requested by the caller ...”. The reason for this assignment is the blocking behavior enforced by the output function of the input_pool generating random numbers for the seeding of the ChaCha20 DRNG. For every bit of random data generated from the input_pool, raw noise that is awarded an heuristic entropy value of at least one bit must have been collected by the Linux-RNG.

The question that must be answered for a conclusion is whether the heuristically determined entropy content estimated by the Linux-RNG is not larger than the real entropy content of the raw noise. Section 6.2 provides this comparison which can be summarized as follows:

- Interrupts: The heuristic entropy applied by the Linux-RNG is one bit of entropy per 64 interrupts, i.e. 1/64th bit of entropy per interrupt[14]. The analysis of the Linux-RNG has shown that the entropy content of each high-resolution time stamp provides more than 4.6 bits (SP800-90B min-entropy estimation), 7.99 bits (Shannon entropy plug-in estimate), or 7.67 bits (Min-entropy plug-in estimate). Irrespective of the used entropy value, these values are significantly higher than the heuristically applied entropy.
- Block devices: The entropy heuristic applied by the Linux-RNG awards on average 0.11 bits of entropy per block device event. The measurement of the raw noise shows the following entropy content per event: more than 6.3 bits (SP800-90B min-entropy estimation), 7.54 bits (Shannon entropy plug-in estimate), or 6.65 bits (Min-entropy plug-in estimate). Again, the heuristic entropy value applied by the Linux-RNG for each event is significantly less than the entropy estimate from the measurements.
- HID: The Linux-RNG entropy heuristic awards on average 0.49 bits of entropy per HID event. The raw noise measurement shows the following entropy content per HID event: more than 6.6 bits (SP800-90B min-entropy estimation), 7.8 bits (Shannon entropy plug-in estimate), or 6 bits (Min-entropy plug-in estimate). Just as for the block device and interrupt noise sources, the Linux-RNG heuristic therefore underestimates the available entropy.
- Scheduler-based noise source: The measurement of the scheduler-based noise source shows that the Linux kernel would not have awarded any heuristic entropy to the collected data. Yet, in a worst case, the scheduler-based noise source may award one time stamp one bit of entropy. The measured data obtained during the boot cycle in virtual environments presented in section 6.3.2.2 shows more than 1.5 bits of entropy (SP800-90B min-entropy estimation), 3.6 bits (Shannon entropy plug-in estimate), or 1.8 bits (Min-entropy plug-in estimate). The entropy estimate measured during the boot cycle on bare-metal discussed in section 6.3.4.2 shows between 0.007 bits and 2.2 bit of entropy for the SP800-90B min-entropy estimation, between 0.6 bits and 2.89 bits of entropy for the Min-Entropy and between 1.4 bits and 3.7 bits for the Shannon entropy. Compared to the heuristic entropy, the estimated entropy still exceeds the awarded entropy.

14 This is the worst case value. It may be the case that more interrupts are collected that are collectively awarded one bit of entropy.

As all heuristic entropy values applied by the Linux-RNG to the entropy estimator steering the blocking behavior of the generation of random numbers from the input_pool are less than the available entropy in the raw noise, the entropy estimator value underestimates the available entropy. As the blocking behavior rests on the entropy estimator, it is guaranteed that each bit produced from the input_pool is backed by at least one bit of fresh entropy obtained from the noise sources.

Therefore, the implementation of the input_pool complies with the requirements set forth by NTG.1.2.

5.2.3 NTG.1.3

The requirement of NTG.1.3 is defined as:

“The RNG provides backward secrecy even if the current internal state and the previously used data for reseeding, resp. for seed-update, are known.”

For performing the analysis, it is assumed that an attacker has the information specified by NTG.1.3:

- the content of the entropy pool s_t at the time index t, as well as
- all data used for the seeding and reseeding for the time index 0, 1, ..., t.

As defined in NTG.1, the attacker wants to obtain the previous random number $out_{(t-1)}$.

- Figure 11 depicts the situation and the relationship of input and output data. Please note that the figure provides a simplified expression of the Linux-RNG processing as the parallel execution and processing of data is not covered. In the parallel processing, several phases may overlap. Yet, the conclusion that can be drawn from figure 11 is identical for a single threaded or parallel processing.

Figure 11 uses the following symbols:

- s_t refers to the entropy pool content at the time index t
- e_t refers to new entropy from the noise sources at time index t
- h_t refers to the Blake2s hash of the entropy pool at the time index t
- out_t refers to the output data provided by the Linux-RNG to the caller at the time index t

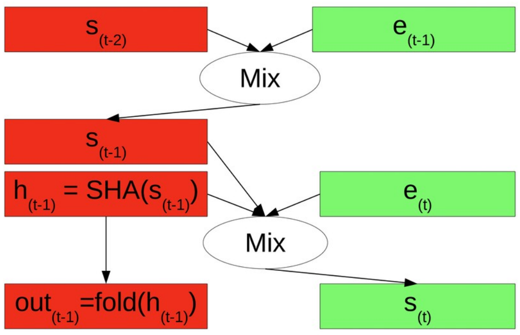

Figure 11: Relationship between Linux-RNG processing and attacker-known and unknown data

Figure 11 indicates attacker-known data with a green background color whereas the attacker-unknown data is indicated with a red color.

The attacker wants to determine the value of $out_{(t-1)}$. In order to determine this value, $h_{(t-1)}$ and $s_{(t-1)}$ must be constructed from $e_{(t)}$ and $s_{(t)}$.

It is assumed that the attacker is capable of reversing the LFSR to the state before $e_{(t)}$ is mixed into the entropy pool considering that in the worst case no additional entropy has been mixed into the entropy pool. Yet, this state contains the mixed-in Blake2s output $h_{(t-1)}$ as well as $s_{(t-1)}$. An attacker is unable to extract the $h_{(t-1)}$ after it has been mixed in without knowing $s_{(t-1)}$ due to the properties of Blake2s and the LSFR operation:

- Both functions are non-invertible which implies that the output cannot be used to deduce its internal state.
- For both functions it is hard to obtain a pre-image for a given output. For Blake2s the effort is related to the pre-image resistance which is considered extremely high. For the LFSR, the determination of a pre-image of the entropy pool with a size of 4096 bits is considered, on the outside, to take eons with current computer technology.

With this rationale it can be concluded that the random numbers generated from the input_pool comply with the requirements of NTG.1.3.

The reader should consider the whole picture when assessing the threat to be countered for NTG.1.3: the entropy pool is maintained in the Linux kernel. If an attacker is capable of eavesdropping on the Linux kernel to observe the entropy pool and the mix function, all security barriers of the operating systems have already been breached, it becomes useless to repel subsequent attacks against the Linux-RNG:

- Every read invocation to /dev/random or /dev/urandom or the extraction of random numbers from the input_pool can be monitored to catch every newly generated random number. In addition, the read invocations can be modified to return attacker-crafted “random numbers”.
- The attacker can read all memory regions of the system. This includes the state of DRNGs seeded by /dev/random or /dev/urandom as well as the input_pool state.
- An attacker can monitor, modify or replay all cryptographic operations of the operating systems. The modification applies to the key material as well which is derived from random number generators like /dev/random.

Thus, the discussion around NTG.1.3 must be considered academic in nature with little practical relevance. In practice an attacker will make use of acquired privileges more effectively than monitoring the Linux-RNG and deducing previous random numbers. This discussion is only relevant for cases where the attacker obtains the internal state and tries to deduce previously generated random numbers.

5.2.4 NTG.1.4

The requirement of NTG.1.4 is defined as:

“The RNG generates output for which [assignment: number of strings] strings of bit length 128 are mutually different with probability [assignment: probability].”

The input_pool entropy pool has a size of 4096 bits. In a worst-case scenario when the input_pool has just been fully seeded with fresh entropy it can generate 4096 bits of entropy before it reseeds with newly fresh entropy.

To generate a bit string of 128 bits, the read operation of the input_pool performs one Blake2s 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:

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

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 - ε, with ε = 3.8e-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):

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

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 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 input_pool, it can be concluded that the behavior of the input_pool comes close to the ideal case:

- The input_pool has a large size of 4096 bits. The pre-image for each Blake2s value is therefore extremely large, which implies that the assumption of an equidistribution must be considered to be appropriate. In addition, a big part of the 4096 bit entropy pool is changed by each random number generation.
- Random numbers are only generated when sufficient entropy is present. If insufficient entropy is present, fresh entropy is obtained from the noise sources which again will change the entropy pool content significantly.[15] This will support the assumption of an equidistribution. It should be noted that a reseeding is enforced at the latest after the generation of 6 blocks of 128 bits each.
- Blake2s was subject to significant assessments considering it became RFC7693 that have shown that generated Blake2s values are a close approximation of an equidistribution supported by the avalanche effect.
- The folding operation using XOR is considered to not change the assumed equidistribution.

This allows the conclusion that the output of the input_pool fulfills the requirements of NTG.1.4.

15 The absolute minimum entropy that is required to be present before data is extracted from the input_pool is 1 byte.

5.2.5 NTG.1.5

The requirement of NTG.1.5 is defined as:

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

The execution of the Test Procedure A defined in [AIS2031] on the output of the input_pool is documented in section 7.3.

The requirement for additional statistical tests is covered chapter 6 can be considered to support the stated requirement.

Considering section 7.2.1.1, the assignment of the requirement can be specified as: "... as well as the dieharder test suite[16], the Chi-Squared test and the test of compressing the generated data with gzip, bzip2, xz and lzma".

5.2.6 NTG.1.6

The requirement of NTG.1.6 is defined as:

"The average Shannon entropy per internal random bit exceeds 0.997."

As outlined in section 5.2.2 for NTG.1.2, the testing performed in section 6.2 allows a comparison of the heuristic entropy estimation awarded by the Linux-RNG to each noise source event with the entropy present in this value. The analysis shows that the heuristic entropy estimation is significantly lower than the entropy obtained from the noise sources. This implies that the Linux-RNG is very conservative in its entropy estimation.

The heuristic entropy estimation is maintained for the input_pool that drives the blocking behavior of its output function. This mechanism allows only to draw as many bits from the input_pool as there is entropy present in it. The entropy estimation is increased by the heuristic entropy estimation awarded to the data mixed into the input_pool and is decreased by the number of bits obtained via the input_pool output function. With this concept, the Linux-RNG implies that each data bit generated from the input_pool is backed by one bit of entropy obtained from the noise sources.

The blocking behavior of the input_pool output function implies that only as much entropy can be extracted from the input_pool as entropy went into it. The Linux-RNG uses its heuristic entropy estimation to control this behavior.

As the heuristic entropy estimation is significantly lower than the measured entropy, the following statement holds true: if it can be demonstrated that the measured entropy ensures that the requirement of NTG.1.6 is met, then the heuristic entropy estimation applied by the Linux-RNG to the blocking behavior is sufficient to guarantee the compliance with NTG.1.6 as well.

The following rationale shows for each noise source independently why the measured entropy is sufficient to meet NTG.1.6[17]:

- Interrupts: A set of at least 64 interrupts is awarded one bit of entropy by the heuristic of the Linux-RNG. This means that after obtaining 64 interrupts and mixing it into the LFSR, one bit is allowed to be read from the entropy pool via Blake2s before the blocking behavior stops additional read operations. The estimated entropy for one interrupt using the Shannon entropy plug-in estimate is given in section 6.2.1

16 Test suite is provided at http://www.phy.duke.edu/~rgb/General/dieharder.php.
17 The discussion refers to bit-wise reading from the entropy pool. The Linux-RNG implementation requires a minimum size of one byte. To illustrate the relationship between the measured and heuristic entropy values and the blocking behavior, the discussion assumes a bit-wise read operation is possible. Therefore, this is considered to be a worst-case analysis.

with 7.99 bits. That means, with the collection of 64 interrupts, the Linux-RNG has collected 64 * 7.99 = 511.36 bits of entropy. The use of the fast_pool with its 4 32 bit words can only hold 128 bits of entropy. Assuming that the fast_pool mix operation does not destroy entropy, 64 or more interrupts injected into an entropy pool will have 128 bits of (Shannon) entropy. That means that for each bit of generated random data, 128 bits of Shannon entropy have been obtained from the interrupt noise source.

- Block devices: As shown in section 6.2.2, each block device event is awarded on average 0.11 bits of entropy by the Linux-RNG heuristic. That means that after around 1/0.11 = 9.1 block device events that have been received on average, the input_pool allows the generation of one random bit. The Shannon entropy value estimated for block device events is 19.9 bits of entropy per event as listed in section 6.2.2. Thus for 9.1 block device events, the Linux-RNG has collected 9.1 * 8= 72.8 bits of entropy. This implies that for each generated random bit, the Linux-RNG has obtained 74.8 bits of entropy from the block device noise sources.
- HID: Considering section 6.2.3, the Linux-RNG heuristic awards on average each HID event 0.49 bits of entropy. This means that after 1 / 0.49 ≈ 2 HID events one bit of random data can be produced before the blocking behavior is enforced. Using the estimated Shannon entropy value of 7.958 bits per HID event, it implies that for generating one bit of random data, 7.958 * 2 = 15.9 bits bits of Shannon entropy plug-in estimate are obtained by the Linux-RNG from the HID noise source.
- Scheduler-based noise source: During the measurement, the kernel did not award any entropy to the collected data. But even applying the worst case where the Linux-RNG awards 1 bit of entropy to each time stamp, the estimated data of 7.7 bits of Shannon entropy plug-in estimate per time stamp shows that sufficient entropy is available.

Due to the independence of the noise sources, the Shannon Entropy values can be added when obtaining entropy from all noise sources in parallel.

It can be concluded that for each generated random bit, much more than one bit of Shannon entropy is collected from the noise sources by the Linux-RNG. In a worst case, when the Linux-RNG fills the entire entropy pool with size of 4096 bits with data from the noise source, the amount of Shannon entropy from the individual events that are collected by the Linux-RNG is much more than 4096 bits, because the Linux-RNG applies its conservative heuristic entropy estimation. The maximum amount of entropy that the entropy pool can maintain is equal to its size, i.e. 4096 bits. This means that once the Linux-RNG considers its entropy completely filled, 4096 bits of Shannon entropy are present in that pool.

The input_pool output function allows the generation of 4096 bits of random data from that completely filled entropy pool before the blocking behavior is enforced. Assuming that the LFSR state transition function and the Blake2s-based output function do not destroy entropy, this means that in this worst case scenario one bit of generated random data is backed by one bit of Shannon entropy.

This allows the conclusion that the input_pool meets the requirement of NTG.1.6.

5.2.7 NTG.1 Properties on Different Environments

The purpose of this study is to demonstrate that the Linux-RNG with its input_pool complies with the properties of NTG.1. The testing was executed on a specific hardware type outlined in the Appendix. The conclusions of the NTG.1 assessment apply to other environments considering the following restrictions:

- The Linux system executes on an Intel / AMD x86 CPU.
- The CPU implements the RDTSC instruction.
- The maximum CPU clock frequency is at least 1GHz.
- The Linux system either executes directly on the hardware without a virtual machine monitor or as a guest within one of the the virtual machine monitors assessed in [LRNGVIRT].

- The source code for the Linux-RNG is unchanged compared to the source code discussed in this document.
- The Linux kernel is fully trusted and does not execute any code unknown to the vendor. This implies that the state of the kernel and therefore the state of the Linux-RNG is fully protected.

Any deviations from the mentioned requirements implies that the NTG.1 assessment given in the preceding sections is not applicable and a separate NTG.1 analysis is required.

5.3 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.

5.3.1 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 option must be set: `random.trust_cpu=0` or the kernel compile-time option of `CONFIG_RANDOM_TRUST_CPU` must be disabled.

The (primary) ChaCha20 DRNG is seeded by reading from the input_pool. As outlined in section 5.2, the input_pool complies with the requirements from NTG.1 and is therefore considered an NTG.1 random number generator.

In case the kernel is compiled with NUMA support and only one NUMA node is present – this scenario includes the execution of the kernel with NUMA support on non-NUMA systems – the primary 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 NUMA-node is present in the system, the primary ChaCha20 DRNG reseeds multiple secondary ChaCha20 instances – one per NUMA-node. The output from the multiple secondary ChaCha20 DRNGs is not considered to be DRG.3 compliant. As the primary ChaCha20 DRNG is not accessible by any caller, the conclusion is that the Linux-RNG with NUMA support executing on a system with more than one NUMA-node is not DRG.3 compliant.

This implies that the first selection can be instantiated with the selection “using an NPTRNG of class NTG.1” for the primary ChaCha20 DRNG.

The second selection implies that the DRNG has been seeded before being used by the caller. This is only enforced for the getrandom system call as well as /dev/random which blocks until a set of 64 interrupts have been received four times and 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.

Considering that the interrupt data is mixed into the 256 bit key part of the ChaCha20 state, the maximum entropy that the used memory location can hold is equal to its size, i.e. 256 bits. This implies that after the receipt of 256 interrupts that are mixed into the ChaCha20 key part, the state of the ChaCha20 DRNG contains 256 bits. Thus, the second selection of DRG.3.1 can be instantiated with: "have 256 bits of entropy".

The in-kernel API call of `add_random_ready_callback` allows registering callback functions by kernel subsystems. When using this API, the registered kernel subsystem's callback function is invoked after either the ChaCha20 DRNG is initially seeded with the aforementioned 64 interrupts four times and after the input_pool has received 128 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.

This allows the conclusion that the DRG.3.1 requirements are met for the `getrandom` system call and the /dev/random device file. In addition, when using the in-kernel API `get_random_bytes` after the callback function registered with `add_random_ready_callback` has been triggered implies that the requirements for DRG.3.1 are met as well.

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 `add_random_ready_callback` 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 `add_random_ready_callback` 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.

5.3.2 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.

5.3.3 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.”

After the generation of a random number, the ChaCha20 DRNG updates its internal state using random numbers that it has generated but that are not provided to any caller and that are discarded immediately afterwards. 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.

5.3.4 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].”

Considering that the ChaCha20 block operation has similar characteristics to Blake2s that are outlined in section 5.2.4 for NTG.1.4 compliance, the presented calculation equally applies to ChaCha20. The following characteristics of ChaCha20 are important:

- The output of the ChaCha20 block operation follows an equidistribution.
- The ChaCha20 DRNG is based on an internal state of 512 bits which can maintain a seed value of up to 256 bits. Larger seed values or more seed is XORed such that it fits into the 256 bits. Although the state is significantly smaller as the input_pool, the output function is conceptually identical: using a cryptographic function a new random value is derived from the internal state where the output follows an equidistribution.

The major difference to section 5.2.4 is that the input_pool is frequently reseeded with fresh entropy. The ChaCha20 DRNG is reseeded after five minutes, which allows the generation of large amounts of random data backed by little entropy only in a worst case. Between the reseeds, the quality of the random numbers rests on the quality of the ChaCha20 block operation. As this operation produces data following an equidistribution, the conclusion from section 5.2.4 is still applicable for the deterministic operation phase of the ChaCha20 DRNG.

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

5.3.5 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 by 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 type of the Linux-RNG being a DRG.3 random number generator. 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 by 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.

Compared to the analysis of the raw entropy conducted previously, the reboot testing has been updated to comply 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 greatly:

- 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 64 interrupts one bit of entropy, a single interrupt is implied to have 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 63 correct deductions out of 64 observations, 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 3.5.2.8 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` is invoked to add the Jiffies time stamp for a block device event. Albeit the block device events are not observable as interrupts with their /proc/interrupts file, an attacker is able to trigger block device events and record his trigger times. It is assumed that an attacker is able to resolve the precise Jiffies value considering the coarse resolution of the Jiffies time stamp. Hence, again, the Jiffies value is considered to deliver no entropy which leads to the exclusion of the Jiffies value from consideration in the study's entropy analysis.
- Finally, `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 3.5.2.8. 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[18]. A 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 Jiffies time stamp to a particular HID event. As this Jiffies value suffers from the same issue discussed in section 6.1.2, this value will be disregarded in the entropy analysis of this study.
- Again, for each HID event, the high-resolution time stamp is added. 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 3.5.2.6. The description of the scheduler-based noise source given in section 3.5.2.6 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:

18 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
```

- 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 3.5.2.8 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 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[19].

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 Min-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.

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

Entropy Estimation Type	Entropy Estimate
Most Common Value Estimate	5.425874
Collision Estimate	7.701248
Markov Estimate	7.940488
Compression Estimate	5.016936
t-Tuple Estimate	4.314848
LRS Estimate	
Multi Most Common in Window Prediction Estimate	5.302400
Lag Prediction Estimate	3.779874
MultiMMC Prediction Estimate	3.301727
LZ78Y Prediction Estimate	3.301735

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

The associated Shannon entropy plug-in estimate is 7.943 bits per interrupt event. The Min-Entropy plug-in estimate is 6.032 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 64 interrupts to be collected and mixed into the input_pool. The entire data from 64 interrupt is credited with one bit of entropy. 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.

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	7.879600
Collision Estimate	7.646648
Markov Estimate	7.996720
Compression Estimate	7.166872
t-Tuple Estimate	7.879600
Longest Repeated Substring (LRS) Estimate	7.718814
Multi Most Common in Window Prediction Estimate	7.942932
Lag Prediction Estimate	7.964191
MultiMMC Prediction Estimate	7.955392
LZ78Y Prediction Estimate	7.956101

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 7.925 bits per block device event is calculated with the Min-Entropy plug-in estimate.

In addition to the collection of the noise source data, the test also collected the entropy estimates per block device event applied by the Linux-RNG. The histogram given in figure 12 specifies all possible entropy estimation values from zero to 11 that can be applied by the Linux-RNG. The histogram shows how often the Linux-RNG awards these entropy estimates to the recorded block device events.

Figure 12 also shows that the mean value of all entropy estimates is 0 bits of entropy. This can be interpreted that on average, the Linux-RNG awarded each block device event 0 bits of entropy.

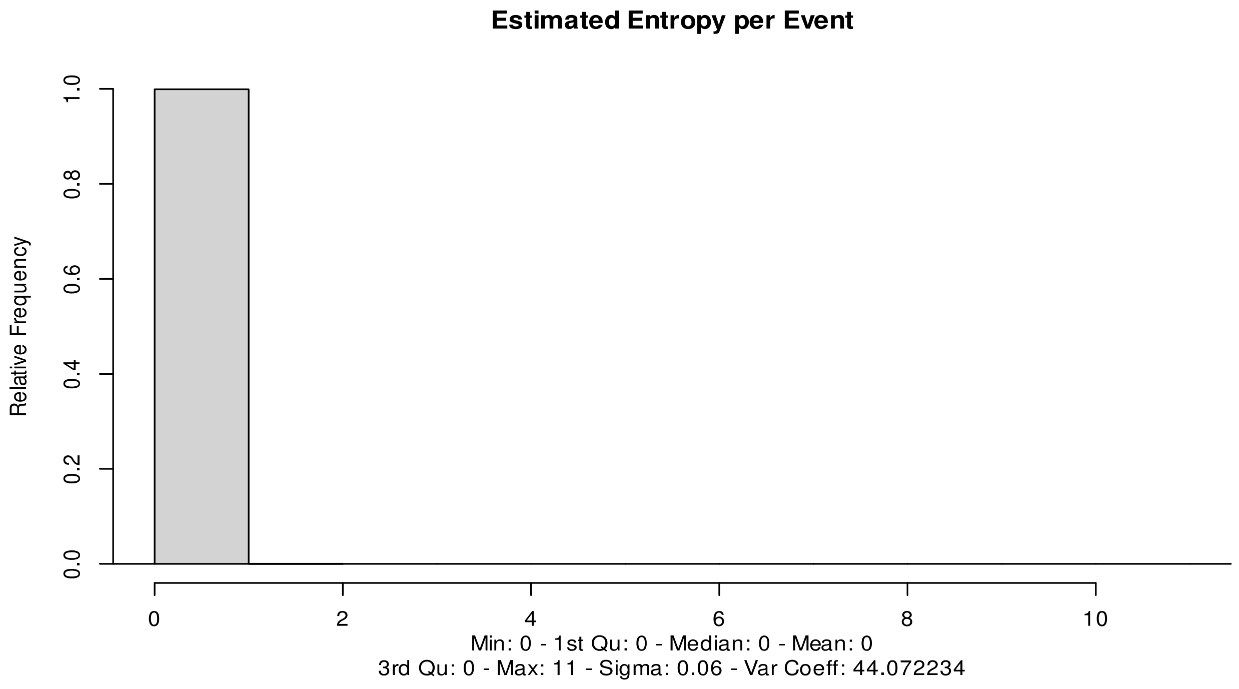

Figure 12: Entropy Estimate per Block Device Event Applied by Linux-RNG – Worst Case

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	7.883765
Collision Estimate	7.584400
Markov Estimate	7.990328
Compression Estimate	7.047840
t-Tuple Estimate	7.314320
Longest Repeated Substring (LRS) Estimate	7.905551
Multi Most Common in Window Prediction Estimate	7.896943
Lag Prediction Estimate	7.950286
MultiMMC Prediction Estimate	7.934351
LZ78Y Prediction Estimate	7.935050

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

The Shannon entropy plug-in estimate shows 8 bits per block device event. 7.941 bits per block device event are calculated with the Min-Entropy plug-in estimate.

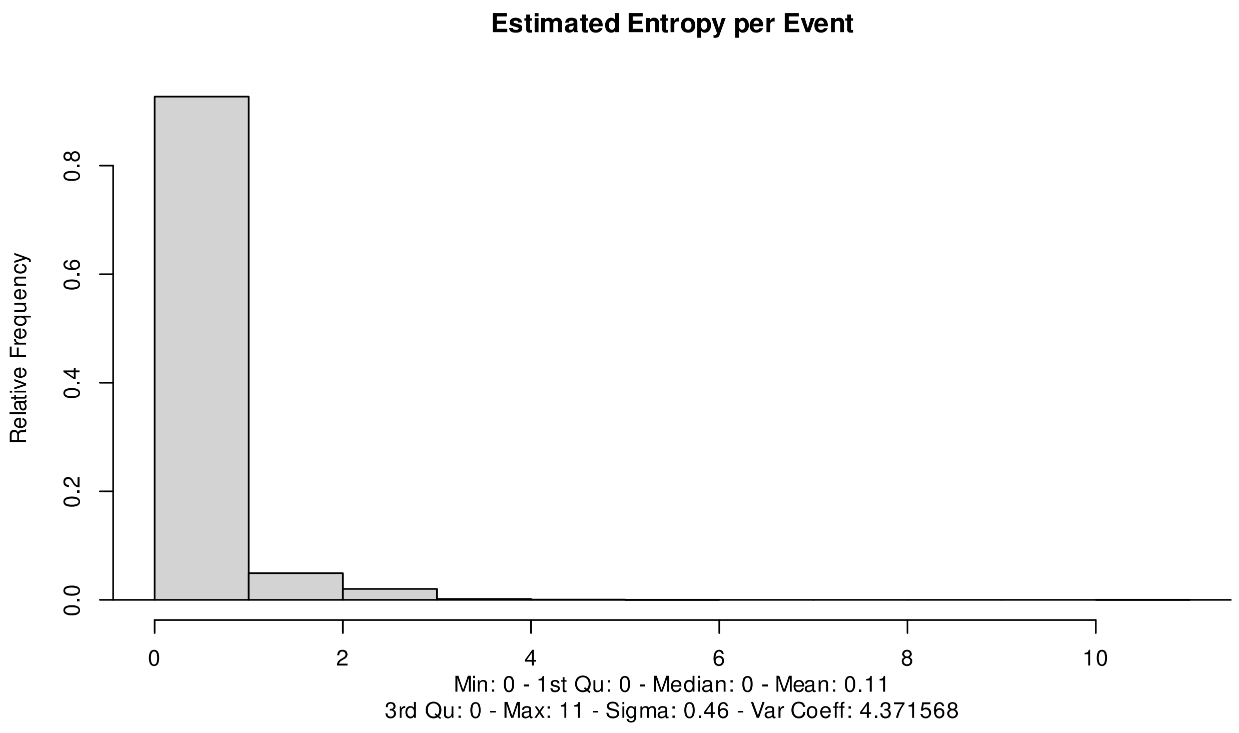

Figure 13: Entropy Estimate per Block Device Event Applied by Linux-RNG – Normal Use Case

Figure 13 shows that the mean value of all entropy estimates for the normal use case is 0.11 bits of entropy. This can be interpreted that on average, the Linux-RNG awarded each block device event 0.11 bits of entropy.

Comparing the result shown in figures 12 and 13 with the min-entropy estimates calculated from the measured time stamps, the following conclusion is drawn: the min-entropy estimates have significantly more than 6 bits of entropy per event. On the other hand, the Linux-RNG considers that each event has on average only up to 0.11 bits of entropy.

This allows the conclusion that the Linux-RNG significantly underestimates the entropy present in the block device noise source data. This significant underestimation implies that the Linux-RNG acts conservatively and thus upholds the cryptographic strength it reports with its entropy estimation.

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	7.879254
Collision Estimate	7.519152
Markov Estimate	7.976592
Compression Estimate	7.262936
t-Tuple Estimate	7.258976
LRS Estimate	7.886864
Multi Most Common in Window Prediction Estimate	7.960063
Lag Prediction Estimate	7.938681
MultiMMC Prediction Estimate	6.638405
LZ78Y Prediction Estimate	6.638399

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

The Shannon entropy plug-in estimate applied on the data set results in 7.958 bits per HID event. 7.502 bits per HID event are calculated when using the Min-Entropy formula.

The test record of the entropy estimate applied by the Linux-RNG for each recorded HID event is depicted with figure 14. This figure lists all possible entropy estimates applied by the Linux-RNG to a HID noise source event ranging from 0 to 11. A histogram is prepared showing all recorded entropy estimates for HID noise source events.

As shown in figure 14, the mean value of the histogram is 0.49 bits. This implies that the Linux-RNG awarded 0.49 bits of entropy to each HID noise source event on average.

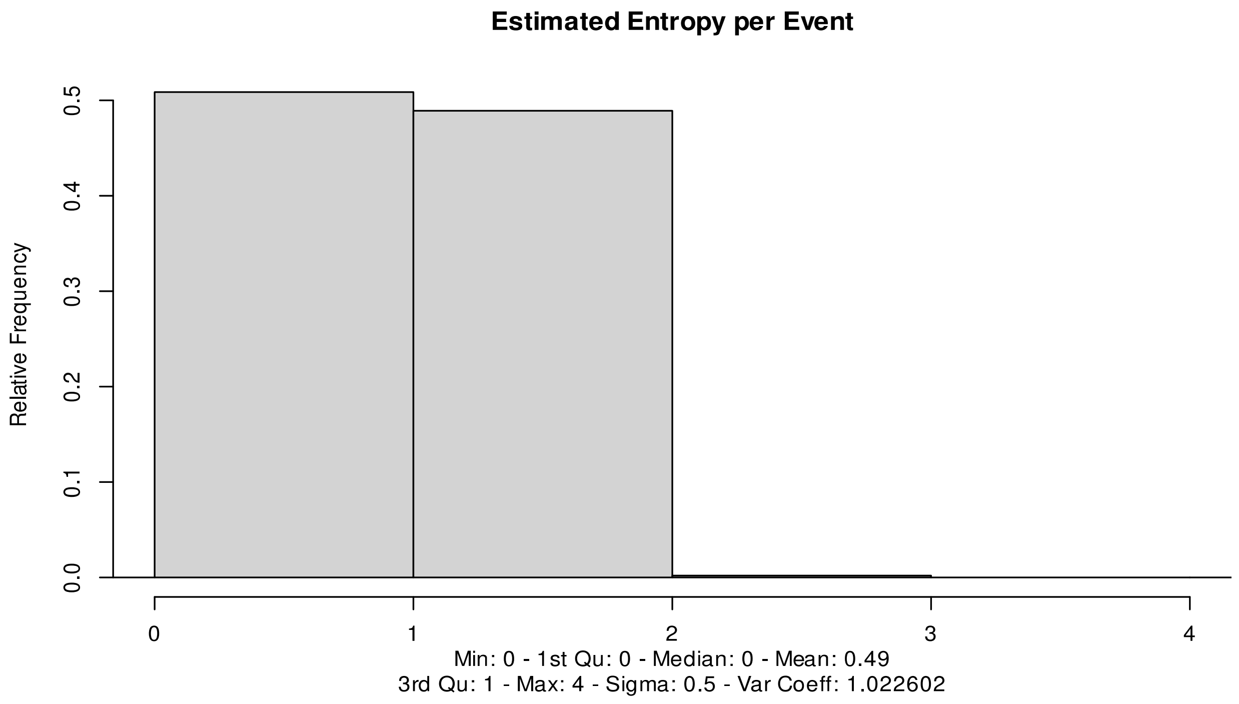

Figure 14: Entropy Estimate per HID Event Applied by Linux-RNG

A conclusion can be reached when comparing the heuristic entropy values applied by the Linux-RNG from figure 14 with the min-entropy estimates. The min-entropy estimates have shown that much more entropy is available as applied by the entropy heuristic.

This comparison allows to conclude that the Linux-RNG again underestimates the available entropy for HID events. This underestimation shows again that, the Linux-RNG applies a conservative entropy estimation and thus upholds the cryptographic strength it reports with its entropy estimation.

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 a very conservative entropy estimate to each individual noise source.

Considering the HID and block device noise sources alone, the combinations of the noise source event data when mixing the data into the input_pool is not considered diminishing any entropy. This is due to the fact that both noise sources are independent. Thus, viewing both noise sources collectively, it can be concluded that the Linux-RNG significantly underestimates the entropy.

Bringing the data from the interrupt noise source into the picture, the interpretation changes as follows: the interrupt noise source has a correlation with the HID and block device noise source as each HID or block device event also triggers an interrupt noise source event. The correlation is assumed to be diminished by the use of the fast_pool which mixes the interrupt data of at least 64 interrupts before injecting the data into the input_pool. Yet a complete diminishing of correlation between the data of the HID and block device noise source on the one hand and the fast_pool content on the other hand cannot be assumed.

However, the Linux-RNG applies a massive underestimation of the available entropy in case of interrupts which gives rise to the following reasoning: the min-entropy estimates show that for 64 interrupts significantly more than 128 bits of entropy are present in the input data. The Linux-RNG awards these 64 interrupts, however, only one bit. This massive underestimation of entropy is considered to outweigh the potentially existing correlation between the HID and block device noise source event data on the one side and the interrupt noise source event data maintained by the fast_pool and injected into the input_pool on the other side.

This finally allows the conclusion that the entropy present in the noise source data collectively is underestimated by the Linux-RNG. Therefore, the Linux-RNG is conservative such that the heuristically determined entropy value awarded to an event and added to the entropy estimator of an entropy pool can be considered to represent at least the cryptographic strength of the data maintained by the Linux-RNG.

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 pools, 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	7.353758	5.880	7.749

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 7.3 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 5.9 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 15 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 15 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.

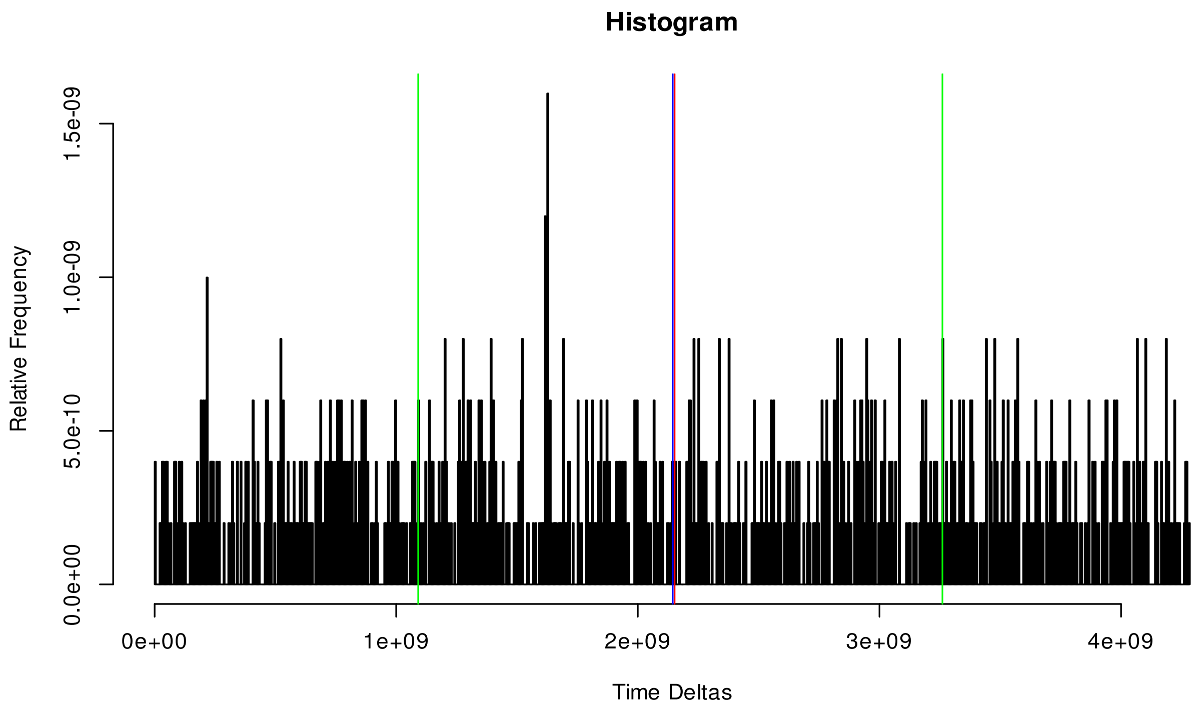

Figure 15: Histogram of Time Deltas for First and Second Interrupt in a Virtual Environment – Column-Wise

The table with the Min-Entropy for the time stamps of the first 1000 interrupts visualized in figure 15 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, only the high and low 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	6.638405	6.057	7.748

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

Similarly to the column-wise assessment, figure 16 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.

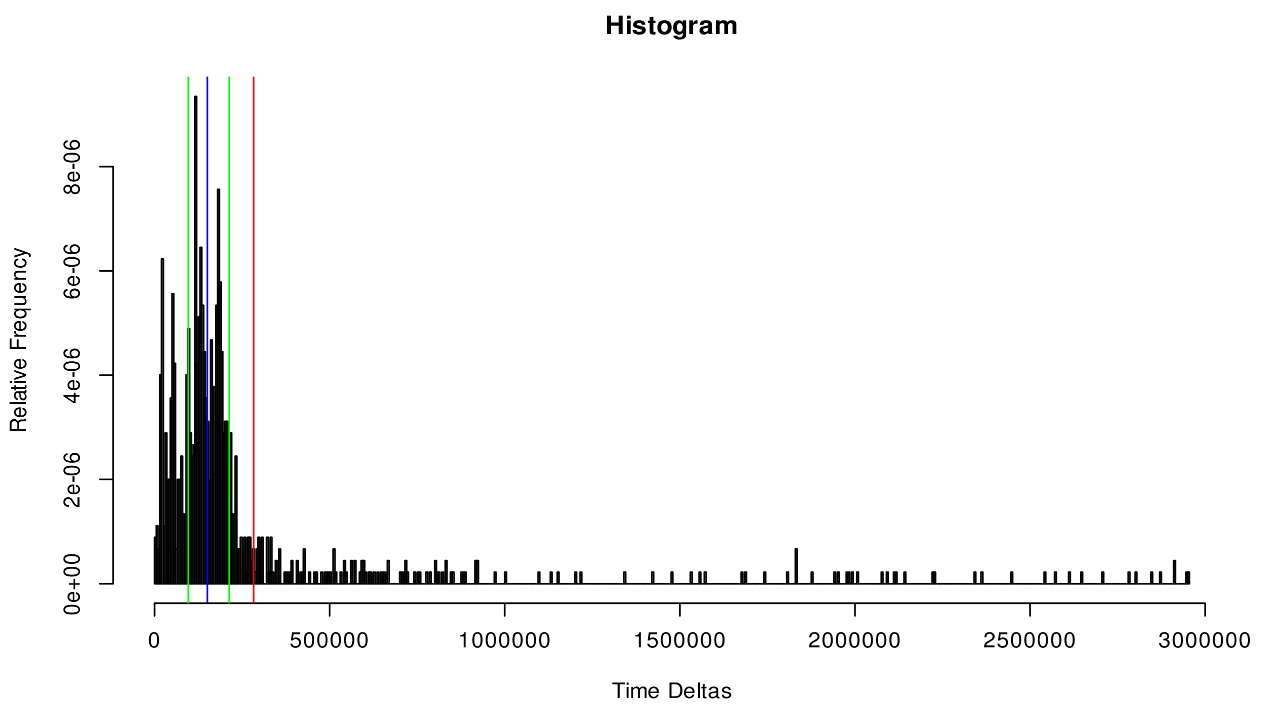

Figure 16: Histogram of Time Deltas for First and Second Interrupt in a Virtual Environment – Row-Wise

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 mandates that an anticipated entropy rate is provided. The entropy rate expected to be present is at least 1 bit of entropy per time stamp. The reason for this value is the following: during boot time the time stamps from 4 times 64 interrupts are injected into the ChaCha20 DRNG. In order to assume the ChaCha20 DRNG to be fully seeded, each time stamp must at least deliver one bit of entropy.

The following data is obtained from the sanity test:

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

6.3.2 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 3.5.2.6. 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 and maximum 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	7.353758	5.967	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 7.3 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 17.

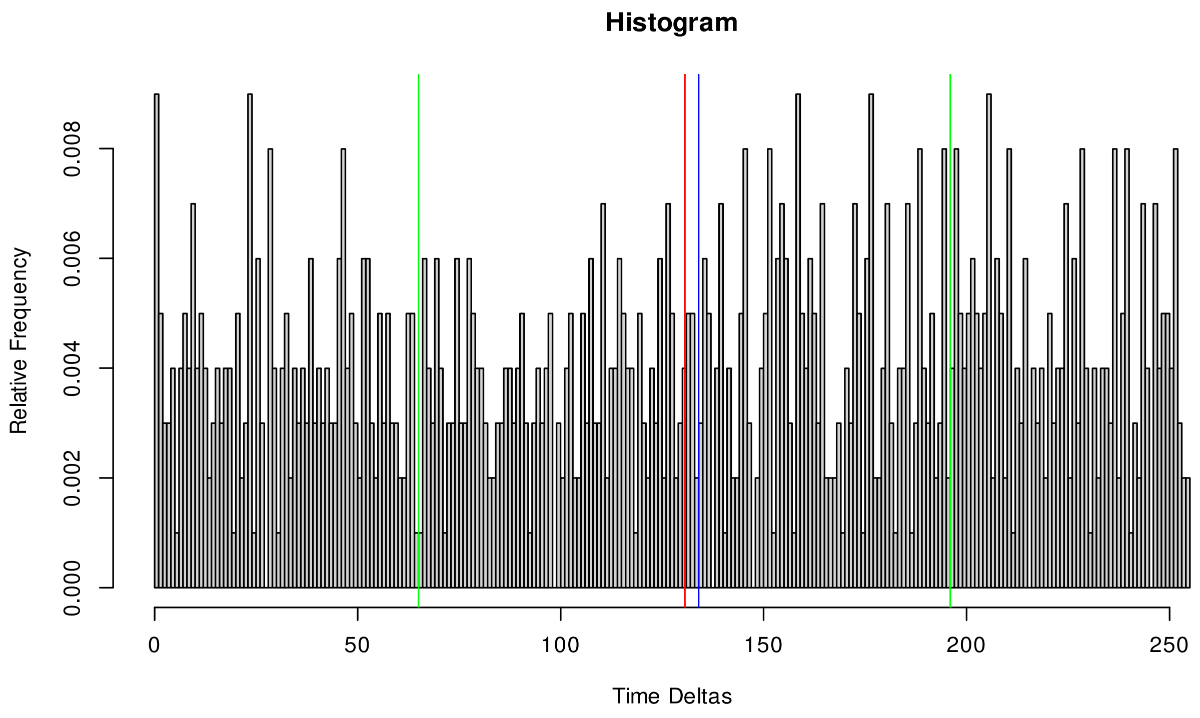

Figure 17: Histogram of Time Deltas for First and Second Scheduler-Based Noise Source Event in a Virtual Environment – Column-Wise

The table with the Min-Entropy for the time stamps of the first 1000 scheduler-based noise source events visualized in figure 17 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 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	3.197303	2.607	3.972

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 18 shows the row-wise time deltas for the first and second event.

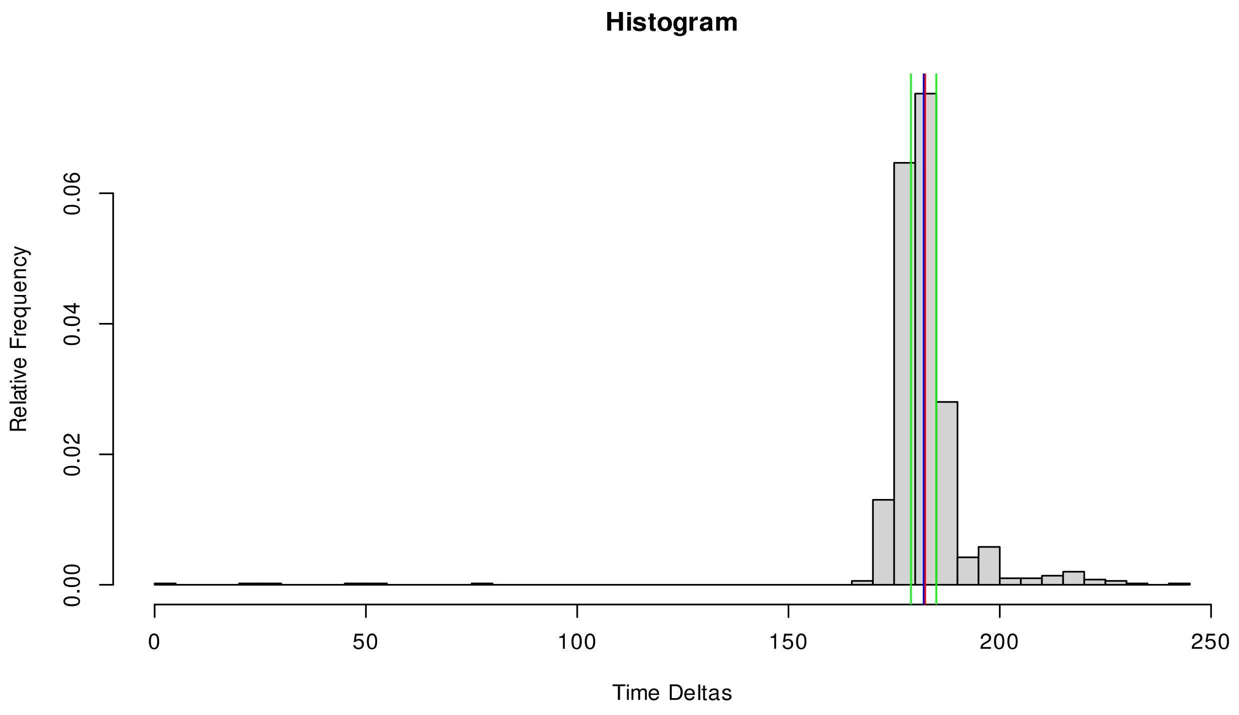

Figure 18: Histogram of Time Deltas for First and Second Interrupt in a Virtual Environment – Row-Wise

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: 63
- 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	6.638399	5.965	7.739

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 19. 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.

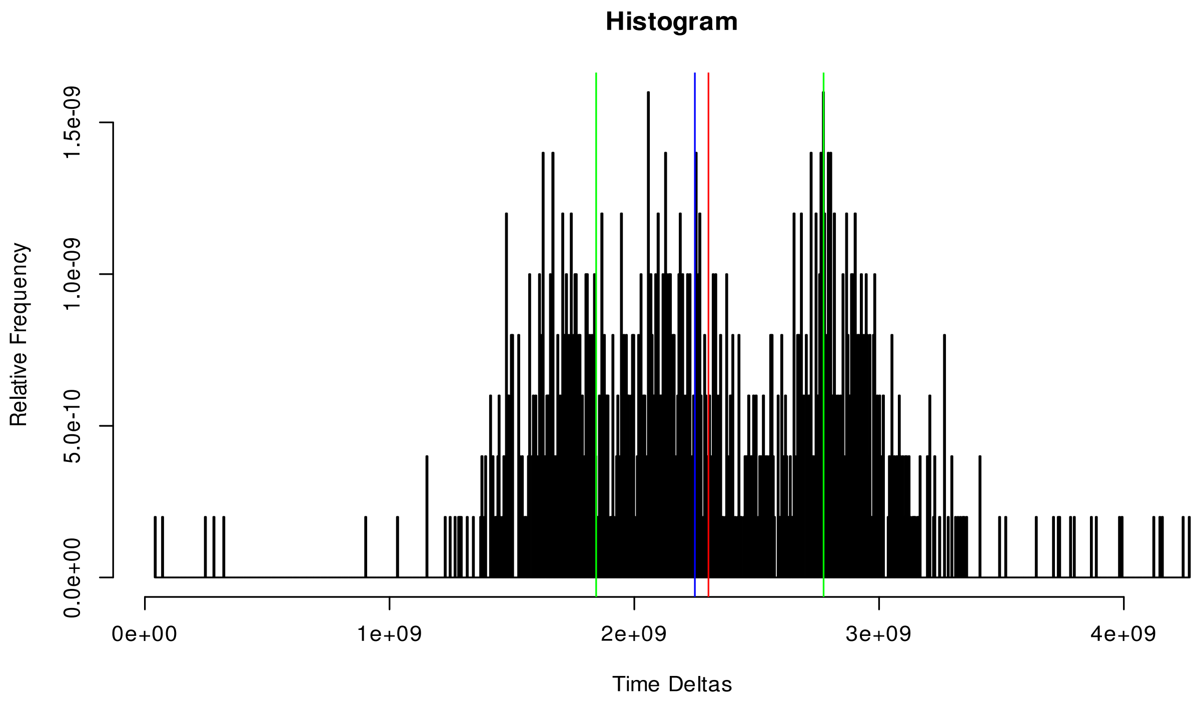

Figure 19: Histogram of Time Deltas for First and Second Interrupt in a Native Environment – Column Wise

Starting with the second time delta depicted in figure 20, 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.

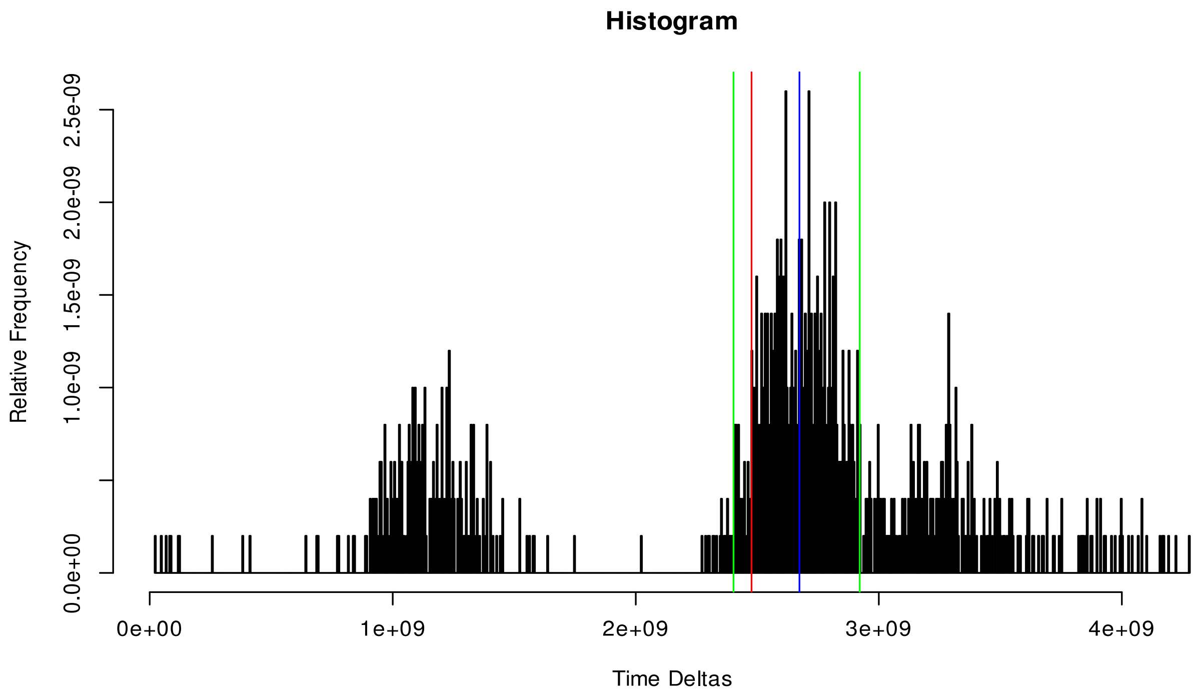

Figure 20: Histogram of Time Deltas for Second and Third Interrupt in a Native Environment – Column-Wise

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 from 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, only the high and low 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	7.860916	5.879	7.735

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

Similar to the column-wise assessment, figure 21 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.

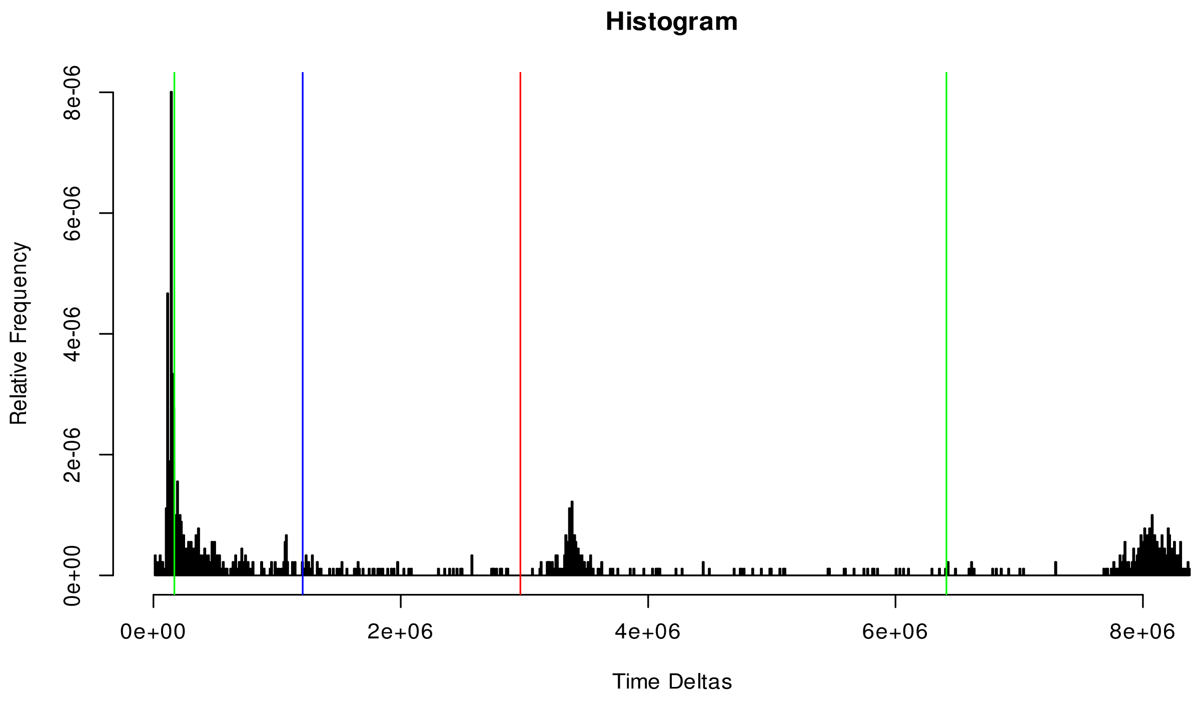

Figure 21: Histogram of Time Deltas for First and Second Interrupt in a Native Environment – Row-Wise

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 and maximum 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	7.241443	5.966	7.736

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 7.2 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 of5.9 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 22.

The table with the Min-Entropy for the time stamps of the first 1000 events visualized in figure 22 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.

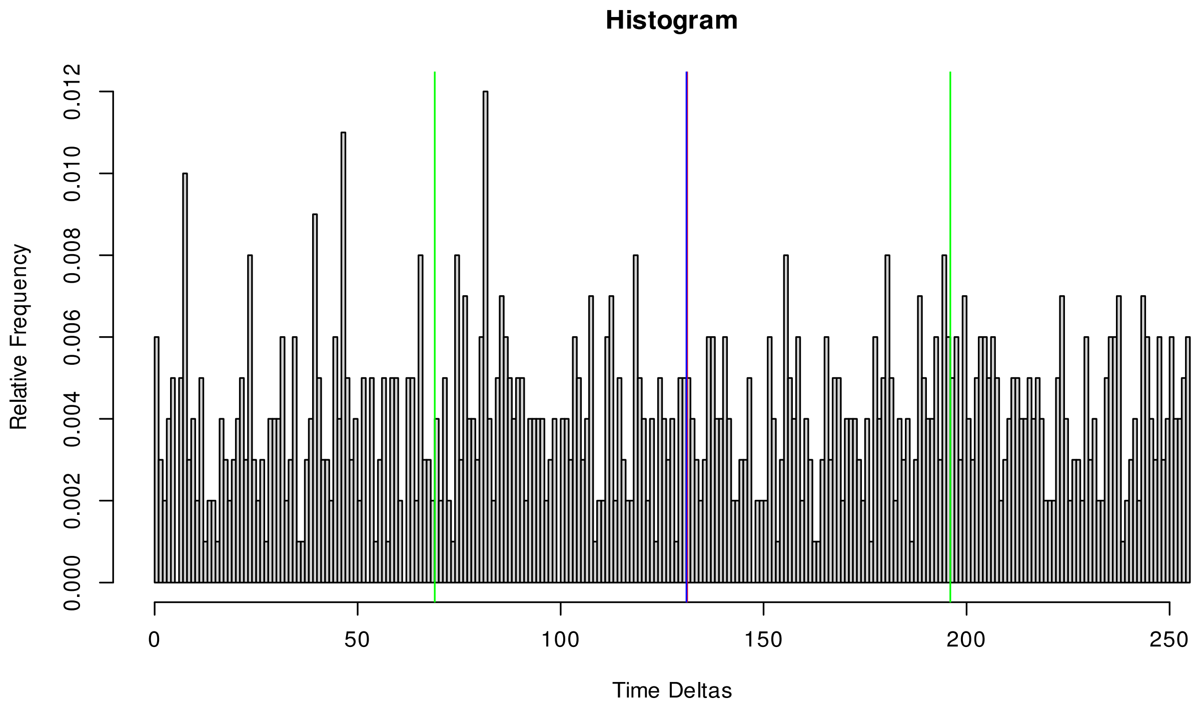

Figure 22: Histogram of Time Deltas for First and Second Scheduler-Based Noise Source Event in a Native Environment – Column-Wise

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	0.078953	0.497	1.304

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 23 shows the row-wise time deltas for the first and second event.

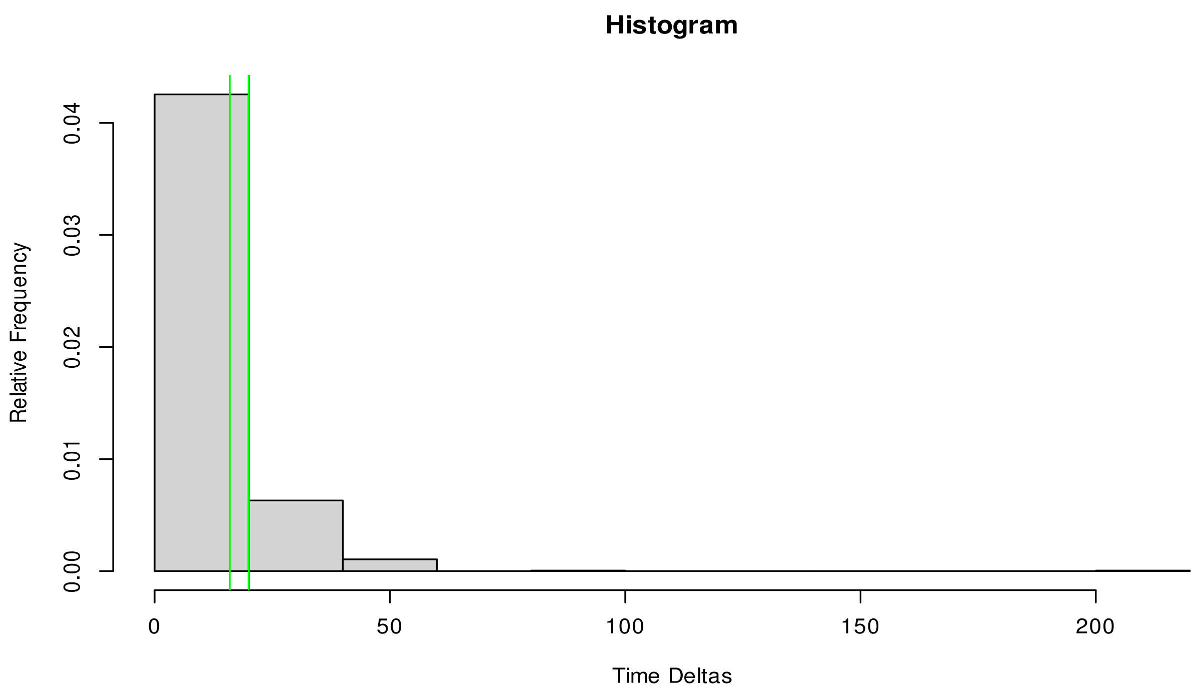

Figure 23: Histogram of Time Deltas for First and Second Scheduler-Based Noise Source Event in a Native Environment – Row-Wise

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. 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: 134
- 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 128 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. The underestimation of the entropy is alleviated to some extent by injecting the first four sets of received 64 interrupts into the ChaCha20 DRNG and marking this DRNG as initially seeded. 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. When reaching the state of being fully seeded and thus having the ChaCha20 DRNG seeded with 256 bits of entropy from at least 256 interrupts and 128 bits of heuristically measured entropy from the noise sources, the `getrandom` system call as well as /dev/random unblocks and generates random numbers. This allows the conclusion that when the two interfaces unblock, sufficient entropy has been accumulated to be 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 ChaCha20 DRNG only after the fast_pool received 64 interrupts. This implies that 4 filled fast_pools are required to be injected into the ChaCha20 DRNG to reach the seeding level of 256 bits of entropy. Considering the presence of multiple CPUs where interrupts may be received by the different CPUs and thus mixed into the respective CPU's fast_pool the following pathological case must be considered. Common systems have multiple CPUs, often 4 CPUs while in virtual environments there is no need for a correspondence of a virtual CPU to a physical or hyperthreaded CPU to allow for an over-commitment of CPUs. Assuming the presence of 4 CPUs, in a pathological case where each of the CPU processes interrupts with an equal chance[20], 256 interrupts are required before even one fast_pool is injected into the ChaCha20 DRNG. Thus, at the time user space starts and data is obtained from /dev/urandom, the ChaCha20 DRNG in a worst case may not be seeded with any data. Naturally, with more CPUs on the system, the pathological case is more severe.

- 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 pathological case for /dev/urandom is still relevant for native hardware, though with a lesser probability.

20 It is quite likely that such pathological case is present. When reviewing /proc/interrupts, for a number of interrupt types a more or less even distribution of interrupts to CPUs can be seen.

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 LFSR operation. When data is injected into the ChaCha20 DRNG, it is mixed with the already present data. In all these cases, the state transition function of the deterministic random number generation mechanism is used to mix-in the received data.

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.

Before the testing is conducted, the properties of the LFSR polynomial are analyzed.

7.1 Properties of the LFSR Polynomials

Using the mathematical tool magma, the LFSR polynomials can be analyzed. The analysis follows [LRSV12] and is performed over GF(2^{32}) considering the formula $Q(X) = \alpha^3 (P(X) - 1) + 1$ where P(X) references the polynomial subject to assessment.

For performing the analysis of the polynomials, the mathematical tool Magma is used. An interface that is freely available can be accessed at http://magma.maths.usyd.edu.au/calc/ where the Magma code listed below can directly be processed.

The following code is used for the analysis of the polynomials applied for the LFSR of the input_pool:

```
// Define finite field of size 2
F2 := FiniteField(2);


// Define a polynomial ring over the finite field of size 2
R2<x> := PolynomialRing(F2);


// This is the CRC-32-IEEE 802.3 polynomial mentioned in Section 3.1.3
// which can be found in
//
https://www.xilinx.com/support/documentation/application_notes/xapp209.pd
f
```

```
G := x^32 + x^26 + x^23 + x^22 + x^16 + x^12 + x^11 + x^10 + x^8 + x^7 +
x^5 + x^4 + x^2 + x + 1;

// Check if this polynomial is irreducible. It must be
// irreducible because otherwise F32 would not be a field
print "is the CRC-32-IEEE 802.3 polynomial irreducible?";
IsIrreducible(G);

// Define the finite field with 2^32 elements with defining
// polynomial G (see above)
F32<alpha> := ext<F2 | G>;

// Define a polynomial ring over the finite field F32
R<y> := PolynomialRing(F32);

// Define the polynomials P and Q used in random.c
P_input := y^128 + y^104 + y^76 + y^51 + y^25 + y + 1;
Q_input := alpha^3*(P_input-1)+1;


// Check if Q(X) is irreducible
print "is Q for input_pool polynomial irreducible?";
IsIrreducible(Q_input);


// Divide Q(x) by its leading coefficient to make it monic.
// Only then it can be tested for primitivity
print "is input_pool polynomial primitive?";
l := LeadingCoefficient(Q_input);
IsPrimitive(Q_input/l);
```

The polynomial used for the input_pool LFSR operation is deemed to be primitive and irreducible with the aforementioned verification.

7.2 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.
  - The LFSR operation requires a logarithm function which is taken from the Linux kernel code.
  - 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:
  - Disabling the secondary ChaCha20 DRNG handling code. This is justified as the test code only analyzes the ChaCha20 state transition function for one instance. This includes the disabling of the NUMA setup code.
  - Disabling the reseed timer enforcement. As mentioned in the ChaCha20 DRNG design, the ChaCha20 DRNG is reseeded every 5 minutes. As this is irrelevant for testing, the respective trigger code is disabled.
  - Removing the locking code.
  - Disabling the special triggers to fetch data from the fast_pools during the initialization.

7.2.1 LFSR State Transition Function

The test code for the LFSR demonstration extracts the LFSR operation into user space to allow analyzing it and drawing conclusions.

The state of the entropy pool is initialized with zeros.

The test mixes 128 bytes into the entropy pool. This number is equal to the number of 32-bit words present in the entropy pool. As the mixing function operates word-wise, after the mix-in of the 128 bytes, all words of the entire entropy pool have been updated once. After the mix-in of each individual byte, the state of the entropy pool is printed to see the filling of the state with data.

This injection operation is performed for 100,000 cycles. A snapshot of the binary representation of the entropy pool state is extracted after each complete mix-in of 128 bytes.

The resulting binary data is processed as follows – all processing is identical with the data processing outlined and discussed in detail in chapter 8:

- The Chi-Squared value calculated by the `ent` tool is obtained.
- The binary data is compressed with the `gzip2`, `bzip2`, `xz` and `lzma` compression tools.

- The binary data is processed with the Test Procedure A defined in [AIS2031].

7.2.1.1 input_pool

To support the discussion of the behavior of the LFSR, the content of the input_pool before the first LFSR operation is set to zero.

After the injection of the first byte which holds a 1, the state of the entropy pool is:

```
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000c8206e
3b
```

The output shows that the majority of the state is still zeros. Only the last byte contains data. This observation can be readily matched to the LFSR operation as follows:

- The byte that was mixed in contains a one. Even the expansion of the one byte into a 32 bit word leaves it at one because the rotation operation leaves it untouched as the `input_rotate` variable is still zero.
- The LFSR polynomial application to the 32 bit value holding a one will not change the value as all taps taken from the other parts of the entropy pool are still zero.
- The low 3 bits of the 32-bit value holding a one are taken as a pointer into the `twist_table` storing the CRC32 constants. The low 3 bits represent a one, thus the value 0x3b6e20c8 is chosen from the table.
- The `twist_table` value is XORed with the 32-bit value that is right-shifted by 3. The 32-bit value contains a one which after a right-shift by 3 turns into a zero. Thus, the twist_table value is XORed with zero leaving only the `twist_table` value.

When considering the fact that the test system is an Intel x86 system which operates in little-endian format, the memory dump represented with the hexadecimal values shown above, and turning the little-endian representation into a big-endian representation which is used for integer operation, it is visible that the entropy pool state contains just the value from the `twist_table`.

After the second byte with a value 2 is mixed into the entropy pool, its content is:

```
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
```

```
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000039c46d07c8206e
3b
```

It is visible that the first 32 bit word is unchanged and only the second 32-bit word is modified. That second word is already affected by the first word due to the LFSR polynomial.

After printing the state of the entropy pool after the 127th byte is mixed in, it contains:

```
000000001e83e686423d37ea46cdb9359e6dff12b5631e9ae4ad4605282607df625218c5f
fdffcfb5c2a131462991d7dd59b10cf08f1f10adbab6097ec9bb92efcc26a9f18a23423e3
41c2b8d298b3e3853c1c197fd22747d47e9c267782503bad8bf6205ea8396220937848950
ea80d8224de0c6fbb7ad74d625de184c99240beef8cd2f33e6f2c1721620d0d0595410c30
52521d687c0e65e744f9a3989a3838d7b67d30faf8f027b654ebe8a5115e2a4f99036bd7f
efce5efbd21dd158bd49e3d15bae538748edc6f73437f90a9e522902f7b7dd0b8d299e48d
3f8d2943976ae550ee591ba45d1952512c020af18e261b208419c15bf49a1e5265c6c2c8f
19a4e8a3365cb214bfc461a6ca5687d7927c1df5f50fc0d9de3cf1ec8af2550ff78779e23
8a2a90a92b5c0899b8ab277951a5ce315e8181f3bf9c1f0449b952778130a714605636623
b33987536ea624a5af62fe5595fd639a75e62885db21fa7de1090337829048d77939f50c7
0069e85e9b687672750346507 13e095e119081e2d60f2bfde5914bcd89b4ae29df70502a6
07f49df8beb9d478fd735b15cae4e4bf6e11868b20fc7058cc8d26d7f412029ad7bd98d74
687deea8a19df6522b3ff1913a22c9a9a5ca315198e38c9006016084e40800a325470310b
e54de8a63c8b549a9fcac549e88e6acd49f8672832571926aa74f80833b39c46d07c8206e
3b
```

It is visible that all words of the entropy pool except the last one is filled. Yet, all words have been modified only once – compare the first and second word with the output listing above.

After the 128th byte is mixed in, the entire entropy pool has been modified once:

```
6b83e8731e83e686423d37ea46cdb9359e6dff12b5631e9ae4ad4605282607df625218c5f
fdffcfb5c2a131462991d7dd59b10cf08f1f10adbab6097ec9bb92efcc26a9f18a23423e3
41c2b8d298b3e3853c1c197fd22747d47e9c267782503bad8bf6205ea8396220937848950
ea80d8224de0c6fbb7ad74d625de184c99240beef8cd2f33e6f2c1721620d0d0595410c30
52521d687c0e65e744f9a3989a3838d7b67d30faf8f027b654ebe8a5115e2a4f99036bd7f
efce5efbd21dd158bd49e3d15bae538748edc6f73437f90a9e522902f7b7dd0b8d299e48d
3f8d2943976ae550ee591ba45d1952512c020af18e261b208419c15bf49a1e5265c6c2c8f
19a4e8a3365cb214bfc461a6ca5687d7927c1df5f50fc0d9de3cf1ec8af2550ff78779e23
8a2a90a92b5c0899b8ab277951a5ce315e8181f3bf9c1f0449b952778130a714605636623
b33987536ea624a5af62fe5595fd639a75e62885db21fa7de1090337829048d77939f50c7
0069e85e9b68767275034650713e095e119081e2d60f2bfde5914bcd89b4ae29df70502a6
07f49df8beb9d478fd735b15cae4e4bf6e11868b20fc7058cc8d26d7f412029ad7bd98d74
687deea8a19df6522b3ff1913a22c9a9a5ca315198e38c9006016084e40800a325470310b
e54de8a63c8b549a9fcac549e88e6acd49f8672832571926aa74f80833b39c46d07c8206e
3b
```

The test now dumps this data after the 128th byte has been mixed in as a binary bit stream. This dumping is performed again every time when 128 bytes have been mixed into the entropy pool. This operation is performed for 100,000 cycles.

The binary output shows the following characteristics:

- The Chi-Squared result when processing the data bit-wise is 28.00. The result indicates that the output shows the characteristics of an ideal random number generator.

- The Chi-Squared result for byte-wise processing of the binary data is 30.38. This result also indicates that the data shows the characteristics of an ideal random number generator.
- For all compression algorithms, the “compressed” data is larger than the binary data. Thus, the compression algorithms could not find patters that allow the data to be compressed. This is another indication for the data showing the characteristics of an ideal random number generator.
- All tests pass the test procedure A, providing another indication for data showing the characteristics of an ideal random number generator.

All results combined strongly suggest that the data shows characteristics of an ideal random number generator which is expected for an LFSR operation. This allows the conclusion that the LFSR implementation does not exhibit flaws that are considered to diminish entropy.

Considering that LFSRs with primitive polynomials are expected to produce data showing the characteristics of an ideal random number generator and collecting and compressing entropy, it can be concluded that the LFSR used for the input_pool state transition is appropriate for the maintenance of data holding entropy.

7.2.2 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 fast load phase (4 copies of fast_pool) causes the state to be non-cryptographically mixed as the input data is XORed with the present data.
- ChaCha20 DRNG reseed causes the state to be non-cryptographically mixed as the input data is XORed with the present data.
- The generation of data results in data showing the characteristics of an ideal random number generator even though the seed is deterministic.
- The backtrack resistance causes the state to be cryptographically mixed.

The test provides a memory for the ChaCha20 state which is filled with zeros to allow the operation of the DRNG to be made visible.

After the initialization of the ChaCha20 state, it has the following content:

```
65787061 6e642033 322d6279 7465206b 00000000 01000000 02000000 03000000
04000000 05000000 06000000 07000000 08000000 09000000 0a000000 0b000000
```

That ChaCha20 state shows the expected content:

- The first four words contain the ASCII character representation of the words “`expand 32-byte k`”. According to the rules of ASCII, the small letter representation of the alphabet starts hex 61 for ‘a’ and ends with hex 7A for ‘z’. Thus, ‘e’ is transformed into hex 65, ‘x’ is transformed into hex 78, ‘p’ is transformed into hex 70 and so on. These hexadecimal numbers are the ones found for the first four words.
- The following words are filled with “random” numbers from the seed source according to the initialization of the ChaCha20 DRNG in the Linux-RNG. As described, the test tool’s “seed source” is a counter that is incremented by one as it is visible in the different words. The initialization is performed

using integer arithmetic which implies that the bit stream shows the little-endian representation of the integer values.

The next step is the first loading of the fast_pool. All bytes in that fast_pool are set to ones. The resulting ChaCha20 state is as follows:

```
65787061 6e642033 322d6279 7465206b 01010101 00010101 03010101 02010101
04000000 05000000 06000000 07000000 08000000 09000000 0a000000 0b000000
```

This state shows that the 4 words of the first fast_pool are injected into the words four through seven which stores the first 128 bits of the ChaCha20 key according to [RFC7539], section 2.3. The mix-in of the four words whose bytes are all set to one is performed using XOR. This XOR result is therefore clearly visible compared to the previous state.

The second fast_pool content is set to all bytes containing a two. The resulting state after the mix-in is:

```
65787061 6e642033 322d6279 7465206b 01010101 00010101 03010101 02010101
06020202 07020202 04020202 05020202 08000000 09000000 0a000000 0b000000
```

The state now clearly shows that the words eight to eleven are modified by the XOR operation. These words represent the second 128 bit part of the 256 bit ChaCha20 key as defined in [RFC7539], section 2.3.

The third fast_pool content is set to all bytes containing the value three. The content of the ChaCha20 DRNG looks like:

```
65787061 6e642033 322d6279 7465206b 02020202 03020202 00020202 01020202
06020202 07020202 04020202 05020202 08000000 09000000 0a000000 0b000000
```

This is expected as the four fast_pool words are XORed with the first 128 bits of the key part of the ChaCha20 DRNG state.

The fourth fast_pool content is set to all bytes holding the value four. As expected, the content is now:

```
65787061 6e642033 322d6279 7465206b 02020202 03020202 00020202 01020202
02060606 03060606 00060606 01060606 08000000 09000000 0a000000 0b000000
```

This content shows that the second part of the 128 bits of the ChaCha20 key are modified by the XOR operation.

The injection of the fast_pool is followed by a full reseed of all words defining the key. Again, the reseed uses predictable content based on two counters (one for the time stamp and one defining the input_pool result):

```
65787061 6e642033 322d6279 7465206b 0b070707 0b070707 0b070707 0b070707
17030303 17030303 17030303 17030303 08000000 09000000 0a000000 0b000000
```

In the test, the next operation after a full reseed is the generation of one ChaCha20 block used as random number. The internal state after the random number generation is:

```
65787061 6e642033 322d6279 7465206b 0b070707 0b070707 0b070707 0b070707
17030303 17030303 17030303 17030303 09000000 09000000 0a000000 0b000000
```

The only difference to the previous state is the increment of the counter word – word 13 – by one (note the use of little endian integer representation). This state is used for the next ChaCha20 block operation. The result of the state after the next ChaCha20 operation differs, as expected, by another increment of one of the counter word. This behavior is conceptually identical to the counter block chaining mode specified in [SP800-38A]:

```
65787061 6e642033 322d6279 7465206b 0b070707 0b070707 0b070707 0b070707
17030303 17030303 17030303 17030303 0a000000 09000000 0a000000 0b000000
```

This operation continues to satisfy one request for random numbers of an arbitrary length which generates the required ChaCha20 blocks. For the testing, the ChaCha20 output is recorded as a bit-stream for 100,000 ChaCha20 blocks. This bit stream is assessed in the following.

After one request is satisfied, the internal state is updated by XORing the “left-over” parts of the last ChaCha20 block (if more than 256 bits are unused) or by using the 256 MSB of a new ChaCha20 block into the key part of the ChaCha20 state:

```
65787061 6e642033 322d6279 7465206b d718cf6a 9082279e 9c1f8d30 965c3e00
71c54c0d 28e2e298 7dd93068 9b3ce29f a8860100 09000000 0a000000 0b000000
```

This state shows that the counter value (in little endian) is decimal 100,008 as expected after the generation of 100,000 ChaCha20 blocks (note, the counter started with the value 8 due to the ChaCha20 initialization). In addition, the key part of the state is XORed with the output of the ChaCha20 block operation. This means that this state update is the only time when the state is affected by a cryptographic operation.

The entire state update discussion demonstrates that:

- The first four words of the state are never changed.
- Any data mixed into the ChaCha20 state is mixed into the key part of the state.
- The counter value is a monotonically increasing counter.
- The nonces are not modified.

The output of the 100,000 ChaCha20 block shows the following statistical properties:

- The Chi-Squared value when treating the data stream as bit-wise is 78.26 which indicates the characteristics of an ideal random number generator.
- The Chi-Squared value for a byte-wise processing of the data stream is 21.44 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.

7.3 AIS 20/31 Test Procedure A for Entropy Pools

With the testing outlined in section 7.2.1, the LFSR behavior is demonstrated. This is achieved by injecting a monotonic increasing counter into the LFSR. With the injection of deterministic operation that has a strong structure, the analysis can be performed whether the LFSR operation transforms such highly structured data into data exhibiting the characteristics of an ideal random number generator. The output of this test is also a byte-stream resulting from the LFSR operation. This byte-stream is analyzed with the AIS20/31 Test Procedure A.

With the obtained data, a binary string can be obtained that shows whether the LFSR implementing the state transition function of the entropy pools guarantees that the data in the entropy pool follows an ideal random number generator.

The test code extracts the LFSR state after injecting 128 counter values. This ensures that all words of the LFSR state were updated.After generating 100,000 snapshots of the LFSR state with the test code and concatenating all data, a binary string is present that shows that all sub-tests of Test Procedure A pass.

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
```

- Using the data generated for the `dieharder` testing, 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 implements an NTG.1 or DRG.3, 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.

The following constraints must be observed to claim that data is obtained from a DRG.3:

- Either the Linux kernel NUMA support must not be compiled or the NUMA-aware Linux kernel must only be executed on a system with one NUMA node. Non-NUMA systems executing with a NUMA-aware Linux kernel are treated to have exactly one NUMA node and thus comply with this constraint.
- Only Linux-RNG interfaces that may block are DRG.3 compliant. Thus, DRG.3 compliant random numbers are generated when using one of the following methods:
  - /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 `add_random_ready_callback` was invoked,
  - invoking the in-kernel `get_random_bytes` API call after the `wait_for_random_bytes` API call returns – note, service functions like the `get_random_XXX_wait` API call family where XXX is either u32, u64, int or long fall into this category.

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 5.7

Previously assessed Linux kernel version: 5.6 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.6.tar.xz

Currently assessed Linux kernel version: 5.7 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.7.tar.xz

Assessment of changes: All changes do not affect the NTG.1 and DRG.3 properties. 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

The following changes are applied to random.c:

- The code initializing the ChaCha20 DRNG state has been updated without affecting the functionality. The changes can be considered as editorial. The changes are applied to handle CPU noise sources which behave differently during boot and runtime.
- The access operation for the state variables for storing the first, second and third discrete derivation have been changed into atomic operations. The functionality of the Linux-RNG remains unchanged from this modification.
- The RNGs accessible through the interface functions of get_random_u64 and get_random_u32 are not seeded by a CPU noise source any more. As these RNGs are not considered part of the Linux-RNG discussed in this document, this change does not affect the operation of the Linux-RNG.

None of the changes affect the functionality of the Linux-RNG.

10.1.1.2 File include/linux/random.h

The following changes are applied to random.h:

- Addition of the CPU noise source handler for early boot to support the functionality discussed for random.c.

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 `hv_stimer0_vector_handler` 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

No new interface was added.

10.2 Linux Kernel 5.8

Previously assessed Linux kernel version: 5.7 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.6.tar.xz

Currently assessed Linux kernel version: 5.8 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.8.tar.xz

Assessment of changes: All changes do not affect the NTG.1 and DRG.3 properties. The configuration requirements given in chapter 9 apply unchanged.

10.2.1 Changes to the Linux-RNG Implementation

10.2.1.1 File drivers/char/random.c

The following changes are applied to random.c:

- A code change without affecting the functionality at all is added by renaming SHA-1 macros in the kernel service functions. These changed names are applied to the Linux-RNG as well.
- One word of the fast_pool currently processed with add_interrupt_randomness that is selected by the least 2 bits of the high-resolution cycle counter is copied into a separate buffer every time add_interrupt_randomness is invoked. The separate buffer is used to "seed" a small-scale, cryptographic non-secure random number generator unrelated to the Linux-RNG. This implies that data that is believed to contain entropy and is credited with entropy by the Linux-RNG is duplicated and re-purposed outside of the Linux-RNG. This operation will diminish the entropy available to the Linux-RNG out of the fast_pool. 32 data bits out of 128 data bits maintained by the fast_pool are duplicated. Considering that after the receipt of at least 64 interrupt events the fast_pool is credited with 1 or 2 bits of entropy which is massively underestimating the available entropy, it is believed that the extraction of the 32 data bits will still leave the Linux-RNG to underestimate the available entropy. Yet, it is currently unclear how to perform a quantitative analysis on this operation and thus the fast_pool behavior in general.

None of the changes affect the functionality of the Linux-RNG.

10.2.1.2 File include/linux/random.h

The following changes are applied to random.h:

- Creation of a per-CPU memory buffer to hold the data extracted from the fast_pool.

10.2.1.3 File include/uapi/linux/random.h

No changes.

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

The following changes are applied to archrandom.h:

- The header file used to define the Intel-CPU instruction mnemonic to invoke RDSEED and RDRAND. The macros for the mnemonic are now removed as the assembler compiler gained support for these instructions. This change does not have an impact to the functionality of the Linux-RNG.

10.2.2 Changes to Invocation of Entropy Gathering Functions

10.2.2.1 add_input_randomness

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

10.2.2.2 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.

10.2.2.3 add_disk_randomness

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

10.2.2.4 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.

10.2.3 Definition and Use of new Interfaces

No new interface was added.

10.3 Linux Kernel 5.9

Previously assessed Linux kernel version: 5.8 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.8.tar.xz

Currently assessed Linux kernel version: 5.9 – https://www.kernel.org/p99/linux/kernel/v5.x/linux-5.9.tar.xz

Assessment of changes: All changes do not affect the NTG.1 and DRG.3 properties. The configuration requirements given in chapter 9 apply unchanged.

10.3.1 Changes to the Linux-RNG Implementation

10.3.1.1 File drivers/char/random.c

No changes.

10.3.1.2 File include/linux/random.h

The following changes are applied to random.h:

- All random32 definitions (i.e. definitions unrelated to Linux-RNG) are extracted to include/linux/prandom.h. The changes are editorial only and do not affect the functionality of the Linux-RNG.

10.3.1.3 File include/uapi/linux/random.h

No changes.

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

No changes.

10.3.2 Changes to Invocation of Entropy Gathering Functions

10.3.2.1 add_input_randomness

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

10.3.2.2 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.

10.3.2.3 add_disk_randomness

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

10.3.2.4 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.

10.3.3 Definition and Use of new Interfaces

No new interface was added.

10.4 Linux Kernel 5.10

Previously assessed Linux kernel version: 5.9 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.9.tar.xz

Currently assessed Linux kernel version: 5.10 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.10.tar.xz

Assessment of changes: All changes do not affect the NTG.1 and DRG.3 properties. The configuration requirements given in chapter 9 apply unchanged.

10.4.1 Changes to the Linux-RNG Implementation

10.4.1.1 File drivers/char/random.c

The following changes are applied to random.c:

- The change introduced with kernel version 5.8 that copies parts of the fast_pool to seed another non-cryptographic random number generator has been reverted. The data maintained in the fast_pool now again is used exclusively for Linux-RNG operations. The respective hint stated in section 4.4 has been removed.

None of the changes affect the functionality of the Linux-RNG.

10.4.1.2 File include/linux/random.h

No changes.

10.4.1.3 File include/uapi/linux/random.h

No changes.

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

No changes.

10.4.2 Changes to Invocation of Entropy Gathering Functions

10.4.2.1 add_input_randomness

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

10.4.2.2 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.

10.4.2.3 add_disk_randomness

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

10.4.2.4 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.

10.4.3 Definition and Use of new Interfaces

No new interface was added.

10.5 Linux Kernel 5.11

Previously assessed Linux kernel version: 5.10 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.10.tar.xz

Currently assessed Linux kernel version: 5.11 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.11.tar.xz

Assessment of changes: All changes do not affect the NTG.1 and DRG.3 properties. The configuration requirements given in chapter 9 apply unchanged.

10.5.1 Changes to the Linux-RNG Implementation

10.5.1.1 File drivers/char/random.c

The following changes are applied to random.c:

- The declarations for SHA-1 have been moved into a separate header file. Thus, the new header file replaced the old one.

None of the changes affect the functionality of the Linux-RNG.

10.5.1.2 File include/linux/random.h

No changes.

10.5.1.3 File include/uapi/linux/random.h

No changes.

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

No changes.

10.5.2 Changes to Invocation of Entropy Gathering Functions

10.5.2.1 add_input_randomness

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

10.5.2.2 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.

10.5.2.3 add_disk_randomness

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

10.5.2.4 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.

10.5.3 Definition and Use of new Interfaces

No new interface was added.

10.6 Linux Kernel 5.12

Previously assessed Linux kernel version: 5.11 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.11.tar.xz

Currently assessed Linux kernel version: 5.12 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.12.tar.xz

Assessment of changes: All changes do not affect the NTG.1 and DRG.3 properties. The configuration requirements given in chapter 9 apply unchanged.

10.6.1 Changes to the Linux-RNG Implementation

10.6.1.1 File drivers/char/random.c

The following changes are applied to random.c:

- A fix has been applied to the RNDRESEEDCRNG IOCTL which requests the primary DRNG to pull data from the input pool. Before that change, the reseed operation was essentially non-functional.
- The function add_interrupt_randomness is modified to not use the CPU entropy source (e.g. RDSEED) anymore. The reason is that on some architectures, the call is too costly in interrupt context. This also implies that the additional crediting of 1 more bit of entropy due to pulling data from the CPU entropy source is also removed. The design documentation is updated accordingly.

None of the changes affect the functionality of the Linux-RNG.

10.6.1.2 File include/linux/random.h

No changes.

10.6.1.3 File include/uapi/linux/random.h

No changes.

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

No changes.

10.6.2 Changes to Invocation of Entropy Gathering Functions

10.6.2.1 add_input_randomness

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

10.6.2.2 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.

10.6.2.3 add_disk_randomness

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

10.6.2.4 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.

10.6.3 Definition and Use of new Interfaces

No new interface was added.

10.7 Linux Kernel 5.13

Previously assessed Linux kernel version: 5.12 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.12.tar.xz

Currently assessed Linux kernel version: 5.13 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.13.tar.xz

Assessment of changes: All changes do not affect the NTG.1 and DRG.3 properties. The configuration requirements given in chapter 9 apply unchanged.

10.7.1 Changes to the Linux-RNG Implementation

10.7.1.1 File drivers/char/random.c

The following changes are applied to random.c:

- Remove of dead code following the removal of the blocking pool for kernel 5.6. The code changes do not affect the operation of the Linux-RNG at all as the code paths were not executed at all.
- Initialize the ChaCha20 constants of the ChaCha20 DRNG with a common callback provided by the kernel-internal ChaCha20 stream cipher implementation. The change makes the initialization of the ChaCha20 state compliant to RFC7539. Previously, on big-endian systems, the initialization string of “expand 32-byte k” is inserted in reverse order making it formally inconsistent with RFC7539. Yet, the change does not exhibit a change that affects the cryptographic strength of the ChaCha20 DRNG.

None of the changes affect the functionality of the Linux-RNG.

10.7.1.2 File include/linux/random.h

No changes.

10.7.1.3 File include/uapi/linux/random.h

No changes.

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

No changes.

10.7.2 Changes to Invocation of Entropy Gathering Functions

10.7.2.1 add_input_randomness

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

10.7.2.2 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.

10.7.2.3 add_disk_randomness

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

10.7.2.4 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.

10.7.3 Definition and Use of new Interfaces

No new interface was added.

10.8 Linux Kernel 5.14

Previously assessed Linux kernel version: 5.13 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.13.tar.xz

Currently assessed Linux kernel version: 5.14 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.14.tar.xz

Assessment of changes: All changes do not affect the NTG.1 and DRG.3 properties. The configuration requirements given in chapter 9 apply unchanged.

10.8.1 Changes to the Linux-RNG Implementation

10.8.1.1 File drivers/char/random.c

No changes.

10.8.1.2 File include/linux/random.h

No changes.

10.8.1.3 File include/uapi/linux/random.h

No changes.

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

No changes.

10.8.2 Changes to Invocation of Entropy Gathering Functions

10.8.2.1 add_input_randomness

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

10.8.2.2 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.

10.8.2.3 add_disk_randomness

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

10.8.2.4 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.

10.8.3 Definition and Use of new Interfaces

No new interface was added.

10.9 Linux Kernel 5.15

Previously assessed Linux kernel version: 5.14 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.14.tar.xz

Currently assessed Linux kernel version: 5.15 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.15.tar.xz

Assessment of changes: All changes do not affect the NTG.1 and DRG.3 properties. The configuration requirements given in chapter 9 apply unchanged.

10.9.1 Changes to the Linux-RNG Implementation

10.9.1.1 File drivers/char/random.c

No changes.

10.9.1.2 File include/linux/random.h

No changes.

10.9.1.3 File include/uapi/linux/random.h

No changes.

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

No changes.

10.9.2 Changes to Invocation of Entropy Gathering Functions

10.9.2.1 add_input_randomness

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

10.9.2.2 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.

10.9.2.3 add_disk_randomness

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

10.9.2.4 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.

10.9.3 Definition and Use of new Interfaces

No new interface was added.

10.10 Linux Kernel 5.16

Previously assessed Linux kernel version: 5.15 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.15.tar.xz

Currently assessed Linux kernel version: 5.16 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.16.tar.xz

Assessment of changes: All changes do not affect the NTG.1 and DRG.3 properties. The configuration requirements given in chapter 9 apply unchanged.

10.10.1 Changes to the Linux-RNG Implementation

10.10.1.1 File drivers/char/random.c

No changes.

10.10.1.2 File include/linux/random.h

No changes.

10.10.1.3 File include/uapi/linux/random.h

No changes.

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

No changes.

10.10.2 Changes to Invocation of Entropy Gathering Functions

10.10.2.1 add_input_randomness

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

10.10.2.2 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.

10.10.2.3 add_disk_randomness

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

10.10.2.4 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.

10.10.3 Definition and Use of new Interfaces

No new interface was added.

10.11 Linux Kernel 5.17

Previously assessed Linux kernel version: 5.16 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.16.tar.xz

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

Assessment of changes: No changes affect the NTG.1 and DRG.3 properties. The configuration requirements given in chapter 9 apply unchanged.

10.11.1 Changes to the Linux-RNG Implementation

10.11.1.1 File drivers/char/random.c

The maintenance of random.c was taken over by a new developer. This resulted in a number of code changes that are editorial in nature and do not alter the behavior of the Linux-RNG. This includes the following aspects:

- Use of current data types (e.g. u8 instead of __u8, u16 instead of unsigned short).
- ChaCha20 constants are set at compile time instead of during runtime initialization.
- Code formatting changes.

In addition to the editorial changes, the following functional changes are present:

- The data output function generated from the LFSR state used to be SHA-1 which was folded in half, extracting 80 bits of data in one round. This is replaced by using the message digest Blake2s where only the 16 most significant bytes are returned in one round. The remaining 16 bytes are discarded by the output function, but the entire message digest is mixed back into the LFSR (see below). This change entails that all SHA-1 related code is replaced. The change is visible in the function extract_buf which calculates a Blake2s message digest and returns the 16 MSB. The Blake2s salt and personal fields are filled by the CPU random number generator (e.g. RDRAND). If it is not available, the Blake2s state is left unchanged. Once the Blake2s message digest is calculated, the entire message digest is mixed back into the LFSR without crediting any entropy.
- The code replaces the runtime resolution of the LFSR tap and property definitions with macros using the same values. This change implies that these values are not resolved at runtime but at compile time. Therefore, the respective variables used for the runtime-resolution are removed.
- The broken and unneeded FIPS 140-2 continuous test is removed including its variables struct entropy_store→last_data_init and entropy_store→last_data.
- The code removes the possibility for multiple entropy pools. A number of functions received a pointer as parameter to the used entropy pool. These pointers are all removed. The code now uses the static definition of the input pool instead. This change does not alter the behavior.
- All operations which are to be invoked after the ChaCha20 DRNG is fully initialized are now moved to a central function crng_finalize_init. The operations are identical to the old code.
- The selection whether a secondary ChaCha20 instance on NUMA-enabled systems or the primary ChaCha20 is to be used is moved into a separate function, select_crng. Yet, the code operates identically to the old code.
- Use of atomic access to the time variables when the ChaCha20 DRNG instance was used last time.
- When random numbers are generated from the ChaCha20 DRNG, the ChaCha20 nonce 32-bit word was XORed with the output from the CPU entropy source (e.g. RDRAND). This is now removed due to performance issues on platforms other than Intel systems.
- The declaration of add_interrupt_randomness is changed to remove the IRQ flags parameter. This parameter was not used by random.c at all and thus this change does not cause a functional alteration.
- The proc files managed by random.c used to be registered during boot time by a different part of the kernel. This is removed. Therefore, random.c received a function random_sysctl_init which is invoked during the boot process to register the proc files.

- When the ChaCha20 DRNG is not yet fully seeded, potential data from hardware random number generators are now inserted into both, the ChaCha20 DRNG and the input_pool. The data inserted into the LFSR, however, is not credited with entropy.

10.11.1.2 File include/linux/random.h

The declaration of the function add_interrupt_randomness removes the interrupt flags parameter which is not provided any more by the interrupt handler as discussed for the changes to random.c above.

None of the changes affect the functionality of the Linux-RNG.

10.11.1.3 File include/uapi/linux/random.h

No changes.

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

No changes.

10.11.2 Changes to Invocation of Entropy Gathering Functions

10.11.2.1 add_input_randomness

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

10.11.2.2 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.

10.11.2.3 add_disk_randomness

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

10.11.2.4 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.

10.11.3 Definition and Use of new Interfaces

No new interface was added.

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 2.8 used for the virtual environment early boot entropy tests documented in 6.3.1:

- 8 virtual CPUs corresponding with the 4 cores and their 2 hyperthreads provided by each core of the host system
- Intel Core Processor (Kaby 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

Abbreviation	Description
	Information Security)
CTR	Counter mode as defined in SP800-38A
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