Is there noise down there?

Gianmarco Lusvardi*

October 17, 2023

1 Introduction

In this short document, I will explain the work I did in fulfilment of the requirements of the Kernel Hacking exam.

The problem setup is entropy generation on a real-time operating system. The Linux kernel gathers entropy data from a variety of non-physical noise sources, however, most of them are unavailable in this scenario. Therefore, the idea is to find other sources of entropy that can be used in place of the ones that are not available.

Although the target operating system kernel is not Linux, in this document the Linux kernel will be analysed as a starting point for analysing the problem in a known environment.

2 The Linux RNG

The Linux kernel provides a RNG[1] that uses hardware events as a noise source to seed a DRNG[2] that in turn will generate random data upon request.

The Linux kernel gathers noise data from a variety of non-physical noise sources:[1]

- Human Interface Devices (HID) such as mice and keyboards (via the interface `add_input_randomness`)
- *Some* block devices, for instance hard drives, but not SSDs (via the interface `add_disk_randomness`)
- Interrupts (via the interface `add_interrupt_randomness`)
- *Some* device drivers (via the interface `add_device_randomness`)
- The scheduler (but only at boot time [1, section 3.5.2.6]).

*229563@studenti.unimore.it

[1] Random Number Generator

[2] Deterministic Random Number Generator: "algorithm for generating sequences of data approximating those of random numbers. The output of a DRNG is determined by its initial seed data"[1, section 3.2].

All the noise collected from the aforementioned sources goes into a data structure called *Entropy Pool* (with the only exception be the interrupt noise that gets collected in a structure called *fast pool* which is mixed in the entropy pool only at a later stage). Noise data gets modified while is added to the pool, and data from the pool to the DRNG based on the ChaCha20 stream cipher gets hashed beforehand using Blake2s.

An important feature of the entropy pool data structure is that it also holds the estimated bits of entropy available. "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"[1], while "when random data is obtained from the entropy pool, the number of generated random bytes is simply subtracted from entropy estimator"[1].

The estimator of entropy is needed for ensuring that, when we extract data from the pool, we do not extract predictable data, but actual (processed) noise. The idea is that the only unpredictable data is the noise, so it is imperative to make sure that any manipulation of such noise happens on actual noise and not predictable results of previous manipulations of noise [see for instance 1, section 3.3.1.3].

The ChaCha20 DRNG has a simple internal data structure which output depends from [1, section 3.3.2]. Populating this data structure with data got from the entropy pool is an action known as *seeding*.

The path from the noise sources to the generation of random data in the Linux kernel is:

1. A noise source collects noisy data and passes it along to the entropy pool through the `mix_pool_bytes` interface.
2. The entropy estimator is updated; this depends on the noise source.
3. Every 5 minutes, the ChaCha20 DRNG is reseeded with random data from the entropy pool up to 256 bits. If no entropy is available, the DRNG is not reseeded.

There is an exception at boot, where the interrupt noise is fundamental to bring the ChaCha20 to an initial seeded state. Another element in initial seeding is the seed file: at the shutdown of the system, some random data should be written to disk which is then used in the next boot to stir the entropy pools (but this data is awarded no entropy from the estimator)[1, sections 3.8, 3.11 and 4.1.2].

2.1 Raw entropy analysis

The report [1] does an in-depth analysis of each noise source in the Linux kernel to assess whether the kernel entropy estimations are correct.

What it is needed to know in this analysis is that the author argues that for each source of noise, only the output from the high resolution timestamp is considered, in his analysis, to have entropy. This conservative outlook for the analysis ensures that the actual amount of entropy in the noise is not overestimated.

The author then uses the tools for non-IID[3] input data provided by the NIST SP800-90B standard[6] to measure entropy. They also used the Shannon

[3]As the timestamp value is a monotonically increasing value, it cannot be IID

entropy (equation 1) and the Min-entropy Plug-in estimate[2] (equation 2) as comparisons, but in order to perform the calculations, they used the deltas between adjacent timestamps.

$$H(X) := -\sum_{x \in X} p(x) \log_2(p(x)) \tag{1}$$

$$H_{\min}(X) := -\log_2(\max_{x \in X}\{p(x)\}) \tag{2}$$

3 Alternative noise sources

Thanks to the previous research made by Emiliano Maccaferri, we had some ideas for the new noise sources to employ. These include

- The CPU voltage level
- Wireless connections signals
- CPU Jitter

In the end, the most promising path seemed to be to study the CPU Jitter, so that is what this work focuses on. There are different reasons for this: it seems like it is the high resolution timer that provides the majority of entropy. This claim is supported by [4]. The same source also seem to conclude that this source is indeed a good source of entropy, and it provides source code too.

One drawback of following this approach is that we do not know how it performs at boot time. Specifically, [1, section 6.3.4.2] states that the scheduler based noise source, which is heavily based on the high resolution timestamp, at boot time, does not have a lot of entropy (SP800-90B row-wise testing give an entropy of 0.07 bits per byte). In particular, "SP800-90B considers this noise source as inappropriate which should be treated with zero bits of entropy"[1].

There is conflicting information about this noise source, but it was decided to move forward with this nonetheless.

4 SP800-90b

The tests performed for estimating the amount of entropy are the ones listed in [6] using the software they provided in their repository [5].

The SP800-90B document also outlines the different components an entropy source should have[6]:

- The noise source itself, which can be an analog source which gets digitized, or a digital source. Noise sources **must be** non-deterministic because they are "ultimately responsible for the uncertainty associated with the bitstrings output by the entropy source"[6]. A noise source can be *physical*, that use dedicated hardware, or *non-physical*, that use system data to generate randomness. The output of the noise source is called *raw data*.
- Health tests must be run on the raw data to ensure that the noise source "continue to operate as expected"[6].

- Raw data may be processed through a conditioning component, which is responsible for "reducing bias and/or increasing the entropy rate of the resulting output bits"[6].

The entropy source should be designed so that it can be shown it is actually a source of entropy and that it outputs entropy "at a rate that meets or exceeds a specified value"[6]. So we need to be able to estimate a priori the amount of entropy by sampling its noise source, even though entropy cannot be measured[4].

In order to collect data to be analyzed, the standard specifies that it is necessary to collect:

1. "A *sequential* dataset of at least [one million] sample values obtained directly from the noise source"[6]. Possibly, more data can be collected for validation (and this will increase the accuracy of entropy estimation). In case it is impossible to get one million consecutive samples, "the concatenation of several smaller sets of consecutive samples [...] is allowed"[6].

2. In some cases (listed in [6]), if the entropy source uses a conditioning component, it is necessary to collect one million consecutive samples from the conditioning component output.

3. The entropy source must be restarted one thousand times. For each restart, one thousand consecutive samples must be collected from the noise source. This data form a matrix M, where $M_{i,j}$ represent the j^{th} sample from the i^{th} restart.

The standard then defines two different "tracks" to be used: the *IID-track* and the *non-IID track*. The IID-track must be used only when the conditions set in the document [6] are met.

As far as the CPU Jitter noise source is concerned, we do not have a mathematical proof that it is IID. However, as it can be seen by experiments in section 6.1, while the collection of the last 8 bits of the deltas are **not** IID because they fail both the chi-square and the permutation tests, if we collect only 4 bits such tests pass, therefore, it *might* be IID. We follow the non-IID track so that we do not have an overestimation of the collected entropy.

Then the entropy is estimated by taking the minimum value between:

1. The theoretical analysis of the entropy of the noise source (which is missing in this case)

2. The min-entropy estimated per sample using the methods described in SP800-90B

3. The min-entropy estimated per bit using the methods described in SP800-90B on the first million bits of the output

4.1 Analysis of CPU Jitter as a noise source according to SP800-90B

The CPU Jitter as a noise source:

- Should be unpredictable if we assume that[4]

[4]By definition of entropy [1, section 3.5.2]

  - No attacker having hardware level privileges is present.
  - No attacker having physical access to the CPU is present.
  - The CPU must be connected to peripherals; in this case, the RAM or any other component connected through a bus is assumed to be a peripheral. If no peripherals are connected to the CPU, some software is expected to run fully deterministically hence no entropy can be collected, but in this case general purpose computing is not allowed

- Is probably **not** stationary: at boot I expect it to generate less noisy data, but it's probably ok since the two moments are clearly separated and the boot time does not happen more than once and, in the end, the analysis will be performed at boot time.
- It should be protected against adversarial knowledge or influence, but I have no way to check for it. A possible test one can make is to run two entropy collection threads in two different CPU cores as much simultaneously as possible and then try to see if there is correlation between the two collected datasets. This, of course, is worth to be done at boot time.
- It should exhibit random behavior: the fact that data simply collected via a loop equivalent to the one shown in algorithm 1, in certain cases explained in section 6.1, **fails** the chi-square test done both by [5] and by the tool `ent`[7] it might serve as a hint that noise collected in this way is not always ready to be used as a random number yet.
- does **not** produce fixed-length strings, but it's easy to make it so: the final length of the noise data is fixed to 8 bits or less, but always the same number of bits.

*Note: a health test for the noise source **must** be provided. [4] and [3] provide one.*

5 A review of CPU Jitter by Müller

A good work that provides an analysis of CPU Jitter as a noise source is [4] by Müller.

Different processes in a modern computer can show difference in timings such as:

- Instruction pipelines
- Jitter between CPU and memory clock
- CPU frequency scaling and power management
- Caches used jointly by multiple CPUs
- Branch predictions
- Process migration between CPUs

Due to such impredictable components, the execution of instructions may have minuscule variations in execution time. The idea is to extract noise from those variations.

```
void main(void){
    uint64_t time1, time2, delta;
    for (int i = 0; i < samples; i++)
    {
        time1 = rdtsc();
        time2 = rdtsc();
        delta = time2 - time1;
        /* Collect delta somewhere */
    }
}
```

Listing 1: Simple CPU Jitter "entropy" collection

With this approach, the author concludes that, in kernel space, the Shannon Entropy for this noise source varies significantly from 1.3 to 3 bits.

6 My Implementation

Unlike Müller's implementation, which not only included a way to process the collected noise too, but also used different ways to compute the delta (see also [3]), the implementation I did for collecting CPU Jitter **conceptually** uses only a loop similar to the one presented in algorithm 1.

The implementation consists of:

- A kernel module that exposes three interfaces

  sleepns An interface that specifies how much time to wait before returning a measurement or how much time to wait between two measurements of the high resolution timer. The waiting is implemented through the function `ndelay`.

  lastdelta An unused interface that measures the time delta between two consecutive measurements of the high resolution timer, which can be separated by a delay if `sleepns != 0`.

  ticks An interface that waits `sleepns` nanoseconds (if `sleepns != 0`) and then returns the value of the high resolution timestamp.

- A python script that reads the `ticks` value from the kernel module, in a loop, reads the returned ticks value and writes on a file the list of measured deltas truncated to a certain number B of bits. The truncation is performed by taking the B least significant bits. The rationale for this choice is a test performed where B was set to $B = 24$. Then, each delta was split into three bytes $\Delta_k = B_1^k|B_2^k|B_3^k$ and three data streams have been formed: $S_1 = S_1^1|S_2^1|\cdots|S_n^1$, $S_2 = S_1^2|S_2^2|\cdots|S_n^2$, $S_3 = S_1^3|S_2^3|\cdots|S_n^3$. Then, each data stream has been analysed as described in section 6.1: it turns out most of the entropy is in the least significant bits, so inside $B_3^k \; \forall k = 1, \ldots, n$

This configuration has a lot of noise compared to an equivalent implementation done completely inside the kernel.

6.1 Analysis of the results

The analysis of the results has been done in the same way [1] does for the noise sources of Linux RNG. So I took the *unprocessed* noise provided by the Jitter

Byte (from MSB)	Min est.[5] entropy	Max est. entropy	Avg est. entropy
0	0	0	0
1	0.562	0.700	0.631
2	6.638	6.990	6.749

Table 1: byte-per-byte analysis

Min est. entropy	Max est. entropy	Avg est. entropy
6.638	6.98	6.730

Table 2: Tests with last 8 bits collection

noise "collector" and I analysed it using the approach described in [1, section 6.2], which uses the metrics published in [6] and implemented in [5].

Actually, SP800-90B[6] requires an initial entropy estimate for the noise source output. However, since a detailed analysis of the noise source is missing, also this entropy estimate is missing in this work, so every consideration made about entropy must be taken with a grain of salt until a theoretical analysis of the noise source is able to obtain such estimate; a theoretical estimate is present in [4], but it is based on Müller's more complex architecture.

Each test was performed 10 times, with `sleepns = 0` and with no user space sleeping.

The analysis has been conducted in the following scenarios:

1. As stated before, deltas were collected with a precision of 24 bits and then a byte-per-byte analysis followed. 1000000 delta samples with a precision of 24 bits have been collected and then they were split as decribed in section 6. The result of such analysis is collected in table 1

   As we can see from the table, most of the noise is concentrated in the least significant byte, hence the tests have been performed only on bits of the LSB. The script used for executing this analysis is `test-noise-bytes.sh 10 1000000 deltas_all`

2. Collection of the least significant byte of 1000000 deltas. Results are shown in table 2. The script used for executing this analysis is `collect-raw-noise.sh 10 1000000 8 deltas_8`

3. Collection of the least significant 4 bits of least significant byte of 1000000 deltas. Results are shown in table 3. Note that in this case the maximum of entropy is 4. The script used for executing this analysis is `collect-raw-noise.sh 10 1000000 4 deltas_4`

4. The restart test was conducted only once due to the fact that it has little sense to run this in userspace as the noise source is so unpredictable (and impossible to restart) that it can only be successful. In fact, the only reason I run this test is to help anyone else who will conduct additional testing to do a restart test in an environment where it provides a meaningful result. In order to run this test, 1000 restarts of the noise source must be done; for each restart, 1000 samples are collected. Those samples

[5] estimated by using the SP800-90B tool[5]

Min est. entropy	Max est. entropy	Avg est. entropy
3.421	3.659	3.538

Table 3: Tests with last 4 bits collection

are organized in a matrix $M_{i,j} \in \mathbb{R}^{1000\times1000}$ where an element $m_{i,j}$ is the j^{th} sample of the i^{th} restart. A restart in this case means something that resets the noise source in some ways. In this particular case, if the CPU Jitter is collected during the boot phase, doing 1000 reboots and collect the CPU Jitter at boot **could** be an idea to take into consideration. A peculiarity of this test is that an initial entropy estimate (H_i) must be provided. The test will then execute either the IID test or the non-IID one (depending on the noise source) on two different datasets:

Row dataset which is composed of the concatenation of the rows of matrix M: $m_{1,1}|m_{1,2}|\cdots|m_{1,1000}|m_{2,1}|\cdots|m_{2,1000}|\cdots|m_{1000,1}|\cdots|m_{1000,1000}$

Column dataset which is composed of the concatenation of the columns of matrix M: $m_{1,1}|m_{2,1}|\cdots|m_{1000,1}|m_{1,2}|\cdots|m_{1000,2}|\cdots|m_{1,1000}|\cdots|m_{1000,1000}$

Two different entropy values will be obtained, H_c from the analysis of the column dataset and H_r for the analysis of the row dataset. Then, two types of tests are performed, the *validation test* checks if $\min\{H_r, H_c\} < \frac{H_i}{2}$, in which case it **fails**. The *sanity check* "checks the frequency of the most common value in the rows and the columns of the matrix M. If this frequency is significantly greater than the expected value, given the initial entropy estimate H_i[...], the restart test fails". If either test fail, no entropy should be awarded to the noise source.

A more detailed analysis of the details of these tests is available in [6].

As for the value of H_i, it **should** be a good idea to keep it conservative, so that even in the worst case scenarios, there is no overstimation of entropy.

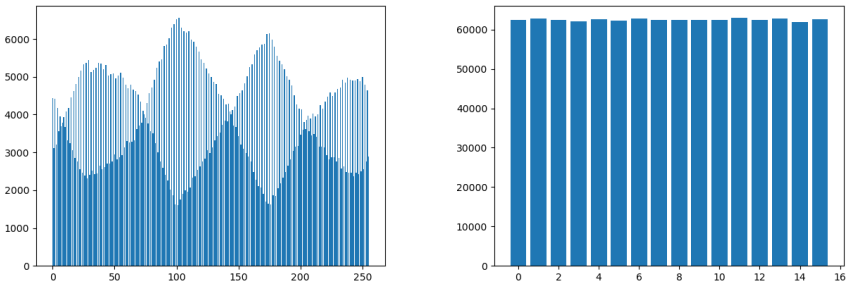
The reason I also did a 4 bit test is the analysis of the frequency graph for the previous tests. Figure 1 show very different distribution shapes for the 4 and 8 least significant bits datasets. Such distributions are pretty much the same for all the 10 experiments.

Moreover, tests done with 4 bits deltas pass the IID test found in [5][6]. This does **not** mean they are IID, as to be considered IID there are more considerations to be done (see [6, section 3.1.2]).

Finally, one can extract 2000000 4-bit deltas, and then, by using the script `compact.py`, it is possible to create a sequence of 8-bits values simply by "stiching" all the values together neglecting the zeroes (for instance `0x05 0x0A 0x0C 0x00` becomes `0x5A 0xC0`). This file can be analysed with the `ent` tool, which is normally applied on random data generated by the full random number generator (for instance, see [1, section 8])[7]. A sample output of that tool is given in listing 2 which shows that it's not perfect, but it's more random than a sequence of "native" 8-bit deltas.

[6] In order to run such tests, run the `ea_iid` program on the output of `extractor.py`

[7] In this case, the entropy was a bit overestimated



(a) Probability distribution of collected 8 bit values

(b) Probability distribution of collected 4 bit values

Figure 1: Probability distribution of collected values

```
Entropy = 7.999818 bits per byte.

Optimum compression would reduce the size
of this 1000000 byte file by 0 percent.

Chi square distribution for 1000000 samples is
    251.93, and randomly
would exceed this value 54.27 percent of the times.

Arithmetic mean value of data bytes is 127.5241
    (127.5 = random).
Monte Carlo value for Pi is 3.138372553 (error 0.10
    percent).
Serial correlation coefficient is -0.000223 (totally
    uncorrelated = 0.0).
```

Listing 2: Sample output of `ent` applied to a file obtained from 4-bits deltas stitched together

Note that the entropy estimation is useful only as a part of a serie of indicators to see if a sequence has enough randomness. For instance, the sequence $0, 1, 2, 3, \ldots, 255$ has 8 bits of Shannon entropy, but it is definitely **not** random. Remember that, by definition, entropy cannot be measured[1, section 3.5.2], so all calculations of entropy should be taken as estimations. A huge missing piece in this work is a theoretical analysis of the entropy of the CPU Jitter raw noise source.

7 Small documentation of provided scripts

Note: none of the following scripts check for the existence of a file before overwriting it. So be careful not to accidentally overwrite useful data.

7.1 analyse_file.py

The syntax for calling this script is `analyse_file.py <length of a word in bits> <filename> <whole length: true|false>` where:

- `length of a word in bits` is the length in bits of the deltas in the file (this can assume the values 1, 2, 4 or 8)
- `filename`, which is the input filename
- `whole length`, which can be either `true` or `false`, tells if the bits are all in the least significant part of each byte or if the content of the file should be treated as a stream of bits. For example, if the file contains `0x4C 0xA1 0x34`, the length in bits is 4 and `whole length` is `true`, then the extracted numbers will be `0x0C, 0x01, 0x34`, while if `whole length` is `false`, the extracted numbers will be `0x04, 0x0C, 0x0A, 0x01, 0x03, 0x04`.

This script computes the Shannon entropy and the min-entropy according to equations 1 and 2 and display a frequency graph for each symbol (defined as above) present in the file `filename`.

7.2 collect-raw-noise.sh

This script loads the kernel module and repeatedly calls `extractor.py` (see section 7.5) to collect samples of raw noise data. This data is then processed and the minimum, maximum and average of the estimates is computed and shown.

The syntax for calling this script is `collect-raw-noise.sh <repetitions> <samples> <bits to collect> <output directory>`:

- `repetitions` is the number of times the script `extractor.py` is called
- `samples` is the number of samples that `extractor.py` extracts
- `bits to collect` is the number of least significant bits that are collcted for each sample
- `output directory` is the directory where files are generated

The generated file names are composed of an incremental number from 1 to `repetitions` (inclusive) and the suffix `.bin`.

7.3 compact.py

This script takes a file generated with `extractor.py` where the least significant 4 bits of each delta are collected, takes the least significant 4 bits of each two consecutive bytes and concatenates them together in a single byte (as described in section 6.1, for instance `0x05 0x0A 0x0C 0x00` becomes `0x5A 0xC0`).

The syntax for calling this script is `python3 compact.py <filename> <output filename>`:

- `filename` is the path of the input filename
- `output filename` is the name of the output filename

7.4 `extract-and-evaluate.sh`

This script generates a file with `extractor.py` and then splits each byte in different files according to the number of bits given as input using the `splitter.py` script (see section 6: each delta is split into b bytes $\Delta_k = B_1^k|B_2^k|\cdots|B_b^k$ and b data streams are formed: $S_1 = S_1^1|S_2^1|\cdots|S_n^1$, $S_2 = S_1^2|S_2^2|\cdots|S_n^2$, ..., $S_b = S_1^b|S_2^b|\cdots|S_n^b$, each of which is saved in a different file). Those files are then analysed using the non-IID tool provided by NIST[5] and report the min-entropy output from such program.

The syntax for calling this script is `./extract-and-evaluate.sh <bits to extract> <samples> <output filename> [-prefix <prefix>] [-sync] [-pid <pid>]`:

- `bits to extract` is the number of bits `extractor.py` will extract. This number should be greater than 8, otherwise the default behavior would be to just copy the file, hence, the script does not accept to be called with a number of bits less than or equal to 8.
- `samples` the number of samples are collected by `extractor.py`.
- `output filename` the name of the file to put the whole, non-split, samples.
- `prefix` the split files will be named `prefix-output_filename<i>`, where `i` is a monotonically increasing integer; by default this prefix is `f`, but it can be customized with this option.
- `sync` if set, it waits for reception of a SIGUSR1 signal before continuing; if this option is set, the PID of the process is output on standard error.
- `pid` if set, when the entropy collection starts, SIGUSR1 is sent to the process specified by `pid`.

7.5 `extractor.py`

This script extracts data by repeatedly querying the `ticks` interface of the `deltats` module.

The syntax for calling this script is `python3 extractor.py <bits to extract> <samples> <time to sleep microseconds> <output filename> <ticks|delta>`:

- `bits to extract` is the number of least significant bits we extract from the relevant value; all remaining bits are either ignored or set to zero.
- `samples` is the number of samples of the relevant value to extract and write to the output file.
- `time to sleep in microseconds`, if different from 0, it makes the script do a call to the `time.sleep` python function to sleep for the specified amount of microseconds.
- `output filename` is the name of the output file where all relevant values are stored
- `ticks|delta` is an option to specify the relevant value: if this option is `ticks`, then the value read from the `ticks` interface is used without any postprocessing, whereas if the option is `delta`, then the relevant value is the difference between two consecutive readings of `ticks`.

7.6 `restart.py`

This script produces a file which should be compliant with the row-file format of SP800-90B[6] in order to analyse it with the appropriate software[5].

The syntax for calling this script is `python3 restart.py <bits to extract> <output filename>`:

- `bits to extract` can be either $1, 2, 4 \text{or} 8$ and it is the number of least significant bits that the called script `extractor.py` will extract.
- `output filename` is the name of the file where all gathered noise will be written into in row-dataset format (see [6, section 3.1.4.1]).

This script does **not** do an automatic analysis of the data because an initial entropy estimate is required to run this test.

7.7 `splitter.py`

This script splits a file containing a stream of bytes into multiple files containing streams of files. That is, given a stream of bytes $\mathbf{B} = B_0|B_1|B_2|\cdots|B_n$ this script produces k files each containing a stream of bytes $S_i = B_i|B_{i+k}|B_{i+2k}|\cdots \forall i = 0, \ldots, k-1$. k must divide n, otherwise although the script does not produce any error message, the k files will have different sizes.

The syntax for calling this script is `python3 splitter.py <bytes number> <input file> <prefix for output>`:

- `bytes number` is the number k of streams.
- `input file` is the file that contains the original stream of bytes $\mathbf{B}$.
- `prefix for output` the output files that are generated by this script are named `<prefix><i>`, where `i` is a monotonically increasing integer that starts from 0 and ends with $k - 1$.

7.8 `test-noise-bytes.sh`

This script is for executing the experiment described in section 6.1, table 1. It is similar in spirit to `extract-and-evaluate.sh` (section 7.4), but some parameters have been fixed to perform the aforementioned experiment and those experiments are repeated more than once to capture basic statistics such as minimum, maximum and average estimated entropy in each byte stream.

The syntax of this script is `test-noise-bytes.sh <repetitions> <samples per test> <output directory>`:

- `repetitions` is the number of byte streams to collect
- `samples per test` is the number of samples each byte stream contains
- `output directory` is the path of the directory where all the files containing the streams will be written into

8 Final considerations

It seems like the entire stack used for these tests is capable of providing some noise to generate good random numbers with, although this statement still needs to be proved following the specifications of the set of standards SP800-90, which this document does not do.

Without much further analysis from experts, I would **not** use this as an entropy source in a production environment where actual randomness of data may be critical (for instance, for cryptographic purposes). In my opinion, a specialized hardware TRNG is more suitable and secure than a non-physical noise source in a (somewhat) predictable environment at providing safe randomness for applications that require it.

9 Improvements

Müller in his articles ([4] and [3]) provide not only other sources of jitter for the CPU (for instance, memory access) but also a way to combine different readings of deltas into a 64-bit random number and demonstrates everything with source code capable of updating the Linux entropy pool by writing entropy data to `/dev/random`. His approach could be better suited for a final product in place of the naïve approach described herein to collect noise. It would be nice to estimate also the entropy with his full architecture at boot time of the real-time operating system used.

References

[1] Federal Office for Information Security. *Documentation and Analysis of the Linux Random Number Generator*. Ed. by Stephan Müller. Apr. 27, 2022.

[2] Wolfgang Killmann and Werner Schindler. *A proposal for: Functionality classes for random number generators*. Sept. 18, 2011.

[3] Stephan Müller. *CPU Time Jitter Based Non-Physical True Random Number Generator*. version 2.2.0 for the current versions of the Linux Kernel. Oct. 24, 2022.

[4] Stephan Müller. *CPU Time Jitter Based Non-Physical True Random Number Generator*.

[5] National Institute of Standards and Technology. *SP800-90B entropy assessment*. 2023. URL: `https://github.com/usnistgov/SP800-90B_EntropyAssessment`.

[6] Meltem Sönmez Turan et al. *NIST Special Publication 800-90B Recommendation for the Entropy Sources Used For Random Bit Generation*. Jan. 2018.

[7] John Walker. *Ent: A Pseudorandom Number Sequence Test Program*. Jan. 28, 2008. URL: `https://github.com/Fourmilab/ent_random_sequence_tester`.

Disclaimer

THERE IS NO WARRANTY FOR THIS WORK, TO THE EXTENT PERMITTED BY APPLICABLE LAW. THE AUTHOR PROVIDE THIS WORK "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THIS WORK IS WITH YOU.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW WILL THE AUTHOR BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THIS WORK (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES), EVEN IF THE AUTHOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

THE DISCLAIMER OF WARRANTIES AND LIMITATION OF LIABILITY PROVIDED ABOVE SHALL BE INTERPRETED IN A MANNER THAT, TO THE EXTENT POSSIBLE, MOST CLOSELY APPROXIMATES AN ABSOLUTE DISCLAIMER AND WAIVER OF ALL LIABILITY.