Federal Office
for Information Security

Documentation and Analysis of the Linux Random Number Generator

Version: 3.6

Document history

Version	Date	Editor	Description
2.0	2018-03-21	Stephan Müller	Covering kernel 4.15 Re-running all tests of chapters 6 and following on 4.15
2.1	2018-04-11	Stephan Müller	Updating the test results for the reboot tests on bare metal with 50,000 reboot cycles
2.2	2018-04-11	Stephan Müller	Covering kernel 4.16
2.3	2018-06-18	Stephan Müller	Covering kernel 4.17
2.4	2018-08-24	Stephan Müller	Covering kernel 4.18
2.5	2018-11-12	Stephan Müller	Covering kernel 4.19 Updated seeding process ChaCha20 DRNG documented
2.6	2019-01-11	Stephan Müller	Covering kernel 4.20
3.0	2019-04-05	Stephan Müller	Covering kernel 5.0 Re-running all tests of chapters 6 and following on 5.0
3.1	2019-04-12	Stephan Müller	Adding results of reboot-tests for 5.0
3.2	2019-05-17	Stephan Müller	Addressing comments from BSI Covering kernel 5.1
3.3	2019-08-13	Stephan Müller	Covering kernel 5.2
3.4	2019-09-26	Stephan Müller	Covering kernel 5.3
3.5	2019-12-13	Stephan Müller	Covering kernel 5.4
3.6	2020-04-07	Stephan Müller	Covering kernel 5.5

Federal Office for Information Security
Post Box 20 03 63
D-53133 Bonn
Internet: https://www.bsi.bund.de
© Federal Office for Information Security 2020

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

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 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 the presentation of the Linux-RNG design, this study is intended to demonstrate that the design of the Linux-RNG complies with the NTG.1 requirements for RNGs specified by AIS 20/31 [AIS2031] and that they are fully met by /dev/random. Also, the study will provide the rationale why /dev/urandom 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 German 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 re-create these tests. Further, the tests are documented with a rationale for why the respective test is 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 all tests are applicable to other architectures as the code implementing the Linux-RNG is independent of the hardware architecture. The 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

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 holds true 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 an entropy pool is based on the SHA-1 hash. 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.
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 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	The processing of a bit-stream may operate only on a subset of it. To reference the

Term	Definition
Bits (MSB)	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 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 and /dev/urandom device files 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	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 a large number of 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 has white noise characteristics.
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.

Term	Definition
White Noise[1]	White noise is defined as noise having a power spectral density that is essentially independent from frequency. A discrete signal with a sequence of serially uncorrelated samples having zero mean and finite variance is defined as white noise.

Table 1: Terminology

2.2 General Architecture

NDRNGs can be found in many different 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 follow a general design pattern outlined in figure 1. This illustration has close relationships to [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 constructs are assessed. [AIS2031] is published and mandated by the German governmental body, BSI.

1 This definition is taken from ISO 389-3:2016.

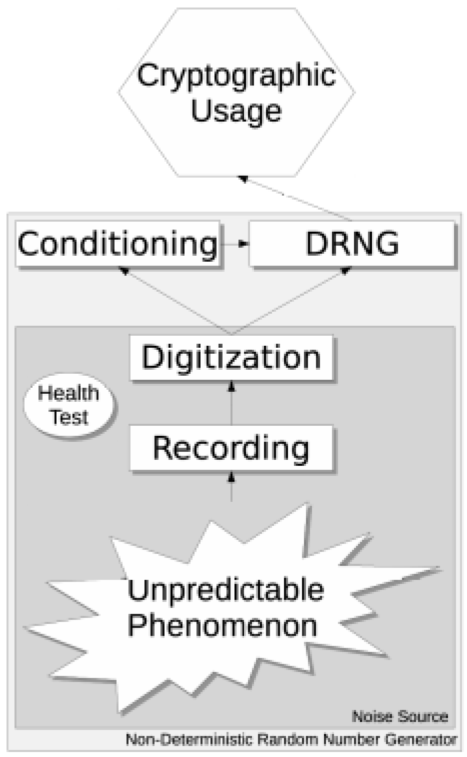

Figure 1: Non-deterministic random number generator architecture

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 assembly 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 DNRGs is fed by the noise source or conditioned noise source data. For example, user space cryptographic daemons using the OpenSSL cryptographic library obtain seed from /dev/urandom and its deterministic component to seed the OpenSSL's SSLeay MD 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 physical phenomenon that is unpredictable in nature, such as thermal noise or shot noise, metastability in bi-stable circuits, or even radioactive decay[2]; or

2 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 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 into white noise where the operation does not reduce the collected entropy. As already mentioned, the key value of those components is to increase the entropy per bit by performing a compression by using XOR. In addition, the conditioner may be used to hide skews in the raw data by applying a Von-Neumann unbias operation or using a 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 following a white noise pattern. 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 is provided by Theodore Ts'o 1994. The design of the Linux-RNG that is still visible today 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. 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.

3.2 Linux-RNG Architecture

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

The Linux-RNG is constructed of different entropy pools. 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. With the entropy pool called blocking_pool, random numbers for the user-space interface of /dev/random are generated. The ChaCha20 DRNG is maintained to produce random numbers for the user space interface of /dev/urandom 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 true 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 SHA-1 hash. 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 area holds true 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.

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 primary entropy pool that has a default size of 4096 bits. This primary entropy pool is called input_pool.

Upon request, this primary entropy pool feeds a secondary entropy pool called the blocking_pool or a DRNG based on the ChaCha20 stream cipher.

The man page random(4) discusses the two device files allowing the Linux-RNG to be accessed from user space:

- /dev/random - this file provides access to the blocking_pool
- /dev/urandom - this file provides access to the ChaCha20 DRNG

The difference between both files are explained in random(4) by referencing the blocking versus non-blocking behavior of both files. The concept of /dev/random is that it delivers only as many random bits as it has entropy in its entropy pool. As soon as the entropy is considered drained, the Linux kernel will put the calling process to sleep until entropy is added back into the blocking_pool. In contrast, the ChaCha20 DRNG will always produce random numbers. It acts as a DRNG that is regularly seeded with the available entropy.

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:

- When invoking `getrandom` where the flag field is zero, the system call accesses the ChaCha20 DRNG identically to /dev/urandom with one exception: whereas /dev/urandom always returns random data, `getrandom` blocks the caller until the ChaCha20 DRNG has been seeded initially with 128 bits of entropy. After reaching that threshold of 128 bits of entropy once, `getrandom` will operate non-blocking for the lifetime of the system.
- When invoking `getrandom` with a flag of `GRND_RANDOM`, it provides accesses to the blocking_pool and behaves exactly like the /dev/random device file.

When generating data from either the input_pool or the blocking_pool, the entire entropy content is hashed using SHA-1. Therefore, the SHA-1 hash operation is the output function used for both entropy pools. The calculated SHA-1 hash value is “folded” in half by XORing the 80 least significant bits with the 80 most significant bits of the SHA-1 hash value. The resulting 80 bits are the random number that is either handed to the caller (in case of the blocking_pool) or injected into the secondary DRNG (in case of the input_pool). To ensure backtracking resistance, the 80 bits are also mixed into the input_pool or blocking_pool, respectively. In case the caller requested more data, the process of generating the SHA-1 hash, folding it, providing it to the caller, and mixing it back into the entropy pool is repeated again until the requested number of bytes are generated or insufficient entropy is detected which leads to a block of the repetition operation. 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 function 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 backtracking resistance. 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 two different entropy pools and a ChaCha20 DRNG to collect, compress and maintain entropy. Figure 2 depicts the relationship between the entropy pools, the ChaCha20 DRNG and the entropy sources. The arrows in this figure explain the flow of information.

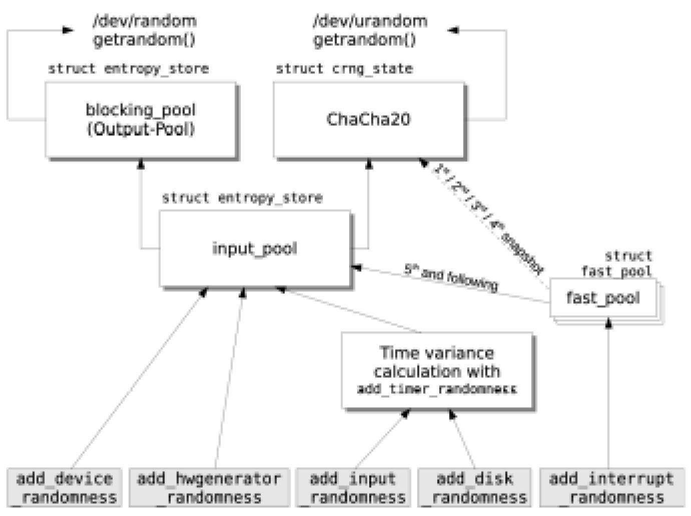

Figure 2: Relationship of entropy pools, ChaCha20-DRNG and entropy sources

The relationship of the entropy pools and the ChaCha20 DRNG to the noise sources and to each other is visible in figure 2:

- The input_pool is the primary entropy pool that collects and compresses the entropy from hardware events. That entropy pool has a default size of 4,096 bits[3]. The purpose of the input_pool is to collect entropy from the noise sources and provide it to the secondary random number generators discussed in the following two bullet points.
- The blocking_pool is fed with true random data from the input_pool. From user space, this entropy pool can be accessed using the /dev/random device file or the `getrandom` system call with the flag `GRND_RANDOM`. The entropy pool has a size of 1,024 bytes.
- The ChaCha20 DRNG obtains its entropic seed data from the input_pool as well and is accessible:
  - from user space via /dev/urandom or the `getrandom` system call without flags, 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 true 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.

3 Larger or smaller entropy pool sizes can be defined during compile time. The remainder of this document discusses the default size. Nonetheless, the discussion can be applied equally to a different entropy pool size.

- 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.

The listed entropy pools can each be viewed as independent random number generators. The input_pool together with the noise sources form a NDRNG. The blocking_pool can be viewed as a separate DRNG which is constantly being seeded by the input_pool. The ChaCha20 DRNG is the third random number generator in the Linux-RNG which is also seeded by the input_pool. The input_pool will exclusively deliver data to either the blocking_pool or the ChaCha20 DRNG only 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 pools can be considered as a DRNG when disregarding the noise sources. This section discusses the state maintenance of the deterministic operation of the entropy pools as well as the ChaCha20 DRNG.

The input_pool as well as the blocking_pool are both managed with the same code logic outlined in the following.

3.3.1 Entropy Pools

The random number generator implementation maintains a separate state for each of the two entropy pools of input_pool and blocking_pool. Both pools are governed by the same data structure which contains the following important information:

```
struct entropy_store {
        /* read-only data: */
        struct poolinfo *poolinfo;
        __u32 *pool;
...
```

```
        struct entropy_store *pull;
...
        unsigned add_ptr;
        unsigned input_rotate;
        int entropy_count;
        unsigned int initialized:1;
...
        __u8 *last_data;
};
```

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

- The variable `poolinfo` contains the definition of the polynomial used to stir new values into the entropy pool. Currently, two polynomials are used, one for the input_pool and another for the blocking_pool. See section 3.3.1.1 for the description on how the polynomials are applied. The following listing shows the definition of the polynomials – note that the first value gives the size of the pool in 32-bit words whereas the following five values define the polynomial that is applied when mixing new data into the pool. The first polynomial is used for the input_pool whereas the second polynomial is used for the blocking_pool:

```
static struct poolinfo {
...
        /* x^128 + x^104 + x^76 + x^51 +x^25 + x + 1 */
        { S(128),       104,    76,     51,     25,     1 },
...
        /* x^32 + x^26 + x^19 + x^14 + x^7 + x + 1 */
        { S(32),        26,     19,     14,     7,      1 },
```

- The variable `pool` contains the pointer to the static data block that holds the actual random pool. The following code shows the definition of the random pool for the input_pool, and the blocking_pool. When the discussed state data structure is initialized, the appropriate variable is registered with pool. The trailing keyword of __latent_entropy refers to the latent entropy GNU Compiler Collection (GCC) plugin which will be discussed in section 3.5.2.6.

```
#define INPUT_POOL_SHIFT        12
#define INPUT_POOL_WORDS        (1 << (INPUT_POOL_SHIFT-5))
#define OUTPUT_POOL_SHIFT       10
#define OUTPUT_POOL_WORDS       (1 << (OUTPUT_POOL_SHIFT-5))
...
static __u32 input_pool_data[INPUT_POOL_WORDS] __latent_entropy;
static __u32 blocking_pool_data[OUTPUT_POOL_WORDS] __latent_entropy;
```

- The variable `pull` holds the reference to the primary pool definition. This implies that this variable is NULL for the input_pool. The blocking_pool references the input_pool with this pointer.
- 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 variable `initialized` is used to identify whether the input_pool was provided with 128 bits of entropy at one point. This identifier has the following theoretical problem: depending on the assumed entropy in the data, zero to 11 bits[4] of entropy may be stirred into the entropy pool per injection. At the same time the entropy pool is seeded with the input containing entropy, callers may read random data from it. For an insufficiently seeded random number generator, this leads to a loss in entropy that is visualized with the following worst-case analogy: when an RNG receives one bit of entropy which is followed by a generation of one or more random numbers, the caller is required to guess one bit to break the state of the RNG. When one new bit of entropy is received after the attacker's gathering of random data, the new state of the RNG will again only have one bit of entropy and not two bits (the addition of the first and second seed). Hence, in a pathological case, the entropy pool may receive 128 bits of entropy in 128 separate seeding steps where an attacker can request random data from the entropy pool between each seeding operation. An attacker has to guess $2^1 \cdot 128$ different states and not 2^{128} – i.e. the amount of guessing to deduce the RNG state is reduced to a manageable level. However, as this variable is only used to decide whether the received entropy is diverted to the blocking_pool, because the input_pool is considered fully seeded, this illustrated issue is not considered a lead to an attack.
- The variable `last_state` holds the last random value extracted for the given pool. That value is used to implement the Federal Information Processing Standard (FIPS) 140-2 continuous test. This variable is initialized only when the kernel is in FIPS mode.

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

3.3.1.1 Entropy Pools State Transition Function

The Linux-RNG maintains two entropy pools: the input_pool and the blocking_pool. Although both have a different size, they are processed identically. The only difference is the use of a different polynomial for the LFSR discussed in this section. Therefore, this section applies to both entropy pools unless explicitly noted.

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 a LFSR. That data is mixed into the random pools using the `mix_pool_bytes` function. The function uses a reference to the 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 with `_mix_pool_bytes`. The 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.
 */
```

4 The explanation for the limit of 11 bits is given in section 3.6.

```
static void mix_pool_bytes_extract(struct entropy_store *r, const void
*in,
                                   int nbytes, __u8 out[64])
{
...
        tap1 = r->poolinfo->tap1;
        tap2 = r->poolinfo->tap2;
        tap3 = r->poolinfo->tap3;
        tap4 = r->poolinfo->tap4;
        tap5 = r->poolinfo->tap5;

        input_rotate = r->input_rotate;
        i = r->add_ptr;

        /* mix one byte at a time to simplify size handling and churn
faster */
        while (nbytes--) {
                w = rol32(*bytes++, input_rotate & 31);
                i = (i - 1) & wordmask;

                /* XOR in the various taps */
                w ^= r->pool[i];
                w ^= r->pool[(i + tap1) & wordmask];
                w ^= r->pool[(i + tap2) & wordmask];
                w ^= r->pool[(i + tap3) & wordmask];
                w ^= r->pool[(i + tap4) & wordmask];
                w ^= r->pool[(i + tap5) & wordmask];

                /* Mix the result back in with a twist */
                r->pool[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 += i ? 7 : 14;
        }


        r->input_rotate = input_rotate;
        r->add_ptr = i;
}
```

The mixing function operates on the entropy pool referenced by the input variable r which either points to the input_pool or the blocking_pool. The mixing function performs the following steps to mix the input data into the random pool state of `r->pool`:

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 logic applies to the blocking_pool polynomial definition equally. The figure assumes that the index of the pool member variable to be changed is 40 (i.e. i==40).

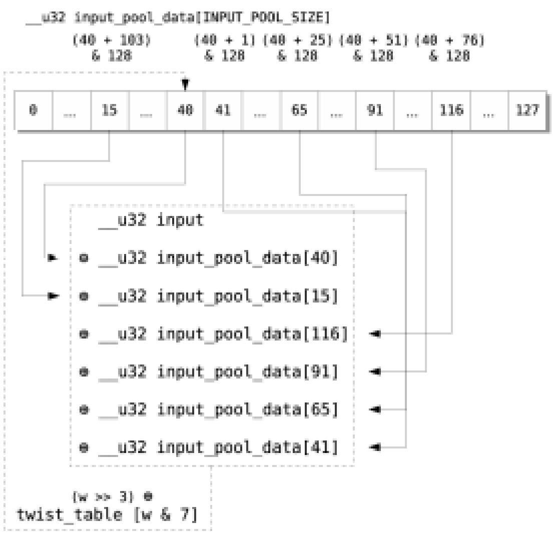
__u32 input_pool_data[INPUT_POOL_SIZE]
(40 + 103) & 128
(40 + 1) & 128
(40 + 25) & 128
(40 + 51) & 128
(40 + 76) & 128
0
...
15
...
40
41
...
65
...
91
...
116
...
127
__u32 input
⊕ __u32 input_pool_data[40]
⊕ __u32 input_pool_data[15]
⊕ __u32 input_pool_data[116]
⊕ __u32 input_pool_data[91]
⊕ __u32 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 all of 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 15 – 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. For both of the entropy pools of input_pool and blocking_pool, 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 respective 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.

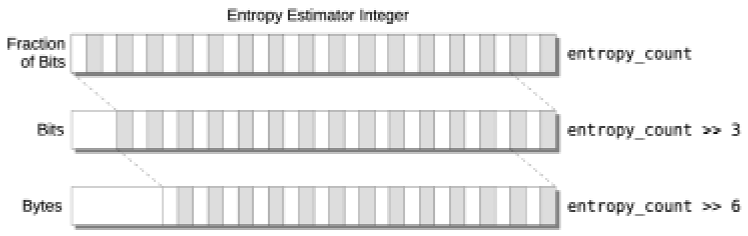

Figure 4: Entropy estimator integer value conversion

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 pools is implemented with the following functions:

- `extract_entropy_user` extracts data from the blocking_pool for a user space caller. This function may check for pending signals and may sleep – the check and the sleep operationare used to ensure that if a process is blocked, it still receives signals and it cannot trigger arbitrarily large kernel operations which would dominate other processes. Furthermore, it copies the gathered data to user space using the common kernel function of `copy_to_user`.

- `extract_entropy` 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, both functions implement the same steps by invoking the function `_extract_buf` which implements the output function for the entropy pools. 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(struct entropy_store *r, size_t nbytes, int min,
                      int reserved)
{
...
        entropy_count = orig = ACCESS_ONCE(r->entropy_count);
        ibytes = nbytes;
        /* If limited, never pull more than available */
        have_bytes = entropy_count >> (ENTROPY_SHIFT + 3);

        if ((have_bytes -= reserved) < 0)
                have_bytes = 0;
        ibytes = min_t(size_t, ibytes, have_bytes);
...
        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 SHA-1 hash is calculated of the entire entropy pool. But before the SHA-1 hash is calculated, the SHA-1 hash is initialized. The memory used for the SHA-1 operation is initialized with the SHA-1 constants defined in [FIPS180-4]. However, if the CPU provides a hardware random number generator, the memory holding the SHA-1 constants are overwritten with the output of that random number generator. This means that in this case, the official SHA-1 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 SHA-1.

2 The entire entropy pool is processed by the SHA-1 transformation which implies that the SHA-1 hash is calculated across the entire entropy pool content.

3 The resulting SHA-1 hash is mixed into the entropy pool with the process discussed in section 3.3.1.1.

4 Fold the calculated SHA-1 hash in half by XORing the first 4 bytes with the fourth 4 bytes, the second 4 bytes with the fifth 4 bytes. The third 4 byte component is XORed with itself by taking the first 2 bytes and XORing them with the second 2 bytes. The folding implies that the original SHA-1 hash of length 160 bits is reduced to 80 bits.

5 Return the 10 bytes calculated in step (4).

The folding of the SHA-1 hash is implemented with the following code snippet:

```
static void extract_buf(struct entropy_store *r, __u8 *out)
{
...
        /*
         * In case the hash function has some recognizable output
         * pattern, we fold it in half. Thus, we always feed back
         * twice as much data as we output.
         */
        hash[0] ^= hash[3];
        hash[1] ^= hash[4];
        hash[2] ^= rol32(hash[2], 16);
...
```

Finally, either the 10 bytes of the SHA-1 calculation are returned, or the requested number of bytes if that number is less than 10 bytes.

The function `extract_buf` is intended to be invoked repeatedly to generate as many random bytes as requested by the caller.

3.3.1.4 Initialization

When the Linux-RNG is initialized, all random pools 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 each of the random pools.

```
static void init_std_data(struct entropy_store *r)
{
...
        ktime_t now = ktime_get_real();
...
        r->last_pulled = jiffies;
        mix_pool_bytes(r, &now, sizeof(now));
        for (i = r->poolinfo->poolbytes; 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(r, 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. 4.9.0 for the kernel version of 4.9.0 – 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;
        spinlock_t      lock;
};
```

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.

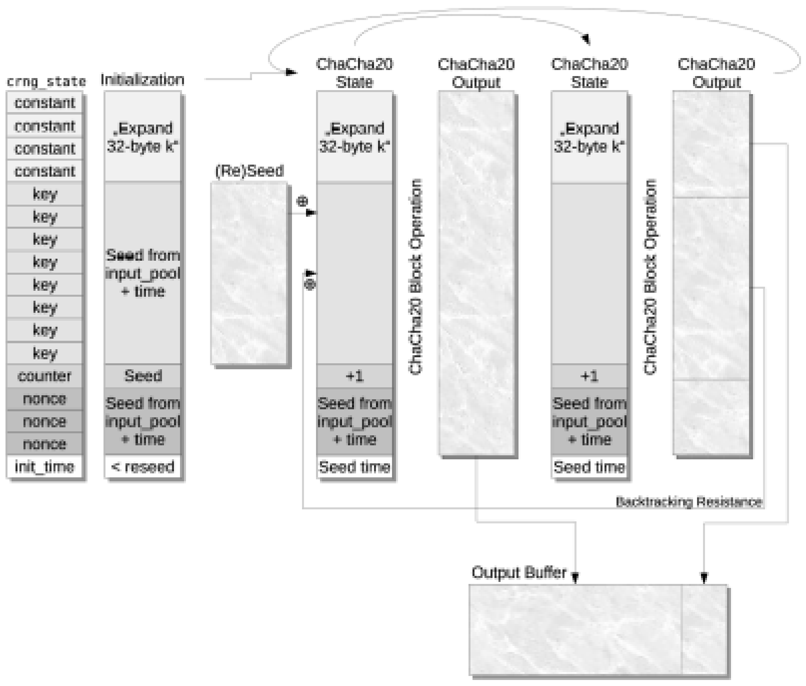

Figure 5: ChaCha20 DRNG state and state transition

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 key part, the counter, and the nonce are XORed with the output of the CPU random number generator if one is present. If it is not present, one high-resolution time stamp obtained with the kernel function `random_get_entropy` word is XORed with the key part (note, figure 5 illustrates the case

where the CPU random number generator is not present and the time stamp is obtained). On Intel x86 systems, this function invokes the `RDTSC` processor instruction.

- 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 backtracking support. 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. Therefore, the entropy used to seed the ChaCha20 DRNG is distributed evenly over the generated random numbers 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 that is accessible via /dev/urandom. 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 (/dev/urandom, or the `getrandom` system call from user space, or `get_random_bytes` from kernel space), 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.

3.3.2.3 ChaCha20: Initially or Fully Seeded

The Linux-RNG maintains the concept of an initially[5] or fully[6] 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.
- 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 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.
- 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 {
```

5 The kernel prints the log message "random: fast init done" in this case.
6 The kernel prints the log message "random: crng init done".

```
        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 entropy estimator is larger than a given threshold, 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 entropy is too low. The threshold can be modified at runtime by writing a positive integer value into /proc/sys/kernel/random/read_wakeup_threshold. 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. The entropy pool blocking_pool is drained as outlined above using the `extract_entropy_user` function. If that function returns that zero bytes were obtained – which is determined by the calculation of the `account` function – the calling process is put to sleep until the entropy estimator for the input_pool reaches 64 bits. The following code demonstrates the discussed function.

```
static ssize_t
_random_read(struct file *file, char __user *buf, size_t nbytes, loff_t
*ppos)
{
...
        nbytes = min_t(size_t, nbytes, SEC_XFER_SIZE);
...
                n = extract_entropy_user(&blocking_pool, buf, nbytes);
...
                wait_event_interruptible(random_read_wait,
                        ENTROPY_BITS(&input_pool) >=
                        random_read_wakeup_bits);
...
```

/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, but the user-provided data is mixed into the entropy pool.

```
static int
write_pool(struct entropy_store *r, const char __user *buffer, size_t
count)
{
...
        __u32 buf[16];
...

        while (count > 0) {
                bytes = min(count, sizeof(buf));
...
                count -= bytes;
...
                mix_pool_bytes(r, 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 /dev/random

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 all entropy pools and reset their contents as discussed in section 3.3.1.4. This IOCTL is restricted to the capability of `CAP_SYS_ADMIN`.
- `RNDCLEARPOOL`: See `RNDZAPENTCNT`.

- RNDRESEEDING: 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[7]:

- GRND_RANDOM – If this bit is set, then random bytes are drawn from the /dev/random pool instead of the /dev/urandom pool. The /dev/random pool is limited based on the entropy that can be obtained from environmental noise. If the number of available bytes in /dev/random is less than requested by the caller of the system call, the call returns just the available random bytes. If no random bytes are available, the behavior depends on the presence of GRND_NONBLOCK.
- GRND_NONBLOCK – By default, when reading from /dev/random, getrandom blocks if no random bytes are available, and when reading from /dev/urandom, it blocks if the entropy pool has not yet been initialized. 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.

The main difference between /dev/urandom and getrandom is that the system call 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 behave identically to reading from /dev/urandom.

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 pools. 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 a 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

7 The following listing is taken from the getrandom(2) man page.

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[8]. The difference, however, is that an in-kernel caller cannot 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.

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 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. Deactivate the other noise source.
    c. Set the reseed threshold to the regular value defined by 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.

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.

8 Note that getrandom blocks until the primary ChaCha20 DRNG is either initially or fully seeded.

- /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/read_wakeup_threshold: threshold that when reached by the entropy_avail value triggers a wakeup of readers. When waking up sleeping processes, the process will find some entropy present in the Linux-RNG that can be picked up from /dev/random.
- /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;
        unsigned dont_count_entropy:1;
};

#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 variable `dont_count_entropy` is currently unused. 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 three classes of entropy sources. For each 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[9]. 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)
                disk->random = state;
}
```

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

```
struct fast_pool {
        __u32           pool[4];
        unsigned long   last;
        unsigned short  reg_idx;
        unsigned char   count;
};
...
```

9 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.

```
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[10].

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:

10 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.

```
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.

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 pools, 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.

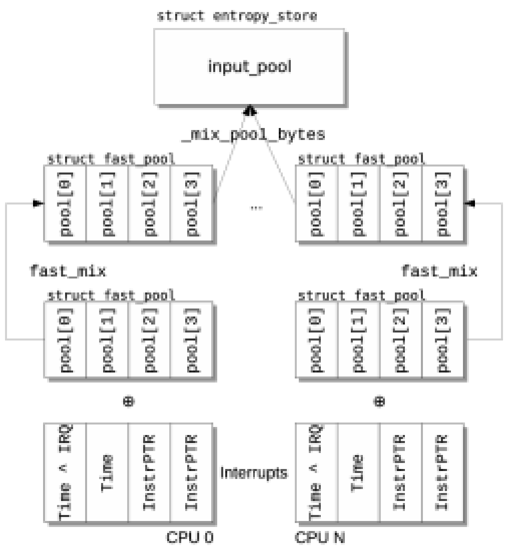

Figure 6: Processing of interrupts by fast_pools and connection to input_pool

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 3.5.2.7. However, one key difference to section 3.5.2.7 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, int irq_flags)
{
...
        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, int irq_flags)
{
...
        if ((fast_pool->count < 64) &&
            !time_after(now, fast_pool->last + HZ))
                return;

        r = &input_pool;
```

```
...
        fast_pool->last = now;
        __mix_pool_bytes(r, &fast_pool->pool, sizeof(fast_pool->pool));
...
        /*
         * If we have architectural seed generator, produce a seed and
         * add it to the pool.  For the sake of paranoia don't let the
         * architectural seed generator dominate the input from the
         * interrupt noise.
         */
        if (arch_get_random_seed_long(&seed)) {
                __mix_pool_bytes(r, &seed, sizeof(seed));
                credit = 1;
        }
...

        fast_pool->count = 0;

        /* award one bit for the contents of the fast pool */
        credit_entropy_bits(r, credit + 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 If a CPU hardware random number generator is present[11], inject the output of that RNG. On 32-bit systems, `RDSEED` fills a variable with a size of 32 bit. On 64-bit systems, the variable's size filled with `RDSEED` output is 64 bit.

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

6 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.

In addition, if `RDSEED` is present, the

32 bits or 64 bits filled by RDSEED

11 At the time of writing, this is only present with the Intel `RDSEED` operation.

are mixed into the input_pool.

That means that the 64 interrupts[12] 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.

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.

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

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.

In Linux kernels before version 5.0, the `add_disk_randomness` function was called from the central code path in the block device layer by the function `blk_update_bidi_request`. With version 5.0, this call was removed due to some structuring changes in the generic block layer.

The conditions for the call have remained the same, and are described as follows:

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, unsigned int bidi_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 starting with 3.18 unsets the flag for disks that are known to not have rotational disks. Such unsetting of the flag is done for:

- 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 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. This implies that the blocking_pool does not benefit from the data obtained via `add_device_randomness` during boot. Such approach is considered appropriate because the blocking_pool will only generate data if it is seeded with fresh data from noise sources.

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[13], 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.

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

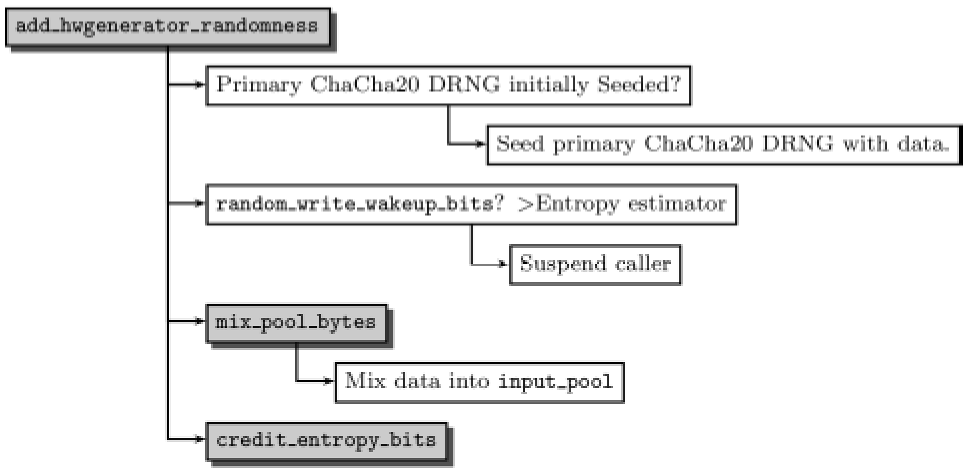
add_hwgenerator_randomness
Primary ChaCha20 DRNG initially Seeded?
Seed primary ChaCha20 DRNG with data.
random_write_wakeup_bits? >Entropy estimator
Suspend caller
mix_pool_bytes
Mix data into input_pool
credit_entropy_bits

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 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 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 Latent Entropy GCC Plugin

Starting with kernel 4.9, a GNU Compiler Collection (GCC) plugin named “latent_entropy” is added. 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 variablemay 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 kernel entropy 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.

3.5.2.7 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 blocking_pool or 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;
        r = &input_pool;
        mix_pool_bytes(r, &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[14]

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. One exception to this rule is the injection of data from the interrupt noise source: `add_interrupt_randomness` applies one or two bits of entropy per fast_pool to input_pool transferal as specified in section 3.5.2.2.

The Linux-RNG estimates the entropy of a hardware event and modifies the entropy estimator of the input_pool accordingly. The entropy estimator of the blocking_pool is simply modified when data from the input_pool is mixed in or when data is read. 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.

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.
...
#define ENTROPY_SHIFT 3
#define ENTROPY_BITS(r) ((r)->entropy_count >> 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 timestamps 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`:

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

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

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

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

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

2 Calculate the $\log_2$ of the minimum delta as an integer operation – i.e. value after the decimal point is discarded. The implementation of the logarithmic operation is achieved by dividing the minimum delta value by 2 and obtain the 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(struct entropy_store *r, 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 poo)l 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.

As this approach applies to both the input_pool and blocking_pool entropy estimator, the following conservatism in the entropy estimate is evident: to outline the issue, the extreme case is applied that the input_pool entropy estimator is close to its ceiling. In this case, the heuristic entropy estimate awarded to the just received event will increase the entropy estimator of the input_pool by a tiny fraction only. When pulling data out of the input_pool to seed the blocking_pool, the input_pool entropy estimator is decreased by the read amount of bits. As the blocking_pool, however, is subject to the same entropy estimate addition handling, the blocking_pool entropy estimator is increased by a fraction of the bits obtained from the input_pool. Ultimately that implies that the heuristic entropy estimate awarded to one hardware event is reduced during insertion of the event data into the input_pool and implicitly further reduced when transferring it from the input_pool to the blocking_pool.

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

3.6.1 Storing of “Superfluous” Entropy

The logic discussed for changing the entropy estimator shows that in case the input_pool is considered to be full of entropy, the hardware events only stir the pool but do not contribute to the entropy estimation.

The following approach is used to fill the blocking_pool in case of a filled input_pool. This means that the output pool is also used as a store of entropy and the entirety of the Linux-RNG consisting of the different entropy pools can collect more entropy than the size of the input_pool.

```
static void credit_entropy_bits(struct entropy_store *r, int nbits)
{
...
        if (r == &input_pool) {
                int entropy_bits = entropy_count >> ENTROPY_SHIFT;
                struct entropy_store *other = &blocking_pool;
...
                /* If the input pool is getting full, and the blocking
                 * pool has room, send some entropy to the blocking
                 * pool.
                 */
                if (!work_pending(&other->push_work) &&
                    (ENTROPY_BITS(r) > 6 * r->poolinfo->poolbytes) &&
                    (ENTROPY_BITS(other) <= 6 * other->poolinfo-
>poolbytes))
                        schedule_work(&other->push_work);
...
```

If the entropy estimation of the input_pool rises above a threshold and the entropy estimation of the destination entropy pool is below a certain threshold, a kernel work queue is triggered which transmits entropy asynchronously. The threshold for the input_pool is the wakeup threshold for the reading/writing as already discussed. The threshold for the destination pools is 75% of the size of the entropy pool.

The blocking_pool has a work queue which triggers the standard transfer mechanism from the input_pool:

```
/*
 * Used as a workqueue function so that when the input pool is getting
 * full, we can "spill over" some entropy to the output pools.  That
 * way the output pools can store some of the excess entropy instead
 * of letting it go to waste.
 */
static void push_to_pool(struct work_struct *work)
{
...
        _xfer_secondary_pool(r, random_read_wakeup_bits/8);
...
```

The code uses the same function for transferring entropy between the entropy pools as discussed in section 3.3.1.3.

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.

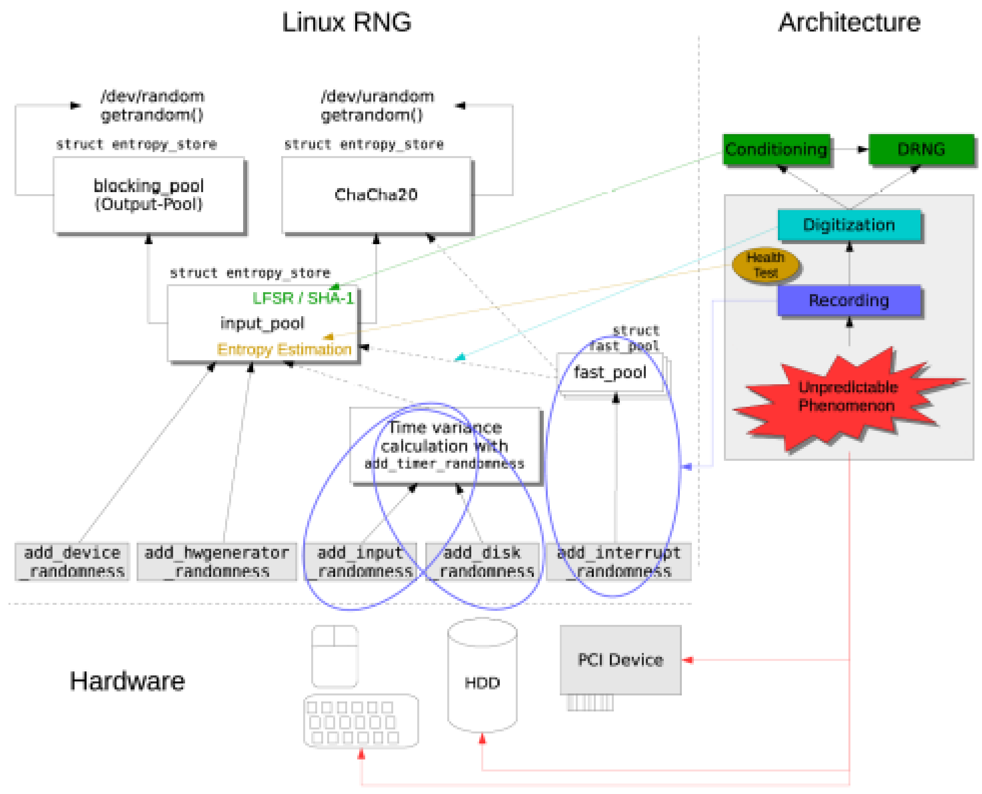

Figure 8: Linux-RNG architecture comared with the generic architecture

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. 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.

- 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.
- 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 time stamp of each event. Now, when this entropy estimation is zero, the input_pool does not provide any data to the callers of the ChaCha20 DRNG or the blocking_pool. 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.
- The DRNG part is implemented by the output function to read the input_pool: the output function calculates a SHA-1 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 blocking_pool can be considered as a standalone, independent DRNG which is seeded from the NDRNG of the input_pool.

Similarly, the ChaCha20 DRNG is a standalone, independent DRNG seeded from the input_pool. This characterization is clearly evident when considering the author's implementation of the ChaCha20 DRNG as an independent user space implementation available at http://www.chronox.de/chacha20_drng.html as well as an independent kernel space implementation provided at http://www.chronox.de/lrng.html (see the file lrng_standalone.c).

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 changes introduced with the addition of ChaCha20, specifically the seeding of the ChaCha20 DRNG with the first filled fast_pool instances alleviates the issues discussed below. 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
```

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 either /dev/random or /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 either /dev/random or /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.

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.

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 int arch_get_random_long(unsigned long *v)
{
        return 0;
}
static inline int arch_get_random_int(unsigned int *v)
{
        return 0;
}
...
static inline bool arch_get_random_seed_long(unsigned long *v)
{
        return 0;
}
static inline bool arch_get_random_seed_int(unsigned int *v)
{
        return 0;
}
...
#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 0 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[15].

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

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.

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:

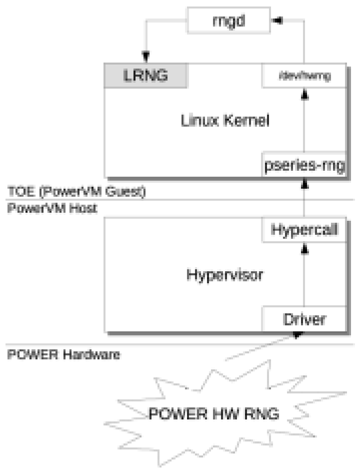

Figure 9: Flow of random numbers in a PowerVM environment

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, performs statistical tests as outlined in rngd(8) which includes a monobit test and a poker test.

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.

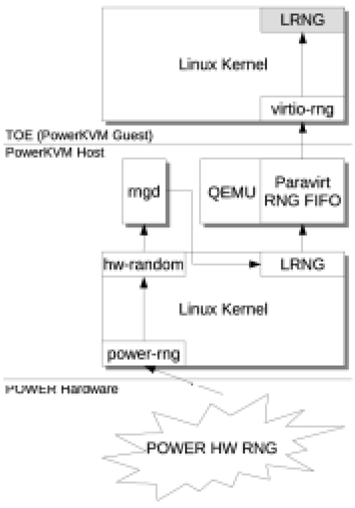

Figure 10: Flow of random numbers in a PowerKVM environment

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 and performs statistical tests as outlined in the rngd man page which includes a monobit test and a poker test.

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[16].

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. These service functions are briefly explained below.

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

The functions `get_random_int` and `get_random_long` are provided to support a lean but not cryptographically secure RNG. It obtains an unsigned integer value from one of the following sources:

- If the architecture implements a hardware random number generator (currently only Intel's `RDRAND` is registered here), it obtains the data from that RNG and returns.
- Otherwise the following mechanism is used:
  - obtain a per-CPU global value that is used to store the memory for the following operations,
  - add the value of the calling task's PID, the Jiffies and the value of the current high-precision time stamp (e.g. `RDTSC`) to the first 32 bits of that value,
  - calculate an MD5 hash of this value,
  - store the MD5 hash as the new value in the global per-CPU variable, and
  - return the first 32 bits of the MD5 hash to the caller.

The function `get_random_bytes_arch` allows callers to obtain random numbers from the CPU hardware RNG. If the current CPU does not provide such support, the service function transparently falls back to calling the Linux-RNG API function `get_random_bytes`.

`randomize_page` is a function which fills memory with random values obtained from the aforementioned function `get_random_long` where the page pointer is adjusted to a page boundary.

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 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 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, those user space DRNGs either do not reseed automatically at all (like it is the case with OpenSSL's SSLeay DRNG) 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 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 Fedora 25 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 blocking_pool or 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.

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. Furthermore, as outlined by Lacharme et al., the kernel maintains a min-entropy content in the input_pool which is reserved for /dev/random and cannot be obtained by any operation with the ChaCha20 DRNG.

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.

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.

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 behavior of /dev/random complies with all requirements of NTG.1 on contemporary x86 hardware having a high-resolution time stamp. This analysis does not provide a statement for other hardware architectures, as measurements are only applied on x86 systems. Since the entropy estimation partially depends on architecture-specific high-resolution time stamps, the results and conclusions of this chapter cannot be readily applied to other hardware architectures. At least the requirement of NTG.1.6 for empirical measurements is not covered on other architectures.
- The implementation of the ChaCha20 DRNG feeding /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 5.2.1. 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.3defines procedural requirements only.

To claim NTG.1 compliance, the ATH9K device driver provided for Atheros Chips based Wi-Fi cards must be compiled with the option `CONFIG_ATH9K_HWRNG` set to “N” during the kernel compilation. When this option is enabled, the hardware random number generator with its noise source invokes the Linux-RNG `add_hwgenerator_randomness` function to provide data and update the entropy estimator. At the time of writing, the noise source is not independently analyzed nor is its design publicly available. Thus, it is unclear whether the NTG.1 properties are met by the Linux-RNG when the Atheros random number generator is present and activated.

5.1 /dev/random: NTG.1

This section analyzes the NTG.1 properties implemented by /dev/random.

In addition, when considering the input_pool as a separate NDRNG, the analysis applies to the input_pool as well. This is due to the fact that the input_pool has the very same characteristics as observed for /dev/random with respect to the NTG.1 requirements. The following characteristics relied upon by the NTG.1 discussion below are identical for both:

- Use of an LFSR with a primitive and irreducible polynomial as state transition function. This finding is discussed in detail in section 7.1
- Use of a SHA-1 based output function
- Blocking behavior, i.e. prevention of the generation of random numbers if the entropy estimation is too low
- The entropy estimation maintenance is identical for both. Note that the entropy estimation maintenance applied by the Linux-RNG reduces the heuristic entropy estimation even further if the amount of entropy assumed to be present in the entropy pool is already high. The following discussion disregards this detail which makes the Linux-RNG entropy estimation maintenance even more conservative than already outlined below.

The following sections discuss the blocking_pool. Considering the identical properties of the blocking_pool and input_pool, all statements are equally applicable to the input_pool. Thus, the input_pool by itself can be considered to be an NTG.1 compliant random number generator as well.

5.1.1 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 (and thus implicitly enforced for the blocking_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. If the Intel CPU instruction of RDSEED is present, two bits of entropy are awarded to the 64 or more interrupts. 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 /dev/random 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.

5.1.2 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 blocking_pool generating random numbers for /dev/random. For every bit of random data generated for /dev/random, 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 when the RDSEED instruction is present is 2 bits of entropy per 64 interrupts, i.e. 1/32th bit of entropy per interrupt[17]. The analysis of the Linux-RNG has shown that the entropy content of each high-resolution time stamp provides more than 2 bits (SP800-90B min-entropy with 4-bit window), more than 4 bits (SP800-90B min-entropy with an 8-bit window), 19.2 bits (Shannon entropy), or 12.1 bits (AIS 20/31 min-entropy). 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 0.21 bits of entropy per block device event. The measurement of the raw noise shows the following entropy content per event: more than 2.7 bits (SP800-90B min-entropy with a 4-bit window), more than 5.5 bits (SP800-90B min-entropy with an 8-bit window), 17.7 bits (Shannon entropy), or 13.5 bits (AIS 20/31 min-entropy). Again, the heuristic entropy value applied by the Linux-RNG for each event is significantly less than the measured entropy.
- HID: The Linux-RNG entropy heuristic awards on average 1.29 bits of entropy per HID event. The raw noise measurement shows the following entropy content per HID event: more than 1.8 bits (SP800-90B min-entropy with a 4-bit window), more than 4 bits (SP800-90B min-entropy with an 8-bit window), 15.6 bits (Shannon entropy), or 10.1 bits (AIS 20/31 min-entropy). Just as for the block device and interrupt noise sources, the Linux-RNG heuristic therefore underestimates the available entropy.

As all heuristic entropy values applied by the Linux-RNG to the entropy estimator steering the blocking behavior of /dev/random 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 by /dev/random is backed by at least one bit of fresh entropy obtained from the noise sources.

Therefore, the implementation of /dev/random complies with the requirements set forth by NTG.1.2.

5.1.3 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

17 This is the worst case value. It may be the case that more interrupts are collected that are collectively awarded 2 bits of entropy.

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 SHA-1 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

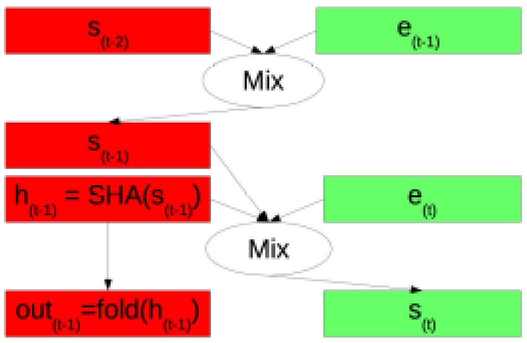

Figure 11: Relationship between Linux-RNG processing and attacker-known and unknown data

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 SHA-1 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 SHA-1 and the LSFR operation:

- Both functions are non-invertible.
- For both functions it is hard to obtain a pre-image for a given output. For SHA-1 the effort is related to the collision resistance which is considered extremely high. For the LFSR, the determination of a pre-image of the entropy pool with a size of 1024 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 for /dev/random complies 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 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.

- 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 hardly any practical relevance. In practice an attacker will make use of acquired privileges more effectively than monitoring the Linux-RNG and deducing previous random numbers.

5.1.4 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 blocking_pool entropy pool has a size of 1024 bits. In a worst-case scenario when the blocking_pool has just been fully seeded with fresh entropy it can generate 1024 bits of entropy before it reseeds with newly fresh entropy.

To generate a bit string of 128 bits, /dev/random performs two SHA-1 operations which produce 80 bits each after the folding operation. Both values will be concatenated where the trailing 32 bits will be discarded.

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

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

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

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

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

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

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

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

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

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

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

Stating the obtained results differently, k > 2^{55} bit strings of size 128 bits can be generated where no collisions occur with a probability of P > 1 - 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 /dev/random is resistant for an attacker with a high the attack potential.

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

- The blocking_pool has a large size of 1024 bits. The pre-image for each SHA-1 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 1024 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 input_pool which again will change the entropy pool content significantly.[18] 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.
- SHA-1 was subject to extensive assessments that have shown that generated SHA-1 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 /dev/random fulfills the requirements of NTG.1.4.

Side note: The rationale also applies to the extraction of data from the input_pool, since the procedure for the generation of data is identical for the blocking_pool and the input_pool.

5.1.5 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 test procedure A [assignment: additional test suites].”

The execution of the Test Procedure A defined in [AIS2031] on the output of /dev/random is documented in section 8.1.

The requirement for additional statistical tests is covered with section 8.1 which refers to other statistical methods. 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[19], the Chi-Squared test and the test of compressing the generated data with gzip, bzip2, xz and lzma”.

5.1.6 NTG.1.6

The requirement of NTG.1.6 is defined as:

18 The absolute min-entropy that is required for a transport is 1 byte.
19 Test suite is provided at http://www.phy.duke.edu/~rgb/General/dieharder.php

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

As outlined in section 5.1.2 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 blocking_pool that drives the blocking behavior of the /dev/random output function. This mechanism allows only to draw as many bits from /dev/random as there is entropy present in the blocking_pool. The entropy estimation is increased by the heuristic entropy estimation awarded to the data mixed into the blocking_pool and is decreased by the number of bits obtained via /dev/random. With this concept, the Linux-RNG implies that each data bit generated for /dev/random out of the blocking_pool is backed by one bit of entropy obtained from the noise sources.

The blocking behavior of /dev/random implies that only as much entropy can be extracted from the blocking_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[20]:

- Interrupts: A set of at least 64 interrupts is awarded 2 bits of entropy by the heuristic of the Linux-RNG. This means that after obtaining 64 interrupts and mixing it into the LFSR, 2 bits are allowed to be read from the entropy pool via SHA-1 before the blocking behavior stops additional read operations. The measured entropy for one interrupt using the Shannon entropy is given in section 6.2.1 with 19.2 bits. That means, with the collection of 64 interrupts, the Linux-RNG has collected 64 * 19.2 = 1228,8 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 2 bits 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 0.21 bits of entropy by the Linux-RNG heuristic. That means that after around 5 block device events that have been received on average, the blocking_pool allows the generation of one random bit. The Shannon entropy value measured for block device events is 17.7 bits of entropy per event as listed in section 6.2.2. Thus for 5 block device events, the Linux-RNG has collected 5 * 17.7 = 88.5 bits of entropy. This implies that for each generated random bit, the Linux-RNG has obtained 88.5 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 1.29 bits of entropy. This means that after 4 HID events around 4 * 1.29 ≈ 5 bits of random data can be produced before the blocking behavior is enforced. Using the measured Shannon entropy value of 15.6 bits per HID event, it implies that for generating 5 bits of random data, four blocks with 15.6 bits of Shannon entropy each having in total 62.4 bits of Shannon entropy are obtained by the Linux-RNG from the HID noise source.

20 The discussion refers to bit-wise reading from the entropy pool. The Linux-RNG implementation requires a minimum size of one byte and a minimum transfer size of data from the input_pool to the blocking_pool of 64 bits. 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.

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 1024 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 1024 bits, because the Linux-RNG applies its conservative heuristic entropy estimation. Though the maximum amount of entropy that the entropy pool can maintain is equal to its size, i.e. 1024 bits. This means that once the Linux-RNG considers its entropy completely filled, 1024 bits of Shannon entropy are present in that pool.

The Linux-RNG output function allows the generation of 1024 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 SHA-1-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 /dev/random meets the requirement of NTG.1.6.

5.1.7 NTG.1 Properties on Different Environments

The purpose of this study is to demonstrate that the Linux-RNG with its /dev/random device file 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.

5.2 ChaCha20 DRNG: DRG.3

In addition to the /dev/random analysis, the ChaCha20 DRNG backing /dev/urandom, the `get_random_bytes` API and the `getrandom` system call is analyzed to comply with DRG.3.

5.2.1 DRG.3.1

The requirement of DRG.3.1 is defined as:

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

The seeding of the ChaCha20 DRNG allows the use of CPU-based noise sources which contribute entropy. These noise source, however, cannot be analyzed or tested and thus must be assumed to have zero bits of entropy for this discussion. This implies that to ensure these noise sources do not contribute entropy, 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 set to "NO" (i.e. disabled).

The ChaCha20 DRNG is seeded by reading from the input_pool. As outlined in section 5.1, the input_pool complies with the requirements from NTG.1 and is therefore considered an NTG.1 random number generator.

This implies that the first selection can be instantiated with the selection "using an NPTRNG of class NTG.1".

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 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 measured for the interrupt events in sections 6.3.1 and 6.3.2, 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. 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.

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, and
- 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.

5.2.2 DRG.3.2

The requirement of DRG.3.2 is defined as:

"The RNG provides forward secrecy."

The forward secrecy is guaranteed by the ChaCha20 DRNG as follows: The ChaCha20 DRNG maintains an internal state which holds a key that is unknown to the caller. Furthermore, the ChaCha20 DRNG increments the counter by one after each generated block. Assuming that the ChaCha20 block function is irreversible for an observer that does not have access to the ChaCha20 state with its key, a caller cannot deduce subsequent random numbers from his obtained random number.

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

These properties therefore guarantee forward secrecy.

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

5.2.3 DRG.3.3

The requirement of DRG.3.3 is defined as:

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

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.

5.2.4 DRG.3.4

The requirement of DRG.3.4 is defined as:

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

Considering that the ChaCha20 block operation has similar characteristics to SHA-1 that are outlined in section 5.1.4 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 blocking_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 5.1.4 is that the blocking_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 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 5.1.4 is still applicable for the deterministic operation phase of the ChaCha20 DRNG.

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

5.2.5 DRG.3.5

The requirement of DRG.3.5 is defined as:

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

Section 8.2 provides a rationale for the execution of the Test Procedure A defined in [AIS2031].

Additional statistical tests are applied as covered in section 8.2, 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 8.2, 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 an NTG.1 and 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.

The 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.

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 an attacker is able to deduct 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 that 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 3.5.2.7 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 3.5.2.7. 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 following command executed without privilege requirements turns into a perfect key logger[21]. 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.

21 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
```

- Like for `add_disk_randomness`, `add_input_randomness` invokes `add_timer_randomess` 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.

6.2 Min-Entropy 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 the test provided in the test directory linux-entropy-sp80090b. This test uses several SystemTap scripts which are enumerated in the following:

- HID measurement: to measure the high-resolution time stamp of HID events, the SystemTap script recording/raw_entropy_hid.stp instruments add_timer_randomness to read out the high-resolution time stamp from the `sample` data structure (see section 3.5.2.7 for details about this data structure). In conjunction, a second SystemTap script record/entropy_per_event_hid.stp is used to record the entropy estimation applied by the Linux-RNG to the HID events. To allow other reviewers to assess the quality of the event values and Jiffies values, they are recorded but disregarded in the subsequent assessments.
- Block device measurement: the SystemTap scripts of record/raw_entropy_disk.stp and record/entropy_per_event_disk.stp are used to record the same data for block devices as outlined for HID devices above. Again, the event values and the Jiffies values are recorded for third-party verification. However, they are again not considered in the analysis below.
- Interrupt measurement: the SystemTap script raw_entropy_irq.stp instruments `add_interrupt_randomness`. It obtains the high-resolution time stamp for this interrupt. There is no SystemTap script measuring the entropy estimation applied to interrupts as the Linux-RNG applies a fixed estimate of one bit (in case of Intel x86 systems with `RDRAND` it is two bits) per injection of a fast_pool content into the input_pool.

The recorded data set is simply a set of 32 bit integer values holding the high-resolution time stamps for each recorded interrupt. To make testing easier and more repeatable, the script recording/gendata.sh is provided which invokes the SystemTap scripts appropriately. That script 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 content 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 values are calculated defined by [SP800-90B]:

- Most Common Value Estimate
- Collision Estimate
- Markov Estimate
- Compression Estimate
- t-Typle 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. To allow processing the time stamps with the aforementioned min-entropy estimation calculations, the application validation/extractlsb.c obtains the 4 least significant bits of the time stamp and concatenates all 4 LSB of all time stamps into a bit stream. This means that the input data width is now 4 bits instead of 32 bits. The calculation of the min-entropy estimations using 4 bits instead of 32 bits is considered to support the conservative assessment of this study. In addition, the mentioned application also extracts the 8 LSB[22] of each time stamp and concatenates them into a bit-stream. This allows the calculation of the min-entropy estimation of the input data with 6 bit width. The following tables therefore provide the entropy estimation for 4 bit and 8 bit input data widths. The tool used to calculate the SP800-90B min-entropy values is available at NIST GitHub repository.

For comparison, the min-entropy and the Shannon entropy defined by [AIS2031] are calculated as well. The used formulas are provided in section 2.3.2 [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 Minimum 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.

6.2.1 Interrupt Noise Source Min-Entropy Estimates

The collection of data for interrupts was conducted twice: once with a normal use case and once with a worst-case. In the normal use case the test environment was made to resemble regular usage where Internet searches and regular office duties were performed. The worst-case covered the test system in a virtual environment where the host system 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.

22 The reason for selecting 8 LSB is to support the Markov min-entropy value calculation which can only be calculated for small data blocks.

Entropy Estimate	4 Bits Width	8 Bits Width
Most Common Value Estimate	3.9684	7.9656
Collision Estimate	3.7452	7.576
Markov Estimate	3.9796	7.9648
Compression Estimate	3.4848	7.1328
t-Typle Estimate	3.7268	7.436
LRS Estimate	3.8868	7.9712
Multi Most Common in Window Prediction Estimate	3.9604	7.9736
Lag Prediction Estimate	3.9884	7.9896
MultiMMC Prediction Estimate	3.9708	7.9736
LZ78Y Prediction Estimate	3.97	7.968

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

The associated Shannon entropy value is 17.147 bits per interrupt event. The min-entropy value according to [AIS2031] is 14.472 bits per interrupt event.

The normal use case returned the following data.

Entropy Estimate	4 Bits Width	8 Bits Width
Most Common Value Estimate	3.9792	7.9784
Collision Estimate	3.6128	7.5656
Markov Estimate	3.9824	7.9864
Compression Estimate	3.472	7.336
t-Typle Estimate	3.7268	7.4856
LRS Estimate	3.9796	7.9648
Multi Most Common in Window Prediction Estimate	3.9824	7.9864
Lag Prediction Estimate	3.796	7.9776
MultiMMC Prediction Estimate	3.9776	7.984
LZ78Y Prediction Estimate	3.9804	7.9816

Table 3: Interrupts: SP800-90B Min-Entropy Measurements - Normal Use Case

Applying the Shannon entropy formula on the data set, a value of 18.412 bits per interrupt event is calculated. Using the min-entropy formula according to [AIS2031], a result of 13.954 bits per interrupt event is measured.

The conclusions that can be drawn from the numbers follow. Regardless of the worst-case or normal case, the high-resolution time stamp of each interrupt will return significantly more than two 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 (two bits when `RDRAND` is present). 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 Min-Entropy Estimates

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 mounting and unmounting a block device. This causes the buffer cache to be irrelevant for the disk accesses caused by the mount operations, as the buffer cache is flushed with each unmount operation of a file system. The worst-case produced the following data:

Entropy Estimate	4 Bits Width	8 Bits Width
Most Common Value Estimate	3.9684	7.9656
Collision Estimate	3.7452	7.592
Markov Estimate	3.9796	7.9808
Compression Estimate	3.48448	7.1328
t-Typle Estimate	3.7268	7.436
Longest Repeated Substring (LRS) Estimate	3.8868	7.9712
Multi Most Common in Window Prediction Estimate	3.96	7.9736
Lag Prediction Estimate	3.9888	7.9896
MultiMMC Prediction Estimate	3.9708	7.9736
LZ78Y Prediction Estimate	3.97	7.968

Table 4: Block Devices: SP800-90B Min-Entropy Measurements

Using the Shannon entropy formula, 19.592 bits per block device event is calculated. A value of 16.762 bits per block device event is calculated as the min-entropy according to [AIS2031].

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 0.01 bits of entropy. This can be interpreted that on average, the Linux-RNG awarded each block device event 0.01 bits of entropy.

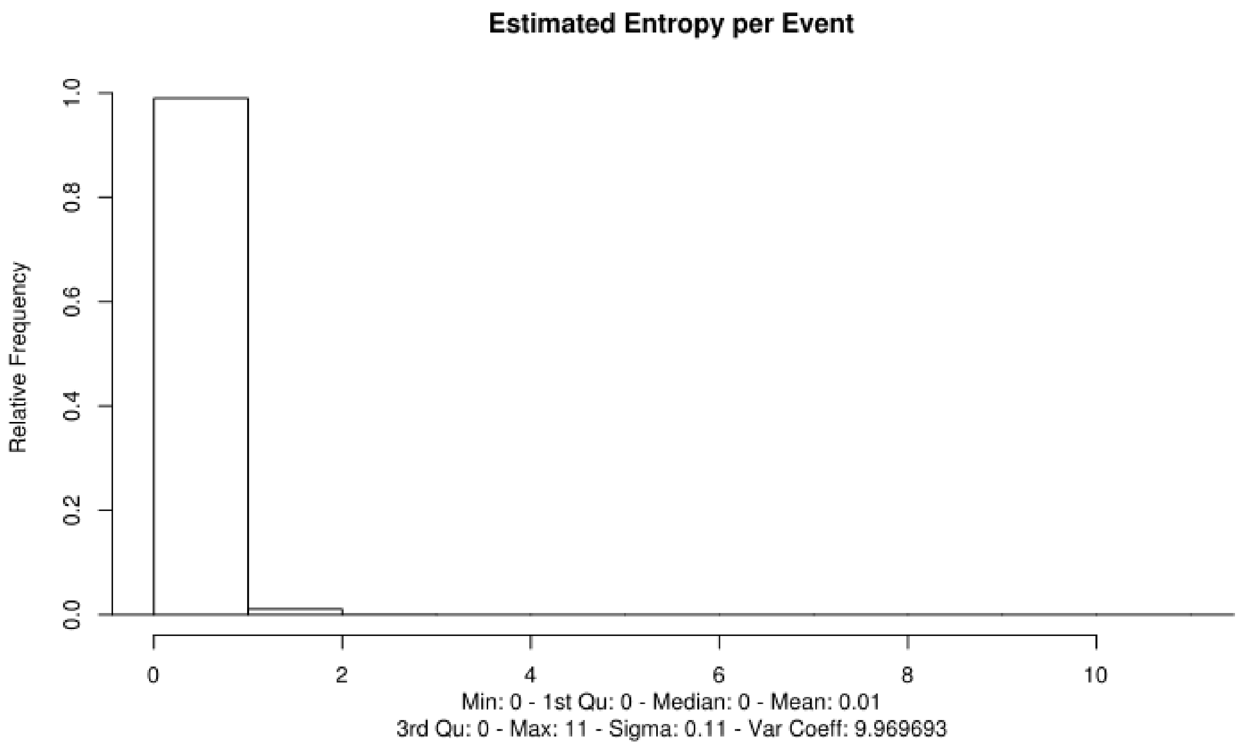

Figure 12: Entropy Estimate per Block Device Event Applied by Linux-RNG

Comparing the result shown in figure 12 with the min-entropy estimates calculated from the measured time stamps, the following conclusion is drawn: the min-entropy estimates have significantly more than 3 bits of entropy per event. On the other hand, the Linux-RNG considers that each event has on average only 0.01 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 Min-Entropy Estimates

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, only 500.000 samples of HID noise source events were recorded. The entropy estimates for the high-resolution time stamp applied to those events are listed in the table below.

Entropy Estimate	4 Bits Width	8 Bits Width
Most Common Value Estimate	3.9832	7.9808
Collision Estimate	2.2932	8
Markov Estimate	3.99	7.988
Compression Estimate	0.9824	3.0696
t-Typle Estimate	0.4304	0.944
LRS Estimate	0.4948	0.6384
Multi Most Common in Window Prediction Estimate	2.7196	6.3704
Lag Prediction Estimate	1.364	1.5072
MultiMMC Prediction Estimate	0.5272	1.112
LZ78Y Prediction Estimate	3.7952	4.6632

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

The Shannon entropy formula applied on the data set results in 17.155 bits per HID event. 13.049 bits per HID event are calculated when using the min-entropy formula according to [AIS2031].

The test record of the entropy estimate applied by the Linux-RNG for each recorded HID event is depicted with figure 13. 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 13, the mean value of the histogram is 0.42 bits. This implies that the Linux-RNG awarded 0.42 bits of entropy to each HID noise source event on average.

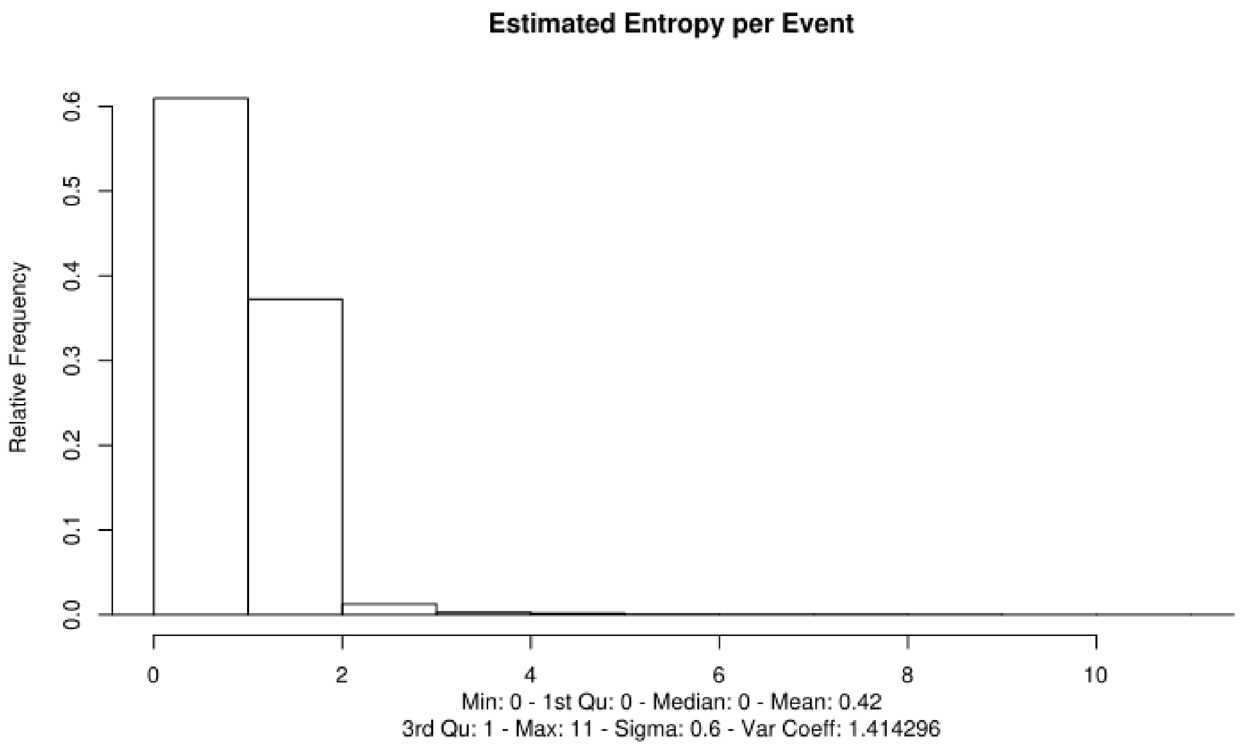

Figure 13: Entropy Estimate per HID Event Applied by Linux-RNG

A conclusion can be reached when comparing the heuristic entropy values applied by the Linux-RNG from figure 13 with the min-entropy estimates. The min-entropy estimates have shown that at least that amount of entropy is available (0.43 bits) as applied by the entropy heuristic (0.42 bits) when considering the lowest SP800-90B min-entropy values.

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 with 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 to diminish 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 concern: 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 (in the presence of `RDRAND` 2 bits are applied). 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 are initialized to allow users to interact with the system. Therefore, testing is limited to measure interrupt event data only. As outlined in section 6.1.1, only the high-resolution time stamp recorded for interrupts is of interest to entropy measurements.

The Linux kernel has been modified with the patch boottime/boottime_test.diff. This patch records the high-resolution time stamps obtained for the first 128 interrupts. A user space shell script boottime/boottime_test_record.sh stores these 128 time stamps to disk and initiates a reboot.

The test is performed for 50,000 reboot cycles for the virtual environment as well as for the bare-metal environment. At the end of the testing, 50,000 times 128 time stamps are collected and analyzed.

The first analysis performs an SP800-90B min-entropy estimate calculation discussed in section 6.2.1. Such a min-entropy estimate is calculated for each of the 128 32-bit time stamps individually. This means that the min-entropy estimate for the first till the 128th 32-bit time stamp of all boot cycles is calculated. Therefore, the full result contains 128 entries with min-entropy estimates. To limit the amount of space in this report for presenting the data, the tables in the following subsections only list the lowest min-entropy estimate out of all estimate types enumerated in section 6.2 for all 128 different interrupt occurrences.

In addition to the calculation of the min-entropy according to SP800-90B, the min-entropy and the Shannon entropy according [AIS2031] has been calculated as well. Both formulas are provided in section 2.3.2 of [AIS2031] and are not repeated here. As the time stamp is a monotonically increasing integer, the entropy lies in the deltas of the time stamps and the distribution of those deltas. Therefore, to perform the calculation for the Minimum 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.

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 128 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.

6.3.1 Early Boot Entropy Testing in a Virtual Environment

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
1	3.164152	6.321312	12.159	15.844
2	3.164152	6.321312	12.274	15.764
3	3.164612	6.977632	12.274	15.771
4	3.225552	6.666392	13.274	15.643
5	3.092844	6.602600	12.537	15.27
6	3.359940	6.348272	11.859	15.16
7	3.117876	6.920120	12.159	15.131
8	3.445524	6.231656	12.052	15.129
9	3.297788	6.681840	12.159	15.119
10	3.232592	6.726816	12.052	15.105
11	3.361196	6.530800	11.859	15.103
12	3.081280	6.758688	11.952	15.091
13	3.242416	7.036352	12.159	15.078
14	3.216636	6.570752	12.052	15.062
15	3.270588	6.343104	12.052	15.063
16	3.191524	6.346608	11.859	15.05
17	3.343716	6.263152	11.859	15.063
18	3.330696	6.369176	11.952	15.056
19	3.116880	6.568680	12.052	15.059
20	3.446528	6.649480	12.052	15.067

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
21	3.176908	6.364352	12.052	15.064
22	3.036088	6.607968	11.689	15.053
23	3.461780	6.406040	11.952	15.055
24	3.273912	6.875080	11.952	15.053
25	3.280780	6.983944	11.689	15.051
26	3.188164	6.381624	11.859	15.048
27	3.131684	6.697856	12.052	15.074
28	3.179088	6.708536	11.952	15.055
29	3.226916	6.852416	11.772	15.118
30	3.151240	6.630416	12.689	15.564
31	3.291272	7.142880	13.859	15.837
32	3.143308	6.454064	12.859	15.601
33	3.144060	6.307624	12.689	15.465
34	3.104256	6.417872	12.537	15.221
35	3.404188	6.411456	12.537	15.68
36	3.254288	6.487048	13.052	15.615
37	3.050188	6.802104	13.052	15.545
38	3.230192	6.474176	12.689	15.55
39	3.340920	6.689888	13.052	15.612
40	3.341084	7.121680	12.859	15.705
41	3.221800	6.617536	11.537	15.338
42	3.348848	6.406392	11.159	15.043
43	3.344880	6.802808	13.859	15.848

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
44	3.138980	6.776728	13.859	15.841
45	3.044384	6.663920	12.689	15.583
46	3.216300	6.474176	12.537	15.552
47	3.024200	6.727024	13.537	15.824
48	3.130916	6.477184	13.859	15.817
49	3.214240	6.555840	13.537	15.775
50	3.493144	6.890160	13.537	15.739
51	3.195340	6.431824	13.537	15.731
52	3.477344	6.250976	13.537	15.725
53	3.187896	6.320472	13.859	15.727
54	3.294976	7.028488	13.537	15.717
55	3.353060	6.841864	13.537	15.709
56	3.259816	6.611384	13.537	15.731
57	3.211132	6.143544	13.859	15.795
58	3.223824	6.523312	14.274	15.843
59	3.162372	6.736776	13.537	15.827
60	3.327736	7.020096	13.859	15.751
61	3.252712	6.663056	13.537	15.72
62	3.265756	6.705760	13.859	15.706
63	3.440552	6.360896	13.537	15.698
64	3.110100	6.525888	13.537	15.703
65	3.081140	6.659008	13.537	15.697
66	3.032832	6.390752	13.537	15.689
67	3.382332	7.012120	13.274	15.681

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
68	3.156640	6.737816	13.274	15.675
69	3.116740	6.440936	13.274	15.677
70	3.352636	6.667944	13.537	15.664
71	3.444560	7.033192	13.537	15.668
72	3.319580	6.679504	13.537	15.662
73	3.373844	6.625000	13.274	15.661
74	3.265196	6.582832	13.537	15.662
75	3.331448	6.590368	13.537	15.671
76	3.186888	6.930712	13.537	15.682
77	3.267740	6.547672	13.537	15.673
78	3.152324	6.898800	13.274	15.664
79	3.130872	6.609872	13.274	15.666
80	3.086236	6.559224	13.537	15.668
81	3.080592	6.565632	13.537	15.67
82	3.177792	6.466224	13.859	15.709
83	3.115764	6.545856	13.859	15.778
84	3.311148	6.898600	14.274	15.804
85	3.173292	6.773008	13.859	15.78
86	3.149712	6.661808	13.537	15.696
87	3.040060	6.457128	13.274	15.646
88	3.187388	6.895712	13.274	15.652
89	3.081628	6.986384	13.537	15.648
90	3.122720	6.747464	13.537	15.651
91	3.067516	6.493744	13.537	15.665

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
92	3.264896	6.540808	13.859	15.715
93	3.081388	6.594856	13.052	15.736
94	3.177652	6.629000	13.537	15.671
95	3.156068	6.615192	13.537	15.639
96	3.102948	6.522048	13.537	15.652
97	3.276060	6.867016	13.537	15.723
98	3.238848	6.772560	13.859	15.8
99	2.578068	5.524936	13.859	15.824
100	2.487492	5.463768	13.537	15.797
101	2.813840	6.691064	13.537	15.738
102	2.413476	5.238568	13.537	15.702
103	2.582076	5.079576	13.537	15.676
104	2.344500	5.781840	13.537	15.664
105	2.969664	5.638736	13.537	15.662
106	2.602308	5.665384	13.537	15.662
107	2.677696	5.526312	13.537	15.653
108	2.941272	5.371176	13.274	15.653
109	2.787420	5.422736	13.537	15.644
110	2.386012	5.769432	13.537	15.642
111	2.741884	5.672008	13.537	15.648
112	2.542964	5.477464	13.537	15.647
113	2.667744	6.080816	13.537	15.644
114	2.661780	5.580488	13.274	15.644
115	2.745624	6.468296	13.537	15.641

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
116	2.746308	5.692912	13.274	15.639
117	2.581732	5.759480	13.537	15.639
118	2.467832	5.735216	13.537	15.643
119	2.446332	5.671016	13.537	15.644
120	2.777600	5.899432	13.537	15.638
121	2.521128	5.399248	13.274	15.642
122	2.859536	6.037480	13.537	15.644
123	2.303792	5.161488	13.537	15.646
124	2.647892	5.501936	13.274	15.647
125	2.606484	6.132192	13.537	15.667
126	2.418868	5.516688	13.537	15.673
127	2.612740	6.306040	13.537	15.679
128	2.837576	5.676232		

Table 6: Interrupts: Early Boot SP800-90B Min-Entropy Measurements in Virtual Environment

The table shows that the high-resolution time stamp of each of the first 128 interrupts has an estimated min-entropy of at least 8 bits in the 4 LSB of the time stamp. In addition, the table shows that for the 8 LSB of the time stamp, at least 5 bits are measured for each of the first 128 interrupts. Considering the min-entropy according to AIS 20/31 applied to the time deltas (i.e. the difference of two adjacent time stamps), the range of values is between 11 and 14 bits per interrupt event. The Shannon entropy values for the time deltas are even higher and range above 15 bits per interrupt event.

To allow the reader to get a graphical view of the time stamp distribution, figure 14 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 14 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. All other interrupts exhibit a similar distribution pattern. To make the graphic more readable, only the 90% quartile of the time delta data is depicted. The remaining 10% cover such a large value span with so little probability of occurrence that they would render the graphic unreadable.

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. The red dotted line indicates how a Gaussian standard normal distribution would look like when using the standard derivation of the data set.

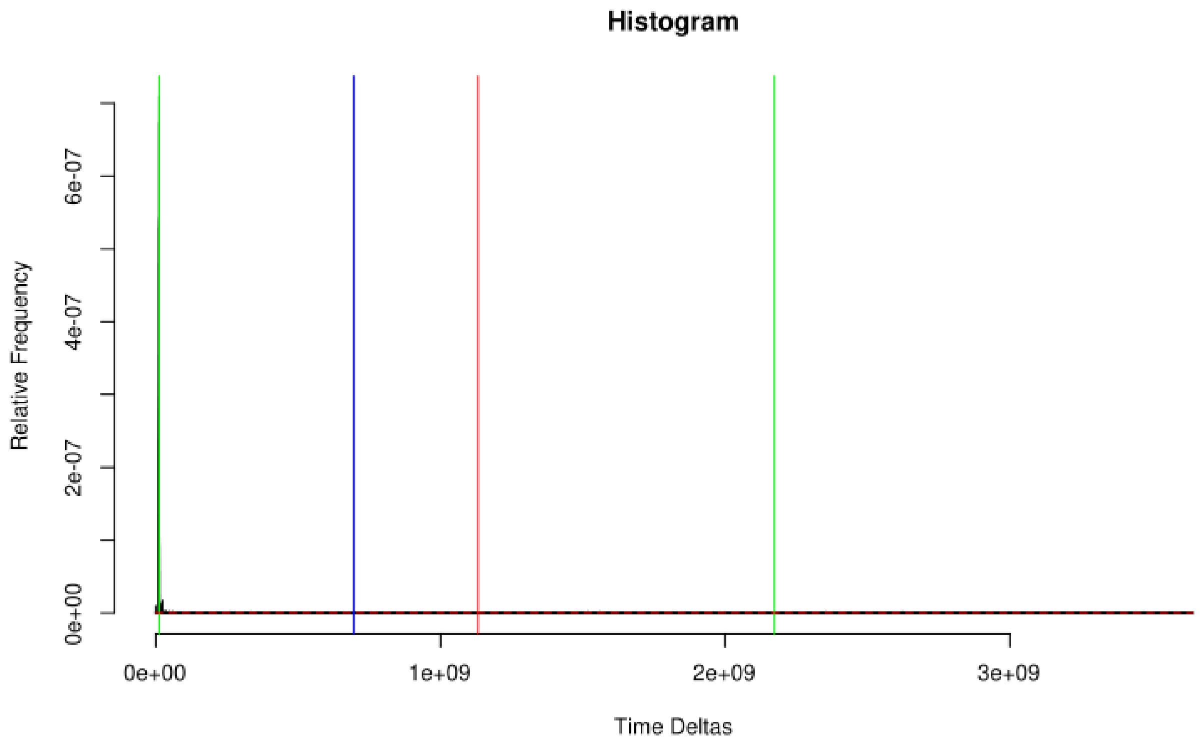

Figure 14: Histogram of Time Deltas for First and Second Interrupt in a Virtual Environment

The table with the min-entropy estimates for the time stamps of the first 128 interrupts visualized in figure 14 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 128 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.

6.3.2 Early Boot 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 less 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.

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
1	2.523752	6.657072	8.799	11.1
2	3.243216	6.657072	7.482	10.184
3	3.233932	6.711256	6.777	9.254
4	3.112624	6.522896	6.618	9.208
5	3.126408	6.866816	7.924	13.058
6	3.117260	6.764768	11.632	14.515
7	3.233428	6.853248	10.077	13.206
8	3.021152	6.276872	12.31	15.18
9	3.313892	6.476072	11.824	15.216
10	3.180784	6.318520	10.774	13.106
11	3.068548	6.298288	10.877	14.799
12	3.053152	7.085320	9.093	12.519
13	2.975800	6.826672	13.632	15.608
14	2.966264	6.314264	11.931	14.961
15	3.023292	6.662832	12.31	15.207
16	3.034524	6.438768	12.632	15.202
17	3.017228	6.573744	11.24	14.307
18	3.062660	6.839248	10.774	14.398
19	3.138460	6.463064	10.678	14.268
20	3.232848	6.519888	9.904	14.529
21	3.067868	6.555048	10.587	14.554
22	3.224884	6.372264	12.632	15.234
23	3.147372	6.622224	11.544	14.616
24	3.243900	6.291568	12.824	15.364

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
25	3.097536	6.626352	13.632	15.583
26	3.054600	6.910352	11.384	14.211
27	3.235292	6.713536	12.632	15.238
28	2.979912	6.673488	11.824	14.115
29	3.009916	6.690504	12.31	15.353
30	3.077064	6.999272	11.544	14.118
31	3.192548	6.950480	12.824	15.488
32	3.221872	6.308184	12.172	14.769
33	3.170064	6.185320	12.31	15.02
34	2.984208	6.737384	11.931	14.164
35	3.093872	6.528520	12.632	15.3
36	3.132196	6.368656	11.31	14.166
37	3.019472	6.267216	12.824	15.418
38	3.163796	6.442912	11.824	14.329
39	3.160292	6.368056	12.047	14.966
40	3.136232	6.582504	11.544	14.148
41	3.153084	6.861312	12.31	15.323
42	3.132992	6.522656	11.824	14.559
43	3.130900	6.794104	13.31	15.598
44	3.146536	6.916976	12.172	15.082
45	3.379284	6.538008	12.172	15.292
46	3.144336	6.894248	11.931	14.945
47	3.258212	6.417568	11.544	15.214
48	3.087040	6.875448	12.31	15.227

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
49	3.198252	6.473072	13.31	15.513
50	3.113904	6.358544	13.047	15.513
51	3.294476	6.702296	12.172	15.216
52	3.381812	6.483592	13.632	15.616
53	3.272460	6.314160	13.31	15.572
54	2.918928	6.416688	13.047	15.546
55	3.073476	6.865888	12.824	15.554
56	3.048224	6.383248	13.632	15.556
57	3.077248	6.679928	14.047	15.593
58	3.234284	6.942920	12.824	15.477
59	3.084016	6.700680	13.047	15.496
60	3.126624	6.690640	13.632	15.569
61	3.187556	6.433992	13.632	15.561
62	3.173180	6.315504	14.047	15.602
63	3.012424	6.193688	13.632	15.588
64	3.214220	6.830648	13.31	15.535
65	3.273944	6.398200	13.047	15.523
66	3.267136	6.450952	12.824	15.499
67	3.178368	6.557112	13.047	15.516
68	3.033760	6.454536	13.31	15.536
69	3.268784	6.473024	13.31	15.528
70	3.224404	7.061808	12.824	15.488
71	3.031392	6.532432	13.047	15.563
72	2.938388	6.381848	13.31	15.53

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
73	3.284808	6.289568	13.632	15.553
74	3.237712	6.465232	13.31	15.533
75	3.166976	6.980784	13.31	15.45
76	3.158248	6.521680	12.462	15.241
77	3.377052	6.624920	13.047	15.44
78	3.298104	6.317968	13.31	15.456
79	3.277244	6.393984	13.31	15.525
80	3.070432	6.891128	12.632	15.372
81	3.121220	7.019472	13.047	15.409
82	3.235860	6.336552	12.462	15.372
83	3.424640	6.510512	12.632	15.395
84	3.301660	6.315360	12.824	15.443
85	3.153000	6.631992	13.047	15.453
86	3.088300	6.549000	12.632	15.225
87	3.162404	6.352120	12.632	15.324
88	3.205232	6.485600	12.462	15.232
89	3.260064	6.366632	12.824	15.427
90	2.972028	6.850064	12.172	15.349
91	3.048044	6.633472	12.462	15.144
92	3.511388	6.600576	12.632	15.433
93	3.070232	6.455864	12.462	15.302
94	3.439240	6.504992	12.632	15.262
95	3.076072	7.109392	12.31	15.091
96	2.991204	6.320072	12.462	15.189

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
97	3.142976	7.033808	13.31	15.438
98	3.106980	6.744144	10.077	14.217
99	3.193568	6.666944	12.31	15.292
100	3.241764	6.543856	11.172	14.407
101	3.191532	6.635120	10.503	13.767
102	3.309320	6.555568	13.632	15.552
103	3.302180	6.520360	13.31	15.558
104	3.071644	6.511568	12.172	15.085
105	3.109848	6.591264	13.31	15.435
106	3.379844	6.365384	12.824	15.434
107	3.250672	6.546048	13.047	15.432
108	2.976388	6.548752	12.824	15.427
109	3.155624	6.375024	12.824	15.4
110	3.045116	6.791640	12.824	15.388
111	3.241400	7.183808	13.047	15.376
112	3.131784	6.415952	12.824	15.364
113	3.013964	6.455496	13.047	15.375
114	3.206112	6.700768	12.824	15.422
115	3.246716	6.235520	12.632	15.364
116	3.214056	6.547528	12.31	15.361
117	2.979380	6.171472	12.632	15.344
118	3.215396	6.665712	13.31	15.494
119	3.161804	6.642616	13.31	15.513
120	3.237372	6.369456	13.31	15.5

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
121	3.083428	6.329824	13.31	15.487
122	3.240476	6.529224	12.824	15.472
123	3.246416	6.380088	12.632	15.476
124	3.155656	6.554800	12.824	15.475
125	3.208796	6.502376	13.047	15.479
126	3.123564	6.506184	12.824	15.471
127	3.144524	6.673000	13.047	15.476
128	3.287312	6.179592		

Table 7: Interrupts: Early Boot SP800-90B Min-Entropy Measurements on Native Hardware

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 15. 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 between the occurrence of both interrupts. As discussed for the virtual environment, the 90% quartile is depicted.

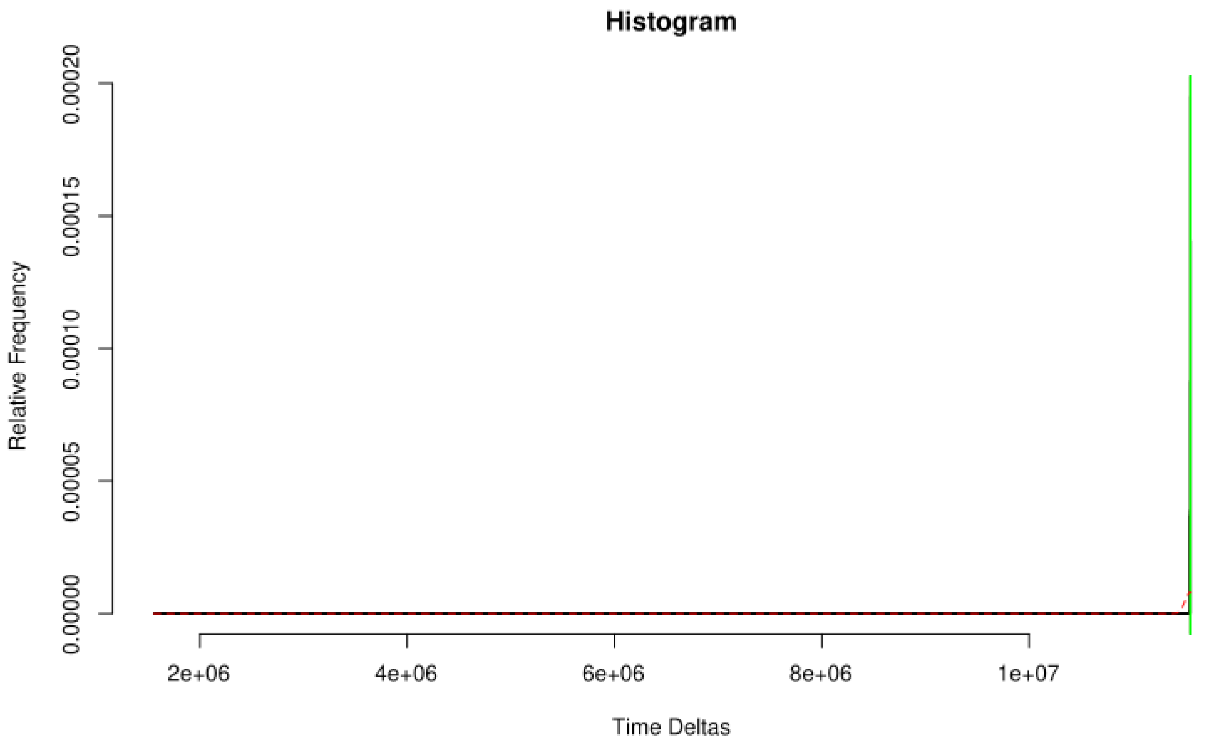

Figure 15: Histogram of Time Deltas for First and Second Interrupt in a Native Environment

The pattern of the histogram is very similar to virtual environments shown in figure 14 albeit the concentration of values is a bit higher. That is supported by the fact that the entropy value for the first time delta is a bit lower than the rest of the time deltas. Starting with the second time delta depicted in figure 16, 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.

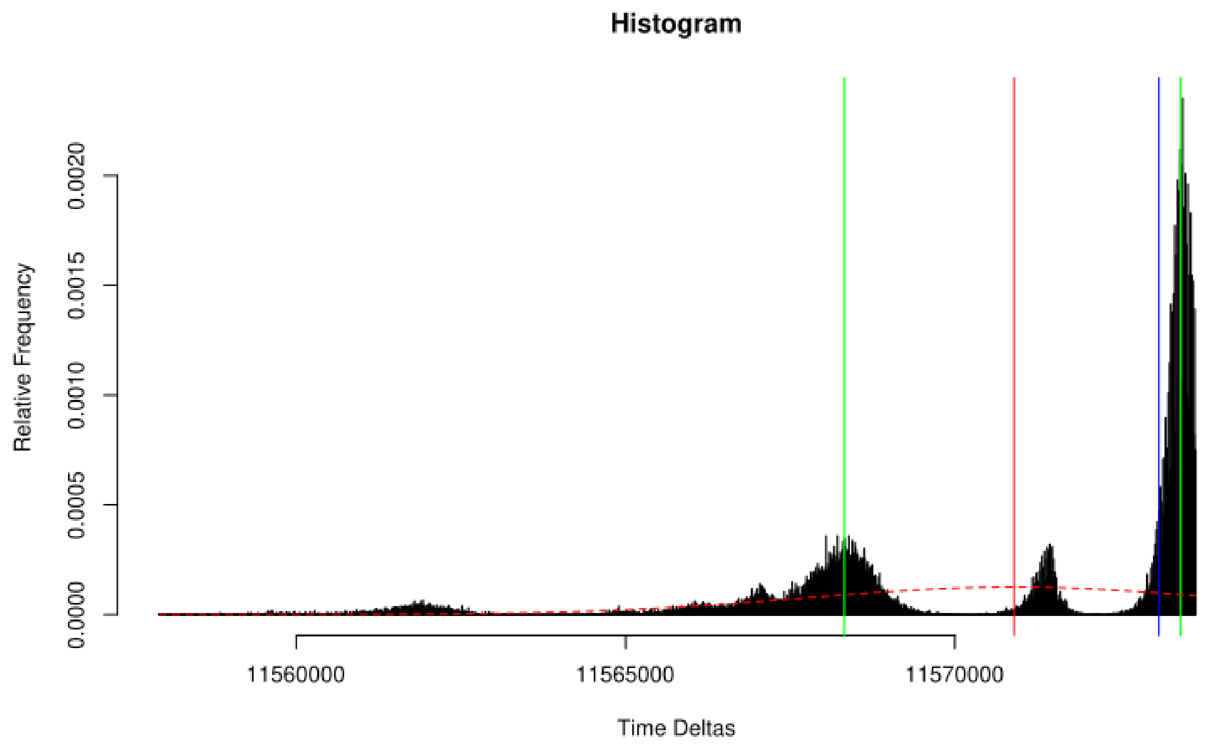

Figure 16: Histogram of Time Deltas for Second and Third Interrupt in a Native Environment

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, 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.

6.3.3 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 considerably exceeds one bit.

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 unblocks and generates random numbers. This allows the conclusion that when the `getrandom` system call unblocks, 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 is considered to always provide 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 boot. 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 and injected into the ChaCha20 DRNG only once one of the fast_pool received 64 interrupts. 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[23], 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.

23 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. A very similar LFSR operation is performed when data is injected into the blocking_pool. When data is injected into the ChaCha20 DRNG, it is mixed with the already present data. In all three 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 so that a counter starting at one is increased by one with each request and provides the input data. The state after the state transition function operation 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 white noise.

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 with.

The following code is used for the analysis of the polynomials applied for the LFSR of the input_pool and blocking_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;

P_output := y^32 + y^26 + y^19 + y^14 + y^7 + y + 1;
Q_output := alpha^3*(P_input-1)+1;

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



// 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);
```

```
print "is blocking_pool polynomial primitive?";
l := LeadingCoefficient(Q_output);
IsPrimitive(Q_output/l);
```

The polynomial used for the input_pool LFSR operation is deemed to be not primitive but irreducible over GF(2^{32}) as explained in [LRSV12].

Similarly, the polynomial used for the blocking_pool LFSR operation is identified to be not primitive, but irreducible over GF(2^{32}) as explained in [LRSV12].

The implementation switched from primitive and irreducible to these polynomials between kernel version 3.12 and 3.13.

Another set of polynomials are suggested in [FV17] for the input_pool as well as the blocking_pool. Both suggested polynomials are assessed with the Magma code outlined above considering $Q(X) = \alpha^4 (P(X) - 1) + 1$ with P(X) referencing the respective polynomial. Both polynomials were identified to be primitive and irreducible over GF(2^{32}). Thus, it is suggested to use the polynomials presented in [FV17] for the LFSR operation of the input_pool as well as the blocking_pool.

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 and the blocking_pool use the output data from the input_pool. In both cases, 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 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 code for the LFSR demonstration is provided in the test directory of lfsr_demonstration/. This code contains the LFSR with the polynomials for the input_pool and the blocking_pool. Therefore, this code can be used to validate both entropy pools.

The state of the entropy pool is initialized with zeros.

The test mixes 128 bytes (input_pool) or 32 bytes (blocking_pool) into the entropy pool. This number is equal to the number of 32-bit words present in the entropy pool (128 and 32, respectively). As the mixing function operates word-wise, after the mix-in of the 128 bytes / 32 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 / 32 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:

```
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000039c46d07c8206e
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:

```
00000001e83e686423d37ea46cdb9359e6dff12b5631e9ae4ad4605282607df625218c5f
fdffcfb5c2a131462991d7dd59b10cf08f1f10adbab6097ec9bb92efcc26a9f18a23423e3
41c2b8d298b3e3853c1c197fd22747d47e9c267782503bad8bf6205ea839622093784895 0
ea80d8224de0c6fbb7ad74d625de184c99240beef8cd2f33e6f2c1721620d0d0595410c30
52521d687c0e65e744f9a3989a3838d7b67d30faf8f027b654ebe8a5115e2a4f99036bd7f
efce5efbd21dd158bd49e3d15bae538748edc6f73437f90a9e522902f7b7dd0b8d299e48d
3f8d2943976ae550ee591ba45d1952512c020af18e261b208419c15bf49a1e5265c6c2c8f
19a4e8a3365cb214bfc461a6ca5687d7927c1df5f50fc0d9de3cf1ec8af2550ff78779e23
8a2a90a92b5c0899b8ab277951a5ce315e8181f3bf9c1f0449b952778130a714605636623
b33987536ea624a5af62fe5595fd639a75e62885db21fa7de1090337829048d77939f50c7
0069e85e9b68767275034650713e095e119081e2d60f2bfde5914bcd89b4ae29df70502a6
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 is white noise.
- The Chi-Squared result for byte-wise processing of the binary data is 30.38. This result also indicates tjat the data is white noise.
- 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 being white noise.
- All tests pass the test procedure A, providing another indication for white noise data.

All results combined strongly suggest that the data shows characteristics of white noise 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 white noise 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.

7.2.1.2 blocking_pool

The same analysis performed for the input_pool is performed for the blocking_pool. As the state transition function applied to the blocking_pool is identical to the one used for the input_pool with the exception of the used LFSR polynomial, this section relies on the previous section for the rationale.

After mixing in the first byte holding a one into the empty blocking_pool, its contents is:

```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000c8206e3b
```

Again, the data is identical to the content of the input_pool as the same operation is applied. This is to be expected, since the application of the LFSR polynomial only XORs zeros onto the input data and thus does not alter the input data. Only the application of the `twist_table` modifies the data, as in the case of the input_pool.

The mixing in the second byte with a value of two changes the state of the entropy pool to:

```
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000039c46d07c8206e3b
```

Again it is visible that the data is identical to the state of the input_pool after the mix-in of the second byte. At this point, the only applied LFSR tap is the use of the first word. The LFSR polynomials for the input_pool and the blocking_pool are identical for this tap.

After the mix-in of the 32nd byte, the entropy pool contains:

```
d33dc01d7194435279d5238b0d414acdf31691cb1f9d0a9ca810485c09fa305c36e5ed1c7
ab810e85354d044f870bffd8b554bcb8a284e223f39bd552ee280342c96985a40d55976da
0b86c439151b6aac093e7964b3905a52017c1eda8a63c8ac8dc4fbac549e88e6acd49f867
2832571926aa74f80833b39c46d07c8206e3b
```

With this mix-in of the 32nd byte, all words in the blocking_pool have been modified once. A snapshot of the entropy pool is recorded as a bit-stream.

Just like for the input_pool, 100,000 cycles of a complete modification of the blocking_pool are performed where a snapshot of the entropy pool state is recorded after each complete modification.

This binary data bit stream shows the following characteristics:

- The Chi-Squared result when processing the data bit-wise is 88.36. The result indicates that the output is white noise.
- The Chi-Squared result for byte-wise processing of the binary data is 31.38. This result also indicates that the data is white noise.
- For all compression algorithms, the “compressed” data is larger than the binary data. Thus, the compression algorithms could not find patterns that allow the data to be compressed. This is another indication that the data is white noise.
- All tests pass the test procedure A, providing another indication for white noise data.

The conclusion that can be drawn from these results is the same as for the input_pool: The LFSR operation is considered to be appropriate for the maintenance of data containing entropy.

7.2.2 ChaCha20 State Transition

To demonstrate the ChaCha20 state transition behavior, the test code provided in the directory chacha20/ is used.

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 white noise 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 "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 fast 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. 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 white noise.
- The Chi-Squared value for a byte-wise processing of the data stream is 21.44 which also indicates white noise.
- 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 is white noise.
- All tests defined by the test procedure A are passed.

7.3 AIS 20/31 Test Procedure A for Entropy Pools

Using the test code input_pool/, the input_pool as well as the blocking_pool are observed. The SystemTap test code takes a snapshot of the entropy pool after an amount of bytes has been mixed in that equals its size in words. That means that a snapshot is taken of the input_pool after 128 bytes have been mixed in.

Similarly, a snapshot of the blocking_pool has been taken after 32 bytes have been mixed into the blocking_pool. After the mix-in of the stated amount of bytes, all words in the entropy pool have been changed by the LFSR operation.

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 white noise in the entropy pool.

7.3.1 input_pool

After generating 1,000,020 snapshots with the test code and concatenating all data, a binary string is present that can be analyzed as follows:

- The Chi-Squared value using the ent tool shows a value < 0.01 (bit-wise) and 296.24 (byte-wise) which indicates white noise.
- The test procedure A is passed by the bit string.

The test results for the input_pool confirm the test results obtained during the analysis presented in section 7.2.

7.3.2 blocking_pool

Similarly to the input_pool, the blocking_pool is analyzed after taking 60728 snapshots. Again, the data can be characterized:

- The Chi-Squared value provided with the ent tool shows the value of 1.27 (bit-wise) and 228.53 (byte-wise) which indicates white noise.
- The test procedure A is also passed by the bit string.

This implies that the test results confirm the results obtained from the analysis provided in section 7.2.

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 and 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 white noise are present.

The testing is conducted on the output received by callers via the /dev/random device (data is received from the blocking_pool) and the /dev/urandom device (data is received from the ChaCha20 DRNG).

The test code used for the functional verification of the output functions is provided in the test code directory of DRNG-output. The conducted testing can be summarized as follows. The device files /dev/random and /dev/urandom are 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 is not white noise. The calculation of the Chi-Squared value is considered an easy approach to identify white noise due to the following: if the Chi-Squared test fails, then no white noise is assumed to be present. However, there could be false positives in the sense that the Chi-Squared result indicates white noise data 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 is not white noise, 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 either /dev/random or /dev/urandom. To apply all tests implemented in the `dieharder` statistical tool to the output of /dev/urandom, the following call is executed:
  ```
  cat /dev/urandom | dieharder -a -g 200
  ```
  For processing /dev/random, the same command is used.

- 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 and /dev/urandom. 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.

As /dev/random blocks if no entropy is present, a tool is devised that injects zero buffers into /dev/random and increases the entropy estimator via the `RNDADDENTROPY` IOCTL. By injecting a zero buffer into /dev/random, the `dieharder` test validates the strength of the /dev/random output function and not the potential cryptographic strength of the inserted data.

8.1 Output of blocking_pool

When obtaining data from /dev/random with the different block sizes, 8 out of the 4096 data sets showed a Chi-Squared value which was outside the aforementioned range. When rerunning the test on these block sizes, the Chi-Squared value was again back in the expected range. Thus, these outliers were a false positive due to the small data size. This allows the conclusion that the Chi-Squared testing does not indicate that the random number generation function used for /dev/random exhibits implementation errors.

The compression operation of the output data always showed that the “compressed” file is larger than the original data. Again, this indicates that the compression algorithms were unable to detect patterns in the data file which confirms that no implementation errors were detected in the random number generating function backing /dev/random.

The dieharder testing shows two “weak” results. All other tests passed.

When executing the test procedure A on the data obtained for the dieharder tests, all tests pass. This indicates white noise data and thus again confirms the previous results.

8.2 Output of ChaCha20 DRNG

Just like for the blocking_pool, the output data from the ChaCha20 DRNG that backs /dev/urandom 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/urandom.

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”. No failed or “weak” test result is present. This type of result is expected for white noise data. Thus, the dieharder test result confirms the initial test results.

All tests pass the test procedure A, indicating white noise data confirming the results of the previous tests.

8.3 Conclusion of the Output Function Testing

The testing has shown that both the output function generating random numbers for /dev/random and the one for /dev/urandom produce white noise. Thus, no implementation errors that would diminish the entropy in the random numbers were identified.

9 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.

9.1 Linux Kernel 4.10

Previously assessed Linux kernel version: 4.9 – http://www.kernel.org/pub/linux/kernel/v4.x/linux-4.9.tar.xz

Currently assessed Linux kernel version: 4.10 – http://www.kernel.org/pub/linux/kernel/v4.x/linux-4.10.tar.xz

The following subsections contain the assessment of potential changes.

9.1.1 Changes to the Linux-RNG Implementation

9.1.1.1 File drivers/char/random.c

File showing deltas: see kernelupdates/4.10/random.c.diff

The following changes are visible:

- Replacement of asm/uaccess.h include file with linux/uaccess.h. This is due to a move of C code from the old to the new file. This change does not have any effect on the functionality of the Linux-RNG.

9.1.1.2 File include/linux/random.h

No changes to the file are present.

9.1.1.3 File include/uapi/linux/random.h

No changes to the file are present.

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

No changes to the file are present.

9.1.2 Changes to Invocation of Entropy Gathering Functions

9.1.2.1 add_input_randomness

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

9.1.2.2 add_interrupt_randomness

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

9.1.2.3 add_disk_randomness

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

9.1.2.4 add_hwgenerator_randomness

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

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

9.1.3 Definition and Use of new Interfaces

No functions implemented in random.c are newly defined with one of the EXPORT_SYMBOL*() macros. Therefore, no new interfaces are exported.

9.2 Linux Kernel 4.11

Previously assessed Linux kernel version: 4.10 – http://www.kernel.org/pub/linux/kernel/v4.x/linux-4.10.tar.xz

Currently assessed Linux kernel version: 4.11 – http://www.kernel.org/pub/linux/kernel/v4.x/linux-4.11.tar.xz

The following subsections contain the assessment of potential changes.

9.2.1 Changes to the Linux-RNG Implementation

9.2.1.1 File drivers/char/random.c

File showing deltas: see kernelupdates/4.11/random.c.diff

The following changes are visible:

- Several changes are applied to cleanup the code suggested by the author of this study, including: removal of unused `random_min_urandom_seed` variable from operational code (the variable is left as it is a user-space interface but it has no purpose any more), removal of the `urandom_init_wait` wait queue, removal of the `limit` variable in entropy pool data structure and all code used for the maintenance of the nonblocking_pool (which is not used any more), removal of unused function `maybe_reseed_primary_crng`. These changes do not affect the Linux-RNG functionality as only unused code paths are removed.
- The implementation of `get_random_int` and `get_random_long` have been replaced with the functions `get_random_u32` and `get_random_u64` which use the ChaCha20 DRNG from the Linux-RNG. These two new functions are marked as an interface function with the EXPORT_SYMBOL macro. This implies that these two new functions are new in-kernel interfaces to the Linux-RNG as documented in section 3.4.3. The new interface does, however, not constitute a change in the logic flow of the Linux-RNG.

9.2.1.2 File include/linux/random.h

File showing deltas: see kernelupdates/4.11/random.h.diff

The following changes are visible:

- The new API calls of `get_random_u32` and `get_random_u64` mentioned above are defined. The old API calls of `get_random_int` and `get_random_long` are replaced with a call to the new API calls.

9.2.1.3 File include/uapi/linux/random.h

No changes to the file are present.

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

No changes to the file are present.

9.2.2 Changes to Invocation of Entropy Gathering Functions

9.2.2.1 add_input_randomness

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

9.2.2.2 add_interrupt_randomness

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

9.2.2.3 add_disk_randomness

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

9.2.2.4 add_hwgenerator_randomness

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

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

9.2.3 Definition and Use of new Interfaces

The functions of `get_random_u32` and `get_random_u64` discussed above are marked with the `EXPORT_SYMBOL` macro. Therefore, these functions are considered new in-kernel APIs.

9.3 Linux Kernel 4.12

Previously assessed Linux kernel version: 4.11 – http://www.kernel.org/pub/linux/kernel/v4.x/linux-4.11.tar.xz

Currently assessed Linux kernel version: 4.12 – http://www.kernel.org/pub/linux/kernel/v4.x/linux-4.12.tar.xz

The following subsections contain the assessment of potential changes.

9.3.1 Changes to the Linux-RNG Implementation

9.3.1.1 File drivers/char/random.c

File showing deltas: see kernelupdates/4.12/random.c.diff

The following changes are visible:

- Fix race condition in edge conditions on M68K architectures when accessing an Linux-RNG internal variable atomically.
- Fix race conditions in the CPU-local buffer accesses for the `get_random_u32` and `get_random_u64` API functions.

9.3.1.2 File include/linux/random.h

No changes to the file are present.

9.3.1.3 File include/uapi/linux/random.h

No changes to the file are present.

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

No changes to the file are present.

9.3.2 Changes to Invocation of Entropy Gathering Functions

9.3.2.1 add_input_randomness

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

9.3.2.2 add_interrupt_randomness

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

9.3.2.3 add_disk_randomness

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

9.3.2.4 add_hwgenerator_randomness

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

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

9.3.3 Definition and Use of new Interfaces

No functions implemented in random.c are newly defined with one of the EXPORT_SYMBOL*() macros. Therefore, no new interfaces are exported.

9.4 Linux Kernel 4.13

Previously assessed Linux kernel version: 4.12 – http://www.kernel.org/pub/linux/kernel/v4.x/linux-4.12.tar.xz

Currently assessed Linux kernel version: 4.13 – http://www.kernel.org/pub/linux/kernel/v4.x/linux-4.13.tar.xz

The following subsections contain the assessment of potential changes.

9.4.1 Changes to the Linux-RNG Implementation

9.4.1.1 File drivers/char/random.c

File showing deltas: see kernelupdates/4.13/random.c.diff

The following changes are visible:

- The implementation of `add_device_randomness` has been changed such that all received data is directly mixed into the ChaCha20 DRNG before the ChaCha20 DRNG is considered to be initially seeded. Section 3.5.2.4 has been updated accordingly.
- The function `warn_unseeded_randomness` has been added which generates kernel log messages about callers of `get_random_bytes` that request random data before the ChaCha20 DRNG is initially seeded. This change is intended to aid kernel development to notify users when requests for random data are made too early. This change does not affect the cryptographic aspects of the Linux-RNG.
- The API function call of `wait_for_random_bytes` is added. This API call is documented in section 3.4.3.

9.4.1.2 File include/linux/random.h

File showing deltas: see kernelupdates/4.13/random.h.diff

The following changes are visible:

- The declaration of the aforementioned new API function of `wait_for_random_bytes` is added. This is accompanied by helper functions of `get_random_bytes_wait`, `get_random_u32_wait`, `get_random_u64_wait`, `get_random_long_wait` and `get_random_int_wait` which all invoke the respective functions without the "_wait" prefix after the ChaCha20 DRNG has been initially seeded.
- The helper function of `get_random_canary` is added which invokes the Linux-RNG `get_random_long` API function and zeroizes the 8 LSB of the generated 64 bit random integer variable on 64-bit systems. On 32-bit systems, a 32-bit random integer value is generated which remains unchanged. This function call is intended to support the stack canary initialization to protect against C string overflows. This change does not affect the Linux-RNG operation.

9.4.1.3 File include/uapi/linux/random.h

No changes to the file are present.

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

No changes to the file are present.

9.4.2 Changes to Invocation of Entropy Gathering Functions

9.4.2.1 add_input_randomness

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

9.4.2.2 add_interrupt_randomness

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

9.4.2.3 add_disk_randomness

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

9.4.2.4 add_hwgenerator_randomness

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

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

9.4.3 Definition and Use of new Interfaces

The API function call of `wait_for_random_bytes` is added and marked with EXPORT_SYMBOL. This API call is documented in section 3.4.3.

9.5 Linux Kernel 4.14

Previously assessed Linux kernel version: 4.13 – https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.13.tar.xz

Currently assessed Linux kernel version: 4.14 – https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.14.tar.xz

Assessment of changes: Changes do not affect the functionality of the Linux-RNG.

The following subsections contain the assessment of potential changes.

9.5.1 Changes to the Linux-RNG Implementation

9.5.1.1 File drivers/char/random.c

No changes to the file are present.

9.5.1.2 File include/linux/random.h

The file received an additional comment regarding the software license. This change does not affect any part of the functionality defined with this file.

9.5.1.3 File include/uapi/linux/random.h

The file received an additional comment regarding the software license. This change does not affect any part of the functionality defined with this file.

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

No changes to the file are present.

9.5.2 Changes to Invocation of Entropy Gathering Functions

9.5.2.1 add_input_randomness

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

9.5.2.2 add_interrupt_randomness

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

9.5.2.3 add_disk_randomness

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

9.5.2.4 add_hwgenerator_randomness

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

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

9.5.3 Definition and Use of new Interfaces

No functions implemented in random.c are newly defined with one of the EXPORT_SYMBOL*() macros. Therefore, no new interfaces are exported.

9.6 Linux Kernel 4.15

Previously assessed Linux kernel version: 4.14 – https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.14.tar.xz

Currently assessed Linux kernel version: 4.15 – https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.15.tar.xz

Assessment of changes: Changes do not affect the functionality of the Linux-RNG.

The following subsections contain the assessment of potentially relevant changes.

9.6.1 Changes to the Linux-RNG Implementation

9.6.1.1 File drivers/char/random.c

The implementation uses the macro READ_ONCE instead of the now deprecated ACCESS_ONCE macro to perform an atomic read operation of a variable. This change does not alter the behavior of the Linux-RNG.

9.6.1.2 File include/linux/random.h

No changes.

9.6.1.3 File include/uapi/linux/random.h

No changes.

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

The file contains an editorial change to the assembler code which does not alter the behavior of the implemented functionality or the functionality of the Linux-RNG.

9.6.2 Changes to Invocation of Entropy Gathering Functions

9.6.2.1 add_input_randomness

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

9.6.2.2 add_interrupt_randomness

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

9.6.2.3 add_disk_randomness

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

9.6.2.4 add_hwgenerator_randomness

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

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

9.6.3 Definition and Use of new Interfaces

No functions implemented in random.c are add by means of any of the EXPORT_SYMBOL*() macros. Therefore, no new interfaces are exported.

9.7 Linux Kernel 4.16

Previously assessed Linux kernel version: 4.15 – https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.15.tar.xz

Currently assessed Linux kernel version: 4.16 – https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.16.tar.xz

Assessment of changes: Changes do not affect the functionality of the Linux-RNG.

The following subsections contain the assessment of potentially relevant changes.

9.7.1 Changes to the Linux-RNG Implementation

9.7.1.1 File drivers/char/random.c

The diff of the file shows numerous changes. All of these changes, however, can be categorized into the following modifications:

- Use aligned memory access for ChaCha20: The ChaCha20 block transformation function operates on 32-bit words. This applies to input as well as output buffers the ChaCha20 block transformation is applied to. When accessing memory as 32-bit words, the CPU will always access the memory on a 4-byte word boundary. In order to ensure that all bytes of the input/output buffer are accessed by the ChaCha20 block transformation (and not up to 3 bytes before or after), it is mandatory that the pointer to the input/output buffer is aligned on a 4-byte boundary. So far, the input/output buffers were treated as an array of 8-bit words. Technically it was therefore possible that the buffer pointers are not aligned on a 4-byte boundary. The modification changes all input/output buffer pointers to be aligned to a 4-byte boundary. This change in the code does not affect the x86 32-bit or 64-bit processor architecture, because the GCC compiler will align even 8-bit buffer pointers on either a 4-byte or 8-byte boundary. However, on other architectures, this is not guaranteed. Therefore, the change does not affect the operation of the Linux-RNG on an x86 architecture.
- Change of the poll infrastructure: The Linux kernel poll infrastructure received changes unrelated to the Linux-RNG implementation. This change also affects the interface functions exported by the poll infrastructure to other kernel parts. These interface changes imply that the Linux-RNG invocation of the poll infrastructure must be changed accordingly. The changes required in the Linux-RNG, however, are minuscule: a rename of 3 flags whose purpose did not change and the change of the data type of a variable without a change of the meaning of the variable. These changes do not affect the operation of the Linux-RNG.

9.7.1.2 File include/linux/random.h

No changes.

9.7.1.3 File include/uapi/linux/random.h

No changes.

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

No changes.

9.7.2 Changes to Invocation of Entropy Gathering Functions

9.7.2.1 add_input_randomness

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

9.7.2.2 add_interrupt_randomness

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

9.7.2.3 add_disk_randomness

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

9.7.2.4 add_hwgenerator_randomness

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

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

9.7.3 Definition and Use of new Interfaces

No functions implemented in random.c are add by means of any of the EXPORT_SYMBOL*() macros. Therefore, no new interfaces are exported.

9.8 Linux Kernel 4.17

Previously assessed Linux kernel version: 4.16 – https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.16.tar.xz

Currently assessed Linux kernel version: 4.17 – https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.17.tar.xz

Assessment of changes: Changes do not affect the functionality of the Linux-RNG. The NTG.1 properties of /dev/random are not affected by the changes.

The following subsections contain the assessment of potentially relevant changes.

9.8.1 Changes to the Linux-RNG Implementation

9.8.1.1 File drivers/char/random.c

The diff of the file shows numerous changes. All of these changes, however, can be categorized into the following modifications:

- Although the following change is only in one line, it has user-visible impact: The Linux-RNG now considers the ChaCha20 DRNG fully seeded after it received 128 bit of entropy from the noise sources. Previously it was sufficient that it received at least 256 interrupts. The Linux-RNG author announced this change as a big security update indicating that the previous implementation affecting the behavior of the getrandom(2) system call was considered insecure. Based on the measurements of the amount of entropy considered to be present in the interrupts provided with section 6.3, such security impact cannot be detected at least for Intel x86-based systems. Furthermore, it is assumed that such security relevance is equally not applicable for all platforms with a high-resolution timer. Nonetheless, the change does not affect the security of the Linux-RNG. Changes to the design description in the preceding sections have been applied.
- The use of the function wq_has_sleepers is added to verify the presence of waiting tasks in the rand_read_wakeup wait queue before invoking the function to wake all up. This addition is a cosmetic change to prevent the invocation of the wakeup function if there are no tasks waiting. The change does not affect the security of the Linux-RNG.
- The safety-check in credit_entropy_bits_safe to prevent overflowing the entropy estimator is modified to cap the maximum entropy level to be added to the size of the entropy pool instead of to a value that is mathematically safe. This change is considered to be a safety-related change and has no relevance to security, because the input to the function must already be correct to ensure that the entropy estimator holds the proper entropy value. Therefore, the change does not affect the security of the Linux-RNG.
- The allocation of the per-NUMA-node ChaCha20 DRNG instances is now moved into a work queue for asynchronous allocation of the instances. While the instances are not yet allocated, the initial ChaCha20 DRNG instance is used. Thus, the change does not affect the security of the Linux-RNG.
- Add the function crng_slow_load which is called from the add_device_randomness noise source with the goal to inject the seed data into the ChaCha20 DRNG state. The goal is to inject the data with a code path that does not have the time constraints of interrupts. Besides, the data is mixed directly into the ChaCha20 DRNG state using a small LFSR to ensure that each bit of the ChaCha20 DRNG is touched when injecting data. The change does not affect the security of the Linux-RNG.
- The Linux-RNG prints out warnings at early boot time in case data is obtained when it is not yet initially or fully seeded. Changes are added to rate-limit these logs and to allow a boot-time flag to toggle the rate of the log data. The change does not affect the security of the Linux-RNG.
- The change introduces a flag crng_global_init_time which is used to trigger a reseed of each ChaCha20 DRNG instance if the newly added IOCTL of RNDRESEEDCRNG is invoked. The new IOCTL is documented in section 3.4.1.3. The change does not affect the security of the Linux-RNG.
- Remove of the unused entropy pool state variable dont_count_entropy. The change does not affect the security of the Linux-RNG.

9.8.1.2 File include/linux/random.h

The following changes are visible:

- The change log of the change reads: Always fill buffer in get_random_bytes_wait. In the unfortunate event that a developer fails to check the return value of get_random_bytes_wait, or simply wants to make a "best effort" attempt, for whatever that's worth, it's much better to still fill the buffer with _something_ rather than catastrophically failing in the case of an interruption. This is both a defense-in-depth measure against inevitable programming bugs, as well as a means of making the API a bit more useful. The change does not affect the security of the Linux-RNG.

9.8.1.3 File include/uapi/linux/random.h

The following changes are visible:

- The definition of the new IOCTL of RNDRESEEDCRNG is added. The change does not affect the security of the Linux-RNG.

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

No changes.

9.8.2 Changes to Invocation of Entropy Gathering Functions

9.8.2.1 add_input_randomness

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

9.8.2.2 add_interrupt_randomness

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

9.8.2.3 add_disk_randomness

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

9.8.2.4 add_hwgenerator_randomness

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

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

9.8.3 Definition and Use of new Interfaces

No functions implemented in random.c are add by means of any of the EXPORT_SYMBOL*() macros. Therefore, no new interfaces are exported.

However, a new IOCTL is added as discussed above.

9.9 Linux Kernel 4.18

Previously assessed Linux kernel version: 4.17 – https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.17.tar.xz

Currently assessed Linux kernel version: 4.18 – https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.18.tar.xz

Assessment of changes: Changes do not affect the functionality of the Linux-RNG. The NTG.1 properties of /dev/random are not affected by the changes.

The following subsections contain the assessment of potentially relevant changes.

9.9.1 Changes to the Linux-RNG Implementation

9.9.1.1 File drivers/char/random.c

The diff of the file shows one change:

- The Linux-RNG allows inserting data with the IOCTL of RNDADDENTROPY as documented in section 3.4.1.3. The handling function obtaining the data from user space is write_random. This function is changed to XOR each 32-bit word of the user-provided input data with 32 bits obtained from the RDRAND CPU instruction. The cause of the change was the addition of the Jitter RNG to the Linux user space daemon of rndg whose purpose is to provide additional entropy. The Linux-RNG author publicly claimed that he did not trust the Jitter RNG output.

9.9.1.2 File include/linux/random.h

No changes.

9.9.1.3 File include/uapi/linux/random.h

No changes.

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

No changes.

9.9.2 Changes to Invocation of Entropy Gathering Functions

9.9.2.1 add_input_randomness

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

9.9.2.2 add_interrupt_randomness

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

9.9.2.3 add_disk_randomness

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

9.9.2.4 add_hwgenerator_randomness

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

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

9.9.3 Definition and Use of new Interfaces

No functions implemented in random.c are add by means of any of the EXPORT_SYMBOL*() macros. Therefore, no new interfaces are exported.

9.10 Linux Kernel 4.19

Previously assessed Linux kernel version: 4.18 – https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.18.tar.xz

Currently assessed Linux kernel version: 4.19 – https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.19.tar.xz

Assessment of changes: Changes do affect the functionality of the Linux-RNG with respect to the seeding process of the ChaCha20 DRNG supporting /dev/urandom and the get_random_bytes in-kernel interface. The NTG.1 properties of /dev/random are not affected by the changes. However, the DRG.3 properties of the ChaCha20 DRNG are affec ted.

The following subsections contain the assessment of potentially relevant changes.

9.10.1 Changes to the Linux-RNG Implementation

9.10.1.1 File drivers/char/random.c

The diff of the file shows one change:

- The initial seeding process of the ChaCha20 DRNG has been altered. The CPU-based noise sources may now be considered “trusted” by the Linux-RNG causing the ChaCha20 to be fully initialized at the time the ChaCha20 DRNG state is initialized. Details are given in section 3.3.2.3. This affects the entropy estimate of a newly initialized ChaCha20 DRNG. As it has an impact on the entropy estimation for the ChaCha20 DRNG, it also has an impact on the DRG.3 properties of the ChaCha20 DRNG. Therefore, section 5.2.1 has been updated.
- The Linux kernel received a new kernel command line argument “random.trust_cpu”. This argument allows toggling the default trust into the CPU-based noise sources at boot time. This relates to the aforementioned initialization of the ChaCha20 DRNG and therefore that security implication statement applies here as well.
- Locking in add_timer_randomness() has been updated. This does no affect the Linux-RNG behavior.

9.10.1.2 File include/linux/random.h

The diff shows the addition of the function prototypes added with the changes discussed for random.c above. This implies that these changes do not affect the Linux-RNG behavior.

9.10.1.3 File include/uapi/linux/random.h

No changes.

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

No changes.

9.10.2 Changes to Invocation of Entropy Gathering Functions

9.10.2.1 add_input_randomness

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

9.10.2.2 add_interrupt_randomness

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

A function call to add_interrupt_randomness is added to the function of `hv_stimer0_isr` part of the Microsoft Hyper-V code. This function is triggered when the stimer0 is operating in Direct Mode. Direct Mode does not use the VMbus or any VMbus messages that would trigger `vmbus_isr`. Hence, the addition is considered appropriate as an event triggered by Hyper-V is detected.

9.10.2.3 add_disk_randomness

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

9.10.2.4 add_hwgenerator_randomness

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

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

9.10.3 Definition and Use of new Interfaces

No functions implemented in random.c are add by means of any of the EXPORT_SYMBOL*() macros. Therefore, no new interfaces are exported.

9.11 Linux Kernel 4.20

Previously assessed Linux kernel version: 4.19 – https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.19.tar.xz

Currently assessed Linux kernel version: 4.20 – https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.20.tar.xz

Assessment of changes: Changes do not affect the functionality of the Linux-RNG. The NTG.1 properties of /dev/random are not affected by the changes.

The following subsections contain the assessment of potentially relevant changes.

9.11.1 Changes to the Linux-RNG Implementation

9.11.1.1 File drivers/char/random.c

The diff of the file shows one change:

- The ChaCha20 block operation interface function change applied to 4.16 has been reverted. The ChaCha20 block operation again works on byte boundaries, not requiring word-aligned memory. Thus the Linux-RNG buffer used to invoke the ChaCha20 block operation has been changed from a word-array to a byte-array. The change does not affect the security of the Linux-RNG.

9.11.1.2 File include/linux/random.h

No changes.

9.11.1.3 File include/uapi/linux/random.h

No changes.

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

No changes.

9.11.2 Changes to Invocation of Entropy Gathering Functions

9.11.2.1 add_input_randomness

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

9.11.2.2 add_interrupt_randomness

No change to the invocation in `handle_irq_event_percpu`, `vmbus_isr` and `hv_stimer0_isr` and thus no effect on the Linux-RNG.

9.11.2.3 add_disk_randomness

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

9.11.2.4 add_hwgenerator_randomness

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

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

9.11.3 Definition and Use of new Interfaces

No functions implemented in random.c are add by means of any of the EXPORT_SYMBOL*() macros. Therefore, no new interfaces are exported.

9.12 Linux Kernel 5.0

Previously assessed Linux kernel version: 4.20 – https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.20.tar.xz

Currently assessed Linux kernel version: 5.0 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.0.tar.xz

Assessment of changes: Changes do not affect the functionality of the Linux-RNG. The NTG.1 properties of /dev/random are not affected by the changes.

The following subsections contain the assessment of potentially relevant changes.

9.12.1 Changes to the Linux-RNG Implementation

9.12.1.1 File drivers/char/random.c

The diff of the file shows one change:

- The ChaCha20 block operation interface function has remained functionally unchanged. Compile-time constants (CHACHA20_BLOCK_SIZE and CHACHA20_KEY_SIZE) have been renamed to omit the "20" in their name. The same applies for the chacha20.h header file name.

9.12.1.2 File include/linux/random.h

No changes.

9.12.1.3 File include/uapi/linux/random.h

No changes.

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

No changes.

9.12.2 Changes to Invocation of Entropy Gathering Functions

9.12.2.1 add_input_randomness

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

9.12.2.2 add_interrupt_randomness

No change to the invocation in `handle_irq_event_percpu`, `vmbus_isr` and `hv_stimer0_isr` and thus no effect on the Linux-RNG.

9.12.2.3 add_disk_randomness

The Linux kernel version 5.0 has undergone some changes in the block layer that have caused the call to add_disk_randomness to be removed from `blk_update_bidi_request`. The call of the function from `scsi_end_request` has remained unchanged. This implies that less disk events are detected and processed by the Linux-RNG with this change.

The conditions under which the `add_disk_randomness` function is called (such as the exclusion of storage devices that are not suitable for gathering entropy from timing variance measurements) have not changed, and therefore do not affect the Linux-RNG.

9.12.2.4 add_hwgenerator_randomness

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

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

9.12.3 Definition and Use of new Interfaces

No functions implemented in random.c are added by means of any of the EXPORT_SYMBOL*() macros. Therefore, no new interfaces are exported.

9.13 Linux Kernel 5.1

Previously assessed Linux kernel version: 5.0 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.0.tar.xz

Currently assessed Linux kernel version: 5.1– https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.1.tar.xz

Assessment of changes: Changes do not affect the functionality of the Linux-RNG. The NTG.1 properties of /dev/random are not affected by the changes.

The following subsections contain the assessment of potentially relevant changes.

9.13.1 Changes to the Linux-RNG Implementation

9.13.1.1 File drivers/char/random.c

No changes.

9.13.1.2 File include/linux/random.h

No changes.

9.13.1.3 File include/uapi/linux/random.h

No changes.

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

No changes.

9.13.2 Changes to Invocation of Entropy Gathering Functions

9.13.2.1 add_input_randomness

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

9.13.2.2 add_interrupt_randomness

No change to the invocation in `handle_irq_event_percpu`, `vmbus_isr` and `hv_stimer0_isr` and thus no effect on the Linux-RNG.

9.13.2.3 add_disk_randomness

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

9.13.2.4 add_hwgenerator_randomness

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

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

9.13.3 Definition and Use of new Interfaces

No functions implemented in random.c are added by means of any of the EXPORT_SYMBOL*() macros. Therefore, no new interfaces are exported.

9.14 Linux Kernel 5.2

Previously assessed Linux kernel version: 5.1 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.1.tar.xz

Currently assessed Linux kernel version: 5.2 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.2.tar.xz

Assessment of changes: Changes do not affect the functionality of the Linux-RNG. The NTG.1 properties of /dev/random are not affected by the changes.

The following subsections contain the assessment of potentially relevant changes.

9.14.1 Changes to the Linux-RNG Implementation

9.14.1.1 File drivers/char/random.c

The diff of the file shows one change:

- C code comments have been added to (partially) document the Linux-RNG interfaces. The change does not affect the security of the Linux-RNG.
- A set of small code style fixes have been applied: Remove an unused variable, constify a data structure that is only read and make a global variable only visible to the code in the file random.c. The change does not affect the security of the Linux-RNG.
- A code change has been added to only read from /dev/random after its pool has received 128 bits. To implement that change, a variable has been re-purposed which implies that the code using that variable previously required a slight update. Yet, apart from the supply of data after 128 bits have been received, no cryptographically visible changes are present. The change does not affect the security of the Linux-RNG.
- A fix to the ChaCha20 RNG initialization has been applied when trusting the CPU-based noise data. Previously, in this case the NUMA RNG instances were not initialized and thus the per-NUMA-node ChaCha20 RNG was not present. The change does not affect the security of the Linux-RNG.
- Invoke the rand_initialize function of the Linux RNG earlier in the boot process to ensure that the stack canary initialization benefits from the initialized Linux-RNG. The change does not affect the security of the Linux-RNG.
- Fix deadlock when using an auxiliary interface provided by the Linux-RNG (the data generated by this interface is not subject to assessment in this document). The change does not affect the security of the Linux-RNG.
- Fix a soft-lockup when trying to read from an uninitialized blocking pool. The change does not affect the security of the Linux-RNG.

9.14.1.2 File include/linux/random.h

The function declaration of the Linux RNG initialization for the early boot-process initialization change has been added. The change does not affect the security of the Linux-RNG.

9.14.1.3 File include/uapi/linux/random.h

No changes.

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

No changes.

9.14.2 Changes to Invocation of Entropy Gathering Functions

9.14.2.1 add_input_randomness

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

9.14.2.2 add_interrupt_randomness

No change to the invocation in `handle_irq_event_percpu`, `vmbus_isr` and `hv_stimer0_isr` and thus no effect on the Linux-RNG.

9.14.2.3 add_disk_randomness

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

9.14.2.4 add_hwgenerator_randomness

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

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

9.14.3 Definition and Use of new Interfaces

No functions implemented in random.c are added by means of any of the EXPORT_SYMBOL*() macros. Therefore, no new interfaces are exported.

9.15 Linux Kernel 5.3

Previously assessed Linux kernel version: 5.2 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.2.tar.xz

Currently assessed Linux kernel version: 5.3 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.3.tar.xz

Assessment of changes: No changes.

9.15.1 Changes to the Linux-RNG Implementation

9.15.1.1 File drivers/char/random.c

No changes.

9.15.1.2 File include/linux/random.h

No changes.

9.15.1.3 File include/uapi/linux/random.h

No changes.

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

No changes.

9.15.2 Changes to Invocation of Entropy Gathering Functions

9.15.2.1 add_input_randomness

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

9.15.2.2 add_interrupt_randomness

No change to the invocation in `handle_irq_event_percpu`, `vmbus_isr` and `hv_stimer0_isr` and thus no effect on the Linux-RNG.

9.15.2.3 add_disk_randomness

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

9.15.2.4 add_hwgenerator_randomness

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

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

9.15.3 Definition and Use of new Interfaces

No functions implemented in random.c are added by means of any of the EXPORT_SYMBOL*() macros. Therefore, no new interfaces are exported.

9.16 Linux Kernel 5.4

Previously assessed Linux kernel version: 5.3 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.3.tar.xz

Currently assessed Linux kernel version: 5.4 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.4.tar.xz

Assessment of changes: Changes are applied that do affect the functionality of the Linux-RNG entropy collection. The NTG.1 properties of /dev/random are not affected by the changes unless the newly added Linux kernel configuration option CONFIG_RANDOM_TRUST_BOOTLOADER is set to "Y".

9.16.1 Changes to the Linux-RNG Implementation

9.16.1.1 File drivers/char/random.c

The following changes are applied to random.c:

- The implementation gained a new noise source which is triggered when a caller requests random numbers via either the getrandom(2) system call or via the in-kernel service function wait_for_random_bytes. The new entropy source is a latch of two timer: one low-resolution timer operating at 1000 Hz and one high-resolution timer operating in the nanosecond range. The high-resolution time stamp is taken in a tight loop which is interrupted by a rescheduling event. When the low-resolution timer expires, the current value of the high-resolution timer is sampled and injected into the entropy pool. At the time the high-resolution time stamp is injected into the entropy pool and the entropy estimator is increased by one (bit of entropy). The author of this document already played with such intermix of two different clocks for a long time as part of the Jitter RNG development. Based on this work, a user-space equivalent of the noise source is already present. This user space noise source implementation was slightly updated to match the in-kernel equivalent. When executing and processing the raw output data with the SP800-90B min entropy tool set, about 1.6 bits of entropy per high-resolution time stamp sampled with the expiry of the low-resolution time stamp on an x86 test system. Although this is an initial make-shift test which requires more in-depth study, the new noise source seems to deliver the claimed entropy rate. Note, this new noise source is deactivated if no high-resolution timer is identified.

  The new noise source is only applied during boot time when the ChaCha20 DRNG is not yet fully seeded. The new noise source only generates data until the ChaCha20 DRNG is fully seeded. This implies that the noise source data mixed into the entropy pool and the associated increase in the entropy estimator is immediately "consumed" by the ChaCha20 DRNG. Thus, this new entropy is not available to /dev/random.

  As only the getrandom(2) system call and its in-kernel equivalent are affected, /dev/random and its NTG.1 property are not affected by this.

- The code gained a new interface allowing a boot loader to directly inject data into the entropy pool. This data is processed with the add_device_randomness function discussed in the design above. Therefore, the data from the boot loader is not awarded any entropy. However, if the kernel is compiled with the compilation option of CONFIG_RANDOM_TRUST_BOOTLOADER, the data is processed with the add_hwgenerator_randomness function discussed in the design above causing the entropy estimator of the entropy pool to be increased by the number of bits injected by the boot loader. In this case, the data from the boot loader is assumed to bear full entropy.

  If CONFIG_RANDOM_TRUST_BOOTLOADER is not set, the NTG.1 property of /dev/random is unaffected. If, however, this option is set to "Y" then the boot loader acts as an entropy-provider. In this case, the data must be analyzed for its entropy content to preserve the NTG.1 claim. As the boot loader is outside of the scope of this document, its data potentially provided to the Linux /dev/random cannot be assessed here.

9.16.1.2 File include/linux/random.h

This file adds the declaration of the new boot loader interface function.

9.16.1.3 File include/uapi/linux/random.h

No changes.

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

No changes.

9.16.2 Changes to Invocation of Entropy Gathering Functions

9.16.2.1 add_input_randomness

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

9.16.2.2 add_interrupt_randomness

No change to the invocation in `handle_irq_event_percpu`, `vmbus_isr` and `hv_stimer0_isr` and thus no effect on the Linux-RNG.

9.16.2.3 add_disk_randomness

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

9.16.2.4 add_hwgenerator_randomness

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

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

The function is also called by the boot loader interface function discussed in section 9.16.1.1. This section outlines the impact on the NTG.1 claim.

9.16.3 Definition and Use of new Interfaces

A new interface function of add_bootloader_randomness is added. This is a wrapper to either add_device_randomness or add_hwgenerator_randomness and is therefore not considered a stand-alone interface in itself. Further details are provided in section 9.16.1.1.

9.17 Linux Kernel 5.5

Previously assessed Linux kernel version: 5.4 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.4.tar.xz

Currently assessed Linux kernel version: 5.5 – https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.5.tar.xz

Assessment of changes: All changes do not affect the NTG.1 property. Yet, the configuration option CONFIG_RANDOM_TRUST_BOOTLOADER is present and its setting has an impact on the NTG.1 property as discussed in section 9.16.

9.17.1 Changes to the Linux-RNG Implementation

9.17.1.1 File drivers/char/random.c

The following changes are applied to random.c:

- The VFS layer of the kernel now requires that the “file system operations” function pointer data structure contains a function for the “compat_ioctl” entry. This function is used in case a 32-bit user space application is executed on a 64-bit kernel and wants to perform an IOCTL system call on a file associated with the file system operations function pointer data structure. Thus, the /dev/random and /dev/urandom file system operations function pointer data received such an entry. The registered generic function unconditionally will invoke the IOCTL handling function documented in section 3.4.1.3. This implies that for the new kernel, the 32-bit user space executing on a 64-bit kernel invokes the same IOCTL handler as it used to before. Thus, this update to the Linux-RNG code base does not introduce any functional change to the Linux-RNG.

9.17.1.2 File include/linux/random.h

No changes

9.17.1.3 File include/uapi/linux/random.h

No changes.

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

No changes.

9.17.2 Changes to Invocation of Entropy Gathering Functions

9.17.2.1 add_input_randomness

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

9.17.2.2 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.

9.17.2.3 add_disk_randomness

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

9.17.2.4 add_hwgenerator_randomness

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

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

The function is also called by the boot loader interface function discussed in section 9.16.1.1. This section outlines the impact on the NTG.1 claim.

9.17.3 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 using `printk`
- `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.

For the testing performed for this study, the SystemTap mechanism has been chosen as this mechanism covers all aforementioned concerns, is easy to use and is automatable.

Early Boot Test

Even though it was stated above that applying patches and recompiling a kernel is less useful, this approach was chosen for the early boot tests. As the test's intent is to obtain data very early in the kernel boot sequence, it is impossible to use a tracing mechanism that relies on certain commands to start. At the time user space is available, the information to be obtained is long gone.

Thus, the only way for obtaining the intended measurements is to apply the patches to a kernel, re-compile, install and boot it.

SystemTap Test Approach

SystemTap provides a kernel infrastructure to facilitate the collection of data from the runtime Linux kernel environment. When using SystemTap, the developer does not need to instrument the Linux kernel with proprietary code, recompile and reboot the system.

To use SystemTap, a command line program is provided that allows the caller to execute a SystemTap script. SystemTap scripts are written in a high-level scripting language to allow for developing and building an instrumentation of the currently executing Linux kernel. Conceptually, SystemTap can be regarded as a programmable debugger.

The SystemTap command line program converts the script into a kernel module which contains additional service functions. Part of these service functions is the ability to export data from kernel space to user space using the DebugFS. This technical detail is hidden by the SystemTap command line application such that the data exported via DebugFS from the SystemTap script can be accessed like any other data on the command line via STDOUT or by writing the data to a file.

SystemTap scripts consist of functions which are invoked when pre-defined events occur. Such events can be:

- Invocation of or returning from a kernel function
- Expiration of a timer-based alarm

The SystemTap infrastructure is part of the assessed Linux kernel.

Measurement Errors with SystemTap

The following problems must be considered when using SystemTap as they may cause measurement errors.

The kernel typically executes on a multi-processor hardware. This means that it is always possible that code sequences in the Linux-RNG are executed in parallel. SystemTap can be used to respond to this parallelism. This implies that the test scripts are always prone to race conditions[24]. In particular, such race conditions are present when the SystemTap script has to examine the source of an invocation of the Linux-RNG code. Considering the Linux-RNG architecture it is clear that the core function implementing the LFSR operation or the modification of an entropy estimator is invoked by noise sources as well as when data is transferred from one entropy pool to another.

Assume the following test sequence that is present in test scripts:

- An event invokes `add_input_randomness`. The test script sets a global status flag to 1 to mark that a relevant event happened.
- In a next step, the LFSR mixing function is invoked. The test script uses the flag set in step 1 to enable the core measurement logic.
- The global variable is set back to zero.

If between steps 1 and 2 a read on, say, /dev/random during normal operation happens where the global variable is set, the LFSR mix operation is invoked to satisfy the /dev/random read operation. But the test script thinks the monitored operation covers the mix-in of new data from the noise source. This conflict causes invalid measurements.

24 A race condition is the uncontrolled modification of a variable when a non-atomic operation accesses this variable by multiple entities. Non-atomic operations have dissimilar times-of-check and times-of-use. This means that after one entity's code completed a check on a variable content but before the content is used, another entity may alter the content. This implies that the first entity's check operation is now invalid without the entity knowing that.

It is also possible that before reaching step 3 to reset the global variable, another LFSR operation is triggered in addition to the monitored one. This implies that now two LFSR operations are recorded.

Such errors are easily detectable during tests conducted in section 6.2. These tests record the event values for each hardware event. For example, the block device event value that was recorded is "8388880" which is the device number of the only disk device on the test system. This number would not be an HID event number. Thus, if an HID event number shows up in the block device records or vise versa, it is clearly identifiable. The obtained test records did not show any such errors. This implies that the likelihood for such errors to occur is very small and is not considered to affect the conclusions drawn from the measurements.

Prerequisite for SystemTap Use

To use SystemTap, the following prerequisites must be met:

- The kernel source code must be accessible for the SystemTap compilation. Commonly, Linux distributions provide so-called "development" packages:
  - Ubuntu: linux-headers-generic
  - Red Hat: kernel-devel
  - SUSE: kernel-*-devel (for kernels that are unchanged to the upstream kernel.org code base, use kernel-vanilla-devel)
  - For self-compiled kernels, the source code must be accessible under /lib/modules/$(uname -r)/build/
- The debug symbols of the running kernel must be installed. Linux distributions provide so-called "debuginfo" packages – it may be possible that additional repositories need to be activated before these packages with the debug symbols are accessible (please see the distribution-specific guidance):
  - Ubuntu: linux-image-$(uname -r)-dbgsym
  - Red Hat: kernel-debuginfo
  - SUSE: kernel-*-debuginfo (for kernels that are unchanged to the upstream kernel.org code base, use kernel-vanilla-debuginfo)
  - For self-compiled kernels, the kernel configuration option CONFIG_DEBUG_INFO must be set to "y". This option can be found under "Kernel Hacking" → "Compile the kernel with debug info (DEBUG_INFO) [N/y/?]" → "Kernel debugging (DEBUG_KERNEL) [Y/n/?]"
- The binaries and libraries forming the user space part of SystemTap must be installed. These software components can be obtained either from the SystemTap Homepage or by installing the following packages:
  - Ubuntu: systemtap
  - Red Hat: stap
  - SLES: systemtap, systemtap-server

Impact of Measurement on Test Results

The SystemTap scripts 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 explain which commands are to be executed to collect the required data. With some of these commands, SystemTap scripts are compiled and loaded.

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 6.3.2:
    - 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:
    - 2 virtual CPUs corresponding with the 2 cores of the host system
    - Intel Core Processor (Broadwell, no TSX)

Appendix B: Test Results on Linux Kernel 4.9

All tests shown in chapters 6 and following have been executed on newer kernels. To allow the reader to compare the test results, this appendix contains the corresponding test results for the kernel 4.9.

If test results discussed in chapters 6 and following are not found in this appendix for kernel 4.9, the test showed the same results. Thus the test discussion in the respective sections applies to the version 4.9 as well.

Testing of the Linux kernel 4.9 was performed on the following environment:

- Thinkpad T530 used for the native hardware early boot entropy tests documented in 6.3.2:
  - 2 core CPU with two hyperthreads per core
  - Intel(R) Core(TM) i7-3520M CPU @ 2.90GHz
- QEMU 2.7.1 used for the virtual environment early boot entropy tests documented in 6.3.1:
  - 4 virtual CPUs corresponding with the 4 hyperthreads of the host system
  - Intel Core Processor (Haswell, no TSX)
- Thinkpad T540p used for all other tests executed on native hardware:
  - 2 core CPU with two hyperthreads per core
  - Intel(R) Core(TM) i7-4600M CPU @ 2.9GHz

Min-Entropy as per SP800-90B

Interrupt Noise Source Min-Entropy Estimates

The collection of data for interrupts was conducted twice: once with a normal use case and once with a worst-case. In the normal use case the test environment was made to resemble regular usage where Internet searches and regular office duties were performed. The worst-case covered the test system in a virtual environment where the host system sent a ping flood to the test system. Each received ICMP request and response triggered an interrupt that was recorded.

The worst-case test execution returned the following data.

Entropy Estimate	4 Bits Width	8 Bits Width
Most Common Value Estimate	3.965	7.883
Collision Estimate	2.944	8
Markov Estimate	3.612	N/A
Compression Estimate	2.748	7.575
t-Typle Estimate	3.181	7.342
LRS Estimate	2.822	7.773
Multi Most Common in Window Prediction Estimate	3.892	7.963
Lag Prediction Estimate	3.180	6.642
MultiMMC Prediction Estimate	3.180	7.631
LZ78Y Prediction Estimate	2.820	7.632

Table 8: Interrupts: SP800-90B min-entropy Measurements - Worst Case
The associated Shannon entropy value is 14.602 bits per interrupt event. The min-entropy value according to [AIS2031] is 11.665 bits per interrupt event.

The normal use case returned the following data.

Entropy Estimate	4 Bits Width	8 Bits Width
Most Common Value Estimate	3.956	7.870
Collision Estimate	1.980	4.406
Markov Estimate	3.289	N/A
Compression Estimate	2.032	4.630
t-Typle Estimate	3.041	7.158
LRS Estimate	2.958	7.308
Multi Most Common in Window Prediction Estimate	3.180	7.660
Lag Prediction Estimate	2.293	7.274
MultiMMC Prediction Estimate	2.293	6.642
LZ78Y Prediction Estimate	3.180	6.642

Table 9: Interrupts: SP800-90B Min-Entropy Measurements - Normal Use Case
Applying the Shannon entropy formula on the data set, a value of 19.204 bits per interrupt event is calculated. Using the min-entropy formula according to [AIS2031], a result of 12.111 bits per interrupt event is measured.

The conclusions that can be drawn from the numbers follow. Regardless of the worst-case or normal case, the high-resolution time stamp of each interrupt will return significantly more than two 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 (two bits when `RDRAND` is present). 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.

Block Device Noise Source Min-Entropy Estimates

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 mounting and unmounting a block device. This causes the buffer cache to be irrelevant for the disk accesses caused by the mount operations, as the buffer cache is flushed with each unmount operation of a file system. The worst-case produced the following data:

Entropy Estimate	4 Bits Width	8 Bits Width
Most Common Value Estimate	3.967	7.890
Collision Estimate	2.728	5.570
Markov Estimate	3.768	N/A
Compression Estimate	2.810	5.883
t-Typle Estimate	3.360	7.342
Longest Repeated Substring (LRS) Estimate	3.260	7.773
Multi Most Common in Window Prediction Estimate	3.180	7.889
Lag Prediction Estimate	3.180	6.642
MultiMMC Prediction Estimate	3.634	7.860
LZ78Y Prediction Estimate	3.634	7.862

Table 10: Block Devices: SP800-90B Min-Entropy Measurements
Using the Shannon entropy formula, 17.683 bits per block device event is calculated. A value of 13.456 bits per block device event is calculated as the min-entropy according to [AIS2031].

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 17 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 17 also shows that the mean value of all entropy estimates is 0.21 bits of entropy. This can be interpreted that on average, the Linux-RNG awarded each block device event 0.21 bits of entropy.

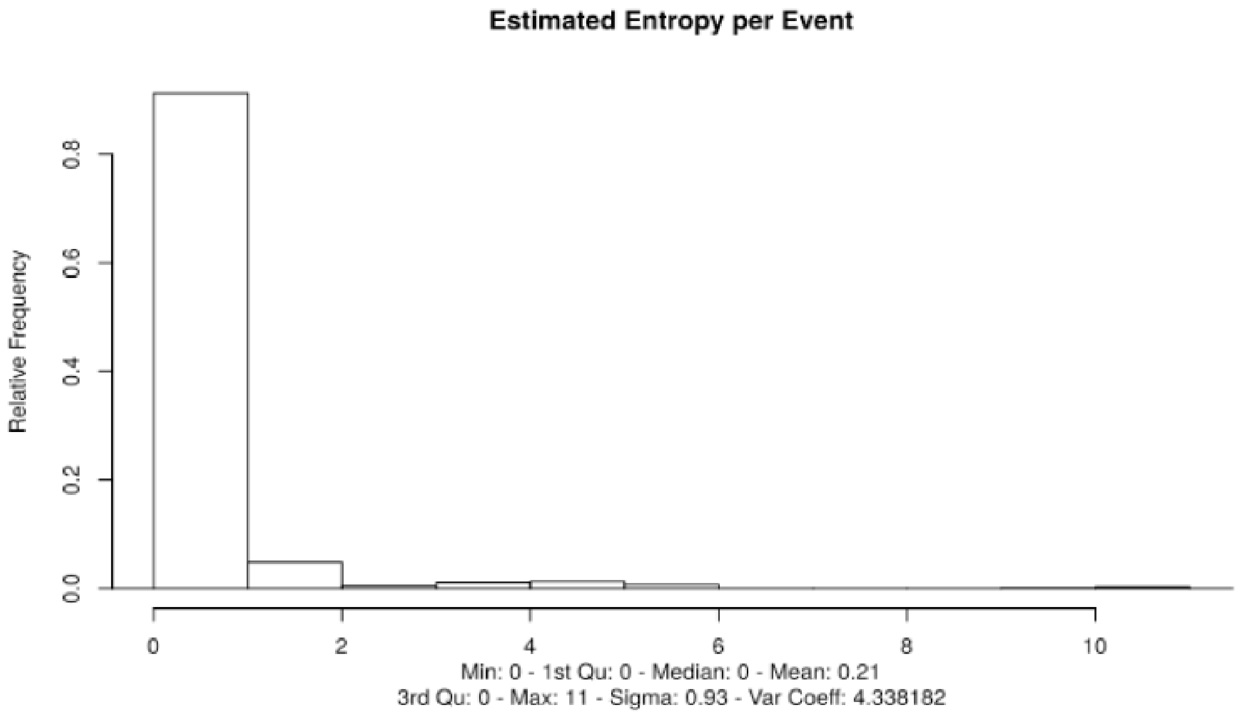

Figure 17: Entropy Estimate per Block Device Event Applied by Linux-RNG

Comparing the result shown in figure 17 with the min-entropy estimates calculated from the measured time stamps, the following conclusion is drawn: the min-entropy estimates have significantly more than 2 bits of entropy per event. On the other hand, the Linux-RNG considers that each event has on average only 0.21 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.

HID Noise Source Min-Entropy Estimates

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, only 200.000 samples of HID noise source events were recorded. The entropy estimates for the high-resolution time stamp applied to those events are listed in the table below.

Entropy Estimate	4 Bits Width	8 Bits Width
Most Common Value Estimate	3.910	7.729
Collision Estimate	1.856	4.137
Markov Estimate	3.183	N/A
Compression Estimate	1.809	4.045
t-Typle Estimate	3.207	7.850
LRS Estimate	2.863	7.512

Entropy Estimate	4 Bits Width	8 Bits Width
Multi Most Common in Window Prediction Estimate	3.300	7.388
Lag Prediction Estimate	2.553	7.069
MultiMMC Prediction Estimate	2.553	7.364
LZ78Y Prediction Estimate	3.288	7.364

Table 11: HID: SP800-90B Min-Entropy Measurements
The Shannon entropy formula applied on the data set results in 15.645 bits per HID event. 10.063 bits per HID event are calculated when using the min-entropy formula according to [AIS2031].

The test record of the entropy estimate applied by the Linux-RNG for each recorded HID event is depicted with figure 18. 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 18, the mean value of the histogram is 1.29 bits. This implies that the Linux-RNG awarded 1.29 bits of entropy to each HID noise source event on average.

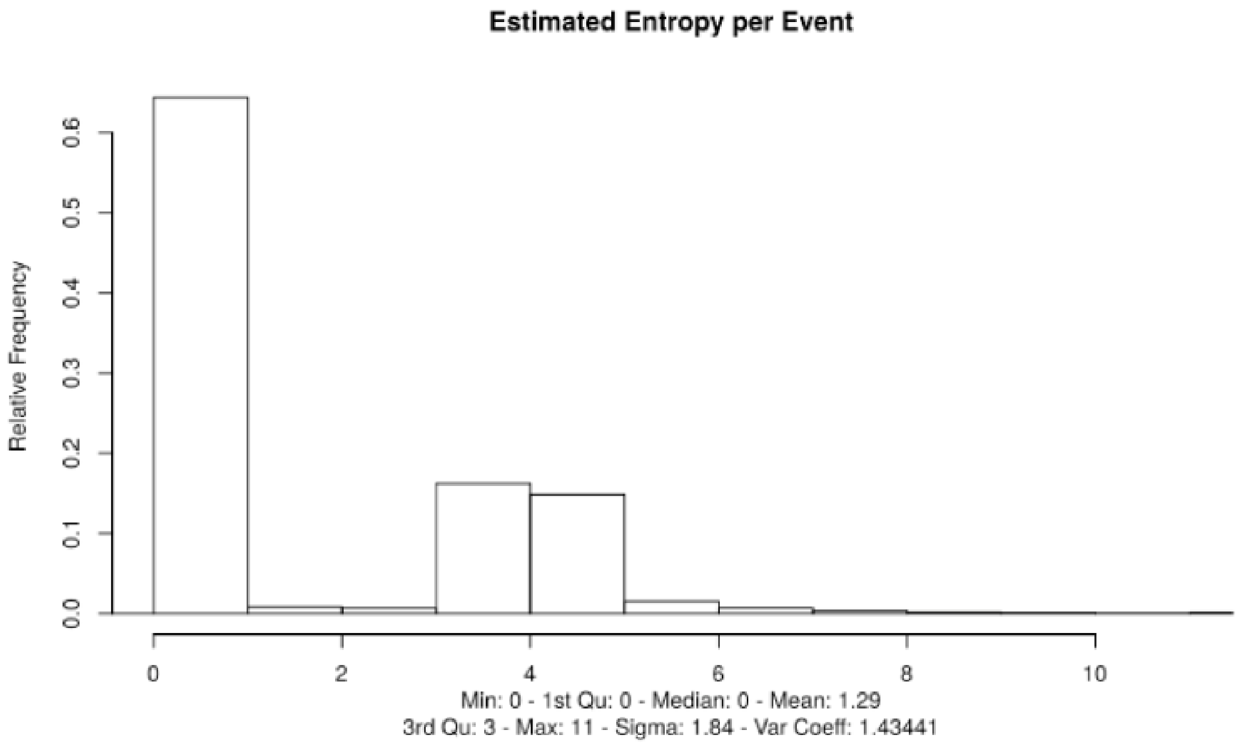

Figure 18: Entropy Estimate per HID Event Applied by Linux-RNG

A conclusion can be reached when comparing the heuristic entropy values applied by the Linux-RNG from figure 18 with the min-entropy estimates. The min-entropy estimates have more than 2 bits of entropy per event (8 bit width), or more than 1.8 bits of entropy per event (4 bits width). On the other hand, the Linux-RNG applies on average 1.29 bits of entropy to a HID event.

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.

Entropy During Early Boot

This appendix applies to the test outlined in section 6.3 for kernel 4.9.

The test is performed for 50,000 reboot cycles. At the end of the testing, 50,000 times 128 time stamps are collected and analyzed.

Early Boot Entropy Testing in a Virtual Environment

This appendix applies to the test outlined in section Fehler: Verweis nicht gefunden for kernel 4.9.

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
1	3.30173	5.09539	13.288	15.388
2	3.30173	5.09539	13.288	15.429
3	3.24271	5.05406	13.61	15.526
4	3.03697	5.09062	13.61	15.541
5	2.94828	5.17785	14.025	15.552
6	3.11278	5.18062	13.61	15.498
7	2.96301	5.16851	13.288	15.536
8	3.09998	4.97885	13.61	15.553
9	2.89757	5.1643	12.61	15.482
10	3.48	5.17752	11.086	14.683
11	3.02415	5.13717	10.909	14.452
12	3.49016	5.10904	10.655	14.143
13	3.18302	5.14208	9.937	13.6
14	3.26001	5.16127	9.937	13.515
15	3.02617	5.12335	11.15	15.023
16	2.92302	5.17071	12.802	15.526
17	2.9831	5.12957	12.802	15.482
18	3.25527	5.06403	12.15	15.474
19	3.04512	5.17675	13.288	15.565
20	3.05607	5.14117	14.025	15.602
21	3.32719	5.18943	13.025	15.51
22	3.04036	5.17868	12.288	15.194

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
23	3.05264	5.05886	12.025	14.941
24	3.06226	5.10445	12.15	15.059
25	3.13945	5.11868	12.288	15.072
26	3.12209	5.16643	11.522	14.95
27	3.14601	5.12833	11.61	14.898
28	2.99096	5.14291	11.703	14.956
29	3.36326	5.07632	12.15	15.175
30	3.1681	5.12096	12.61	15.321
31	2.86473	5.0967	12.288	15.28
32	3.14747	5.10998	12.288	15.167
33	2.99754	5.12061	12.288	15.192
34	3.07821	5.02638	12.288	15.155
35	3.38898	4.99506	12.61	15.2
36	3.13147	5.17215	12.025	15.17
37	2.94192	5.14449	12.15	15.054
38	2.90004	5.15256	11.802	14.993
39	3.27431	5.17919	11.44	14.929
40	3.07125	5.087	11.522	14.924
41	3.03156	5.15921	12.025	14.939
42	3.13509	5.15989	11.909	14.95
43	3.25685	5.11564	11.362	14.954
44	2.93244	5.12911	11.217	14.952
45	3.29685	5.18695	11.61	14.957
46	2.97205	5.16866	11.522	14.956
47	3.37692	5.15097	11.44	14.972
48	3.54018	5.14872	11.61	14.986
49	2.94066	5.16365	11.522	15.011
50	3.09786	5.12336	11.522	15.015
51	3.14017	5.07814	11.217	15.009
52	2.99162	5.13659	11.703	14.99
53	2.93623	5.08187	11.362	14.983
54	3.15187	5.11192	11.362	14.96
55	2.88776	5.12315	11.61	14.958

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
56	2.96044	5.17851	11.909	14.979
57	3.54982	5.13802	12.025	14.989
58	3.02013	5.13341	11.909	14.929
59	3.38898	5.01221	12.025	14.928
60	3.09715	5.12272	11.909	14.935
61	2.96301	5.09843	11.61	14.888
62	3.30662	5.15366	11.703	14.863
63	2.92239	5.17039	11.703	14.869
64	3.03697	5.07835	11.61	14.862
65	2.96237	5.08455	11.44	14.85
66	3.19506	5.0043	11.909	14.893
67	3.34134	5.17747	11.802	14.944
68	3.01411	5.12867	11.522	14.737
69	3.11135	5.15828	11.025	14.45
70	3.12281	5.07231	11.15	14.54
71	3.30499	5.15676	11.802	14.807
72	3.15628	5.12891	11.44	14.925
73	3.03223	5.13207	11.703	14.908
74	3.22717	5.15972	11.288	14.876
75	2.98899	5.03567	11.61	14.857
76	2.94192	5.15561	11.703	14.844
77	2.92302	5.1662	11.44	14.84
78	3.23648	5.11391	11.61	14.896
79	3.088	5.06645	11.909	14.929
80	3.05195	5.06598	11.362	14.645
81	3.20416	5.12219	11.025	14.268
82	3.36241	5.15981	11.086	14.558
83	2.99227	5.02146	11.61	15.099
84	3.19733	5.13076	12.44	15.371
85	2.94892	5.16468	12.288	15.203
86	3.32553	5.19812	11.802	14.976
87	2.85216	5.11442	11.44	14.983
88	3.01211	5.1312	12.025	14.99

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
89	3.13219	4.98788	12.025	15.047
90	3.07751	5.03714	11.909	14.949
91	3.59439	5.09877	11.909	14.912
92	3.17852	5.14809	11.909	14.936
93	3.08239	5.15001	11.909	14.978
94	3.04717	5.224	11.802	14.896
95	3.11635	5.13731	11.802	14.923
96	3.02281	5.17201	12.025	14.944
97	3.04649	5.16605	11.909	14.948
98	3.52732	5.14062	11.909	14.897
99	3.37863	5.09191	11.909	14.972
100	3.12497	5.15368	11.61	14.899
101	3.18377	5.06324	11.61	14.905
102	3.04036	5.12172	11.909	14.925
103	3.08309	5.05332	11.909	14.905
104	3.40988	5.06415	11.802	14.883
105	3.11135	4.88415	11.909	14.986
106	2.9714	5.1129	11.802	14.931
107	3.47081	5.07207	11.703	14.92
108	3.12353	5.13698	12.15	14.962
109	3.11135	5.13487	11.802	14.963
110	3.1733	5.09316	11.703	14.935
111	3.34637	5.1211	11.909	14.962
112	3.14309	5.1175	11.909	14.978
113	2.95019	5.18953	11.802	14.964
114	3.06709	5.14012	11.703	14.93
115	2.94828	5.16127	11.44	15.017
116	3.10779	5.07306	11.802	14.945
117	3.19961	5.16889	11.703	14.942
118	3.02953	4.9848	11.703	14.995
119	3.28554	5.11712	11.802	14.979
120	3.40638	5.07079	11.909	14.969
121	3.12497	5.14678	11.802	15.04

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
122	2.88228	5.11998	11.802	14.965
123	3.30499	5.00985	11.909	14.932
124	2.95083	5.18148	12.025	14.949
125	3.14747	5.07219	11.802	14.97
126	3.17107	5.1625	11.909	14.94
127	3.0789	5.11768	11.802	14.956
128	3.53061	5.00221		

Table 12: Interrupts: Early Boot SP800-90B Min-Entropy Measurements in Virtual Environment
The table shows that the high-resolution time stamp of each of the first 128 interrupts has an estimated min-entropy of at least 3 bits in the 4 LSB of the time stamp. In addition, the table shows that for the 8 LSB of the time stamp, at least 5 bits are measured for each of the first 128 interrupts. Considering the min-entropy according to AIS 20/31 applied to the time deltas (i.e. the difference of two adjacent time stamps), the range of values is between 10 and 13 bits per interrupt event. The Shannon entropy values for the time deltas are even higher and range between 14 and 15.5 bits per interrupt event.

To allow the reader to get a graphical view of the time stamp distribution, figure 19 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 19 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. All other interrupts exhibit a similar distribution pattern. To make the graphic more readable, only the 90% quartile of the time delta data is depicted. The remaining 10% cover such a large value span with so little probability of occurrence that they would render the graphic unreadable.

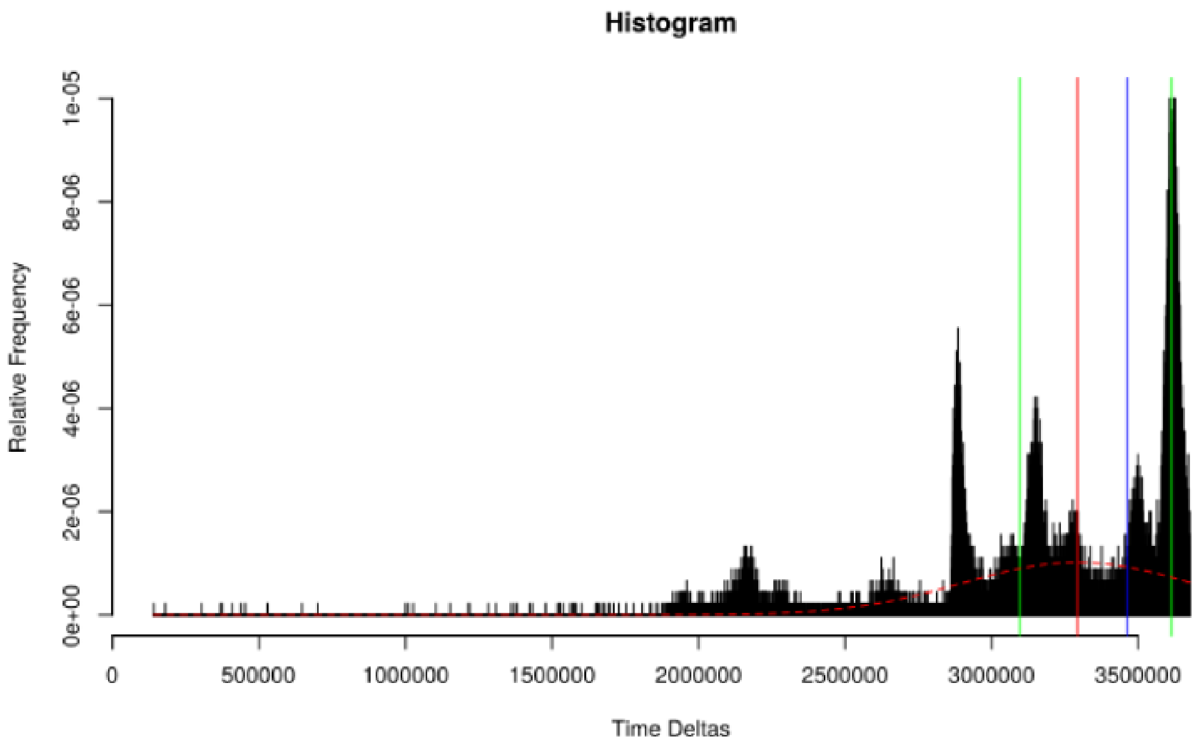

Figure 19: Histogram of Time Deltas for First and Second Interrupt in a Virtual Environment

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 red bar shows the statistical mean value of the histogram. The blue line shows the median and the two green bars show the 25% and 75% quartile of the data set. The red dotted line indicates how a Gaussian standard normal distribution would look like when using the standard derivation of the data set.

The table with the min-entropy estimates for the time stamps of the first 128 interrupts visualized in figure 19 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 128 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.

Early Boot Entropy Testing on Native Hardware

This appendix applies to the test outlined in section Fehler: Verweis nicht gefunden for kernel 4.9.

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
1	3.30499	5.15333	11.15	13.193
2	3.30499	5.15333	9.44	11.65
3	3.09715	5.17964	9.15	11.375

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
4	2.96044	5.16013	8.882	11.155
5	3.06433	5.11205	8.727	11.157
6	3.46169	5.16314	8.48	11.078
7	3.05264	5.16173	8.261	10.922
8	3.07194	4.94216	8.752	11.165
9	2.8841	5.0768	10.855	14.461
10	2.92427	5.10604	11.15	14.488
11	3.11492	5.15423	9.61	11.441
12	3.26794	5.11223	9.828	11.505
13	3.02684	5.18255	11.362	14.423
14	2.99424	5.15633	11.703	14.715
15	3.00216	5.08594	14.025	15.589
16	3.19053	5.13294	12.025	15.369
17	3.06295	5.13405	8.679	11.054
18	3.13727	5.0667	7.56	9.286
19	3.01145	4.97159	6.415	7.84
20	3.03291	5.09018	5.902	7.762
21	2.94319	5.08526	6.937	12.497
22	3.10708	5.17654	10.252	14.923
23	3.04444	5.08905	11.522	14.514
24	3.22794	5.07332	11.217	14.593
25	3.09222	5.11183	9.828	13.507
26	3.00018	4.92814	8.46	12.012
27	3.45488	5.20711	8.055	10.523
28	3.22949	5.11442	7.094	9.507
29	3.24115	5.14023	6.749	9.223
30	2.91055	5.11148	6.587	9.185
31	2.95659	5.06464	6.861	9.369
32	3.2537	5.11438	7.32	9.814
33	2.97789	5.08937	7.348	10.133
34	2.974	5.06181	7.758	12.709
35	2.90004	5.14598	7.789	13.195
36	2.69958	4.90777	10.802	15.189

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
37	2.51508	4.7596	11.802	14.515
38	2.17638	4.63331	12.025	15.131
39	2.15932	4.59699	10.4	13.847
40	2.16709	4.63145	11.288	14.407
41	2.11878	4.6035	9.42	12.479
42	2.05959	4.61474	10.217	13.187
43	2.08179	4.5926	9.855	12.525
44	2.08915	4.54992	10.4	13.282
45	2.10167	4.58879	9.522	12.607
46	2.1102	4.59488	10.44	13.991
47	2.09407	4.58762	8.995	12.587
48	2.0769	4.56889	10.4	14.018
49	2.07447	4.58863	8.937	12.653
50	2.06978	4.61553	10.752	14.48
51	2.07865	4.63452	9.632	13.107
52	2.21937	4.55743	11.362	14.371
53	2.34271	4.58054	9.632	12.974
54	2.32896	4.66535	10.703	13.45
55	2.32027	4.71625	9.27	12.455
56	2.42988	4.65204	10.565	13.749
57	2.39798	4.71931	9.288	12.692
58	2.36891	4.69557	10.288	13.706
59	2.40299	4.59747	9.252	12.653
60	2.40759	4.66288	9.565	13.044
61	2.46272	4.65679	8.752	12.511
62	2.45727	4.74267	10.217	13.769
63	2.41175	4.696	9.306	12.664
64	2.46956	4.64053	10.252	13.681
65	2.43299	4.67243	9.235	12.448
66	2.46272	4.71073	9.679	13.006
67	2.41351	4.63483	8.882	12.576
68	2.34313	4.64441	10.565	14.317
69	2.38974	4.69332	9.882	13.995

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
70	2.45071	4.70943	12.025	15.271
71	2.46136	4.68495	10.855	14.161
72	2.68996	4.76885	11.703	14.849
73	2.56355	4.7644	10.252	13.406
74	2.73596	4.72446	11.522	14.024
75	2.5894	4.79207	10.48	13.259
76	2.46227	4.79052	11.522	14.127
77	2.51508	4.80296	10.183	13.334
78	2.49889	4.76837	11.288	14.382
79	2.60514	4.74695	9.727	13.227
80	2.46774	4.8078	11.288	14.284
81	2.61927	4.79807	9.995	13.199
82	2.7261	4.74498	10.855	13.931
83	2.69529	4.72362	10.44	13.31
84	2.74479	4.81701	10.966	14.071
85	2.6985	4.75939	10.61	13.439
86	2.83086	4.89239	11.802	14.883
87	2.60966	4.87646	10.802	14.591
88	2.87984	4.95871	13.025	15.519
89	2.94129	4.96163	12.15	14.993
90	2.9249	4.90284	12.61	15.215
91	3.09998	4.92943	12.44	15.093
92	2.9714	4.92112	12.288	14.96
93	3.05607	4.95678	12.802	15.204
94	2.81916	4.91172	12.44	15.138
95	2.86353	4.89614	12.802	15.34
96	2.99952	4.99188	12.44	15.272
97	2.98245	4.95782	13.025	15.486
98	3.01145	5.05297	12.15	15.102
99	3.12569	5.14276	12.288	15.336
100	2.92114	5.1058	11.61	14.82
101	3.1014	5.19469	12.61	15.327
102	3.0015	5.16685	12.44	15.414

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 8 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
103	3.20721	5.14491	13.288	15.578
104	3.34805	5.19241	13.61	15.583
105	3.07333	5.1571	13.61	15.579
106	2.92052	5.13835	13.025	15.44
107	3.17367	5.10252	13.025	15.488
108	3.15848	5.07703	12.025	15.216
109	3.05744	5.03846	12.802	15.419
110	3.18302	5.10916	11.217	14.932
111	3.04649	5.10783	12.44	15.412
112	3.01545	5.12331	11.61	15.156
113	3.3838	5.09094	13.025	15.492
114	2.90127	5.08737	12.025	15.376
115	2.88045	5.0769	13.025	15.511
116	3.06019	5.07597	11.217	15.334
117	3.08379	4.98531	13.025	15.471
118	2.99622	5.05527	11.288	15.235
119	3.05607	4.96621	12.288	15.454
120	3.38035	5.1295	10.703	15.201
121	2.98637	5.13873	12.15	15.447
122	2.95019	5.06541	10.61	15.169
123	3.13727	5.07633	12.44	15.449
124	3.1185	5.11505	10.752	15.176
125	3.09081	4.96025	12.44	15.468
126	2.91303	5.12936	11.802	15.264
127	3.03426	5.09961	13.025	15.458
128	3.45082	5.11811	▨	▨

Table 13: Interrupts: Early Boot SP800-90B Min-Entropy Measurements on Native Hardware
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 20. 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 between the occurrence of both interrupts. As discussed for the virtual environment, the 90% quartile is depicted.

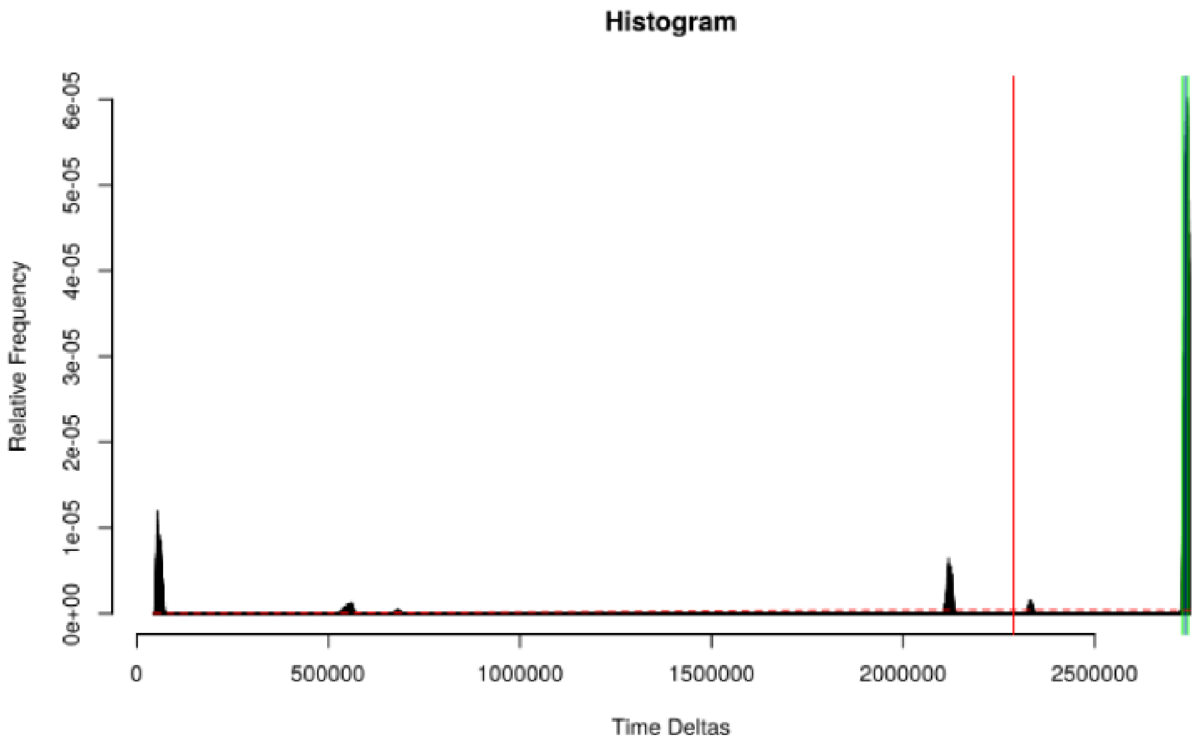

Figure 20: Histogram of Time Deltas for First and Second Interrupt in a Native Environment

It is interesting that the first time delta shows spikes at around 60000 ticks, 2.1 million ticks and 2.7 million ticks. Nonetheless, these spikes exhibit variations such that the Shannon and AIS 20/31 min-entropy values are large. Clearly, the pattern of the histogram is quite different compared to virtual environments shown in figure 19. Starting with the third time delta, the distribution of the time deltas start to “normalize” and look very similar to figure 19.

AIS 20/31 Test Procedure A for Entropy Pools

Using the test code input_pool/, the input_pool as well as the blocking_pool are observed. The SystemTap test code takes a snapshot of the entropy pool after an amount of bytes has been mixed in that equals its size in words. That means that a snapshot is taken of the input_pool after 128 bytes have been mixed in. Similarly, a snapshot of the blocking_pool has been taken after 32 bytes have been mixed into the blocking_pool. After the mix-in of the stated amount of bytes, all words in the entropy pool have been changed by the LFSR operation.

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 white noise in the entropy pool.

input_pool

After generating 1,000,000 snapshots with the test code and concatenating all data, a binary string is present that can be analyzed as follows:

- The Chi-Squared value using the ent tool shows a value 0.56 which indicates white noise.
- The test procedure A is passed by the bit string.

The test results for the input_pool confirm the test results obtained during the analysis presented in section 7.2.

blocking_pool

Similarly to the input_pool, the blocking_pool is analyzed after taking 100,000 snapshots. Again, the data can be characterized:

- The Chi-Squared value provided with the ent tool shows the value of 0.05 which indicates white noise.
- The test procedure A is also passed by the bit string.

This implies that the test results confirm the results obtained from the analysis provided in section 7.2.

Appendix C: Test Results on Linux Kernel 4.15

This appendix chapter holds the test results for the tests of Linux Kernel 4.15 for reference and comparison with future re-tests, as described in chapter 6.2.

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 6.3.2:
  - 2 core CPU with two hyperthreads per core
  - Intel(R) Core(TM) i7-3520M CPU @ 2.90GHz
- QEMU 2.10.1 used for the virtual environment early boot entropy tests documented in 6.3.1:
  - 4 virtual CPUs corresponding with the 4 hyperthreads of the host system
  - Intel Core Processor (Broadwell, no TSX)
- Apple MacBook Pro 2015 used for all other tests executed on native hardware:
  - 2 core CPU with two hyperthreads per core
  - Intel(R) Core(TM) i7-5557U CPU @ 3.10GHz

Min-Entropy 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 the test provided in the test directory linux-entropy-sp80090b. This test uses several SystemTap scripts which are enumerated in the following:

- HID measurement: to measure the high-resolution time stamp of HID events, the SystemTap script recording/raw_entropy_hid.stp instruments add_timer_randomness to read out the high-resolution time stamp from the `sample` data structure (see section 3.5.2.7 for details about this data structure). In conjunction, a second SystemTap script record/entropy_per_event_hid.stp is used to record the entropy estimation applied by the Linux-RNG to the HID events. To allow other reviewers to assess the quality of the event values and Jiffies values, they are recorded but disregarded in the subsequent assessments.
- Block device measurement: the SystemTap scripts of record/raw_entropy_disk.stp and record/entropy_per_event_disk.stp are used to record the same data for block devices as outlined for HID devices above. Again, the event values and the Jiffies values are recorded for third-party verification. However, they are again not considered in the analysis below.
- Interrupt measurement: the SystemTap script raw_entropy_irq.stp instruments `add_interrupt_randomness`. It obtains the high-resolution time stamp for this interrupt. There is no SystemTap script measuring the entropy estimation applied to interrupts as the Linux-RNG applies a fixed estimate of one bit (in case of Intel x86 systems with `RDRAND` it is two bits) per injection of a fast_pool content into the input_pool.

The recorded data set is simply a set of 32 bit integer values holding the high-resolution time stamps for each recorded interrupt. To make testing easier and more repeatable, the script recording/gendata.sh is provided which invokes the SystemTap scripts appropriately. That script 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 content 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 values are calculated defined by [SP800-90B]:

- Most Common Value Estimate
- Collision Estimate
- Markov Estimate
- Compression Estimate
- t-Typle 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. To allow processing the time stamps with the aforementioned min-entropy estimation calculations, the application validation/extractlsb.c obtains the 4 least significant bits of the time stamp and concatenates all 4 LSB of all time stamps into a bit stream. This means that the input data width is now 4 bits instead of 32 bits. The calculation of the min-entropy estimations using 4 bits instead of 32 bits is considered to support the conservative assessment of this study. In addition, the mentioned application also extracts the 6 LSB[25] of each time stamp and concatenates them into a bit-stream. This allows the calculation of the min-entropy estimation of the input data with 6 bit width. The following tables therefore provide the entropy estimation for 4 bit and 6 bit input data widths.

For comparison, the min-entropy and the Shannon entropy defined by [AIS2031] are calculated as well. The used formulas are provided in section 2.3.2 [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 Minimum 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.

Interrupt Noise Source Min-Entropy Estimates

The collection of data for interrupts was conducted twice: once with a normal use case and once with a worst-case. In the normal use case the test environment was made to resemble regular usage where Internet searches and regular office duties were performed. The worst-case covered the test system in a virtual environment where the host system 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.

25 The reason for selecting 6 LSB is that the Markov min-entropy value can only be calculated for data blocks not exceeding 6 bits.

Entropy Estimate	4 Bits Width	8 Bits Width
Most Common Value Estimate	3.9833	5.95955
Collision Estimate	3.333	4.85264
Markov Estimate	3.82232	5.69777
Compression Estimate	3.33175	4.94307
t-Typle Estimate	3.64351	5.81733
LRS Estimate	3.14332	5.69474
Multi Most Common in Window Prediction Estimate	3.98114	5.94717
Lag Prediction Estimate	3.91254	5.95643
MultiMMC Prediction Estimate	3.84493	5.50815
LZ78Y Prediction Estimate	3.8451	5.50815

Table 14: Interrupts: SP800-90B Min-Entropy Measurements - Worst Case

The associated Shannon entropy value is 14.025 bits per interrupt event. The min-entropy value according to [AIS2031] is 11.412 bits per interrupt event.

The normal use case returned the following data.

Entropy Estimate	4 Bits Width	6 Bits Width
Most Common Value Estimate	3.97688	5.94959
Collision Estimate	2.55868	3.93252
Markov Estimate	3.75639	5.64278
Compression Estimate	2.64331	4.12894
t-Typle Estimate	3.42091	5.77394
LRS Estimate	3.23859	5.9087
Multi Most Common in Window Prediction Estimate	3.79055	5.85228
Lag Prediction Estimate	3.59835	5.29956
MultiMMC Prediction Estimate	3.29956	5.6648
LZ78Y Prediction Estimate	2.55868	5.81447

Table 15: Interrupts: SP800-90B Min-Entropy Measurements - Normal Use Case

Applying the Shannon entropy formula on the data set, a value of 19.216 bits per interrupt event is calculated. Using the min-entropy formula according to [AIS2031], a result of 13.163 bits per interrupt event is measured.

The conclusions that can be drawn from the numbers follow. Regardless of the worst-case or normal case, the high-resolution time stamp of each interrupt will return significantly more than two 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 (two bits when `RDRAND` is present). 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.

Block Device Noise Source Min-Entropy Estimates

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 mounting and unmounting a block device. This causes the buffer cache to be irrelevant for the disk accesses caused by the mount operations, as the buffer cache is flushed with each unmount operation of a file system. The worst-case produced the following data:

Entropy Estimate	4 Bits Width	6 Bits Width
Most Common Value Estimate	3.97764	5.94579
Collision Estimate	4	6
Markov Estimate	3.62004	5.50219
Compression Estimate	4	6
t-Typle Estimate	3.28755	5.71189
Longest Repeated Substring (LRS) Estimate	3.43395	5.70321
Multi Most Common in Window Prediction Estimate	3.98959	5.97841
Lag Prediction Estimate	3.29956	5.734
MultiMMC Prediction Estimate	3.29956	5.29956
LZ78Y Prediction Estimate	3.64385	4.41504

Table 16: Block Devices: SP800-90B Min-Entropy Measurements

Using the Shannon entropy formula, 18.912 bits per block device event is calculated. A value of 14.977 bits per block device event is calculated as the min-entropy according to [AIS2031].

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 21 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 21 also shows that the mean value of all entropy estimates is 1.31 bits of entropy. This can be interpreted that on average, the Linux-RNG awarded each block device event 1.31 bits of entropy.

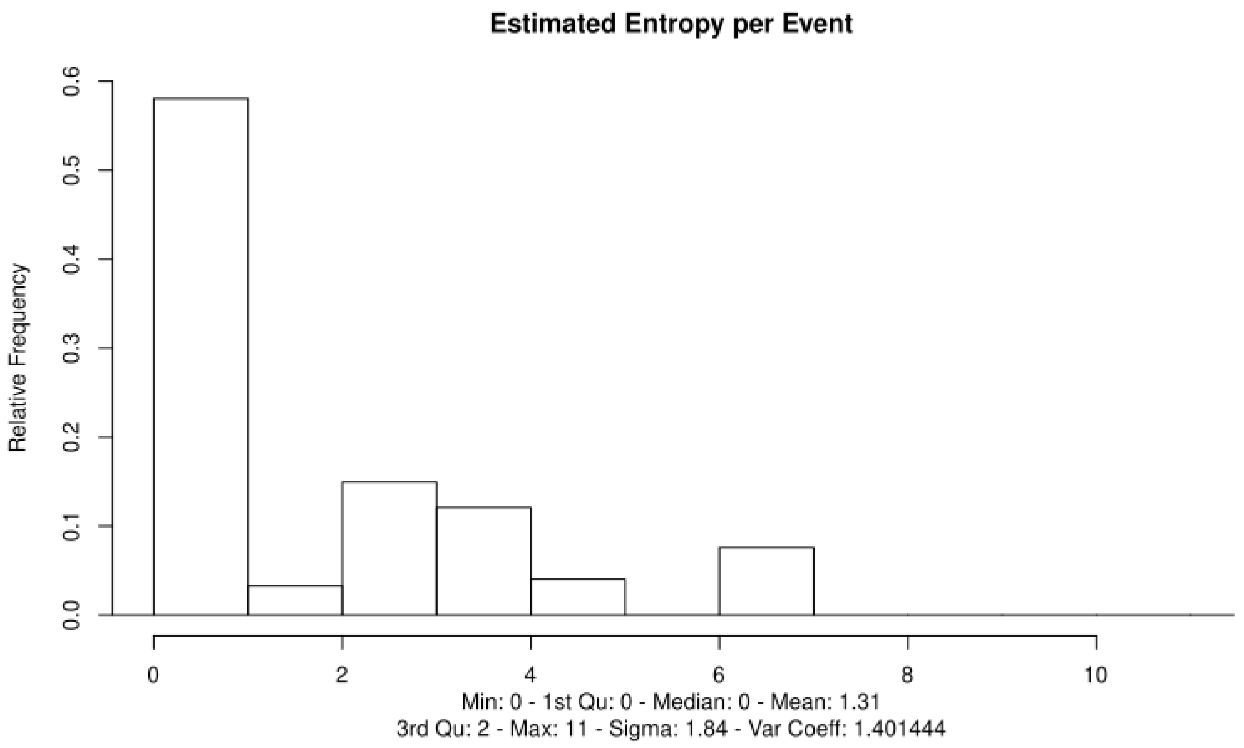

Figure 21: Entropy Estimate per Block Device Event Applied by Linux-RNG

Comparing the result shown in figure 21 with the min-entropy estimates calculated from the measured time stamps, the following conclusion is drawn: the min-entropy estimates have significantly more than 3 bits of entropy per event. On the other hand, the Linux-RNG considers that each event has on average only 1.31 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.

HID Noise Source Min-Entropy Estimates

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, only 500.000 samples of HID noise source events were recorded. The entropy estimates for the high-resolution time stamp applied to those events are listed in the table below.

Entropy Estimate	4 Bits Width	6 Bits Width
Most Common Value Estimate	3.95963	5.93102
Collision Estimate	3.43645	4.63996
Markov Estimate	3.75615	5.59784
Compression Estimate	3.29604	4.76048
t-Typle Estimate	3.32193	5.73607

Entropy Estimate	4 Bits Width	6 Bits Width
LRS Estimate	3.35119	5.86791
Multi Most Common in Window Prediction Estimate	3.63368	5.90624
Lag Prediction Estimate	3.63368	5.64622
MultiMMC Prediction Estimate	3.44475	5.63528
LZ78Y Prediction Estimate	3.63368	5.79679

Table 17: HID: SP800-90B Min-Entropy Measurements

The Shannon entropy formula applied on the data set results in 16.377 bits per HID event. 11.539 bits per HID event are calculated when using the min-entropy formula according to [AIS2031].

The test record of the entropy estimate applied by the Linux-RNG for each recorded HID event is depicted with figure 22. 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 22, the mean value of the histogram is 0.9 bits. This implies that the Linux-RNG awarded 0.9 bits of entropy to each HID noise source event on average.

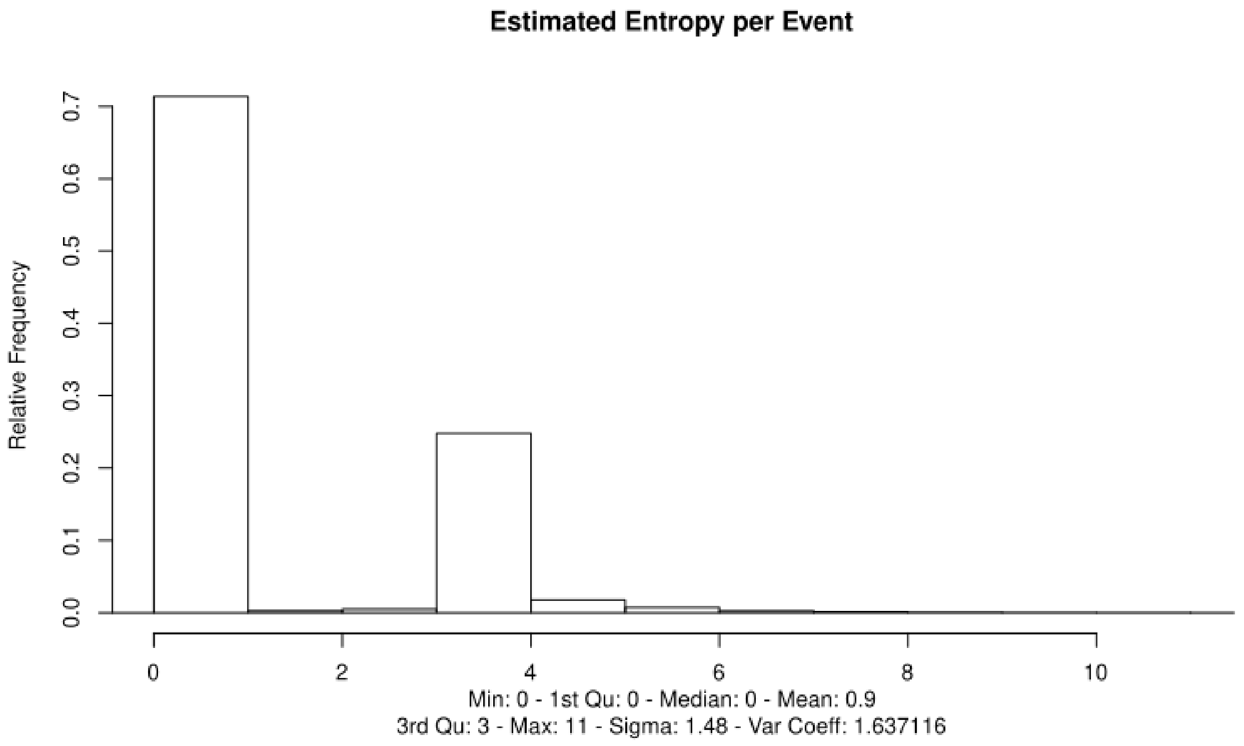

Figure 22: Entropy Estimate per HID Event Applied by Linux-RNG

A conclusion can be reached when comparing the heuristic entropy values applied by the Linux-RNG from figure 22 with the min-entropy estimates. The min-entropy estimates have more than 4 bits of entropy per event (6 bit width), or more than 3 bits of entropy per event (4 bits width). On the other hand, the Linux-RNG applies on average 0.9 bits of entropy to a HID event.

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.

Conclusion of SP800-90B Measurements

The conclusions given for each noise source regarding the SP800-90B measurements are collectively summarized with 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 to diminish 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 concern: 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 (in the presence of `RDRAND` 2 bits are applied). 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.

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 are initialized to allow users to interact with the system. Therefore, testing is limited to measure interrupt event data only. As outlined in section 6.1.1, only the high-resolution time stamp recorded for interrupts is of interest to entropy measurements.

The Linux kernel has been modified with the patch boottime/boottime_test.diff. This patch records the high-resolution time stamps obtained for the first 128 interrupts. A user space shell script boottime/boottime_test_record.sh stores these 128 time stamps to disk and initiates a reboot.

The test is performed for 50,000 reboot cycles for the virtual environment as well as for the bare-metal environment. At the end of the testing, 50,000 times 128 time stamps are collected and analyzed.

The first analysis performs an SP800-90B min-entropy estimate calculation discussed in section 6.2.1. Such a min-entropy estimate is calculated for each of the 128 32-bit time stamps individually. This means that the min-entropy estimate for the first till the 128th 32-bit time stamp of all boot cycles is calculated. Therefore, the full result contains 128 entries with min-entropy estimates. To limit the amount of space in this report for presenting the data, the tables in the following subsections only list the lowest min-entropy estimate out of all estimate types enumerated in section 6.2 for all 128 different interrupt occurrences.

In addition to the calculation of the min-entropy according to SP800-90B, the min-entropy and the Shannon entropy according [AIS2031] has been calculated as well. Both formulas are provided in section 2.3.2 of [AIS2031] and are not repeated here. As the time stamp is a monotonically increasing integer, the entropy lies in the deltas of the time stamps and the distribution of those deltas. Therefore, to perform the calculation for the Minimum 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.

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 128 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.

Early Boot Entropy Testing in a Virtual Environment

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 6 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
1	3.09504	4.42222	11.522	14.318
2	3.09504	4.42222	14.025	15.556
3	2.94828	4.29682	12.802	15.446
4	3.0796	4.43014	13.025	15.434
5	2.97335	4.30794	13.61	15.519
6	3.07681	4.55651	13.61	15.554
7	3.02482	4.26731	12.025	15.438

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 6 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
8	3.03629	4.4367	13.025	15.522
9	3.06295	4.3774	13.288	15.5
10	3.04853	4.41759	13.025	15.498
11	3.01545	4.18854	13.61	15.517
12	3.10708	4.42455	13.288	15.544
13	2.9805	4.25919	13.61	15.541
14	2.96752	4.56009	13.025	15.426
15	3.04512	4.23512	12.44	15.337
16	3.0789	4.21849	12.025	15.223
17	3.0255	4.40835	11.61	14.916
18	3.07751	4.15052	11.44	14.032
19	3.07472	4.35328	10.909	13.442
20	3.30989	4.60105	10.802	13.351
21	3.18302	4.81311	10.61	13.421
22	3.04853	4.45512	10.752	13.578
23	3.24976	4.54635	10.48	13.811
24	3.12858	5.07403	10.61	14.072
25	3.29199	4.36887	10.288	14.307
26	3.3181	4.36708	10.522	14.526
27	3.14893	4.6021	10.656	14.735
28	3.08239	4.33786	10.855	14.871
29	3.29199	4.88301	10.752	14.988
30	3.43466	4.63457	10.752	15.083
31	3.30907	4.32956	11.362	15.171

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 6 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
32	3.35766	4.66228	11.288	15.229
33	3.11278	4.76283	12.288	15.271
34	3.09998	4.38553	12.025	15.295
35	3.40813	4.49365	12.44	15.311
36	3.43109	4.79615	12.288	15.342
37	3.1482	4.39872	12.61	15.356
38	3.18602	4.2282	13.025	15.349
39	3.17778	4.61107	13.025	15.357
40	3.17107	4.7291	13.025	15.362
41	3.12281	4.44046	13.288	15.361
42	3.15775	4.38101	12.44	15.367
43	3.16736	4.19687	13.025	15.366
44	3.16958	4.34753	12.802	15.36
45	3.21869	4.66173	12.802	15.355
46	3.09081	4.51282	12.61	15.339
47	3.21179	4.32608	12.61	15.332
48	3.15628	4.50296	12.802	15.319
49	3.11135	4.20904	12.802	15.298
50	3.05538	4.24659	12.802	15.273
51	3.07612	4.3738	12.61	15.263
52	3.15187	4.41573	12.61	15.245
53	3.19431	4.37066	12.802	15.226
54	3.18602	4.36618	12.61	15.213
55	3.24584	4.43108	12.61	15.2

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 6 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
56	3.45534	4.43776	12.802	15.18
57	3.17107	4.38916	12.44	15.169
58	3.221	4.53325	12.44	15.167
59	3.17256	5.03651	12.44	15.159
60	3.5211	4.47957	12.802	15.154
61	3.13509	4.32087	12.802	15.163
62	3.15451	4.43435	12.288	15.16
63	3.04376	4.62491	12.802	15.162
64	3.10921	4.64158	12.802	15.166
65	3.25527	4.59895	12.802	15.157
66	3.12713	4.73948	12.61	15.156
67	3.37863	4.61002	12.61	15.149
68	3.21639	4.41712	12.61	15.158
69	3.06433	4.19389	12.802	15.166
70	3.05813	4.56162	12.802	15.171
71	3.02348	4.43623	12.802	15.171
72	3.25055	4.54635	12.802	15.182
73	3.1409	4.40284	12.61	15.169
74	3.56147	4.86901	12.61	15.167
75	3.32843	4.38779	12.61	15.167
76	3.24741	4.46561	12.288	15.16
77	3.23648	4.5838	12.802	15.164
78	3.03765	4.3756	12.61	15.175
79	3.18752	4.34005	12.288	15.176

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 6 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
80	3.19885	4.2503	12.61	15.181
81	2.96301	4.4757	12.802	15.175
82	3.26714	4.62759	12.802	15.179
83	3.23959	4.34225	12.802	15.191
84	3.2537	4.39918	12.802	15.187
85	3.12281	4.2981	12.61	15.192
86	3.17554	4.55244	12.15	15.191
87	3.01745	4.23227	12.61	15.195
88	3.10211	4.58744	12.288	15.214
89	3.25055	4.22051	12.44	15.242
90	3.19658	4.70856	12.44	15.28
91	3.32305	4.55958	12.61	15.311
92	3.38725	4.44564	12.61	15.312
93	3.03765	4.18342	12.61	15.318
94	3.23648	4.6984	12.61	15.335
95	3.18002	4.4414	12.802	15.361
96	3.29929	4.57964	12.61	15.343
97	3.27431	4.43435	12.44	15.339
98	3.17703	4.51827	12.61	15.322
99	3.19355	4.68663	12.61	15.325
100	3.12066	4.38417	12.61	15.298
101	3.03358	4.45845	12.61	15.29
102	3.20874	5.04669	12.15	15.254
103	3.11994	4.24126	12.288	15.236

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 6 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
104	3.02281	4.38146	12.44	15.211
105	3.31975	4.78059	12.15	15.199
106	3.1548	4.17713	12.15	15.18
107	3.1548	4.39827	11.909	15.165
108	3.16513	4.33786	12.025	15.143
109	3.01145	4.44801	11.802	15.149
110	2.97076	4.16581	12.15	15.136
111	2.96108	4.39188	12.025	15.152
112	3.07056	4.25195	12.025	15.145
113	3.00018	4.16347	12.15	15.159
114	3.01879	4.43108	12.15	15.154
115	3.07751	4.46705	12.15	15.178
116	3.34469	4.59057	12.15	15.185
117	3.00746	4.41066	12.15	15.199
118	3.14601	4.68384	12.288	15.202
119	2.95083	4.55958	12.288	15.214
120	3.13872	4.39827	12.025	15.209
121	2.98441	4.38734	12.61	15.223
122	3.0817	4.53827	12.288	15.219
123	3.18377	4.19607	12.44	15.233
124	3.12353	4.29384	12.61	15.224
125	3.09857	4.40055	12.44	15.238
126	3.01145	4.4213	12.288	15.239
127	2.99424	4.3729	12.61	15.242

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 6 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
128	3.15628	4.2998		

Table 18: Interrupts: Early Boot SP800-90B Min-Entropy Measurements in Virtual Environment

The table shows that the high-resolution time stamp of each of the first 128 interrupts has an estimated min-entropy of at least 3 bits in the 4 LSB of the time stamp. In addition, the table shows that for the 6 LSB of the time stamp, at least 4 bits are measured for each of the first 128 interrupts. Considering the min-entropy according to AIS 20/31 applied to the time deltas (i.e. the difference of two adjacent time stamps), the range of values is between 10 and 13 bits per interrupt event. The Shannon entropy values for the time deltas are even higher and range between 13 and 15.5 bits per interrupt event.

To allow the reader to get a graphical view of the time stamp distribution, figure 23 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 23 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. All other interrupts exhibit a similar distribution pattern. To make the graphic more readable, only the 90% quartile of the time delta data is depicted. The remaining 10% cover such a large value span with so little probability of occurrence that they would render the graphic unreadable.

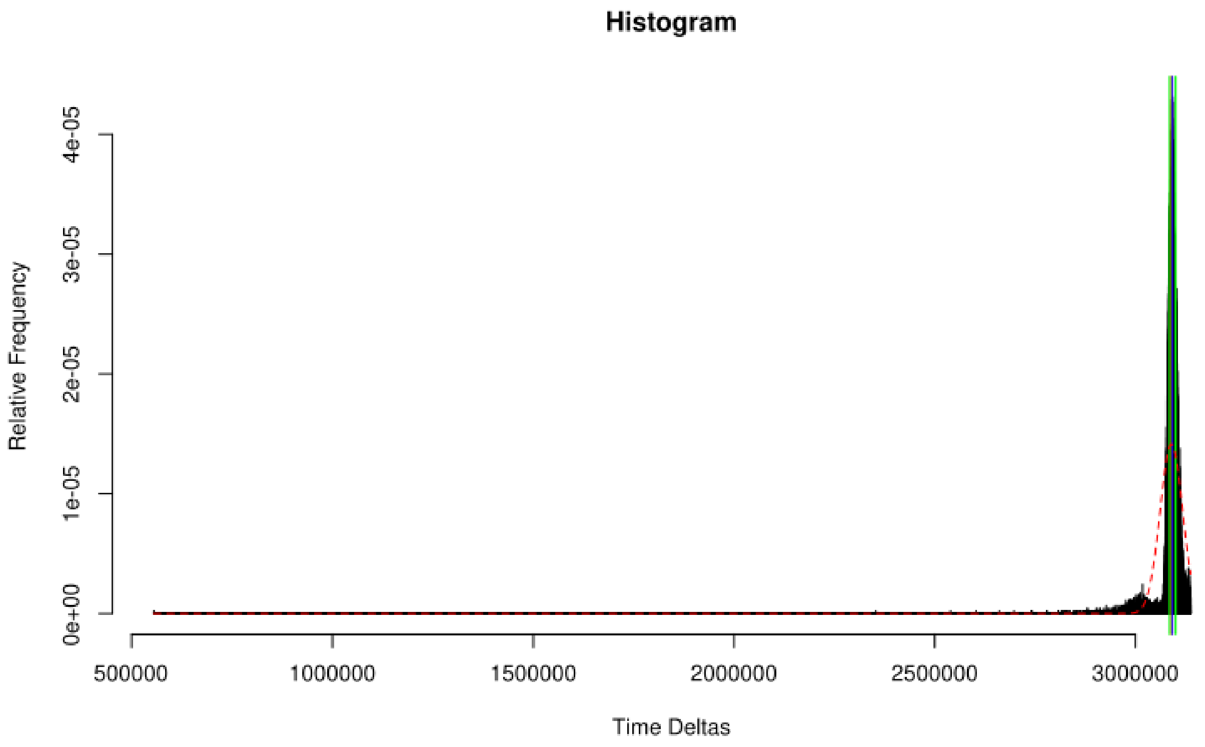

Figure 23: Histogram of Time Deltas for First and Second Interrupt in a Virtual Environment

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. The red dotted line indicates how a Gaussian standard normal distribution would look like when using the standard derivation of the data set.

The table with the min-entropy estimates for the time stamps of the first 128 interrupts visualized in figure 23 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 128 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.

Early Boot 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 less 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.

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 6 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
1	3.41339	4.90751	9.602	11.606
2	3.41339	4.90751	9.123	11.477
3	3.30499	4.87345	8.149	11.301
4	3.02348	4.62011	7.764	11.303
5	3.07751	4.70516	10.284	11.718
6	3.30336	4.63564	10.947	12.68
7	3.10566	4.4815	10.488	12.368
8	3.10637	4.70516	10.14	12.234
9	3.27911	4.24598	13.947	15.519
10	3.19506	4.4917	12.725	15.417

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 6 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
11	3.2608	4.78237	10.175	12.839
12	3.07095	4.63929	10.832	13.438
13	3.06697	4.55047	10.947	14.28
14	3.08464	4.62135	11.947	15.372
15	3.17343	4.72057	9.832	12.375
16	2.97548	4.60863	10.175	12.75
17	3.03407	4.55272	9.284	11.979
18	3.00643	4.57684	9.466	12.249
19	3.04527	4.56994	9.193	11.956
20	3.05688	4.55582	10.284	12.649
21	3.02857	4.57091	9.123	11.887
22	3.03849	4.53796	11.532	14.607
23	3.03825	4.56751	9.445	11.989
24	3.04682	4.51154	9.65	12.524
25	3.02328	4.5501	8.963	11.881
26	3.05099	4.58221	9.778	12.62
27	3.05461	4.56925	9.123	12.134
28	3.00125	4.54954	10.832	13.424
29	3.06636	4.55773	9.21	12.681
30	3.03495	4.6335	9.466	12.754
31	3.00643	4.60766	9.057	12.361
32	3.06936	4.63401	9.832	12.656
33	3.04465	4.59841	9.025	12.496
34	3.05714	4.60311	11.009	13.385

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 6 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
35	3.06914	4.58282	9.403	12.407
36	3.05045	4.62894	9.304	12.28
37	3.04226	4.57772	8.86	11.86
38	2.73311	4.6436	9.602	12.348
39	3.0648	4.60765	8.662	12.248
40	3.03242	4.62036	11.947	14.33
41	3.0472	4.53828	10.778	13.622
42	3.05762	4.59815	12.21	14.618
43	3.06069	4.62506	11.14	13.657
44	3.04279	4.57562	11.284	13.831
45	3.05228	4.59675	11.073	13.281
46	3.03053	4.58534	12.21	14.423
47	3.03832	4.57619	10.947	13.38
48	3.03514	4.58071	11.14	13.444
49	3.0448	4.53612	11.362	14.143
50	3.00643	4.61314	11.14	13.543
51	3.02176	4.42257	11.073	13.171
52	3.04345	4.58944	11.21	13.162
53	3.03132	4.57507	10.947	13.053
54	3.05317	4.597	11.362	13.525
55	3.04889	4.54551	11.445	14.111
56	3.04587	4.55816	11.21	13.44
57	3.0331	4.5562	10.578	12.949
58	3.05636	4.47577	11.445	14.227

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 6 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
59	3.03495	4.6093	11.073	13.56
60	3.05396	4.57878	11.21	13.409
61	3.07104	4.58559	11.009	13.418
62	3.0763	4.52918	10.947	13.375
63	3.04657	4.61301	11.445	13.872
64	3.09028	4.56918	11.832	14.275
65	3.07395	4.55906	11.284	13.732
66	3.04513	4.54165	11.362	13.791
67	3.07426	4.50302	11.947	14.674
68	3.05183	4.61546	11.832	14.072
69	3.06076	4.65317	11.725	13.953
70	3.07784	4.63118	11.626	14.058
71	3.07827	4.62089	11.362	13.919
72	3.0707	4.55758	11.532	14.173
73	3.07065	4.4188	11.725	14.477
74	3.07626	4.59021	10.626	13.526
75	3.0664	4.60811	10.674	13.531
76	3.08913	4.56159	10.832	13.897
77	3.07012	4.60421	11.362	13.918
78	2.74261	4.62105	11.445	14.195
79	3.09774	4.60541	11.832	14.68
80	3.00643	4.62689	11.073	13.594
81	3.06833	4.60641	10.362	13.236
82	3.05705	4.58138	10.21	13.354

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 6 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
83	3.06582	4.58658	10.323	13.262
84	3.04987	4.4188	8.918	13.486
85	3.05821	4.49466	11.445	14.447
86	3.08443	4.56275	9.889	13.243
87	3.09235	4.61514	10.175	13.234
88	3.08204	4.29492	10.073	14.03
89	3.08035	4.58805	10.247	13.344
90	3.08538	4.60573	10.009	13.009
91	3.05802	4.62182	10.14	13.179
92	3.08441	4.6343	10.247	12.971
93	3.07596	4.63328	9.889	13.138
94	3.0574	4.6496	10.106	12.947
95	3.03691	4.4188	9.832	13.1
96	3.03466	4.57671	10.073	12.986
97	3.08159	4.62901	9.725	13.127
98	3.00643	4.4188	10.247	12.994
99	3.07623	4.62187	9.751	13.023
100	3.07669	4.51291	10.14	12.954
101	3.07097	4.48928	9.7	13.025
102	3.08403	4.56284	9.947	12.896
103	3.09169	4.64467	9.578	12.991
104	3.07497	4.6841	9.889	12.856
105	3.08712	4.59934	9.555	12.984
106	3.03786	4.58154	9.751	12.829

Time Stamp Position / Time delta	Lowest SP800-90B min-entropy estimate of 4 Bits Width Time Stamp	Lowest SP800-90B min-entropy estimate of 6 Bits Width Time Stamp	Min-entropy (AIS 20/31) of Time Delta	Shannon Entropy of Time Delta
107	3.08371	4.60841	9.555	13.074
108	3.10183	4.58294	9.86	12.907
109	3.06409	4.65171	9.725	13.146
110	3.09016	4.63152	9.947	12.89
111	3.08407	4.57259	9.343	13.026
112	3.07659	4.5871	9.778	12.763
113	3.05481	4.62262	9.424	12.981
114	2.74261	4.63858	9.21	12.706
115	3.07035	4.60136	9.403	12.903
116	3.08583	4.69585	9.304	12.682
117	3.07398	4.42257	9.383	12.763
118	3.05292	4.63643	9.323	12.614
119	3.07114	4.62449	9.21	12.669
120	3.04157	4.59885	9.229	12.642
121	3.07661	3.67807	8.041	12.857
122	3.08088	4.63655	9.751	13.272
123	3.07156	4.59839	8.466	12.77
124	3.07705	4.57344	9.445	12.873
125	3.08322	4.60368	9.123	13.762
126	3.08688	4.60518	9.532	12.762
127	2.74261	4.65792	9.009	12.427
128	3.10012	4.63142	▨	▨

Table 19: Interrupts: Early Boot SP800-90B Min-Entropy Measurements on Native Hardware

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 24. 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 between the occurrence of both interrupts. As discussed for the virtual environment, the 90% quartile is depicted.

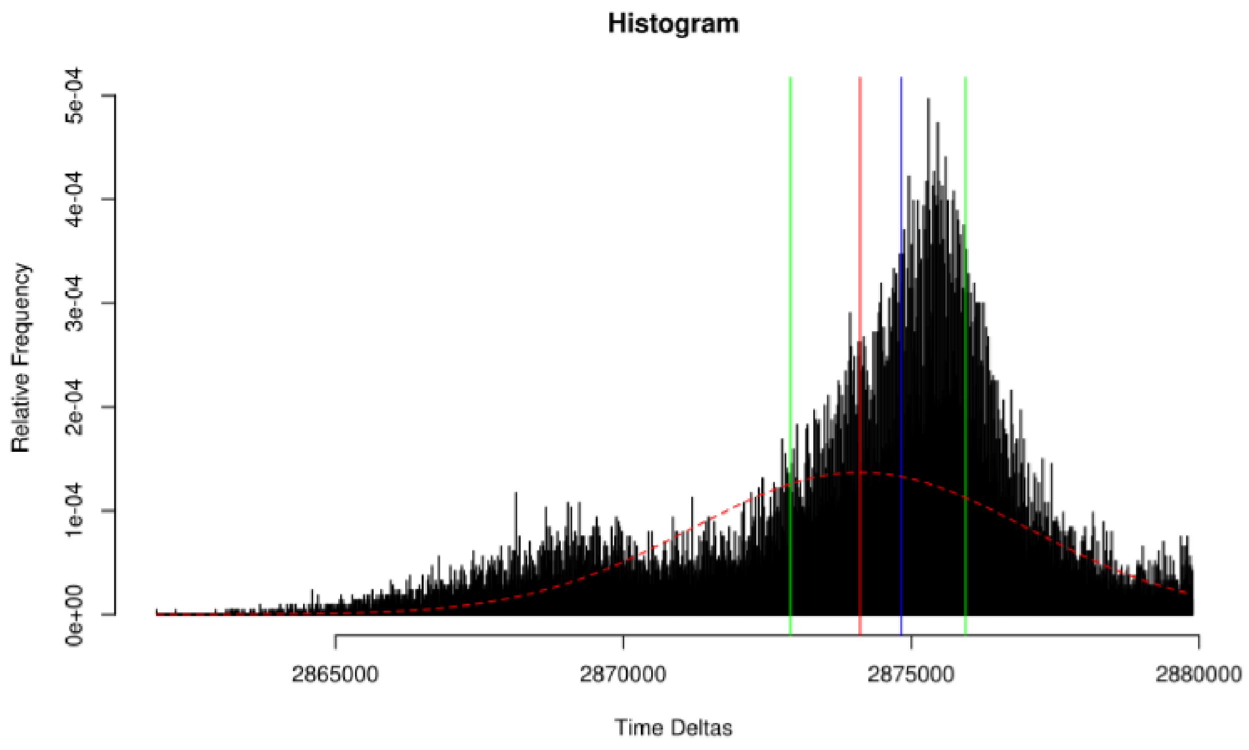

Figure 24: Histogram of Time Deltas for First and Second Interrupt in a Native Environment

The pattern of the histogram is very similar to virtual environments shown in figure 23.

With the obtained results, the same conclusions for the measurements in virtual environments given in section Fehler: Verweis nicht gefunden can be drawn. The listing of the 4 bit width SP800-90B min-entropy contains 9 entropy values that are below one bit. As the corresponding 6 bit width min-entropy values are similar to all other values, the ones below 1 bit are considered statistical artefacts. Disregarding the correlation problem, 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.

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 considerably exceeds one bit.

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 unblocks and generates random numbers. This allows the conclusion that when the `getrandom` system call unblocks, 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 is considered to always provide 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 boot. 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 and injected into the ChaCha20 DRNG only once one of the fast_pool received 64 interrupts. 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[26], 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.

AIS 20/31 Test Procedure A for Entropy Pools

Using the test code input_pool/, the input_pool as well as the blocking_pool are observed. The SystemTap test code takes a snapshot of the entropy pool after an amount of bytes has been mixed in that equals its size in words. That means that a snapshot is taken of the input_pool after 128 bytes have been mixed in.

26 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.

Similarly, a snapshot of the blocking_pool has been taken after 32 bytes have been mixed into the blocking_pool. After the mix-in of the stated amount of bytes, all words in the entropy pool have been changed by the LFSR operation.

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 white noise in the entropy pool.

input_pool

After generating 1,000,000 snapshots with the test code and concatenating all data, a binary string is present that can be analyzed as follows:

- The Chi-Squared value using the ent tool shows a value 0.03 (bit-wise) and 258.02 (byte-wise) which indicates white noise.
- The test procedure A is passed by the bit string.

The test results for the input_pool confirm the test results obtained during the analysis presented in section 7.2.

blocking_pool

Similarly to the input_pool, the blocking_pool is analyzed after taking 100,000 snapshots. Again, the data can be characterized:

- The Chi-Squared value provided with the ent tool shows the value of 0.06 (bit-wise) and 272.56 (byte-wise) which indicates white noise.
- The test procedure A is also passed by the bit string.

This implies that the test results confirm the results obtained from the analysis provided in section 7.2.

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
INTELDRNG	Intel: Intel Digital Random Number Generator (DRNG) Software Implementation Guide
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
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
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
FV17	David Fontaine, Olivier Vivolo: Proposal of primitive polynomials for Linux kernel PRNG https://eprint.iacr.org/2017/726.pdf

Keywords and Abbreviations

Abbreviation	Description
AES	Advanced Encryption Standard (FIPS 197)

Abbreviation	Description
API	Application Programming Interface
BSI	Bundesamt für Sicherheit in der Informationstechnik (Federal Office for Information Security)
CTR	Counter block chaining 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)
NUMA	Non-Uniform Memory Access
RNG	Random Number Generator
UUID	Universally Unique Identifier