CPU Time Jitter Based Non-Physical True Random Number Generator

Stephan Müller <smueller@chronox.de>

July 1, 2022

Abstract

Today's operating systems provide non-physical true random number generators which are based on hardware events. With the advent of virtualization and the ever growing need of more high-quality random numbers, these random number generators reach their limits. Additional sources of entropy must be opened up. This document introduces an entropy source based on CPU execution time jitter. The design and implementation of a non-physical true random number generator, the CPU Jitter random number generator, its statistical properties and the maintenance and behavior of entropy is discussed in this document.

Contents

1 Introduction

Each modern general purpose operating system offers a non-physical true random number generator. In Unix derivatives, the device file `/dev/random` allows user space applications to access such a random number generator. Most of these random number generators obtain their entropy from time variances of hardware events, such as block device accesses, interrupts triggered by devices, operations on human interface devices (HID) like keyboards and mice, and other devices.

Limitations of these entropy sources are visible. These include:

- Hardware events do not occur fast enough to satisfy the ever grown needs of high-quality random numbers. Today's implementation of such hardware event based random number generators provide a hybrid formed by the joining of the hardware event collection which is post-processed by a deterministic whitening function. In case of insufficient entropy, the whitening function ensures that the continuously generated output still behaves random even when new entropy is lacking.

- Virtualized environments remove an operating system from direct hardware access. The properties of the observed hardware events in a virtualized environment do not match with the properties required by the non-physical true random number generators. The implication on the entropy collected by these random number generators is not well researched, but it is safe to assume that the entropy is overestimated by the standard operating system's random number generators.

- Depending on the usage environment of the operating system, entire classes of hardware devices may be missing and can therefore not be used as entropy source. For example, a server system located in a server lab typically does not have any human interface devices attached.

- A number of the operating system's non-physical true random number generators use block devices, such as hard disks as entropy source. The heart of the entropy lies in the timing variances when accessing such disks which depend on the spin angle of the disk or the location of the read heads at the time of the access request. The more and more often used Solid State Disks (SSDs) advertise themselves as block devices to the operating system but yet lack the physical phenomenon that is expected to deliver entropy. The implication is that the SSD block devices cannot be used as entropy source either, although they are partially still treated as entropy source by the standard operating system's random number generators.

- On Linux, the majority of the entropy for the `input_pool` behind `/dev/random` is gathered from the `random_get_entropy` time stamp. However, that time stamp function returns 0 hard coded on several architectures, such as MIPS. Thus, there is not much entropy that is present in the entropy pool behind `/dev/random` or `/dev/urandom`.

- Current cache-based attacks allow unprivileged applications to observe the operation of other processes, privileged code as well as the kernel. Thus, it is desirable to have fast moving keys. This applies also to the seed keys used for deterministic random number generators.

How can these challenges be met? A new source of entropy must be developed that is not affected by the mentioned problems.

This document introduces a non-physical true random number generator, called CPU Jitter random number generator, which is developed to meet the following goals:

1. The random number generator shall only operate on demand. Other random number generators constantly operate in its lifetime, regardless whether the operation is needed or not, binding computing resources.

2. The random number generator shall always return random numbers with a speed that satisfies today's requirement for random numbers. The random number generator shall be able to be used synchronously with the random number consuming application, such as the seeding of a deterministic random number generator.

3. The random number generator shall not block the request for user noticeable time spans.

4. The random number generator shall deliver high-quality random numbers when used in virtualized environments.

5. The random number generator shall not require a seeding with data from previous instances of the random number generator.

6. The random number generator shall work equally well in kernel space and user space.

7. The random number generator implementation shall be small, and easily understood.

8. The random number generator shall provide a decentralized source of entropy. Every user that needs random numbers executes its own instance of the CPU Jitter random number generator. Any denial of service attacks or other attacks against a central entropy source with the goal to decrease the level of entropy maintained by the central entropy source is eliminated. The goal is that there is no need of a central `/dev/random` or `/dev/urandom` device.

9. The random number generator shall provide perfect forward and backward secrecy, even when the internal state becomes known.

Apart from these implementation goals, the random number generator must comply with the general quality requirements placed on any (non-)physical true random number generator:

Entropy The random numbers delivered by the generator must contain true information theoretical entropy. The information theoretical entropy is based on the definition given by Shannon.

Statistical Properties The random number bit stream generated by the generator must not follow any statistical significant patterns. The output of the proposed random number generator must pass all standard statistical tools analyzing the quality of a random data stream.

These two basic principles will be the guiding central theme in assessing the quality of the presented CPU Jitter random number generator.

The document contains the following parts:

- Discussion of the noise source in chapter 2

- Presentation of CPU Jitter random number generator design in chapter 3

- Discussion of the statistical properties of the random number generator output in chapter 4

- Assessment of the entropy behavior in the random number generator in chapter 5

But now away with the theoretical blabber: show me the facts! What is the central source of entropy that is the basis for the presented random number generator?

1.1 Related Work

Another implementation of random number generators based on CPU jitter is provided with HAVEGEd. A similar work is proposed in The maxwell random number generator by Sandy Harris.

An analysis of the system-inherent entropy in the Linux kernel is given with the Analysis of inherent randomness of the Linux kernel by N. Mc Guire, P. Okech, G. Schiesser.

1.2 Applicable Code Version

This document applies and describes the Jitter RNG code revision 3.3.1.

2 CPU Execution Time Jitter

We do have deterministically operating CPUs, right? Our operating systems behave fully deterministically, right? If that would not be the case, how could we ever have operating systems using CPUs that deliver a deterministic functionality.

Current hardware supports the efficient execution of the operating system by providing hardware facilities, including:

- CPU instruction pipelines. Their fill level have an impact on the execution time of one instruction. These pipelines therefore add to the CPU execution timing jitter.

- The CPU clock cycle is different than the memory bus clock speed. Therefore, the CPU has to enter wait states for the synchronization of any memory access where the time delay added for the wait states adds to time variances.

- The CPU frequency scaling which alters the processing speed of instructions.

- The CPU power management which may disable CPU features that have an impact on the execution speed of sets of instructions.

- Instruction and data caches with their varying information – tests showed that before the caches are filled with the test code and the CPU Jitter random number generator code, the time deltas are bigger by a factor of two to three;

- CPU topology and caches used jointly by multiple CPUs;

- Different types of memory (L1, L2, L3 caches, main memory) are operated with different frequencies which require the introduction of wait states to synchronize the CPU accessing these memory components.

In addition to the hardware nondeterminism, the following operating system caused system usage adds to the non-deterministic execution time of sets of instructions:

- CPU frequency scaling depending on the work load;

- Branch prediction units;

- TLB caches;

- Moving of the execution of processes from one CPU to another by the scheduler;

- Hardware interrupts that are required to be handled by the operating system immediately after the delivery by the CPU regardless what the operating system was doing in the mean time;

- Large memory segments whose access times may vary due to the physical distance from the CPU.

2.1 Assumptions

The CPU Jitter random number generator is based on a number of assumptions. Only when these assumptions are upheld, the data generated can be believed to contain the requested random numbers. The following assumptions apply:

- Attacker having hardware level privileges or attacker controlling the execution environment[1] of the operating system are assumed to be not present. With hardware level privilege, on some CPU it may be possible to change the state of the CPU such as that caches are disabled. In addition, millicode may be changed such that operations of the CPU are altered such that operations are not executed any more. The assumption is considered to be unproblematic, because if an attacker has hardware level privilege, the collection of entropy is the least of our worries as the attacker may simply bypass the entropy collection and furnish a preset key to the random numbers-seeking application.

- Attacker with physical access to the CPU interior is assumed to be not present. In some CPUs, physical access may allow enabling debug states or the readout of the entire CPU state at one particular time. With the CPU state, it may be possible to deduct upcoming variations when the CPU Jitter random number generator is executed immediately after taking a CPU state snapshot. An attacker with this capability, however, is also able to read out the entire memory. Therefore, when launching the attack shortly after the entropy is collected, the attacker could read out the key or seed material, bypassing the entropy collection. Again, with such an attacker, the entropy collection is the least of our worries in this case.

If attackers are absent, the assumptions are trivially met.

2.2 Jitter Depicted

With the high complexity of modern operating systems and their big monolithic kernels, all the mentioned hardware components are extensively used. However, due to the complexity, nobody is able to determine which is the fill level of the caches or branch prediction units, or the precise location of data in memory at one given time.

This implies that the execution of instruction may have minuscule variations in execution time. In addition, modern CPUs have a high-resolution timer or instruction counter that is so precise that they are impacted by these tiny variations. For example, modern x86 CPUs have a TSC clock whose resolution is in the nanosecond range.

These variations in the execution time of an identical set of CPU instructions can be visualized. Figure 2.1 illustrates the variation of the following code sequence:

Listing 1: Collection of Time Variances in Userspace

```
static inline void jent_get_nstime(uint64_t *out)
{
...
        if (clock_gettime(CLOCK_REALTIME, &time) == 0)
```

---

[1]Virtual Machine Monitors, Simulators, Emulators, Hypervisors, etc.

```
...
}

void main(void)
{
...
        for(i = 0; (SAMPLE_COUNT + CACHE_KILL) >i ; i++)
        {
...
                jent_get_nstime(&time);
                jent_get_nstime(&time2);
                delta = time2 - time;
...
}
```

The contents of the variable `delta` is not identical between the individual
loop iterations. When running the code with a loop count of 1,000,000 on an
otherwise quiet system to avoid additional time variance from the noise of other
processes, we get data as illustrated in figure 2.1.

Distribution of time delta in userspace up to 64
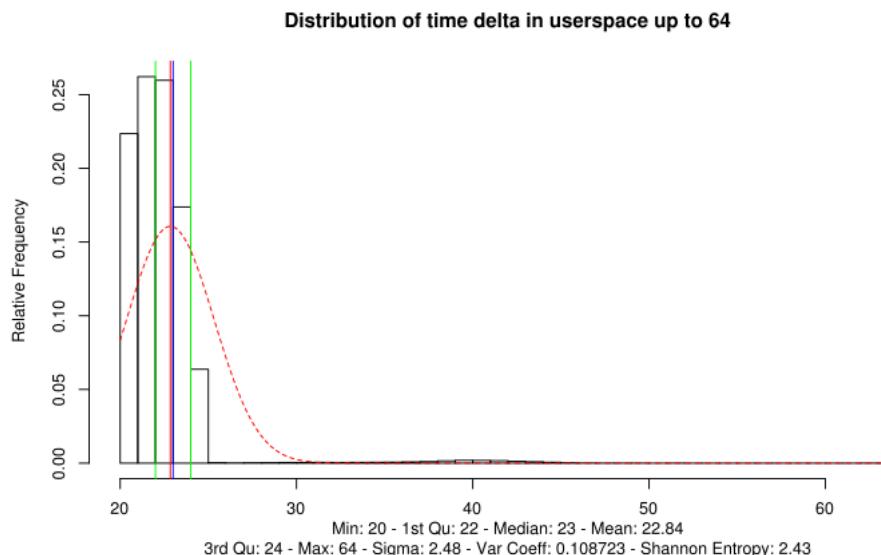
Figure 2.1: Distribution of time variances in user space over 1.000.000 loops

Please note that the actual results of the aforementioned code contains a few
exceptionally large deltas as an operating system can never be fully quiet. Thus,
the test results were processed to cut off all time deltas above 64. The limitation
of the graph to all variations up to 64 can be considered as a "magnification"
of the data set to the interesting values.

Figure 2.1 contains the following information of interest to us:

- The bar diagram shows the relative frequency of the different delta values
  measured by the code. For example, the delta value of 22 (nanoseconds –
  note the used timer returns data with nanosecond precision) was measured
  at 25% of all deltas. The value 23 (nanoseconds) was measured at about
  25% of all time deltas.

- The red and blue vertical lines indicate the mean and median values. The mean and median is printed in the legend below the diagram. Note, they may overlap each other if they are too close. Use the legend beneath the diagram as a guidance in this case.

- The two green vertical lines indicate the first and third quartile of the distribution. Again, the values of the quartiles are listed in the legend.

- The red dotted line indicates a normal distribution defined by the measured mean and the measured standard derivation. The value of the standard derivation is given again in the legend.

- Finally, the legend contains the value for the Shannon Entropy that the measured test sample contains. The Shannon Entropy is calculated with the formula specified in section 5.1 using the observations after cutting off the outliers above the threshold mentioned above.

The graph together with the code now illustrates the variation in execution time of the very same set of operations – it illustrates the CPU execution time jitter for a very tight loop. As these variations are based on the aforementioned complexity of the operating system and its use of hardware mechanisms, no observer can deduce the next variation with full certainty even though the observer is able to fully monitor the operation of the system. And these non-deterministic variations are the foundation of the proposed CPU Jitter random number generator.

As the CPU Jitter random number generator is intended to work in kernel space as well, the same analysis is performed for the kernel. The following code illustrates the heart of the data collection disregarding the details on copying the data to user space.

Listing 2: Collection of Time Variances in Kernel Space

```
static int jent_timer(char *data, size_t len)
{
        __u64 time, time2;
        time = time2 = 0;
...
        time = random_get_entropy();
        time2 = random_get_entropy();
        snprintf(data, len, "%lld\n", (time2 - time));
...
}
```

Although the code sequence is slightly different to the user space code due to the use of the *architecture-dependent* processor cycle function call `random_get_entropy`, the approach is identical: obtaining two time stamps and returning the delta between both. The time stamp variance collection is invoked 30.000.000 times to obtain the graph presented in figure 2.2[2].

_____

[2]The generation of the given number of time deltas is very fast, typically less than 10 seconds. Thus, the shown graph is not fully representative. When re-performing the test, the distribution varies greatly, including the Shannon Entropy. The lowest observed value was in the 1.3 range and the highest was about 3. The reason for not obtaining a longer sample is simply resources: calculating the graph would take more than 8 GB of RAM.

Distribution of time delta in kernel up to 128

Min: 18 - 1st Qu: 21 - Median: 24 - Mean: 25.17
3rd Qu: 24 - Max: 126 - Sigma: 7.43 - Var Coeff: 0.295166 - Shannon Entropy: 2.14
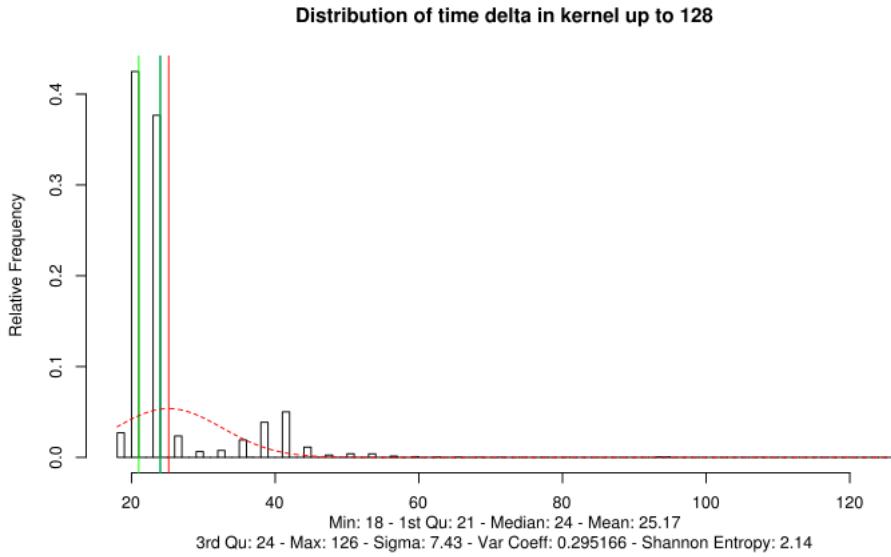
Figure 2.2: Distribution of time variances in kernel space over 10.000.000 loops

Striking differences between the timer variances in kernel and user space can be detected:

- The timer delta value has obvious holes in its distribution. The reason for that is the following observation: the processor cycle counter does not increment in steps of one, but in steps of three[3].

- However, the user space time stamp delta is much more narrowly distributed around the mean of the distribution. The kernel time stamp deltas have a much wider range. Therefore, the Shannon Entropy value of the kernel space distribution is larger than the one from the user space distribution.

Even with the kernel time stamp incremented in steps of three, the user space and the kernel space time stamps show a distribution and fluctuation. When looking at the sequence of time deltas gathered during testing, no pattern can be detected. Therefore, the fluctuation and the resulting distribution are not based on a repeating pattern and must be considered random.

The tests were executed on an Intel Core i7 with 2.7GHz. As the tests always consume much CPU power, the frequency of the CPU was always at 2.7 GHz during the tests. Surprisingly, when setting the maximum speed of the CPU to 800MHz, which is the lowest setting on the test machine, the distribution of the kernel timer variations hardly changes. For user space, the timer variations are larger compared to a fast CPU on an otherwise quiet system as depicted in figure 2.3. As the variations are even on a slower system, all subsequent

---

[3]It is important to note that this was observed on an Intel Core i7-2620 processor. Other processors and especially other architectures may show a different pattern in the incrementation.

discussions will cover the worse case of the fast CPU speed illustrated above as its variations inherently has less entropy.

Distribution of time delta for two sequential timer calls in userspace-800MHz up to 150

Min: 21 - 1st Qu: 22 - Median: 23 - Mean: 26.23
3rd Qu: 24 - Max: 150 - Sigma: 10.51 - Var Coeff: 0.400764 - Shannon Entropy: 3.11
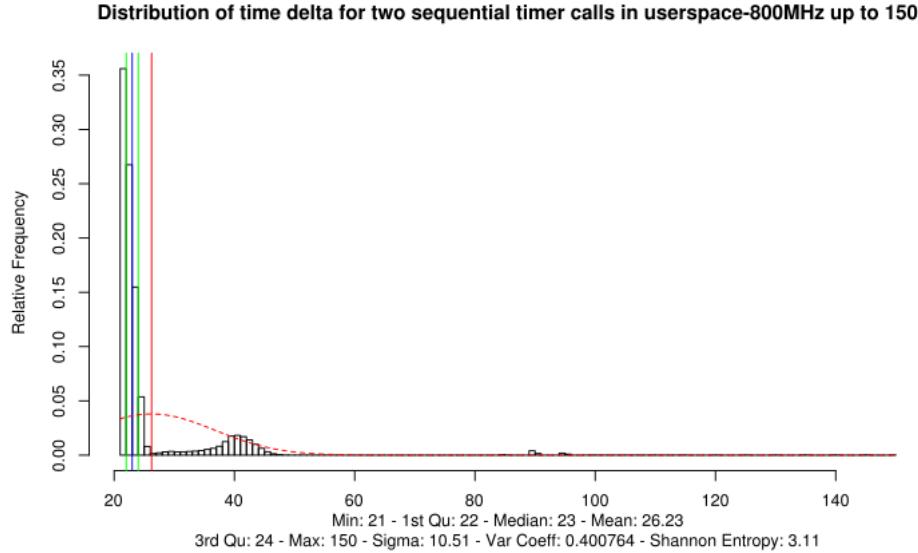
Figure 2.3: Distribution of time variances in user space over 1.000.000 loops at 800 MHz

Now that we have established the basic source of entropy, the subsequent design description of the random number generator must explain the following two aspects which are the basic quality requirements discussed in chapter 1 applied to our entropy phenomenon:

1. The random number generator design must be capable of preserving and collecting the entropy from the discussed phenomenon. Thus, the random number generator must be able to "compress" the entropy phenomenon.

2. The random number generator must use the observed CPU execution time jitter to generate an output bit string that delivers the random numbers to a caller. That output string must not show any statistical anomalies that allow an observer to deduce any random numbers or increase the probability when guessing random numbers and thus reducing its entropy.

With the following chapter, the design of the random number generator is presented. Both requirements will be discussed.

3   Random Number Generator Design

The CPU Jitter random number generator uses the above illustrated operation to read the high-resolution timer for obtaining time stamps. At the same time it performs operations that are subject to the CPU execution time jitter which also impact the time stamp readings.

3.1 Maintenance of Entropy

The heart of the random number generator is illustrated in figure 3.1.
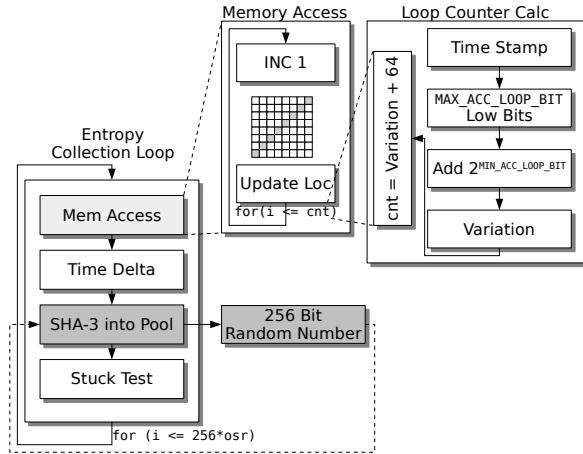
Figure 3.1: Entropy Collection Operation

The basic concept that is implemented with the CPU Jitter RNG can be summarized as follows: The unpredictable phenomenon of variances in the execution time of a given set of instructions is collected and accumulated. The accumulation is implemented using SHA-3 256. The measurement of the execution time jitter is performed over the post-processing logic of the SHA-3 256 and supportive functions. I.e. the CPU Jitter RNG measures the execution time of the SHA-3 256 with its supporting functions (and the additional noise source of the memory access) where the execution time is then injected into the SHA-3-maintained entropy pool.

The random number generator maintains a 256 bit entropy pool, that is indicated with the dark gray shaded boxes in figure 3.1 which identify the entropy pool at two different times in the processing. The size is equal to the SHA-3 256 output size. The light gray shaded boxes indicate the noise sources of the random number generator is based on.

In a big picture, the random number generator implements an entropy collection loop that

1. invokes memory accesses to induce timing variations,

2. fetches a time stamp to calculate a delta to the time stamp of the previous loop iteration,

3. inject the time delta value into the entropy pool by hashing the content of the entropy pool with the new value to form a new state of the entropy pool, and

4. verifies that basic properties of the time stamp are met by applying a health test.

The loop is executed exactly 256 times multiplied by the oversampling rate (osr) as each loop iteration is intended to provide at least $1/_{osr}$ bit of entropy added to the entropy pool. This implies that after the completion of all rounds, at least 256 bits of entropy are collected and compressed into the 256 bit entropy pool – i.e. the entropy pool contains full entropy.

The following subsection discuss every step in detail.

When considering that the time delta is always calculated by calculating the delta to the previous loop iteration, and the fact that the majority of the execution time is spend in the SHA-3 hashing operation, the central idea of the CPU Jitter Random Number Generator is to measure the execution time jitter over the execution of the SHA-3 hashing operation as well as the memory access.

3.1.1 Noise Source: Memory Access

For implementing memory access, memory is allocated during the allocation time of the entropy collector. The memory access operation is defined by the following values:

- Size of a memory block,

- State of a pseudo random number generator which defines the next memory location in the memory block..

To perform a read and write access, one byte is simply incremented by one and wrapped at 255. The code ensures a quasi-random access of all memory locations in the block where each location has the equal probability of being accessed.

3.1.2 Obtaining Time Delta

The time delta is obtained by:

1. Reading a time stamp,

2. Subtracting that time stamp from the time stamp calculated in the previous loop iteration,

3. Storing the current time stamp for use in the next loop iteration to calculate the next delta.

For every new request to generate a new random number, the first iteration of the loop is used to "prime" the delta calculation. All steps of the entropy collection loop are performed. However, this first iteration of the entropy collection loop does not impact the number of iterations used for entropy collection. This is implemented by executing one more loop iteration than specified for the generation of the current random number.

When a new random number is to be calculated, i.e. the entropy collection loop is triggered anew, the previous contents of the entropy pool, which is used as a random number in the previous round is reused. The reusing shall just mix the data in the entropy pool even more. But the implementation does not rely on any properties of that data. The mixing of new time stamps into the entropy pool using SHA-3 ensures that any entropy which may have been

left over from the previous entropy collection loop run is still preserved. If no entropy is left, which is the base case in the entropy assessment, the already arbitrary bit pattern in the entropy pool does not negatively affect the addition of new entropy in the current round.

When the time delta is obtained, also the following properties of the time are measured: the first, second and third derivative of the time. Only when all three values are not 0, the SHA-3 hashing loop iteration is counted as part of the 256 loop iterations. Section 3.1.3 explains how this validity is enforced.

3.1.3 Noise Source: Inject Time Delta Into Entropy Pool Using SHA-3

The SHA-3 hash operation is depicted by the left side of figure 3.1[4].

The hash operation is considered as a noise source as the execution of the hash operation contains variations that is measured with the time delta measurement from the previous step.

To apply the SHA-3 256 hash, the time delta value of 64 bits hashed with the entropy pool ~~and the number of invocations of the SHA3 256 operation (commonly 1) to form a new state of the entropy pool~~. In addition, a SHA3-256 hash of the following data is performed:

- RCT count value,

- APT count value,

- APT cutoff value,

- APT window value,

- APT base value

The resulting message digest is inserted into the entropy pool with a SHA3-256 update as well.

After the hash operation for all bits is completed, the result of the stuck test is enforced: If the stuck test is negative, i.e. the time delta is considered to be not stuck, the received entropy value is counted towards gathering 256 time deltas.

If the stuck test, however, indicates that the time delta is stuck, the received entropy value is not counted. This implies that for generating one block of random numbers, the Jitter RNG gathers 256 times the oversampling rate non-stuck time delta values.

3.2 Generation of Random Number Bit Stream

We now know how one 256 bit random number value is generated. The interface to the CPU Jitter random number generator allows the caller to provide a pointer to memory and a size variable of arbitrary length. The random number generator is herewith requested to generate a bit stream of random numbers of the requested size that is to be stored in the memory pointed to by the caller.

---

[4]If during compilation the macro `JENT_CONF_DISABLE_LOOP_SHUFFLE` is unset, that SHA-3 hash operation is implemented by a loop where the loop counter is not fixed. Effectively this extends the duration of the hash operation. The following discussion disregards this approach as it is not the default behavior.

The random number generator performs the following sequence of steps to fulfill the request:

1. Check whether the requested size is smaller than 256 bits. If yes, generate one 256 bit random number, copy the requested amount of bits to the target memory and stop processing the request. The unused bits of the random number are not used further. If a new request arrives, a fresh 256 bit random number is generated.

2. If the requested size is larger than 256 bits, generate one random number, copy it to the target. Reduce the requested size by 256 bits and decide now whether the remaining requested bits are larger or smaller than 256 bits and based on the determination, follow either step 1 or step 2.

Mathematically step 2 implements a concatenation of multiple random numbers generated by the random number generator.

3.3 Jitter RNG Data Processing

The processing of entropic data from the noise source up to the point data is made available to the caller is performed with the following mathematical operations. The operation $SHA()$ refers to the SHA-3 256 hash operation.

1. Hashing: A set of auxiliary data is obtained from the Jitter RNG state, including the RCT and APT state information such as the number of RCT counts, APT counts and APT base value. This data is not credited with any entropy:

$$AUX_n = SHA(aux\_data) \qquad (3.1)$$

2. Hashing: One 64-bit time delta data value t_n received from measuring the execution time of the SHA-3 operation of the auxiliary data and memory access operations is inserted into the SHA-3 state of the entropy pool using a SHA-3 update operation. Also, the auxiliary data is inserted into the entropy pool using a SHA-3 update operation. This implies that mathematically speaking all time stamps and auxiliary data are concatenated to form the entropy pool state EP_n. The following formula depicts EP_n after the receipt of 768 time delta values – which would be collected if the oversampling rate is 3 (the default) and there would be no stuck time delta. The implementation is provided with function `jent_hash_time`:

$$EP_n = SHA(t_n||AUX_n||t_{n-1}||AUX_{n-1}||...||t_{n-767}||AUX_{n-767}) \quad (3.2)$$

3. Entropy Collection Loop: 256 (multiplied by the oversampling rate – which is considered to be 1 in this example) time delta values t_{n-255} through t_n are obtained and processed with the hashing operation outlined in equation 3.2 as implemented in `jent_random_data`. After processing the last time delta, the entropy pool $EP_{k_{block}}$ is considered to be the output block of the CPU Jitter RNG.

$$EP_{k_{block}} = EP_n\{n = 0..(256 \cdot osr - 1)\} \qquad (3.3)$$

This formula implies that when 256 times the oversampling rate time deltas are to be mixed into the entropy pool.

This implies that for obtaining one output block of the Jitter RNG, the entropy pool is iteratively hashed with the time deltas up to a maximum of $256 \cdot osr$ times.

4. Concatenation with truncation: To generate the requested number of random bytes, the result of equation 3.3 is concatenated with the previous completion of equation 3.3 and so on. If the requested amount of random data is not a multiple of 256 bits, the required most significant bits from the current entropy pool state is used - the implementation is provided with function `jent_read_entropy`:

$$RAND = EP_{(k-sizeof(RAND)/256)_{block}} || \ldots || EP_{(k-1)_{block}} || MSB_{sizeof(RAND) \ mod \ 256}(EP_{k_{block}})$$
(3.4)

3.4 Runtime Health Tests

The Jitter RNG implements the following health tests:

- Stuck Test

- Repetition Count Test

- Adaptive Proportion Test

- Lag Predictor Test

Those tests are detailed in the following sections.

3.4.1 Stuck Test

The stuck test calculates the first, second and third discrete derivative of the time to be processed by the hash. Only if all three values are non-zero, the received time delta is considered to be non-stuck.

3.4.2 Repetition Count Test

The Jitter RNG uses an enhanced version of the Repetition Count Test (RCT) specified in SP800-90B [Turan et al., 2018] section 4.4.1. Instead of counting identical back-to-back values, the input to the RCT is the counting of the stuck values during the generation of one Jitter RNG output block. The data that is mixed into the entropy pool is the time delta, i.e. the first discrete derivative of the time stamp. As the stuck result includes the comparison of two back-to-back time deltas by computing the second discrete derivative of the time stamp, the RCT simply checks that the second discrete derivative of the time stamp is zero. If it is zero, the RCT counter is increased. Otherwise, the RCT counter is reset to zero.

The RCT is applied with $\alpha = 2^{-30}$ compliant to the recommendation of FIPS 140-2 IG 9.8.

During the counting operation, the Jitter RNG always calculates the RCT cut-off value of C. If that value exceeds the allowed cut-off value, the Jitter RNG output block will be calculated completely but discarded at the end. The caller of the Jitter RNG is informed with an error code.

3.4.3 Adaptive Proportion Test

Compliant to SP800-90B [Turan et al., 2018] section 4.4.2 the Jitter RNG implements the Adaptive Proportion Test (APT). Considering that the entropy is present in the least significant bits of the time delta, the APT is applied only to those least significant bits. The APT is applied to the four least significant bits.

The APT is calculated over a window size of 512 time deltas that are to be mixed into the entropy pool. By assuming that each time delta has (at least) one bit of entropy and the APT-input data is non-binary, the cut-off value $C = 325$ as defined in SP800-90B[Turan et al., 2018] section 4.4.2.

3.4.4 Lag Predictor Test

This test is a vendor-defined conditional test that is designed to detect a known failure mode where the result becomes mostly deterministic.

3.5 Initialization

The CPU Jitter random number generator is initialized in two main parts. At first, a consuming application must call the `jent_entropy_init(3)` or `jent_entropy_init_ex(3)` function which validates some basic properties of the time stamp. Only if this validation succeeds, the CPU Jitter random number generator can be used[5].

The second part can be invoked multiple times. Each invocation results in the instantiation of an independent copy of the CPU Jitter random number generator. This allows a consumer to maintain multiple instances for different purposes. That second part is triggered with the invocation of `jent_entropy_collector_alloc(3)` and implements the following steps:

1. Allocation and zeroization of memory used for the entropy pool and helper variables – **struct rand_data** defines the entropy collector which holds the entropy pool and its auxiliary values.

2. Invoking the entropy collection loop once – this fills the entropy pool with the first random value which is not returned to any caller. The idea is that the entropy pool is initialized with some values other than zero. In addition, this invocation of the entropy collection loop implies that the entropy collection loop counter value is set to a random value in the allowed range.

3. If FIPS 140-2 is enabled by the calling application, the SP800-90C compliant oversampling of the noise source for conditioning is applied by obtaining 64 more bits of entropy than extracted from the SHA3-256 operation.

3.6 Memory Protection

The CPU Jitter random number generator is intended for any consuming application without placing any requirements. As a standard behavior, after completing the caller's request for a random number, i.e. generating the bit stream

---

[5]The importance of this call is illustrated in appendix C.31 as well as other sections in appendix C where some CPUs are not usable as an entropy source.

of arbitrary length, another round of the entropy collection loop is triggered. That invocation shall ensure that the entropy pool is overwritten with a new random value. This prevents a random value returned to the caller and potentially used for sensitive purposes lingering in memory for long time. In case paging starts, the consuming application crashes and dumps core or simply a hacker cracks the application, no traces of even parts of a generated random number will be found in the memory the CPU Jitter random number generator is in charge of.

In case a consumer is deemed to implement a type of memory protection, the flag `CRYPTO_CPU_JITTERENTROPY_SECURE_MEMORY` can be set at compile time. This flag prevents the above mentioned functionality.

Example consumers with memory protection are the kernel, and libgcrypt with its secure memory.

3.7   Locking

The core of the CPU Jitter random number generator implementation does not use any locking. If a user intends to employ the random number generator in an environment with potentially concurrent accesses to the same instance, locking must be implemented. A lock should be taken before any request to the CPU Jitter random number generator is made via its API functions.

Examples for the use of the CPU Jitter random number generator with locks are given in the reference implementations outlined in the appendices.

3.8   Programming Dependencies on Operating System

The implementation of the CPU Jitter random number generator only uses the following interfaces from the underlying operating systems. All of them are implemented with wrappers in `jitterentropy-base-*.h`. When the used operating system offers these interfaces or a developer replaces them with accordingly, the CPU Jitter random number generator can be compiled on a different operating system or for user and kernel space:

- Time stamp gathering: `jent_get_nstime` should deliver the high resolution time stamp. This function is an architecture dependent function with the following implementations:

  - User space:

    * On Mach systems like MacOS, the function `mach_absolute_time` is used for a high-resolution timer.
    * On AIX, the function `read_real_time` is used for a righ resolution timer.
    * On POSIX systems, the `clock_gettime` function is available for this operation.

  - Linux kernel space: In the Linux kernel, the `random_get_entropy` function obtains this information. The directory `arch/` contains various assembler implementations for different CPUs to avoid using an operating system service. If `random_get_entropy` returns

0, which is the case on a large number of architectures the kernel-internal call `__getnstimeofday` is invoked which uses the best available clocksource implementation. The goal with the invocation of `__getnstimeofday` is to have a fallback for `random_get_entropy` returning zero. Note, if that clocksource clock also is a low resolution timer like the Jiffies timer, the initialization function of the CPU Jitter Random Number Generator is expected to catch this issue.

- `jent_zalloc` is a wrapper for the `malloc` function call to obtain memory.

- `jent_zfree` is a wrapper for calling the `free` function to release the memory.

- `__u64` must be a variable type of a 64 bit unsigned integer – either unsigned long on a 64 bit system or unsigned long long on a 32 bit system.

The following additional functions provided by an operating system are used without a wrapper as they are assumed to be present in every operating environment:

- `memcpy`

- `memset`

3.8.1 Environments Without High-Resolution Time Stamps

The CPU Jitter RNG depends on the presence of high-resolution time stamps. Yet, it is possible that the operating environment may not offer such time stamps. In this case, the CPU Jitter RNG implements a fallback which allows to still collect entropy but without the use of a high-resolution time stamp provided by the environment.

In this case, the CPU Jitter RNG implements its own high-resolution time stamp maintenance. This is achieved, by creating a thread which implements an counter that is incremented with an endless loop. That counter value acts as the high-resolution time stamp.

To support this fallback, the POSIX threading must be available in the environment. In addition, to ensure that the operating system does not affect the Jitter RNG significantly it is recommended that in this case at least 2 CPUs are at the disposal of the Jitter RNG.

4 Random Generator Statistical Assessment

After the discussion of the design of the entropy collection, we need to perform assessments of the quality of the random number generator. As indicated in chapter 1, the assessment is split into two parts.

This chapter contains the assessment of the statistical properties of the data in the entropy pool and the output data stream.

When compiling the code of the CPU Jitter random number generator with instrumentations added to the code, data can be obtained for the following graphs and distributions. The tests can be automatically re-performed by invoking the `tests_[userspace|kernel]/getstat.sh` shell script which also generates the graphs using the `R-Project` language toolkit.

4.1 Statistical Properties of Entropy Pool

During a testing phase that generated 1,000,000 random numbers, the entropy pool is observed. The observation generated statistical analyses for different aspects illustrated in table 1. Each line in the table is one observation of the entropy pool value of one round of the entropy collection loop. To read the table, assume that the entropy pool is only 10 bits in size. Further, assume that our entropy collection loop count is 3 to generate a random number.

The left column contains the entropy collection loop count and the indication for the result rows. The middle columns are the 10 bits of the entropy pool. The Bit sum column sums the set bits in the respective row. The Figure column references the figures that illustrate the obtained test data results.

Loop count	0	1	2	3	4	5	6	7	8	9	Bit sum	Figure
1	0	1	1	0	0	0	1	0	1	1	N/A	N/A
2	0	0	0	1	0	1	1	1	0	0	N/A	N/A
3	1	1	0	0	1	0	1	0	0	0	4	4.1
Result 1	1	2	1	1	1	1	3	1	1	1	13	4.3
Result 2	1	2	1	2	1	2	0	2	1	1	13	4.5

Table 1: Example description of tests

The "Result 1" row holds the number of bits set for each loop count per bit position. In the example above, bit 0 has a bit set only once in all three loops. Bit 1 is set twice. And so on.

The "Result 2" row holds the number of changes of the bits for each loop count compared to the previous loop count per bit position. For example, for bit 0, there is only one change from 0 to 1 between loop count 2 and 3. For bit 7, we have two changes: from 0 to 1 and from 1 to 0.

The graphs contains the same information as explained for figure 2.1.

The bit sum of loop count 3 is simply the sum of the set bits holds the number of set bits at the last iteration count to generate one random number. It is expected that this distribution follows a normal distribution closely, because only such a normal distribution is supports implies a rectangular distribution of the probability that each bit is equally likely to be picked when generating a random number output bit stream. Figure 4.1 contains the distribution of the bit sum for the generated random numbers in user space.

Distribution of number of set bits per observation in userspace

Min: 10 - 1st Qu: 30 - Median: 34 - Mean: 33.01
3rd Qu: 36 - Max: 60 - Sigma: 5.57 - Var Coeff: 0.168689
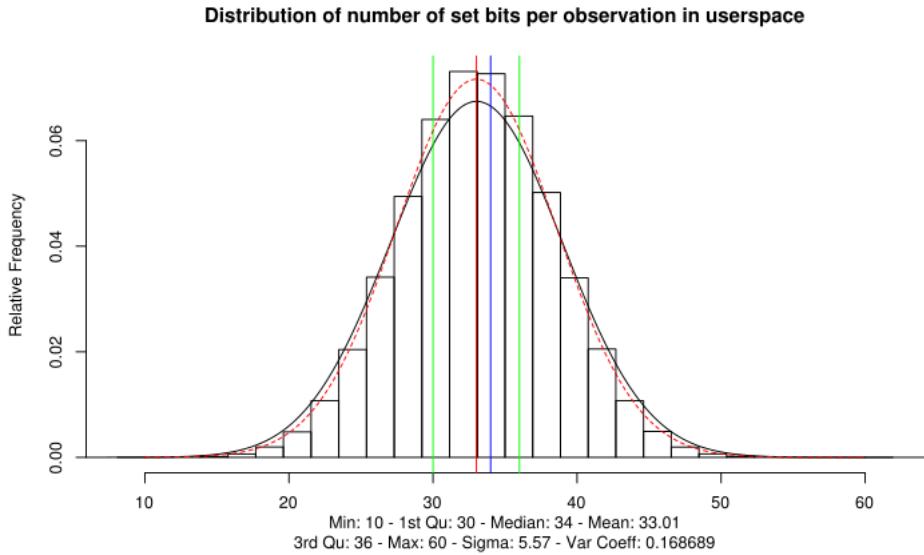
Figure 4.1: Bit sum of last round of entropy collection loop user space

In addition, the kernel space distribution is given in figure 4.2 – they are almost identical and thus show the same behavior of the CPU Jitter random number generator

Distribution of number of set bits per observation in kernel

Min: 6 - 1st Qu: 30 - Median: 34 - Mean: 33
3rd Qu: 36 - Max: 58 - Sigma: 5.56 - Var Coeff: 0.1683822
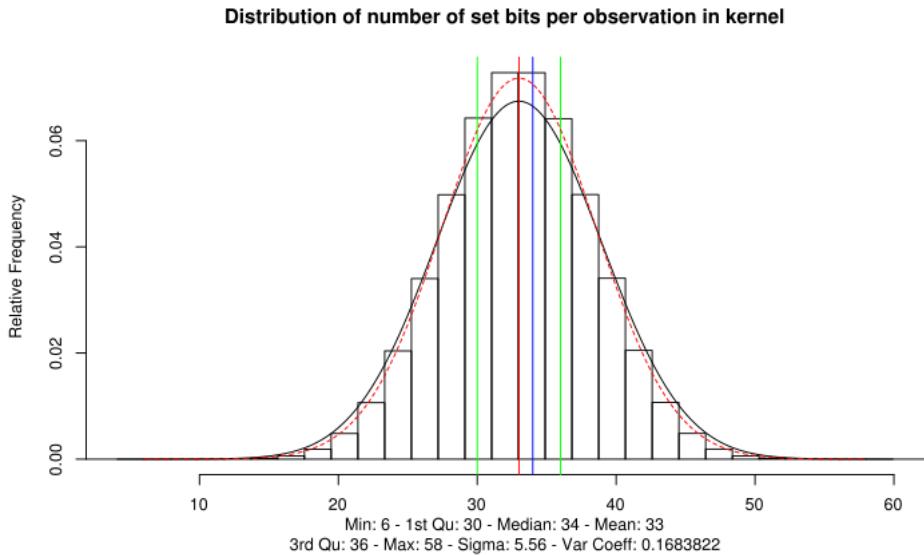
Figure 4.2: Bit sum of last round of entropy collection loop kernel space

Please note that the black line in the graphs above is an approximation of the density of the measurements using the histogram. When more histogram

bars would be used, the approximation would better fit the theoretical normal
distribution curve given with the red dotted line. Thus, the difference between
both lines is due to the way the graph is drawn and not seen in the actual
numbers. This applies also to the bars of the histogram since they are left-
aligned which means that on the left side of the diagram they overstep the
black line and on the right side they are within the black line.

The distribution for "Result 1" of the sum of of these set bits is given in
figure 4.3.
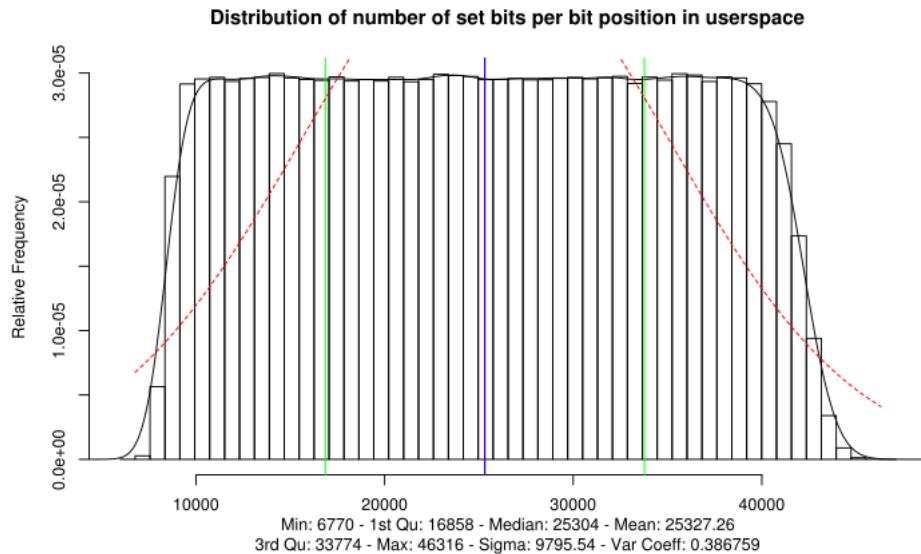
Figure 4.3: Bit sum of set bits per bit position in user space

Again, for the kernel we have an almost identical distribution shown in
figure 4.4. And again, we conclude that the behavior of the CPU Jitter random
number generator in both worlds is identical.

Distribution of number of set bits per bit position in kernel

Min: 6366 - 1st Qu: 16848 - Median: 25308 - Mean: 25327.7
3rd Qu: 33760 - Max: 48516 - Sigma: 9814.39 - Var Coeff: 0.3874965
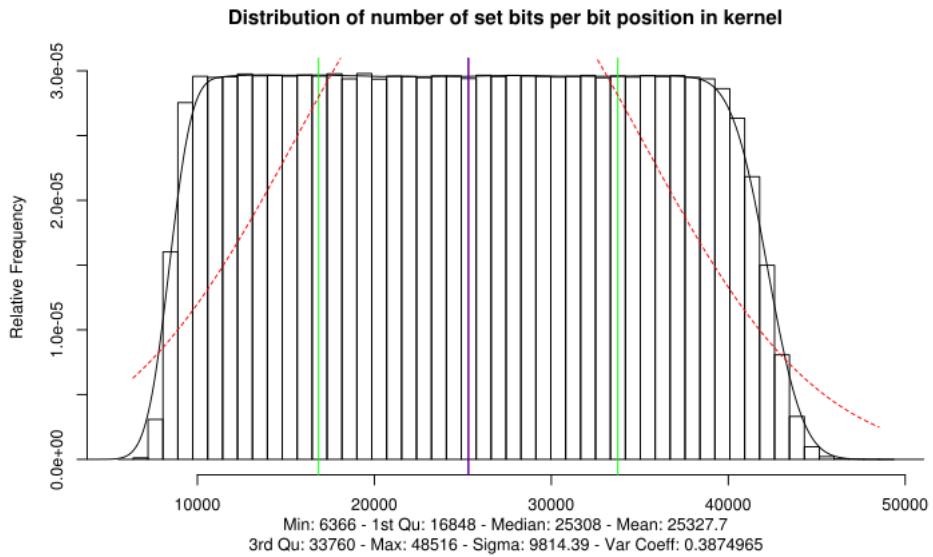
Figure 4.4: Bit sum of set bits per bit position in kernel space

A question about the shape of the distribution should be raised. One can have no clear expectations about the distribution other than it must show the following properties:

- It is a smooth distribution showing no breaks.

- It is a symmetrical distribution whose symmetry point is the mean.

The distribution for "Result 2" of the sum of of these bit variations in user space is given in figure 4.5.

25
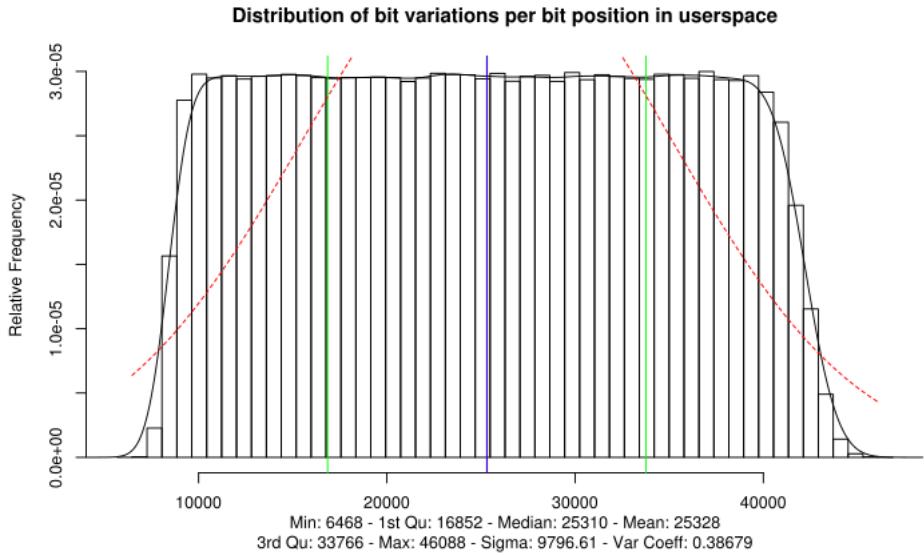
Figure 4.5: Bit sum of bit variations per bit position in user space

Just like above, the plot for the kernel space is given in figure 4.6.
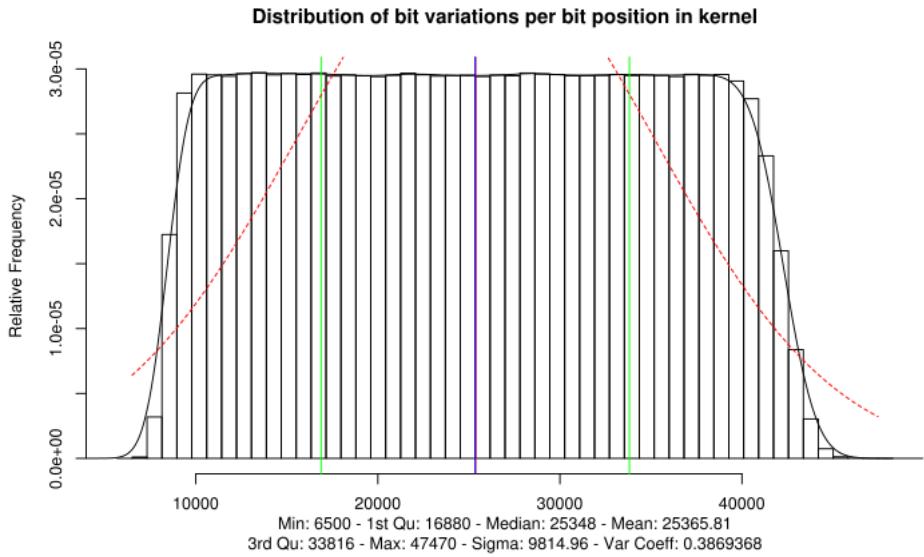
Figure 4.6: Bit sum of bit variations per bit position in kernel space

Just like for the preceding diagrams, no material difference is obvious between kernel and user space. The shape of the distributions is similar to the one for the distribution of set bits. An expected distribution can also not be given apart from the aforementioned properties.

4.2 Statistical Properties of Random Number Bit Stream

The discussion of the entropy in chapter 5 tries to show that one bit of random number contains one bit of entropy. That is only possible if we have a rectangular distribution of the bits per bit position, i.e. each bit in the output bit stream has an equal probability to be set. The CPU Jitter random number block size is 64 bit. Thus when generating a random number, each of the 64 bits must have an equal chance to be selected by the random number generator. Therefore, when generating large amounts of random numbers and sum the bits per bit position, the resulting distribution must be rectangular. Figure 4.7 shows the distribution of the bit sums per bit position for a bit stream of 10,000,000 random numbers, i.e 640,000,000 bits.
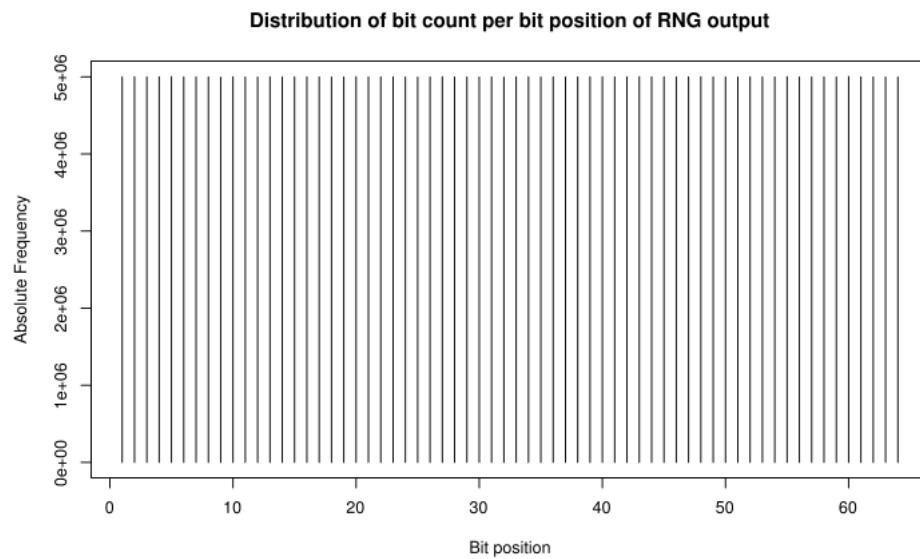
Figure 4.7: Distribution of bit count per bit position of RNG output

Figure 4.7 looks pretty rectangular. But can the picture be right with all its 64 vertical lines? We support the picture by printing the box plot in figure 4.8 that shows the variance when focusing on the upper end of the columns.

Distribution of bit count per bit position of RNG output
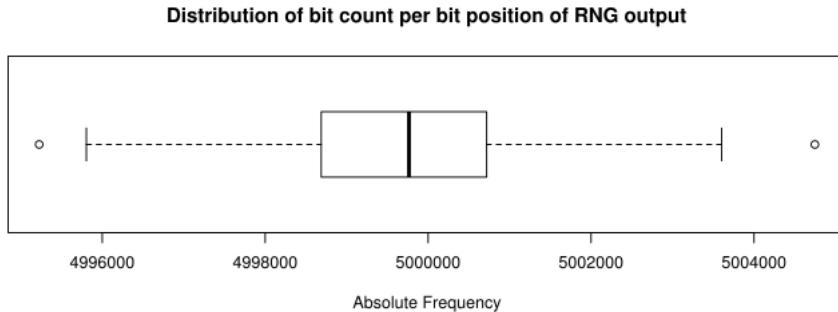
Figure 4.8: Box plot of variations in bit count per bit position of RNG output

The box plot shows the very narrow fluctuation around expected mean value of half of the count of random numbers produced, i.e. 5,000,000 in our case. Each bit of a random number has the 50% chance to be set in one random number. When looking at multiple random numbers, a bit still has the chance of being set in 50% of all random numbers. The fluctuation is very narrow considering the sample size visible on the scale of the ordinate of figure 4.7.

Thus, we conclude that the bit distribution of the random number generator allows the possibility to retain one bit of entropy per bit of random number.

This conclusion is supported by calculating more thorough statistical properties of the random number bit stream are assessed with the following tools:

- `ent`

- `dieharder`

- BSI Test Procedure A

The `ent` tool is given a bit stream consisting of 10,000,000 random numbers (i.e. 80,000,000 Bytes) with the following result where `ent` calculates the statistics when treating the random data as bit stream as well as byte stream:

Listing 3: `ent` statistical test

```
$ dd if=/sys/kernel/debug/jitterentropy/seed of=random.out bs=8 count=10000000

# Byte stream
$ ent random.out
Entropy = 7.999998 bits per byte.

Optimum compression would reduce the size
of this 80000000 byte file by 0 percent.

Chi square distribution for 80000000 samples is 272.04, and randomly
would exceed this value 25.00 percent of the times.

Arithmetic mean value of data bytes is 127.4907 (127.5 = random).
Monte Carlo value for Pi is 3.141600679 (error 0.00 percent).
Serial correlation coefficient is 0.000174 (totally uncorrelated = 0.0).

# Bit stream
$ ent -b random.out
Entropy = 1.000000 bits per bit.

Optimum compression would reduce the size
of this 640000000 bit file by 0 percent.
```

```
Chi square distribution for 640000000 samples is 1.48, and randomly
would exceed this value 25.00 percent of the times.

Arithmetic mean value of data bits is 0.5000 (0.5 = random).
Monte Carlo value for Pi is 3.141600679 (error 0.00 percent).
Serial correlation coefficient is -0.000010 (totally uncorrelated = 0.0).
```

During many re-runs of the `ent` test, most of the time, the Chi-Squared test showed the test result of 50%, i.e. a perfect result – but even the showed 25% is absolutely in line with random bit pattern. Very similar results were obtained when executing the same test on:

- an Intel Atom Z530 processor;

- a MIPS CPU for an embedded device;

- an Intel Pentium 4 Mobile CPU;

- an AMD Semperon processor;

- KVM guest where the host was based on an Linux 3.8 kernel and with QEMU version 1.4 without any special configuration of hardware access;

- OpenVZ guest on an AMD Opteron processor.

In addition, an unlimited bit stream is generated and fed into `dieharder`. The test results are given with the files `tests_userspace/dieharder-res.*`. The result files demonstrate that all statistical properties tested by `dieharder` are covered appropriately.

The BSI Test Suite A shows no statistical weaknesses.

The test tools indicate that the bit stream complies with the properties of random numbers.

4.3   Anti-Tests

The statistical analysis given above indicates a good quality of the random number generator. To support that argument, an "anti" test is pursued to show that the quality is *not* provided by the post-processing of the time stamp data, but solely by the randomness of the time deltas. The post-processing therefore is only intended to transform the time deltas into a bit string with a random pattern and magnifying the timer entropy.

The following subsections outline different "anti" tests.

4.3.1   Static Increment of Time Stamp

The test is implemented by changing the function `jent_get_nstime` to maintain a simple value that is incremented by 23 every time a time stamp is requested. The value 23 is chosen as it is a prime. Yet, the increment is fully predictable and does not add any entropy.

The stuck test ensures that the time stamp is rejected.

4.3.2 Pattern-based Increment of Time Stamp

Contrary to the static increment of the time stamp, this "anti" test describes a pattern-based increment of the time stamp. The time stamp is created by adding the sum of 23 and an additional increment using the following code:

Listing 4: Pattern-based Increment of Time Stamp

```
static unsigned int pad = 0;
static __u64 tmp = 0;
static inline void jent_get_nstime(__u64 *out)
{
        tmp += 23;
        pad++;
        *out = (tmp + pad);
}
```

The code adds 24 in the first loop, 25 in the second, 26 in the third, 27 in the fourth, and so forth.

Again, the stuck test ensures that the time stamp is rejected.

4.3.3 Disabling of System Features

The CPU jitter is based on properties of the system, such as caches. Some of these properties can be disabled in either user space or kernel space. The effect on such changes is measured in appendix 6.1.

5 Entropy Behavior

As the previous chapter covered the statistical properties of the CPU Jitter random number generator, this chapter provides the assessment of the entropy behavior. With this chapter, the second vital aspect of random number generators mentioned in chapter 1 is addressed.

The CPU Jitter random number generator does not maintain any entropy estimator. Nor does the random number generator tries to determine the entropy of the individual recorded time deltas that are fed into the entropy pool. There is only one basic rule that the CPU Jitter random number generator follows: upon completion of the entropy collection loop, the entropy pool contains 256 bit of entropy which are returned to the caller. That results in the basic conclusion of the random number bit stream returned from the CPU Jitter random number generator holding one bit of entropy per bit of random number.

Now you may say, that is a nice statement, but show me the numbers. The following sections will demonstrate the appropriateness of this statement.

Section 5.1 explains the base source of entropy for the CPU Jitter random number generator. This section explains how the root cause of entropy is visible in the CPU Jitter random number generator.

Before we start with the entropy discussion, please let us make one issue perfectly clear: the nature of entropy, which is an indication of the level of uncertainty present in a set of information, can per definition *not* be calculated. All what we can do is try to find arguments whether the entropy estimation the CPU Jitter random number generator applies is valid. Measurements are used to support that assessment. Moreover, the discussion must contain a worst case analysis which gives a lower boundary of the entropy assumed to be present in the random number bit stream extracted from the CPU Jitter random number

generator. The lower boundary, however, is considered for theoretical discussion only because it deactivates one important aspect of the Jitter RNG.

5.1 Base Entropy Source

As outlined in chapter 3, the variations of the time delta is the source of entropy.

The design specification already indicates that multiple noise sources support the operation of the RNG. The following subsections discuss the individual noise sources.

All diagrams include the value of the Shannon Entropy H which is calculated with the following formula:

$$H = -\sum_{i=1}^{N} p_i \cdot log_2(p_i)$$

where N is the number of samples, and p_i is the probability of sample i. As the Shannon Entropy formula uses the logarithm at base 2, that formula results in a number of bits of entropy present in an observed sample.

5.1.1 Noise Sources Depicted

NOTE: This section discusses an older CPU Jitter RNG implementation using an LFSR. The newer implementation using a SHA-3 hash, however, is considered to behave identically.

Unlike the graphs outlined in chapter 2 where two time stamps are invoked immediately after each other, the CPU Jitter random number generator places the hash loop between each time stamp gathering. That implies that the CPU jitter over the hash loop is measured and used as a basis for entropy.

Considering the fact that the CPU execution time jitter over the hash loop is the source of entropy, we can determine the following:

- The hash loop shall inject the time delta value into the entropy pool.

- The delta of two time stamps before and after the folding loop is given to the hash loop to be injected into the entropy pool.

The use cases of the Jitter RNG assume that the entropy of the time delta exceeds 1 bit of entropy – if it is less than one bit of entropy, the caller has to invoke the Jitter RNG more often.

Tests are implemented that measure the variations of the time delta over an invocation of the hash loop.

The design of the LFSR loop in section 3.1.3 explains that the number of LFSR loop iterations varies between 2^0 and 2^4 iterations. The testing of the entropy of the LFSR loop must identify the lower boundary and the upper boundary. The lower boundary is the minimum entropy the LFSR loop at least will have: this minimum entropy is the entropy observable over a fixed LFSR loop count. The test uses 2^0 as the fixed LFSR loop count. On the other hand, the upper boundary of the entropy is set by allowing the LFSR loop count to float freely within the above mentioned range.

It is expected that the time stamps used to calculate the LFSR loop count is independent from each other. Therefore, the entropy observable with the

testing of the upper boundary is expected to identify the entropy of the CPU execution time jitter. Nonetheless, if the reader questions the independence, the reader must conclude that the real entropy falls within the measured range between the lower and upper boundary.

Figure 5.1 presents the lower boundary of the LFSR loophash loop operation executing in user space of the test system. The graph shows two peaks whereas the higher peak is centered around the execution time when the code is in the CPU cache. For the time when the code is not in the CPU cache – such as during context switches or during the initial invocations – the average execution time is larger with the center at the second peak. In addition, figure 5.3 provides the upper boundary of the LFSR loop. With the graph of the upper boundary, we see 16 spikes which are the spikes of the lower boundary scattered by the LFSR loop counter. If the LFSR loop counter is 2^0, the variation of the time delta is centered around a lower value than the variations of a LFSR loop counter of 2^1 and so on. As the variations of the delta are smaller than the differences between the means of the different distributions, we observe the spikes.

The following graphs use the time deltas of 10,000,000 invocations of the LFSRhash loop. To eliminate outliers, time delta values above the number outlined in the graphs are simply cut off. That means, when using all values of the time delta variations, the calculated Shannon Entropy would be higher than listed in the legend of the graphs. This cutting off therefore is yet again driven by the consideration of determining the worst case.

The CPU Jitter RNG is based on two noise sourcesgenerating phenomena. The following graphs depict the LFSRhash noise source separately from the memory access noise source as the memory access noise source may be disabled during allocation. The following graphs shows the memory access noise source with the lowest set memory accesses to show the additional impact of just memory accesses – the graph is marked as "constant memory access´´. To show the additional noise picked up with the variations of the memory access loop, additional graphs are added marked as "variating memory access´´. Graphs for varying memory accesses are not shown any more as they just show a more or less perfect rectangular distribution (i.e. the varying memory accesses now make the noise sources even better).

Distribution of time delta in userspace-foldtime-i7.O0 up to 650

Min: 561 - 1st Qu: 582 - Median: 588 - Mean: 587.45
3rd Qu: 591 - Max: 648 - Sigma: 9.25 - Var Coeff: 0.0157501 - Shannon Entropy: 3.24
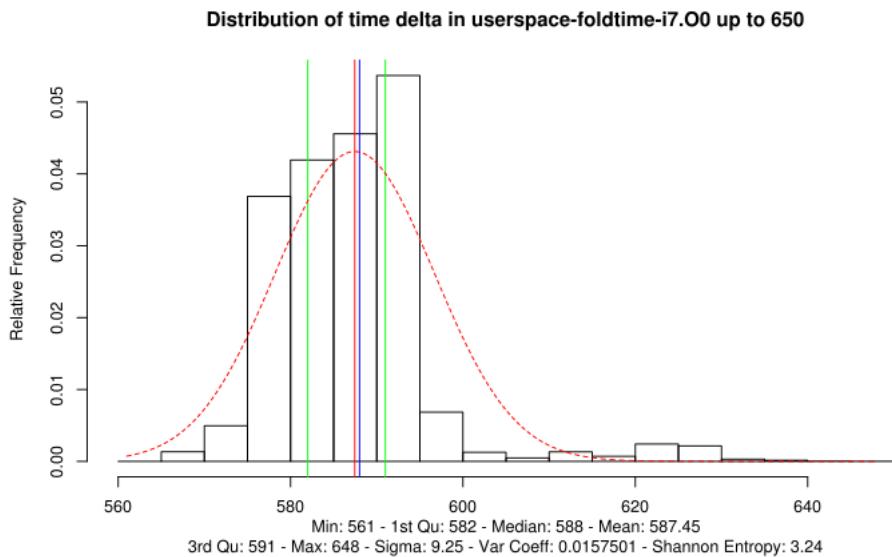
Figure 5.1: ~~Lower boundary of entropy~~Distribution of timing variations over hash~~LFSR~~ loop in user space

Distribution of time delta in userspace-foldtime-i7-memacc.O0 up to 1300

Min: 758 - 1st Qu: 800 - Median: 826 - Mean: 893.26
3rd Qu: 985 - Max: 1300 - Sigma: 133.81 - Var Coeff: 0.149798 - Shannon Entropy: 7.93
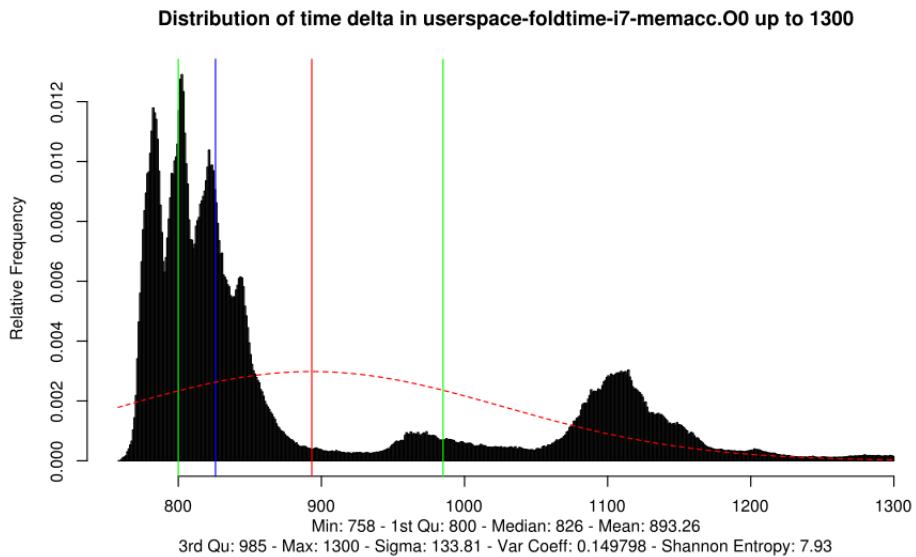
Figure 5.2: ~~Lower boundary of entropy over LFSR~~Distribution of timing variations over hash loop and constant memory access in user space

Distribution of time delta in userspace-foldtime-i7.O0 up to 8500

Min: 573 - 1st Qu: 2157 - Median: 4203 - Mean: 4398.83
3rd Qu: 6264 - Max: 8499 - Sigma: 2363.96 - Var Coeff: 0.5374078 - Shannon Entropy: 8.44
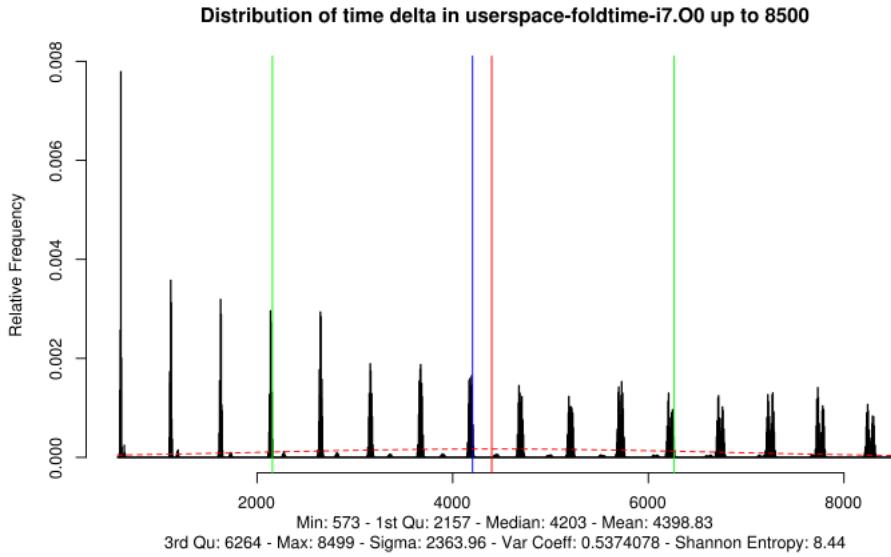
Figure 5.3: ~~Upper boundary of entropy over LFSR loop in user space~~

Distribution of time delta in userspace-foldtime-i7-memacc.O0 up to 4500

Min: 767 - 1st Qu: 1537 - Median: 2260 - Mean: 2266.26
3rd Qu: 2984 - Max: 4500 - Sigma: 848.46 - Var Coeff: 0.3743857 - Shannon Entropy: 11.32
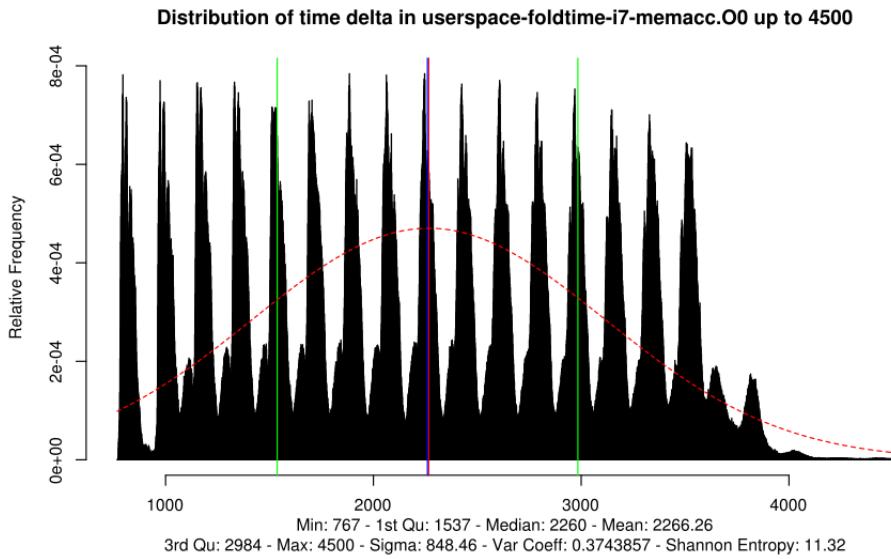
Figure 5.4: ~~Upper boundary of entropy over LFSR loop and constant memory access in user space~~

In addition to the user space measurements, figure~~s~~ 5.5 ~~and 5.6~~ present the ~~lower and upper boundary of the LFSR~~ hash loop execution time variations in kernel space on the same system. Again, the lower boundary is above 2 bits and the upper above 6 bits of Shannon Entropy.

34

Distribution of time delta in kernel-foldtime up to 330

Min: 288 - 1st Qu: 306 - Median: 309 - Mean: 308.46
3rd Qu: 312 - Max: 330 - Sigma: 5.69 - Var Coeff: 0.0184334 - Shannon Entropy: 2.67
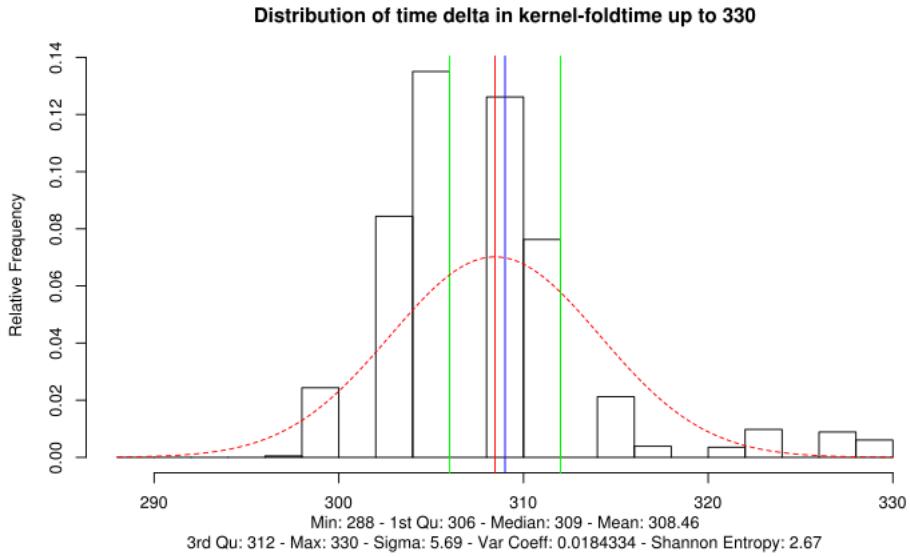
Figure 5.5: Entropy~~Lower boundary of entropy~~ over LFSR loop in kernel space

Distribution of time delta in kernel-foldtime up to 5000

Min: 297 - 1st Qu: 1167 - Median: 2244 - Mean: 2250.54
3rd Qu: 3324 - Max: 4998 - Sigma: 1177.1 - Var Coeff: 0.5230279 - Shannon Entropy: 7.24
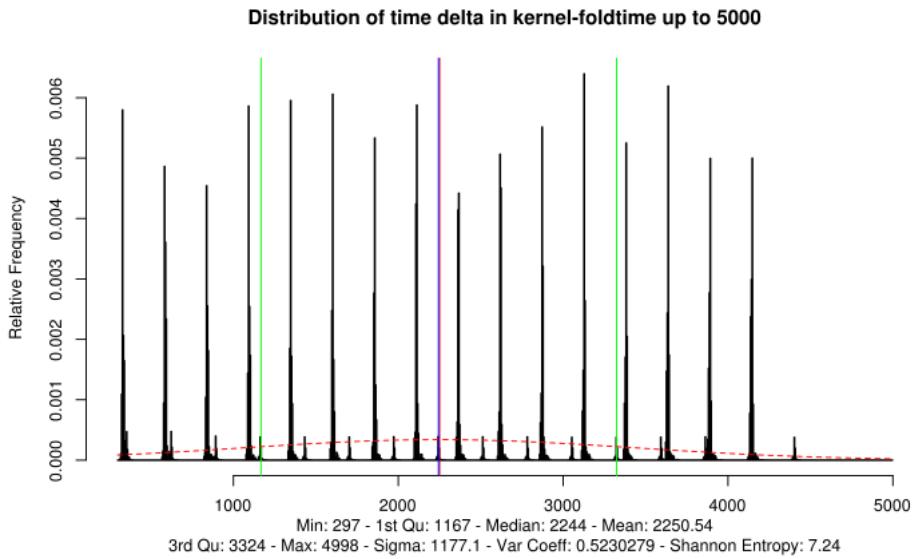
Figure 5.6: ~~Upper boundary of entropy over LFSR loop in kernel space~~

As this measurement is the basis of all entropy discussion, appendix C shows the measurements for many different CPUs. All of these measurements show that the lower and upper boundaries are always much higher than the required one bit of entropy with exceptions. All tests are executed with optimized code as even a worst case assessment and sometimes with the non-optimized compi-

lation to show the difference. For one CPU, section C.26 shows that the lower boundary is below 1 bit of Shannon Entropy. When re-performing the test with non-optimized code, ~~as required in section ??~~as applied for the production version of the Jitter RNG, the lower boundary is well above 2 bits of entropy and therefore sufficient for the entropy collection loop. This test shows that disabling optimizations is vital for the CPU Jitter random number generator. In addition, when enabling the memory access, the timing variations are even much greater, sufficient for the RNG operation.

~~For the other CPUs whose lower entropy is below 1 bit and the `jent_entropy_init` function allows this CPU, statistical tests are performed to verify that no cycles are present. This implies that the entropy is closer to the upper boundary and therefore well above 1 bit. But again, when enabling memory accesses entropy rises way above 1 bit.~~

The reader should also consider that the measured Shannon Entropy is a conservative measurement as the test invokes the ~~LFSR~~hash loop millions of times successively. This implies that for the entire duration of the test, caches, branch prediction units and similar are mostly filled with the test code and thus have hardly any impact on the variations of the time deltas. In addition, the test systems are kept idle as much as possible to limit the number of context switches which would have an impact on the cache hits. In real-life scenarios, the caches are typically filled with information that have an big impact on the jitter measurements and thus increase the entropy.

With these measurements, we can conclude that the CPU execution jitter over the ~~LFSR~~hash loop is always more than double the entropy in the worst case than required. Thus, the measured entropy of the CPU execution time jitter that is the basis of the CPU Jitter random number generator is much higher than required.

The reader may now object and say that the measured values for the Shannon Entropy are not appropriate for the real entropy of the execution time jitter, because the observed values may present some patterns. Such patterns would imply that the real entropy is significantly lower than the calculated Shannon Entropy. This argument can easily be refuted by the statistical tests performed in chapter 4. If patterns would occur, some of the statistical tests would indicate problems. Specifically the Chi-Squared test is very sensitive to any patterns. Moreover, the "anti" tests presented in section 4.3 explain that patterns are easily identifiable.

Fast Fourier Transformation When applying a Fast Fourier Transformation transformation to the raw time delta input data before ~~LFSR~~conditioning operation of the hash, an interesting observation can be made: there is no noticeable pattern in the raw time delta. Only one spike is visible: the expected spike at the zero point. Even when applying the FFT to the oldest or smallest CPUs, no big spikes are visible.

FFTs are calculated on the time deltas when setting the memory access loop numbers and the LFSR loop numbers to the minimum – i.e. without variations added by these different loop sizes. Regardless of the type of CPU the FFT is calculated for, either a perfect rectangular distribution with some bubbling is visible. The second FFT is applied to the time deltas of the normal operation. Again, an almost perfect rectangular distribution is seen.

In the following, example graphs for a small CPU of a MIPS 4Kec V6.8 is shown. Note, the bigger the CPUs the more perfect FFTs are seen. To make the graphs readable, the spike at the zero point is eliminated.
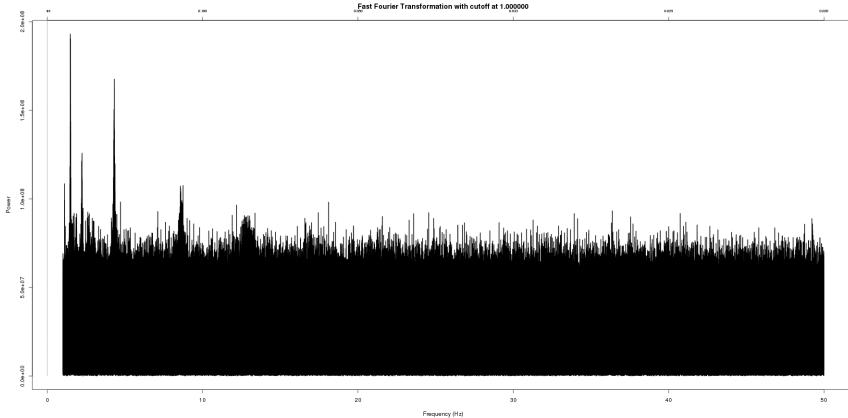
Figure 5.7: FFT of hash loop~~minimum LFSR~~ and memory access loop counts
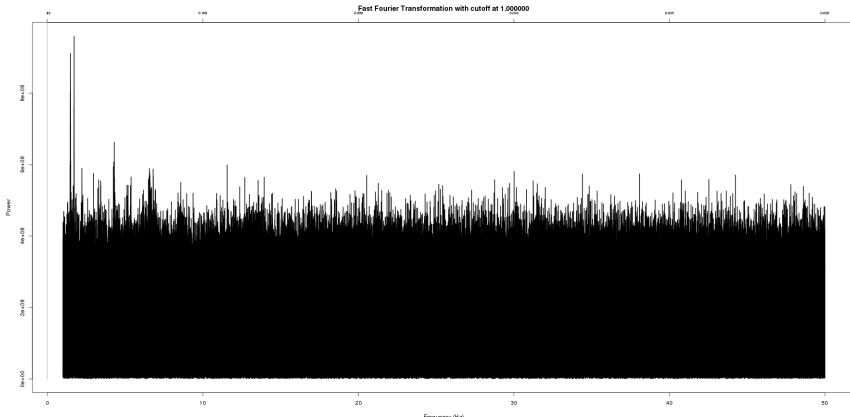
Figure 5.8: ~~FFT of normal operation of LFSR and memory access loop counts~~

Impact of Frequency Scaling and Power Management on Execution Jitter   When measuring the execution time jitter on a system with a number of processes active such as a system with the X11 environment and KDE active, one can identify that the absolute numbers of the execution time of a ~~LFSR~~hash loop is higher at the beginning than throughout the measurement. The behavior of the jitter over time is therefore an interesting topic. The following graph plots the first 100,000 measurements[6] where all measurements of time deltas above 600 were removed to make the graph more readable (i.e. the outliers

---

[6]The measurements of the ~~LFSR~~hash loop execution time were re-performed on the same system that is used for section 5.1. As the measurements were re-performed, the absolute numbers vary slightly to the ones in the previous section.

are removed). It is interesting to see that the execution time has a downward trend that stabilizes after some 60,000 ~~LFSR loops~~jitter RNG operations. The downward trend, however, is not continuously but occurs in steps. The cause for this behavior is the frequency scaling (Intel SpeedStep) and power management of the system. Over time, the CPU scales up to the maximum processing power. Regardless of the CPU processing power level, the most important aspect is that the oscillation within each step has an similar "width" of about 5 to 10 cycles. Therefore, regardless of the stepping of the execution time, the jitter is present with an equal amount! Thus, frequency scaling and power management does not alter the jitter.
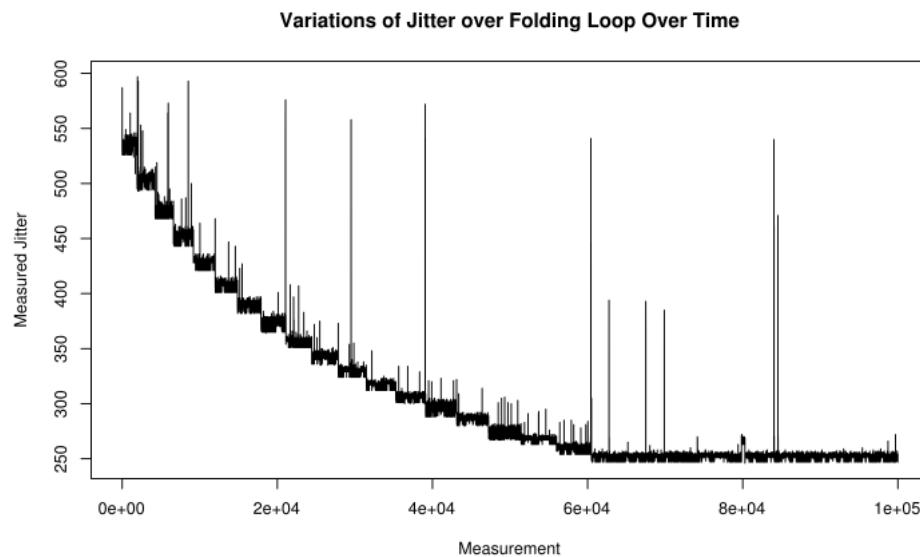
Figure 5.9: Variations of the execution time jitter over time when performing LFSR loop jitter measurements with Frequency Scaling / Power Management

When "zooming" in into the graph at different locations, as done below, the case is clear that the oscillation within each step remains at a similar level.

Variations of Jitter over Folding Loop Over Time
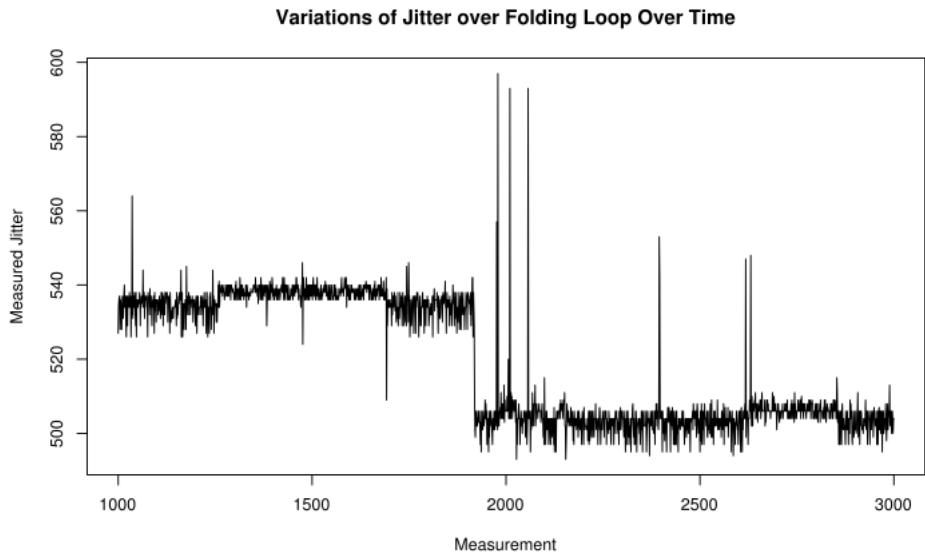
Figure 5.10: Variations of the execution time jitter over time when performing ~~LFSR~~hash loop jitter measurements with Frequency Scaling / Power Management – "zoomed in at measurements 1,000 - 3,000"

Variations of Jitter over Folding Loop Over Time
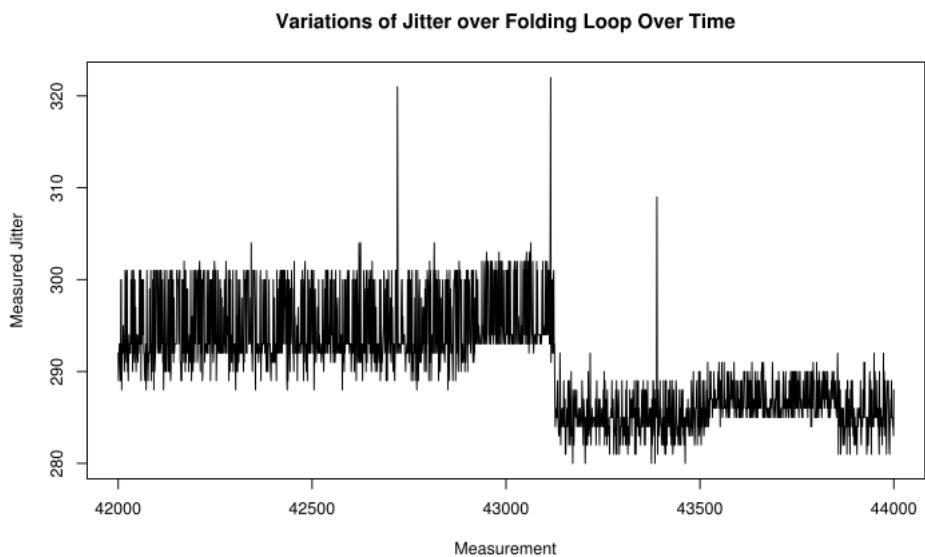
Figure 5.11: Variations of the execution time jitter over time when performing ~~LFSR~~hash loop jitter measurements with Frequency Scaling / Power Management – "zoomed in at measurements 42,000 - 44,000"

The constant variations support the case that the CPU execution time jitter

is agnostic of the with frequency scaling and power management levels.

To compare the measurements with disabled frequency scaling and power management on the same system, the following graphs are prepared. These graphs show the same testing performed.

Variations of Jitter over Folding Loop Over Time
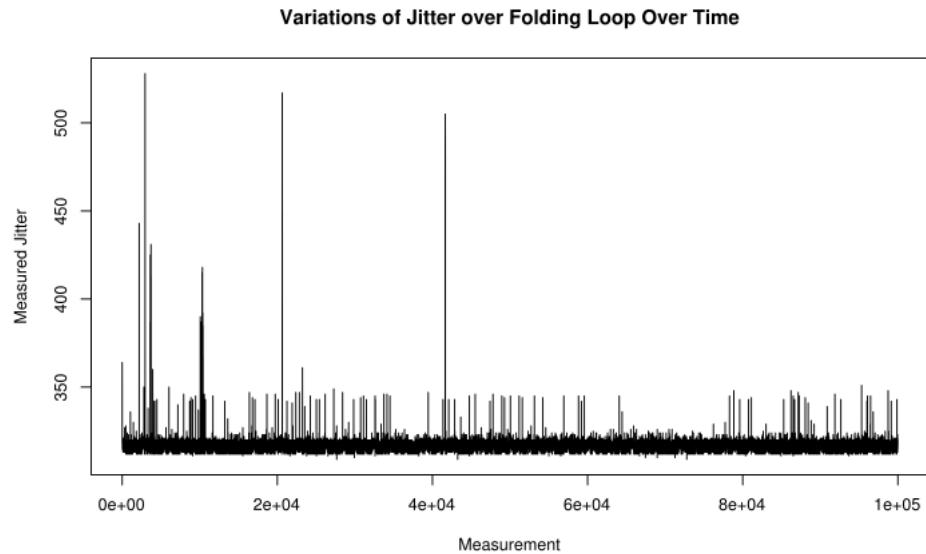
Figure 5.12: Variations of the execution time jitter over time when performing ~~LFSR~~hash loop jitter measurements with Frequency Scaling / Power Management disabled

40

Variations of Jitter over Folding Loop Over Time
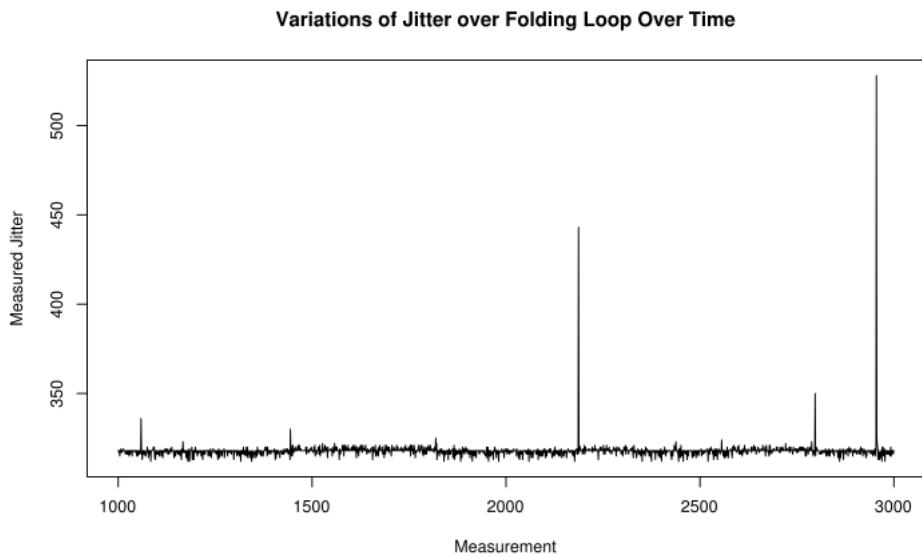
Figure 5.13: Variations of the execution time jitter over time when performing ~~LFSR~~hash loop jitter measurements with Frequency Scaling / Power Management disabled – "zoomed in at measurements 1,000 - 3,000"

Variations of Jitter over Folding Loop Over Time
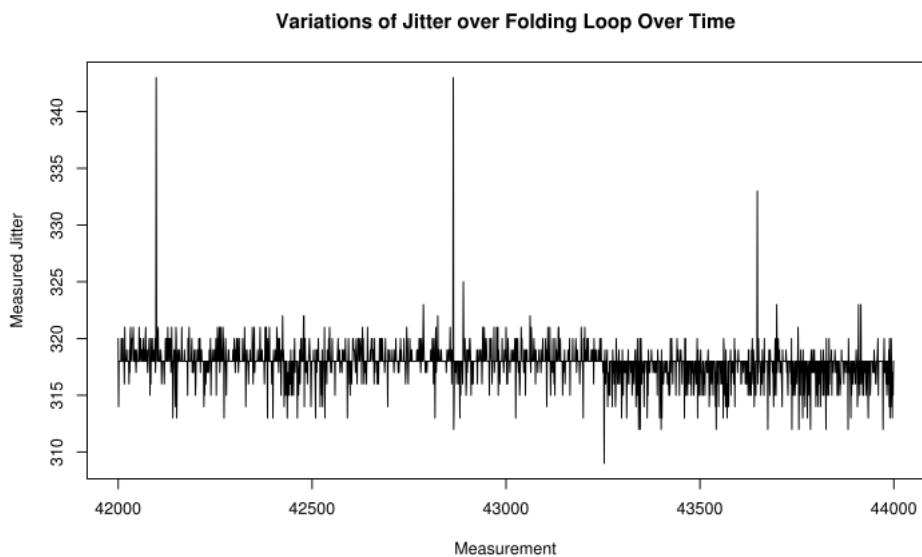
Figure 5.14: Variations of the execution time jitter over time when performing ~~LFSR~~hash loop jitter measurements with Frequency Scaling / Power Management disabled – "zoomed in at measurements 42,000 - 44,000"

6 Assessment of Noise Sources

NOTE: This section discusses an older CPU Jitter RNG implementation using an LFSR. The newer implementation using a SHA-3 hash, however, is considered to behave identically.

The quality of the CPU Jitter RNG rests on the assumption that the execution timing variations cannot be predicted. In addition, any influence that would diminish the entropy below one bit per timing measurement would also adversely affect the CPU Jitter RNG. Finally, any CPU manipulation that has the capability to introduce patterns also will affect the quality of the CPU Jitter RNG.

The following sections discuss the root cause of the noise sources of the CPU Jitter RNG. These tests shall help understand where the noise originates.

Before outlining the different causes of the variations in the Jitter noise source, it is important to understand that there are the two factors contributing to the uncertainty of the timing operation and thus to the gathering of entropy:

- The execution time of a fixed sequence of CPU instructions is not fully predictable. This is due to the complex nature of CPUs, including the branch prediction, and wait states the CPU must add to synchronize with the instruction fetching including the use of the L1 instruction cache (if present) among others.

- The access time to memory is also not fully predictable due to the state of the L1 through L3 caches, and the TLB cache.

Both contributing factors to the gathering of entropy in the Jitter RNG are invoked in a sequential order as outlined in section 3.1. Both factors provide its variations independent of each other. Thus the entropy each individual factor provides are additive to obtain the final entropy rate of the Jitter RNG. This finally implies that if one root cause for one contributing factor, for example the L1 through L3 caches, do provide a sufficient amount of entropy, the other contributing factor is yet unaffected and may still provide sufficient entropy. Therefore, both contributing factors equally contribute to the Jitter RNG entropy rate without dominating each other.

The test results presented in appendix C shows the test results with a Jitter RNG memory size of 2048 bytes. Almost all tested machines have an L1 cache that is larger which implies that the shown measurements show the Jitter RNG behavior when accessing L1 cache plus the CPU instruction timing variations. Almost all shown systems deliver an entropy rate of more than 1 bit of entropy per time delta. The device used for creating figure 6.1 shows still more than 0.4 bits of entropy per time delta even with 1024 bytes. All entropy values are above the heuristic entropy value applied by the Jitter RNG of 0.333 bits per time delta.

The test harness provides a tool to detect the amount of entropy depending on the selected memory size. This tool was used to prepare the figure 6.1. It can be used to examine the intended environment to obtain the most appropriate memory setting.

6.1 CPU Execution Timing Jitter

This analysis tries to answer the question:

Why is jitter visible in execution timing? How can it be quantifed?

The bare metal testing mechanism discussed in section 6.3 offers an analysis tool to gain more insights into the behavior of the CPU without interference by an operating system.

The following tests deviate slightly from all preceding tests by adding various cache flush strategies to observe any potential changes in the jitter measurements to gain an understanding of the CPU behavior. All tests are executed on the same hardware of an Intel Core i7 2nd generation system with the same operating system of Fedora 19 and the Linux kernel 3.11. The test system is configured to execute without power management and with Speedstep disabled to prevent additional interference. The baseline is the execution of the jitter measurement without any special operations. That baseline shows the following entropy numbers:

- Entropy~~Lower boundary of entropy~~: 3.06

- Upper boundary of entropy: 9.21

All of the following tests are performed independently of each other which means that one test does not contain the code changes of another test unless specifically noted. Therefore, the measurements can always be compared to the baseline measurements.

The testing enumerated below can all be re-performed on the bare metal tester outlined in section 6.3 by selecting test case 0 and enabling or disabling the discussed CPU mechanisms.

6.1.1 Serialization Instruction

Using serialization instructions, the execution jitter can be completely eliminated.

- Baseline test code which eliminates all jitter by using a serialization instruction:

```
asm volatile (
        "cpuid\n\t"
        "cpuid\n\t"
        "cpuid\n\t"
);
asm volatile("rdtsc" : "=A" (a));
asm volatile("rdtsc" : "=A" (b));
```

  The delta between the variables `a` and `b` does not vary. *Therefore, the CPU execution timing variations draw from on the internal state of the CPU which a serialization instruction can reset.*

- Now, considering that the serialization operation eliminates the variation as seen in the preceding, an attack against the RNG can be planned. When executing the following code on each CPU, the question now arises, whether the statistical properties of the RNG output would change:

```
void main(void)
{
        asm volatile("cpuid");
}
```

While that code is now executing on all CPUs to ensure that the entropy collection code of the RNG executes on a CPU where this serialization instruction has been executed before hand, the statistical properties of the RNG output does *not* show any weaknesses. The Chi-Squared test result of the binary output of the RNG marks it as white noise.

- Placing the invocation of the serialization instruction into a function:

```
void cpuid(void)
{
        asm volatile(
                "cpuid\n\t"
                "cpuid\n\t"
                "cpuid\n\t"
        );
}
void test(void)
{
        cpuid();
        asm volatile("rdtsc" : "=A" (a));
        asm volatile("rdtsc" : "=A" (b));
}
```

With this invocation of the serialization instruction, small variations start to appear.

- When re-implementing the function call with assembler code, however, still no variations are visible:

```
asm volatile(
        "jl 1f\n"
        "sub:\t"
        "cpuid\n\t"
        "ret\n\t"
        "1:"
        "call sub\n\t"
);
asm volatile("rdtsc" : "=A" (a));
asm volatile("rdtsc" : "=A" (b));
```

Thus, it is not clear what the difference between this code and the preceding code is.

- Even when enlarging the memory space between the CPUID instruction and the time stamp gathering, still no variations are visible:

```
asm volatile(
        "jl 1f\n"
        "sub:\t"
        "cpuid\n\t"
        "ret\n\t"
        "1:"
        "nop\n\t" //This instruction now 500 times
        "call sub\n\t"
);
asm volatile("rdtsc" : "=A" (a));
asm volatile("rdtsc" : "=A" (b));
```

Thus, no proximity of the serialization instruction to the timing has no impact to the absence of variations.

- Placing the serialization instruction inbetween the time reading instructions:

```
asm volatile(
        "cpuid\n\t"
        "cpuid\n\t"
        "cpuid\n\t"
);
asm volatile("rdtsc" : "=A" (a));
asm volatile("cpuid");
asm volatile("rdtsc" : "=A" (b));
```

  This code now shows significant variations of the execution time. This means that the flushing of the CPU state with the serialization instruction varies significantly with no apparent reason.

- When using other serialization instructions, like `WBINVD`, or `INVD`, the same results as with `CPUID` are visible. That means that all serialization instructions have an equal effect on the execution timing jitter. *Therefore, invoking a serialization instruction causes the CPU state that is the basis of the jitter to reset.*

- When replacing the serialization function with a memory barrier using the `MFENCE` instruction, the execution variations did not decrease compared to simply reading the timing values without any special treatment. *Therefore, the memory barrier has no effect on the CPU execution timing variations.*

Additional similar tests are performed. The interpretation of the results, however, is not possible at this point. This means that a theory of the noise source cannot be formulated for the CPU execution timing jitter. Thus, it can only be concluded that noise is visible in normal operation, and even when attacking the RNG with the method causing the variations to vanish by using serialization instructions, the noise source remains operational.

The next sections show other attempts to eliminate the CPU execution timing variations which, however, are not successful.

6.1.2 Prevention of System Call And Branch Prediction Interference

The measurements generated in the following are performed by measuring the time duration of one ~~LFSR~~hash loop and immediately using `printf` to write it to standard out. This write-out involves system calls and thus a modification of the caches, branch prediction, pipelines and TLB beyond the ~~LFSR~~hash operation. The difference now is that instead of simply taking the measurement and writing it out, the test takes the measurements 1.000 times in one row and prints out the last value. That way, the first 999 loop iterations shall cancel out the impact of the preceding `printf` to the current measurement:

```
int i = 0;
for(i=0; i<1000; i++)
        duration = jent_fold_var_stat(NULL, 0);
for(i=0; i<1000; i++)
        duration_min = jent_fold_var_stat(NULL, 1);
printf("%llu %llu\n", duration, duration_min);
```

- Entropy~~Lower boundary of entropy~~: 3.86

- Upper boundary of entropy: 9.48

45

6.1.3 Flush of CPU Instruction Pipeline

The CPU instruction pipeline can be flushed with the MFENCE CPU instruction. The flush of the pipeline is performed right before the invocation of one measurement. The following code illustrates that:

```
#define mb() asm volatile("mfence":::"memory")
mb();
duration = jent_fold_var_stat(NULL, 0);
mb();
duration_min = jent_fold_var_stat(NULL, 1);
mb();
```

- Entropy~~Lower boundary of entropy~~: 3.66
- Upper boundary of entropy: 9.33

6.1.4 Flush of CPU Caches

The different CPU caches can be flushed with the WBINVD CPU instruction. The flush of the caches is performed right before the invocation of one measurement. The following code illustrates that:

```
#define wbinvd() asm volatile("wbinvd": : :"memory");
wbinvd();
duration = jent_fold_var_stat(NULL, 0);
wbinvd();
duration_min = jent_fold_var_stat(NULL, 1);
wbinvd();
```

- Entropy~~Lower boundary of entropy~~: 6.43
- Upper boundary of entropy: 10.58

6.1.5 Disabling of Preemption

The preemption of the execution of the ~~LFSR~~hash loop may imply that scheduling happens while the loop is executing. Inside the kernel, preemption can be disabled as follows:

```
preempt_disable();
duration = jent_fold_var_stat(NULL, 0);
duration_min = jent_fold_var_stat(NULL, 1);
preempt_enable();
```

- Entropy~~Lower boundary of entropy~~: 3.46
- Upper boundary of entropy: 8.68

6.1.6 TLB Flush

The flush of all (non-global) TLB entries is achieved by modifying the CR3 register. Inside the kernel, modification of the CR3 register is performed by reading and writing the register as follows:

```
native_write_cr3(native_read_cr3());
duration = jent_fold_var_stat(NULL, 0);
duration_min = jent_fold_var_stat(1);
```

- Entropy~~Lower boundary of entropy~~: 3.22
- Upper boundary of entropy: 8.52

6.1.7 Pinning of Entropy Collection to one CPU

The pinning of the process that performs the measurements to one CPU can be performed by creating a CPUSET:

```
mkdir /sys/fs/cgroup/cpuset/foldtime
/bin/echo 2 > cpuset.cpus
/bin/echo 0 > cpuset.mems
/bin/echo <PID_of_proc> > tasks
/bin/echo 1 > cpuset.mem_hardwall
```

- Entropy~~Lower boundary of entropy~~: 3.32

- Upper boundary of entropy: 9.12

6.1.8 Disabling of Frequency Scaling and Power Management

Modern CPUs allow frequency scaling, including the Intel SpeedStep technology, the Intel TurboBoost, the power management of the CPU and peripherals. These techniques are used to conserve power. As these mechanisms may add variations, all these mechanisms are deactivated using the BIOS on the test machine.

- Entropy~~Lower boundary of entropy~~: 2.59

- Upper boundary of entropy: 9.25

The lower boundary shows a significant drop in variations by around 0.5 bits of entropy. Yet, the drop does not affect the quality of the RNG. The cause for the drop in variations is the different patterns of variations as outlined in section 5.1.1.

6.1.9 Disabling of L1 and L2 Caches

The next test disables the L1 and L2 caches of the CPU and reperforms the measurement of the jitter again. As the disabling of the caches can only be completed in kernel space, the test was executed using the kernel module and reading the exported interface `/sys/kernel/debug/jitterentropy/stat-fold`.

To disable the caches, the following code was added to the initialization function of the kernel module:

```
__asm__("push    %rax\n\t"
        "mov     %cr0,%rax;\n\t"
        "or      $(1 << 30),%rax;\n\t"
        "mov     %rax,%cr0;\n\t"
        "wbinvd\n\t"
        "pop     %rax"
       );
```

In addition, the MTRR was disabled with the following command before the mentioned file was read:

```
echo "disable=00" >| /proc/mtrr
```

The disabling of the caches is really noticeable as the system gets really slow by orders of magnitudes! So, if you redo the testing, ensure that nothing executes, perform the tests on a console (not within X11) to get a system that is somewhat responsive to your commands.

The measurements of the variation contain a large number of outliers which are removed to calculate the entropy.

- Entropy~~Lower boundary of entropy~~: 8.48

- Upper boundary of entropy: 11.53

As the lower boundary is already that high and due to the problem of removing the outliers from the measurements of the upper boundary, it is questionable whether the value for the upper boundary is helpful as it surely overstates the worst case by a large degree.

6.1.10 Disabling of L1 and L2 Caches And Interrupts

The test documented in appendix 6.1.9 is re-performed with the following code modification in the kernel module function `jent_debugfs_statfold_read`:

```
local_irq_save(flags);
local_irq_disable();
duration = jent_fold_var_stat(NULL, 0);
duration_min = jent_fold_var_stat(NULL, 1);
local_irq_restore(flags);
local_irq_enable();
```

The code change disables all interrupts on the current CPU while executing the ~~LFSR~~hash loop and the time measurement. After the measurement is completed for one round, it is re-enabled again.

Similarly to appendix 6.1.9, the variation measurement contains a large number of outliers. When removing them and limiting to the set of values to consider the worst case, the following lower entropy value is calculated. For the upper boundary value, the removal of the outliers is not really possible. Therefore, the given value may overstate the worst case significantly.

- Entropy~~Lower boundary of entropy~~: 8.37

- Upper boundary of entropy: 12.11

6.1.11 Disabling of All CPU Mechanisms

In the preceding subsections various CPU mechanisms were selectively disabled. This section now combines the disabling of all these mechanisms to analyze whether any combination of disabling CPU mechanisms changes the entropy statement.

The following table contains the test results. It starts with the test that combines the disabling and changing of all the CPU mechanisms outlined in the preceding sections[7]. Each following row allows some more CPU mechanisms as indicated. For each test, the upper and lower boundary of the Shannon entropy is calculated and listed. The tests were executed on an absolute quiet system that excludes X11 and any graphical user interface.

Disabled / Altered CPU mechanisms	Upper	Lower

---

[7]Note, the tests must be executed in kernel space where the CPU pinning capability using cgroups is not available.

Disabled / Altered CPU mechanisms	Upper	Lower
Prevention of System Call interference Flush of CPU instruction pipeline Flush of CPU caches Disabling of Preemption TLB Flush Disabling of Frequency Scaling / Power Mgt Disabling of MTRR Disabling of L1 and L2 caches Disabling of Interrupts	8.36	6.39
Prevention of System Call interference Flush of CPU instruction pipeline Flush of CPU caches Disabling of Preemption TLB Flush Disabling of Frequency Scaling / Power Mgt Disabling of MTRR Disabling of L1 and L2 caches Disabling of Interrupts	N/A	4.28[8]
Prevention of System Call interference Flush of CPU instruction pipeline Flush of CPU caches Disabling of Preemption TLB Flush Disabling of Frequency Scaling / Power Mgt Disabling of L1 and L2 caches Disabling of Interrupts	5.30	1.61
Prevention of System Call interference Flush of CPU instruction pipeline Flush of CPU caches Disabling of Preemption TLB Flush Disabling of Frequency Scaling / Power Mgt Disabling of Interrupts	5.51	1.59
Prevention of System Call interference Flush of CPU instruction pipeline Flush of CPU caches Disabling of Preemption TLB Flush Disabling of Frequency Scaling / Power Mgt	6.88	3.10
Prevention of System Call interference Flush of CPU instruction pipeline Flush of CPU caches TLB Flush Disabling of Frequency Scaling / Power Mgt	6.91	3.06

---

[8] removal of outliers

Disabled / Altered CPU mechanisms	Upper	Lower
Prevention of System Call interference Flush of CPU instruction pipeline Flush of CPU caches Disabling of Frequency Scaling / Power Mgt	6.90	2.65
Prevention of System Call interference Flush of CPU instruction pipeline Flush of CPU caches Disabling of Frequency Scaling / Power Mgt Disabling of Interrupts	5.19	1.46
Prevention of System Call interference Flush of CPU instruction pipeline Disabling of Frequency Scaling / Power Mgt Disabling of Interrupts	5.94	2.28
Prevention of System Call interference Disabling of Frequency Scaling / Power Mgt Disabling of Interrupts	5.94	1.69
Prevention of System Call interference Flush of CPU caches Disabling of Frequency Scaling / Power Mgt Disabling of Interrupts	5.87	1.89

Any CPU mechanism that has not been enabled as per table above will always enlarge the CPU execution jitter based on the analyses on the jitter measurements outlined in the previous sections. Therefore, these tests are disregarded.

The combination of disabled CPU mechanisms which diminishes the CPU execution jitter the most can be illustrated with the following code supplemented by disabling the power management and frequency scaling in the system BIOS:

```
local_irq_save(flags);
local_irq_disable();
wbinvd();
mb();
for(i=0; i<1000; i++)
        duration = jent_fold_var_stat(0);
wbinvd();
mb();
for(i=0; i<1000; i++)
        duration_min = jent_fold_var_stat(1);
wbinvd();
mb();
local_irq_restore(flags);
local_irq_enable();
```

It is interesting that a particular combination of disabling CPU mechanisms causes the jitter to drop more than to disable all CPU mechanisms. Moreover, measuring the effect of disabling each CPU mechanism in isolation – as done in the preceding subsections – shows no significant drop in jitter. A rationale for this behavior cannot be given at this point.

Nonetheless, the measurements of the lower boundary would still much more entropy than needed for the operation of the CPU Jitter RNG, let alone considering the upper boundary.

To support this conclusion, the above listed code was added to the function **jent_debugfs_read_func** to apply the modifications to the regular random

number generation. In addition, the locking found in `jent_drng_get_bytes_raw` must be removed to prevent complaints by the kernel for this test. Also, the Von-Neumann unbiaser must be disabled. After compilation and insertion of the kernel module, the file `/sys/kernel/debug/jitterentropy/seed` is to be read. After the generation of 3MB of data, the smoke test using the `ent` tool is performed to check the statistical behavior. The result is, as expected, appropriate:

```
$ ent /tmp/out && ent -b /tmp/out
Entropy = 7.999936 bits per byte.

Optimum compression would reduce the size
of this 2911816 byte file by 0 percent.

Chi square distribution for 2911816 samples is 257.43, and randomly
would exceed this value 44.55 percent of the times.

Arithmetic mean value of data bytes is 127.5091 (127.5 = random).
Monte Carlo value for Pi is 3.146313017 (error 0.15 percent).
Serial correlation coefficient is -0.000537 (totally uncorrelated = 0.0).
Entropy = 1.000000 bits per bit.

Optimum compression would reduce the size
of this 23314880 bit file by 0 percent.

Chi square distribution for 23314880 samples is 0.06, and randomly
would exceed this value 81.14 percent of the times.

Arithmetic mean value of data bits is 0.5000 (0.5 = random).
Monte Carlo value for Pi is 3.146391175 (error 0.15 percent).
Serial correlation coefficient is -0.000004 (totally uncorrelated = 0.0).
```

6.2   Memory Access Testing

The previous section covered the exclusive analysis of the noise source of the ~~LFSR~~hash operation. This section in addition covers the exclusive assessment of the memory accesses and their impact on the timing variations.

The tests are all conducted with the test tool discussed in section 6.3.

For conducting the memory access testing, the test tool is used to review the impact of the following settings – all other settings are left unchanged at their default:

- Size of memory blocks: The description of the memory access noise source explains that the memory used for measuring access times is segmented into memory blocks. With the configuration of the size of memory blocks the access pattern to the memory is altered.

- Number of memory blocks: In addition to the size of one memory block, the number of used memory blocks defines the size of the entire memory used for access. The modification of the size of the entire memory influences the number of memory addresses the CPU sees.

- Number of loop iterations to access memory: The design of the memory access noise source implies that various bytes out of the allotted memory are accessed. Each loop iteration reads and writes one byte.

6.2.1   Noise Source Discussion

Before measurements are presented, a discussion of the noise source needs to be conducted. The following question must be answered:

51

Where does the noise come from?

In more technical terms, this question can be converted to: Why do memory accesses exhibit variations when measuring the execution time of those memory accesses?

The CPU of a system executes with the speed of the processor clock. That means, the processing of one CPU instruction directly depends on the processor clock when disregarding auxiliary processing, such as fetching the instruction from memory. Now, if the CPU instruction happens to require additional data, memory move instruction(s) must be used to move the data into CPU registers in order to operate on the data. However, when fetching data from memory, the CPU must synchronize itself with the access speed of the memory in order for the memory fetch/store to succeed. The CPU must introduce wait states, because CPU instruction for a memory fetch or store can only be performed if the CPU clock to execute the memory access instruction must be aligned with the clock the memory bus executes with.

Real life, however, is a bit more complicated with the addition of caches. The caches execute at much higher speed as the real memory. The following rule applies: L1 cache is the fastest, L2 is slower than L1, and L3 is again slower than L2. That means, the number of wait states to synchronize the CPU with L1 access windows is less compared to the number of wait states needed to synchronize the CPU with L2. And similarly, the number of wait states for L3 will be higher than the one for L2. Finally, the number of wait states for memory will be higher than the ones for L3.

Now, the noise source rests on the basis that the time duration of wait states is not predictable and observable.

To ensure that sufficient uncertainty is delivered to the random number generator sufficient wait states shall be covered. Memory accesses are cached with the typical caching strategy by the CPU to fill L1 first and try to obtain predictions from it, followed by L2, followed by L3 and finally followed by real memory accesses. As established, accesses to L1 will exhibit the smallest number of wait states.

After performing some initial analyses, it was concluded that the access attempts must be as large to overflow the L1 cache and "spill" over to L2 accesses. This is the reason for setting the size of the memory blocks, the number of memory blocks as well as the number of access loops for the testing. When considering all three variables, the total number of memory accesses must definitely fill L1 and use at least parts of L2. This ensures that the wait states the CPU incurs for accessing L2 deliver the main noise.

6.2.2   Noise Source Measurements

After understanding the root of the noise source, this subsections shows measurements of the behavior of the noise source to answer the question:

Can the memory access timing variations be quantified?

The first test set analyzes the memory access variations in relationship to the memory block size. The second test set analyzes the memory access variations in relationship to number of memory accesses loop rounds.

Analysis of Memory Size   When considering the discussion above around the L1 through L3 caches, the following behavior of the Jitter RNG is expected::

- Having a memory block that is small enough that all memory accesses can fit into an L1 cache exhibit the smallest amount of timing variations. Yet, variations are present considering that even the L1 cache accesses are not as fast as the CPU.

- If the memory block is large enough that its accesses will not fit into the L1 memory but collectively into L1 + L2 cache, the memory access timing variations are significantly larger compared to the sole L1 cache accesses.

- If the memory block is large enough to not fit into the L1 and L2 caches together which implies that the L3 cache is utilized, the memory access timing variations are again significantly larger compared to the L1 + L2 cache accesses.

- Finally, if the memory block is large enough to not fit into the collective L1 through L3 caches together, the CPU is forced to also access real memory. Again, the memory access variations are significantly larger compared to the L1 + L2 + L3 cache accesses.

Figure 6.1 shows the test results where the Jitter RNG is configured with different numbers of memory block sizes. This figure gives an impression of these memory access timing variations in terms of an SP800-90B entropy rate depending on the memory size used by the Jitter RNG. The test system of a RISC-V 64 bit SiFive Freedom U740 SoC contains the following cache setup[9]:

- L1 data cache: 32 kBytes (note, the instruction cache is of no concern here as only memory accesses are analyzed)

- L2 cache: 2 MBytes (note, the L1 cache is integrated into L2 which implies that a memory block of larger than 2 MBytes is already spilling over)

- L3 cache N/A

---

[9]This system was chosen, because it exhibits the smallest amount of execution timing variations discussed in section 6.1 which should be disregarded here.

Memory Access Time Variations
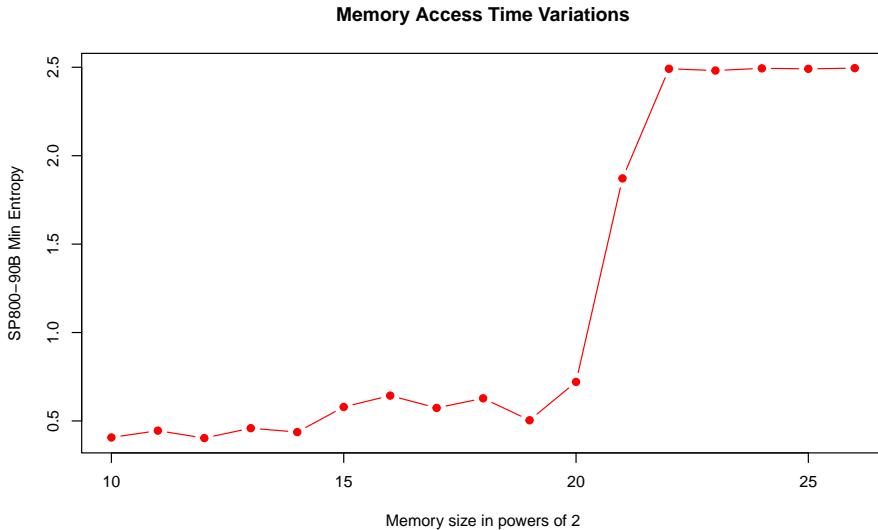
Figure 6.1: Memory Access Timing Variations in Relation to Memory Size

Figure 6.1 shows the increasing of the memory access timing variations by calculating the SP800-90B min entropy rate in relationship to the size of the Jitter RNG memory block. It depicts clearly the stairs of a leap in entropy rate when starting spill over into the next larger, but slower memory type:

- If the memory block size is less than 2^{15} bytes, the L1 cache fully satisfies the access requests. The min entropy value is between 0.4 and 0.5 bits.

- If the memory block size is less than or equal to 2^{20} bytes, the L1/L2 caches fully satisfy the access requests. Yet, the min entropy value is between 0.6 and 0.7 bits.

- If the memory block size is equal or larger than 2^{21} bytes, L1, L2 and the main memory is accessed. The min entropy value is now 2 bits or larger but remains at this level.

In addition, it shows that with a small memory access that remains fully within L1 cache, memory access timing variations exist. For this test, the absolute value of the SP800-90B entropy rate is irrelevant, only the changes depending on the memory size is to be considered.

The test is available with the tests/raw-entropy/recording_userspace/analyze_options.sh tool set.

Analysis of Number of Memory Accesses   The testing sets the number of memory blocks to 64 and the size of one memory block to 32. The measurements are taken by varying the number of memory access loops between 1 loop iteration up to 256 iterations. The following graphs list the number of memory access loops on the abscesses. The random number generator hard codes the number of memory access loops to 128, which is marked with a green line in all the graphs.

Each of the graphs contain three lines:

- The maximum and minimum observed values depicted with the blue lines.

- The mean of the observed values depicted with the red line. As expected, the red line will always be within the two blue lines.

The first measurement shows the execution time of memory accesses depending on the number of accesses. Figure 6.2 shows the execution time (i.e. the difference between an `RDTSC` invocation before the first memory access and an `RDTSC` invocation after the last memory access).
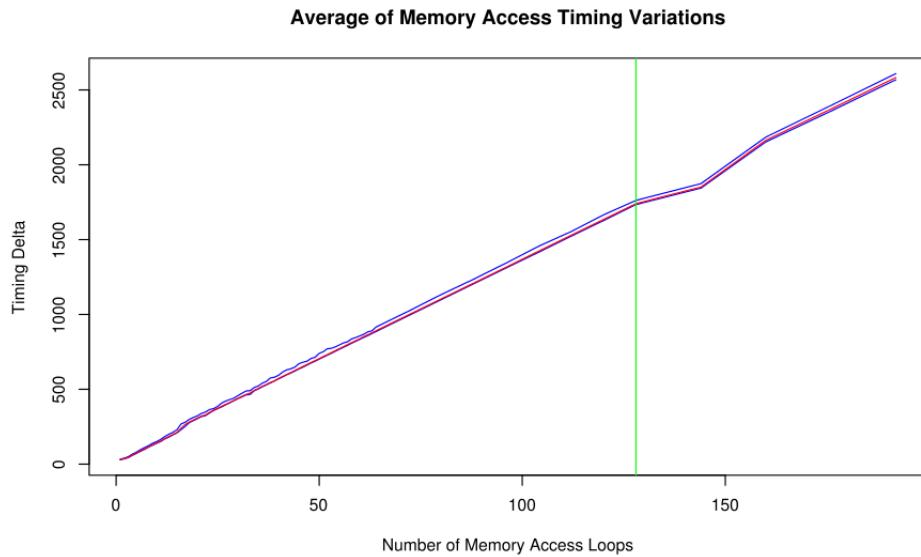
Figure 6.2: Average time duration for memory accesses

As expected, the graph shows a linear increasing of the time duration when memory is accessed. That means, each new memory access adds on average an equal amount of execution time.

The next measurement provided with figure 6.3 shows the standard deviation of the memory access time when increasing the number of memory accesses. With the standard deviation, the size of the timing variation is depicted, i.e. how "large" the variations of the memory access times fluctuate.

Standard Variation of Memory Access Timing Variations
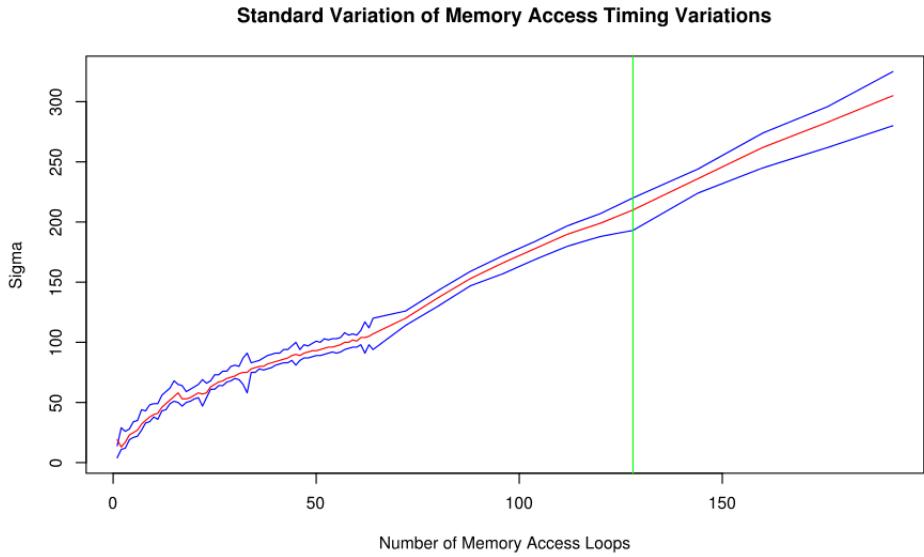
Figure 6.3: Standard deviation of time duration for memory accesses

The graph with the standard deviation clearly shows that it is an almost linear increase of the standard deviation. This graph implies that the execution variations increase linearly with the number of memory accesses. *The conclusion that can be drawn from this result is that each addition memory access attempt will increase the timing variations and thus the uncertainty of the memory access times.*

To allow comparing the standard deviation values for the different memory access times, the variation coefficient can be used. The variation coefficient "normalizes" the standard deviation by dividing it with the mean value of the time measurement. Figure 6.4 shows the variation coefficient for the different memory access loop iterations.
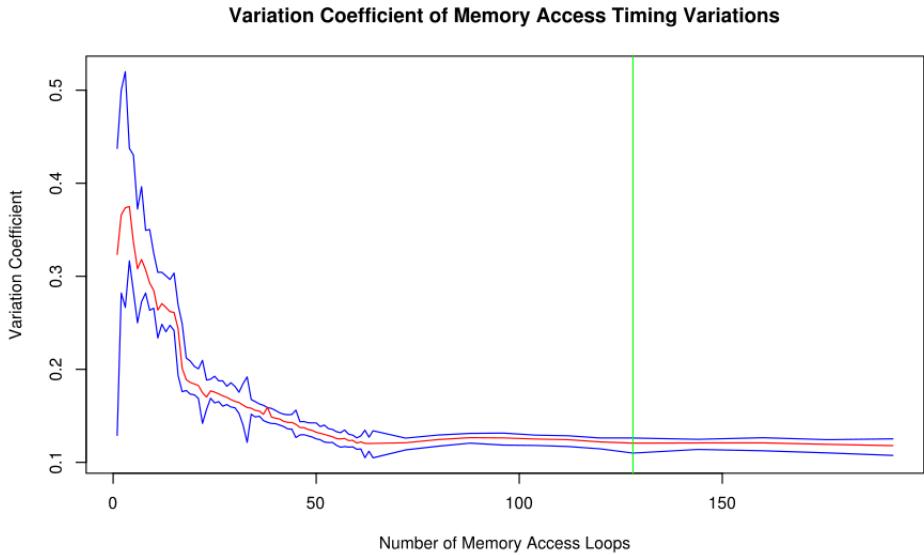
Figure 6.4: Variation coefficient of time duration for memory accesses

The graph nicely show the stabilization of the variation coefficient the higher the number of memory access loops. *That stabilization, i.e. flattening of the curve, demonstrates that the standard deviation in relation to the number of memory accesses increases almost perfectly linearly.* The spikes at the lower number of memory access loops are due to higher impact of slight measuring errors that have are more visible at the smaller measurements. Where do the measuring errors come from? The code that invokes the RDTSC reading also is affected by the memory access variations. When invoking that code twice (for the beginning and ending time stamps) to measure only one or two memory accesses, error added by the time reading is relatively higher than when having several tens or hundreds of memory accesses.

Another enlightening statistic is the count of how many different timing values can be detected. The following graph presented in figure 6.5 now counts how many different memory access times can be detected throughout the measurement.
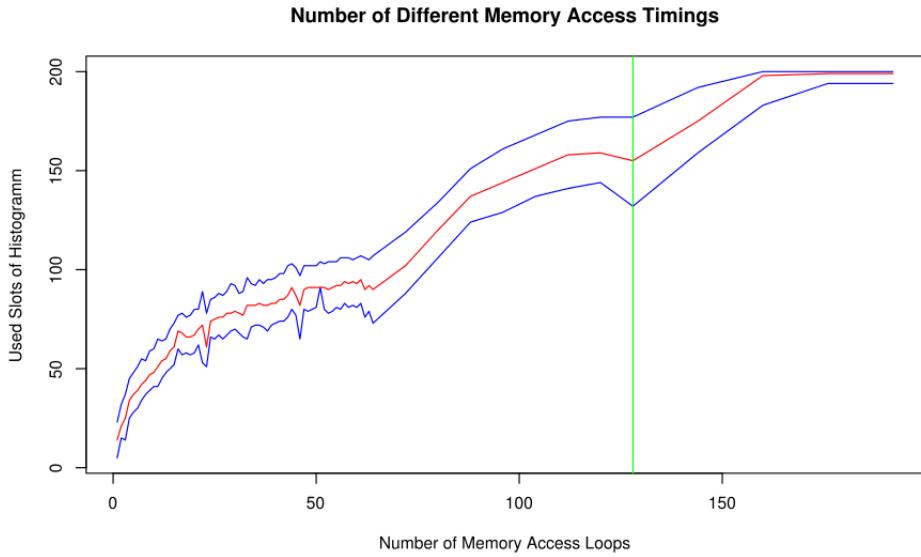
Figure 6.5: Number of different of time durations for memory accesses

Before interpreting the graph, please note that the test is set up to only measure up to 200 different values. When considering the increase of the standard deviation as outlined above, the result of the graph with the number of different time measurements is fully expected and understandable: the more memory access loops are performed, the more different memory access times are measured. This result is fully expected as the increase in the memory access time variation is the factor that increases the standard deviation. *The conclusion can be drawn that the more memory accesses are performed, the stronger the timing variations of these accesses are.*

As a final graph, figure 6.6 shows the calculation of the Shannon Entropy value for the timing measurement. As outlined in section 6.3, the Shannon Entropy values are subject to a calculation error that is up to one bit. As it cannot be identified how large the error is at a given number of memory access loops, the fluctuations in that graph must be interpreted accordingly – i.e. fluctuations within one bit must be considered to show about equal measurements.

Shannon Entropy of Memory Access Timing Variations
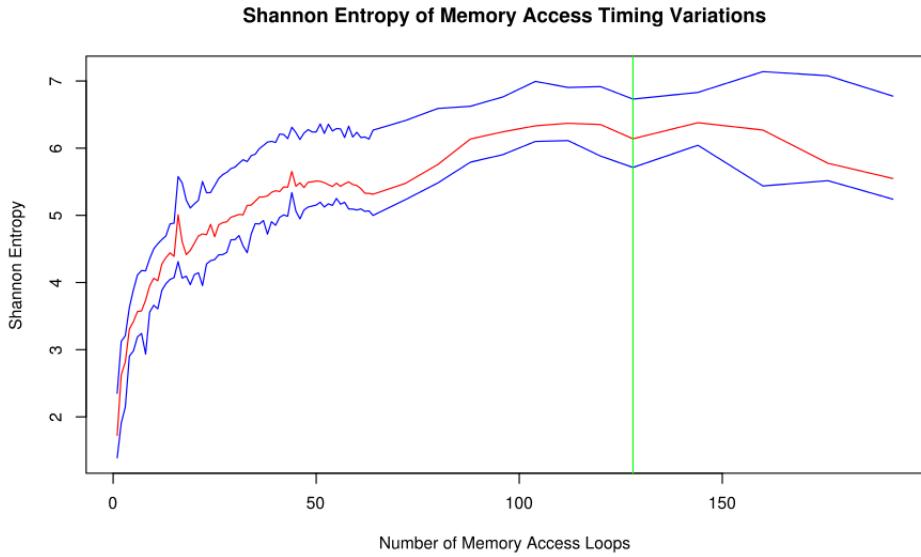
Figure 6.6: Shannon Entropy of time durations for memory accesses

The graph showing the Shannon Entropy values support all previous graphs and conclusions by showing that the variations increase with the increase of memory access loops.

The following final conclusion can be drawn from the measurements: *After having sufficient memory accesses to completely fill L1 and draw from L2 the timing variations and therefore the uncertainty regarding the total time for memory accesses increases linearly with the number of measured memory accesses.* This implies that memory accesses can be considered a good source for entropy.

Note, the tests were partially redone in light of the results in section 6.1. In this section, various CPU mechanisms were disabled with partially having a severe impact on the measured timing variations. When altering the CPU for the memory access timing measurement, the following findings based on figure 6.7 apply:
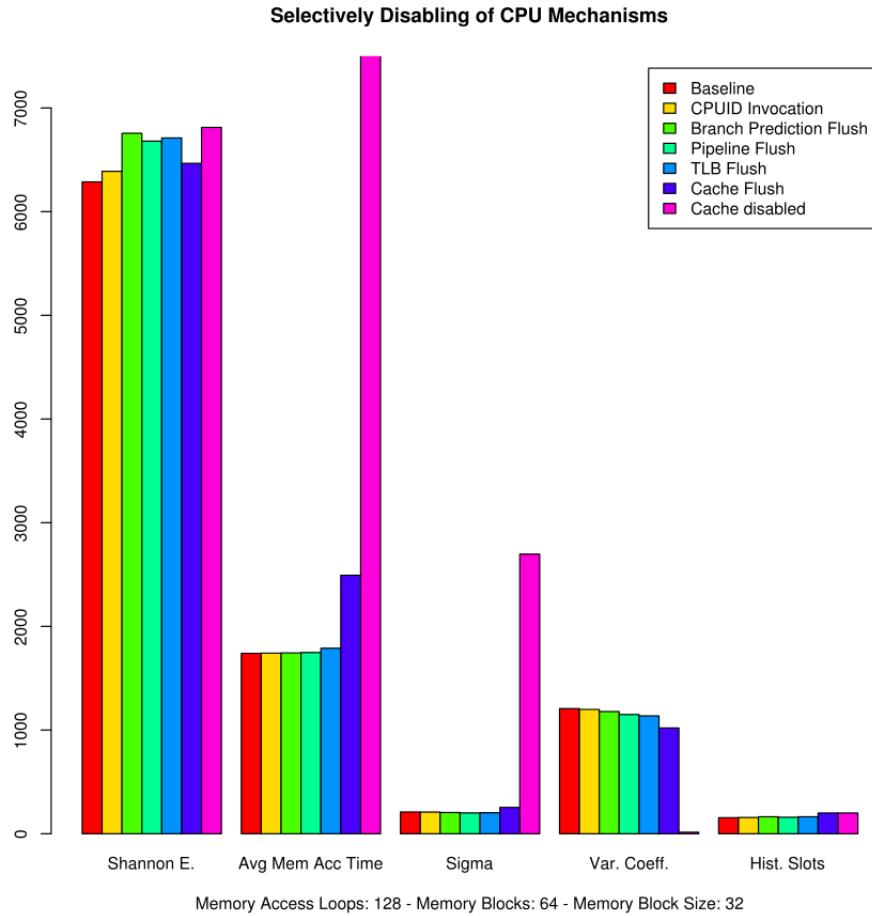
Figure 6.7: Impact of Selectively Disabling of CPU Mechanisms on Memory Access Timing Variations

- Using a serialization instruction like `CPUID` which removed all timing variations for the CPU execution time jitter does not have any impact on the measurements for memory access timing.

- Flushing the branch prediction unit of the CPU does not have a measurable impact on the memory access timing variations.

- Flushing the instruction pipeline or the TLB cache does not show any impact on the memory access timing variations.

- Disabling L1 and L2 caches showed a significant impact by dramatically increasing the memory access timing variations. This is expected, because the CPU now always has to access the main memory. As the accesses to main memory are subject to more wait states than L2 accesses, the variations must increase.

These additional results can be summarized as follows: *Changing the CPU-*

internal mechanisms for code execution has no impact on the memory access timing variations. Changing how the CPU accesses memory by disabling the caches significantly increases variations and thus entropy.

6.2.3   Memory Accesses and ~~LFSR~~Hash Loop

After independently discussing the memory access noise source, a view of the combined noise of memory accesses and the ~~LFSR~~hash loop should be performed.

The impact of the memory accesses to the ~~LFSR~~hash loop can be easily shown by depicting the execution time variations of just the ~~LFSR~~hash loop and then the ~~LFSR~~hash loop together with the memory accesses. The following graphs show the execution timing variations on a test system where just the ~~LFSR~~hash loop is challenged to produce sufficient variations. But when adding the memory access variations, more than enough timing variations are recorded. First, the graphs for the lower boundary are shown.
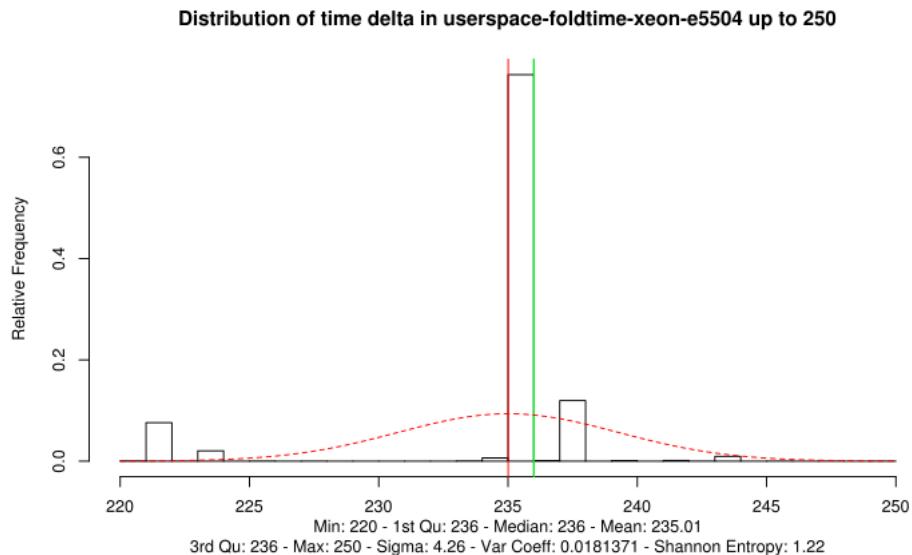
Figure 6.8: ~~LFSR~~hash loop without memory accesses on Intel Xeon E5504 – lower boundary

Distribution of time delta in userspace-foldtime-xeon-e5504-memacc.O0 up to 2500

Min: 1965 - 1st Qu: 1996 - Median: 2065 - Mean: 2092.08
3rd Qu: 2173 - Max: 2500 - Sigma: 111.53 - Var Coeff: 0.0533125 - Shannon Entropy: 6.53
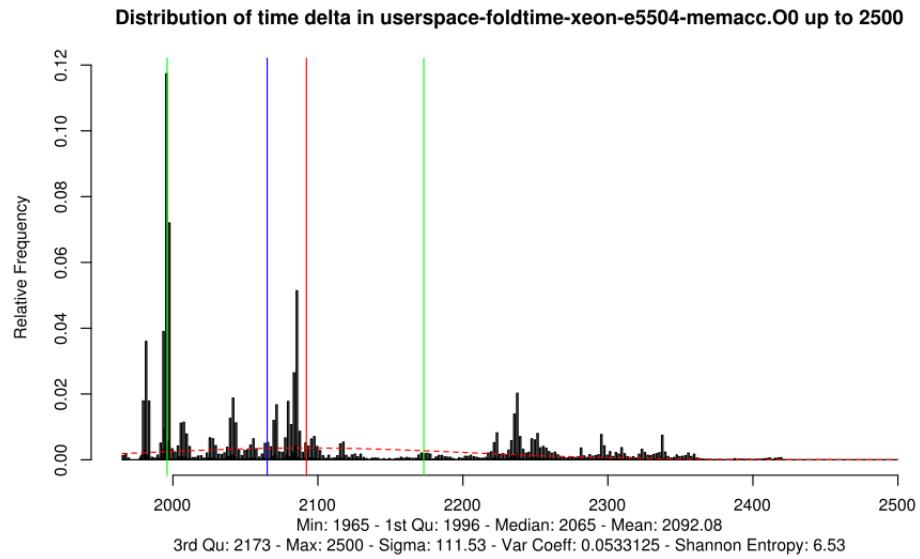
Figure 6.9: ~~LFSR~~hash loop with memory accesses on Intel Xeon E5504 ~~— lower boundary~~

~~And now the graphs for the upper boundary.~~

Distribution of time delta in userspace-foldtime-xeon-e5504 up to 2500

Min: 219 - 1st Qu: 792 - Median: 1290 - Mean: 1260.89
3rd Qu: 1808 - Max: 2488 - Sigma: 575.13 - Var Coeff: 0.456134 - Shannon Entropy: 5.95
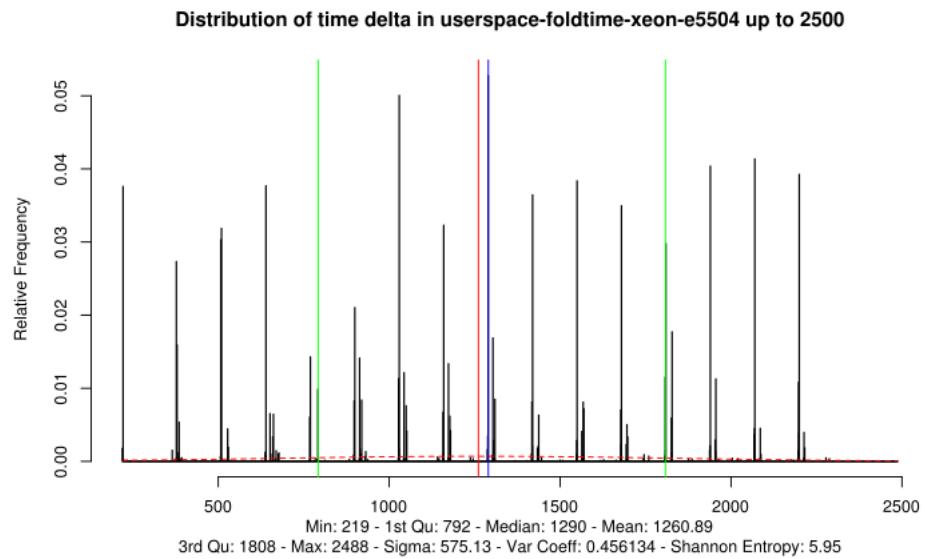
Figure 6.10: ~~LFSR loop without memory accesses on Intel Xeon E5504 - upper boundary~~

Distribution of time delta in userspace-foldtime-xeon-e5504-memacc.O0 up to 8300

Min: 1968 - 1st Qu: 3584 - Median: 5181 - Mean: 5102.09
3rd Qu: 6757 - Max: 8300 - Sigma: 1834.81 - Var Coeff: 0.3596196 - Shannon Entropy: 10.98
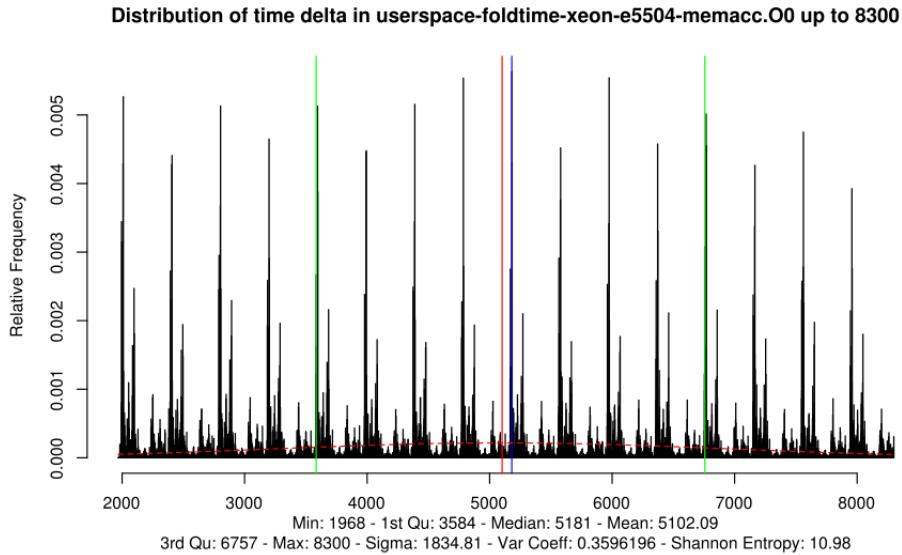
Figure 6.11: ~~LFSR loop with memory accesses on Intel Xeon E5504 – upper boundary~~

Similar results are obtained for other systems. And these results speak for themselves: memory access provide a significant source for variations in addition to just the ~~LFSR~~hash loop.

6.3 Noise Source Testing Without Operating System

The execution timing tests discussed in section 5.1 do not need any specific support from the operating system it runs on. Nonetheless, an operating system is needed to allow the code to be executed on the CPU, i.e. to boot an environment that can execute some code where the results can somehow be conveyed. Or not?

The Memtest86 tool is intended to be started directly from the boot loader without any operating system running. In essence, that tool is its own operating system with the sole purpose of executing some (memory) tests.

This tool now is used to allow running the CPU execution timing tests on bare metal (err, on bare silicon) where no operating system with any parallel threads or tasks can interfere. The Memtest86 tool is modified by removing all memory tests and adding a number of CPU execution timing variation tests. The code for the tool is provided in the directory `test_baremetal/`.

The goal with this testing is to eliminate the impact of the operating system by only and exclusively executing the test cases on the CPU. No scheduling or context switching will occur during the test execution. Even interrupts are not processed while the tests execute. The test is implemented by only printing the results after the completion of each test (i.e. not during the execution of a test). This approach further reduces the impact of the test framework on the measurements.

This measurement is the same measurement used for determining the ~~En-tropy~~lower boundary of entropy throughout this document. This means that here only the worst case is analyzed.

The following tests are implemented:

Test No	Test Case Description
0	This is the baseline test by simply executing two time stamp reads immediately after each other. This test shall help finding the right CPU clearing and flushing operations that eliminate all jitter.
1	This test covers the memory access operation by measuring each memory access individually.
2	The entire entropy gathering operation is tested with this test. The entropy gathering covers the ~~LFSR~~hash loop operation and the memory access operation. When setting a configuration flag, the memory access operation can be disabled to ensure that this test only measures the ~~LFSR~~hash loop operation. The execution time of each loop is measured.
3	This test tries to measure some CPU characteristics by placing well-crafted CPU instructions between the time measurements. However, this test is considered irrelevant for the CPU Jitter RNG measurement.
4	This test implements an automated invocation of test 1. For test 1, the memory block size, the number of blocks and the number of access loop iterations can be defined. This test repeats test 1 after incrementing the counters. Test results are displayed and can be transported to a different machine using the Morse code.

For each test, the following options can be toggled:

- Enabling / Disabling of the L1/L2 cache during the measurements. To ensure a responsive test framework, the caches are always enabled when not executing tests. The caches are disabled as outlined in section 6.1.9.

- Enabling / Disabling L1/L2 Cache Flush: Before the execution of a test, the L1/L2 cache can be flushed as discussed in section 6.1.4.

- Enabling / Disabling of Pipeline Flush: The CPU instruction pipeline can be flushed before the execution of each test as outlined in section 6.1.3.

- Enabling / Disabling of TLB Flush: The TLB can be flushed before the execution of each test as outlined in section 6.1.6.

- Enabling / Disabling of serialization: Before the execution of each test, a serialization instruction is invoked. The code uses the `CPUID` instruction invoked with zero as input parameters in `EAX` and `EDX`.

- Enabling / Disabling of Mem Jitter: This flag toggles whether the memory access operation shall be invoked when testing the operation of the entropy collection loop. By disabling this flag, the testing would only measure the ~~LFSR~~hash loop operation. This flag is only relevant for test 2.

- Enabling / Disabling of Morse Results: When the automated testing is selected, the displayed test results of the statistical data calculated after completing testing can be transferred to another computer using the Morse code. As the test framework has no drivers, the only way to extract data is by Morse code. The individual values are separated by commas and the line separator is a dash. The WPM of the Morse communication is about 21. The following command can be used for capturing data:

```
demorse -f 21 -phw:0 -rCapture  -lCapture -uleft -d8
```

For each test, the following numeric options can be set:

- Branch Pred Fl Lp: The number of branch prediction flush loops defines the number of loops discussed in section 6.1.2.

- Memory Access Loops: The number of memory access loops can be set here.

- Test Loops: The number of test executions before displaying the histogram and various statistical data of measurements can be set here. Per default, the value is set to 10,000. This value only applies to test 4.

- Number of Mem Blks: The number of memory blocks defining the size of the memory to be accessed is set here.

- Jitter ~~LFSR~~hash Iter: The number of loop iterations of the ~~LFSR~~hash loop is defined with this value. If set to zero, the automated selection of the number of loop iterations as found in normal operation of the RNG is done. This value applies to test 2 only.

- Memory Block Size: The size of one memory block in bytes is defined with this configuration option.

For each test case, up to 200 different execution timing values are recorded. These records form a histogram of the execution times. For each seen timing value, the number of occurrences is recorded. Figure 6.12 illustrates the results for the execution of the tests within a KVM instance[10].

---

[10]This figure shall only serve as illustration for the discussion and the explanation on how to interpret the results. The test results on a KVM cannot be interpreted as bare metal testing and are therefore not used for any conclusions.

```
   Jittertest86+ v4.20      | Pass    %
Intel Core Gen2 2494 MHz    | Test    %
L1 Cache:   32K  99779 MB/s | Test #2  [Jitter Entropy Collection Loop Test]
L2 Cache: 2048K  48911 MB/s | Testing:  208K - 1024M 1024M
L3 Cache:      None         | Pattern:
Memory  : 1024M  20446 MB/s |-------------------------------------------------
Chipset/IMC : ***FAIL SAFE***FAIL SAFE***FAIL SAFE***FAIL SAFE***FAIL SAFE***
*** Memtest86+ is running in fail safe mode. Same reliability, less details ***

     Exec Option       |     Flushing     | Jitter :       :    Mem   | Tests
Cache Serial TstLoops | Cach Bran Pipe TLB | Mem Ite : Blck : Size Acce |
 on    off    1024    | off  1    off  off | on   1   : 128  : 1024 64  | 6783
-------------------------------------------------------------------------------
Histogram (<Time Delta>:<Number of occurrences>)
2312 :     1 | 1894 :    1 | 1657 :    2 | 1684 :    1 | 1924 :    1 |
1730 :     2 | 1724 :    1 | 1700 :    1 | 1722 :    1 | 1712 :    2 |
1842 :     1 | 1691 :    1 | 1703 :    1 | 1846 :    1 | 1942 :    1 |
1524 :    26 | 1552 :   12 | 1775 :    1 | 1710 :    1 | 1545 :   16 |
Shannon Entr (div by 1024   )   5256    Min 4565    Max 6568    Med 5352
Delta Mean   (div by 1      )   1543    Min 1535    Max 4559    Med 1684
Sigma        (div by 1      )   53      Min 47      Max 6054    Med 340
Var Coeff    (div by 10000  )   343     Min 303     Max 34476   Med 1872
Switches     (div by 1      )   1008    Min 989     Max 1024    Med 1009
Used Slots   (div by 1      )   100     Min 100     Max 100     Med 100
(ESC)Reboot  (c)configuration  (SP)scroll_lock  (CR)scroll_unlock
```

Figure 6.12: Execution of Bare-Metal CPU Execution Jitter Test on KVM

The test framework presents the following data:

- In the upper right corner, the currently executed test is referenced.

- In the row in the middle of the screen lists the selected options. All options can be set by hitting the character `c` during operation.

- In the lower part, parts of the recorded histogram of timing data is shown. A group of two numbers delimited by a colon shall be interpreted together. The left hand side of the colon is the time duration for one ~~LFSR~~hash loop operation. The right hand side is the number of occurrences that were counted for the test. The groups of value pairs can be considered to form a histogram. For example, the following results are observable from the screenshot above: The ~~LFSR~~hash loop execution duration of 2312 cycles is observed 1 time. The ~~LFSR~~hash loop execution duration of 1552 cycles is observed 12 times. The ~~LFSR~~hash loop execution duration of 1524 cycles is observed 26 times, And so on. Note, only a subset of all recorded histogram slots is depicted due to space constraints.

- Below the histogram, different statistical values are shown with a divisor used as a scaling factor in the following. To keep the test framework minimal, only integer calculation is possible. To limit the calculation error due to truncation performed with divisions, the formulas partially increase the dividend to obtain values which are greater than 1. The reader now must manually divide the shown number by the displayed divisor. For each statistical value, the minimum observed value, the maximum observed value and the mean value during the test cycle is listed. The following statistical values are shown:

    - Shannon Entropy: The calculation of the Shannon Entropy is calcu-

lated with the formula

$$H = \sum_{i=1}^{i\leq 200} p_i \cdot log_2 p_i = \sum_{i=1}^{i\leq 200} \frac{s_i}{l} \cdot log_2 \frac{s_i}{l} = \sum_{i=1}^{i\leq 200} \frac{s_i}{l} \cdot (log_2 s_i - log_2 l)$$

where s_i specifies the number of observations in the histogram for one timing value and l specifies the number of test loops performed. The scaling factor f is added by modifying the formula above as follows

$$H \cdot f = \sum_{i=1}^{i\leq 200} \frac{s_i \cdot f}{l} \cdot (log_2(s_i \cdot f) - log_2(l \cdot f))$$

Warning: Due to the integer calculation for the logarithm, the result of the formula overestimates the Shannon Entropy by up to one bit. Therefore, that value shall not be used as a blank statement of the entropy contained, but rather as a reference to the variations found in the time deltas of the test sample. Nonetheless, the Shannon Entropy is not lower than the calculated value minus one (bit). When comparing this value with Shannon Entropy measurements from other tests, always consider the Sigma and the Variation Coefficient in addition.

– Delta Mean: This value simply calculates the mean of the histogram for one test execution:

$$\overline{x} = \frac{s_i \cdot v_i}{l}$$

where s_i specifies the number of observations in the histogram for one timing value, v_i specifies the timing value in the histogram, and l specifies the number of test loops performed.

– Sigma: The standard derivation is calculated by first calculating the variance followed by calculating the square root of the variance. The following formula is used:

$$\sigma = \sqrt{\sum_{i=1}^{i\leq 200} (v_i - \overline{x}) \cdot (v_i - \overline{x}) \cdot s_i}$$

where the variables have the same meaning as outlined above.

– The variation coefficient is a scaling of the standard derivation to the mean of the data set. The following formula shows the calculation of the variation coefficient V

$$V = \frac{\sigma}{\overline{x}}$$

As the variation coefficient can be smaller than 1, it is scaled with the factor f

$$V \cdot f = \frac{\sigma \cdot f}{\overline{x}}$$

– The number of switches specifies how often a different time delta compared to the immediately previously seen time delta value is recorded. For example, consider the following series of time delta measurements: 100, 100, 101, 102, 102, 100. The number of switches here is 3.

– The value for used slots simply lists how many of the 200 available slots in the histogram are actually filled. That means, how many different time delta values are recorded.

7 Standards Compliance

7.1 FIPS 140-2 Compliance

FIPS 140-2 specifies entropy source compliance in FIPS 140-2 IG 7.18. This section analyzes each requirement for compliance. The general requirement to comply with SP800-90B [Turan et al., 2018] is analyzed in section 7.2.

7.1.1 FIPS 140-2 IG 7.18 Requirement For Statistical Testing

The Jitter RNG is provided with the following testing tools:

- Raw Entropy Tests: The tests obtain the raw unconditioned and unprocessed noise information and records it for analysis with the SP800-90B non-IID statistical test tool. The test tool includes the gathering of raw entropy for one execution run as well as for the restart tests required in SP800-90B section 3.1.4. The tool adjusts the data to be processed by the SP800-90B statistical test tool. The test tool provides the SP800-90B minimum entropy values for the lower and upper boundaries documented in section 5. The testing was successfully conducted on Intel x86-based systems, ARM-based systems including smart phones, embedded devices with MIPS and ARM CPU, IBM POWER, IBM System-Z Mainframes.

- Jitter RNG Output Tests: The Jitter RNG output is captured and processed with the SP800-90B IID statistical tests.

- Health Test Assessments: The Adaptive Proportion Test and Repetition Count Tests are validated independently from the noise source to validate the false-rejection rate as well as false acceptance rate. The test tool invokes these tests while the Jitter RNG is operational. The test verifies whether the online health tests trigger alarms.

In particular the first test covers the test requirement of FIPS 140-2 IG 7.18.

7.1.2 FIPS 140-2 IG 7.18 Heuristic Analysis

FIPS 140-2 IG 7.18 requires a heuristic analysis compliant to SP800-90B section 3.2.2. The discussion of this SP800-90B requirement list is given in section 7.2.

7.1.3 FIPS 140-2 IG 7.18 Additional Comment 1

The first test referenced in section 7.1.1 covers this requirement.

The test invokes the raw noise components feeding the hash, the memory access and the SP800-90B health tests in a tight loop to develop a worst case scenario. The regular Jitter RNG operation adds additional entropy by the processing of the hash and memory access results. Therefore, the test is considered to show the lower boundary of the entropy measurements.

7.1.4 FIPS 140-2 IG 7.18 Additional Comment 2

The lowest entropy yield is analyzed by gathering raw entropy data solely over the hash operation and memory access operations, disregarding additional processing that also delivers entropy. In addition, the raw entropy gathering obtains

the lower and upper boundary raw entropy information as documented in section 5.

The lower boundary, however, is considered to be informative to support the assessment of the Jitter RNG. Its results, however, should always be analyzed with the caution that in production mode, the Jitter RNG does not exhibit this behavior, i.e. the lower boundary is a worst case that deactivates an important feature of the Jitter RNG.

The entropy is not considered to degrade when using the hardware within the environmental constraints documented for the used CPU. The online health tests are intended to detect entropy source degradation. The documentation provided with the jitterentropy(3) man page explains the actions to be taken if such entropy source degradation is detected.

7.1.5 FIPS 140-2 IG 7.18 Additional Comment 3

The Jitter RNG uses SHA-3 256 as conditioning component. This implementation can be tested with the ACVP Parser which contains a backend to cover the Jitter RNG SHA-3 256 implementation.

Using the mentioned tool, successful test runs were conducted.

The requested power-up self test is provided by the Jitter RNG function `sha3_tester` which is invoked during the execution of `jent_entropy_init(3)`. The test vector is obtained from a successful ACVP test execution.

7.1.6 FIPS 140-2 IG 7.18 Additional Comment 4

The restart test is covered by the first test documented in section 7.1.1.

7.1.7 FIPS 140-2 IG 7.18 Additional Comment 6

The entropy assessment usually shows this conclusion – tests performed on Intel x86-based systems, ARM-based systems including smart phones, embedded devices with MIPS and ARM CPU, IBM POWER, IBM System-Z Mainframe show the following conclusions:

The entropy rate for all devices validated with the raw entropy tests outlined in section 7.1.1 show that the minimum entropy values are always above one bit of entropy per four data bits. The data bits are the least significant bits of the time deltas generated by the raw noise.

Assuming the worst case that all other bits in the time delta have no entropy, that entropy value above one bit of entropy applies to one time delta.

The Jitter RNG gathers at 256 time deltas for returning 256 bits of random data and it uses SHA-3 256 which is considered to provide full entropy as outlined in Turan et al. [2018] section 3.1.5.1.2. Thus, the Jitter RNG collected 256 times more than one bit of entropy for its 256 bit output.

As the Jitter RNG maintains a 256 bit entropy pool, its entropy content cannot be larger than the pool itself. Thus, the entropy content in the pool after collecting 256 time deltas is the maximum of 256 bits and measured entropy value from the previous steps. As long as the entropy measurement shows that each time delta has more than one bit of entropy, the entropy rate of the Jitter RNG random numbers is 256 bits of entropy per 256 bit data block.

This implies that the Jitter RNG data has 1 bit of entropy per data bit.

7.1.8 FIPS 140-2 IG 7.18 Additional Comment 9

N/A as the raw entropy is a non-IID source and processed with the non-IID SP800-90B statistical tests as documented in section 7.1.1.

7.2 SP800-90B Compliance

This chapter analyzes the compliance of the Jitter RNG to the SP800-90B Turan et al. [2018] standard considering the FIPS 140-2 implementation guidance 7.18 which alters some of the requirements mandated by SP800-90B.

7.2.1 SP800-90B Section 3.1.1

The collection of raw data for the SP800-90B entropy testing documented in section 7.1.1 uses 1,000,000 consecutive time deltas obtained in one execution round.

The gathering random numbers generated by the SHA-3 operation using the test documented in section 7.1.1 includes 1,000,000 consecutive 256 bit blocks. The individual Jitter RNG blocks are concatenated to form a bit stream.

The restart tests documented in section 7.1.1 perform 1,000 restarts collecting 1,000 consecutive time deltas.

7.2.2 SP800-90B Section 3.1.2

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

7.2.3 SP800-90B Section 3.1.3

Please see section 7.1.7: The entropy of the raw noise source is believed to have more than one bit of entropy per time delta to allow to conclude that one output block of the Jitter RNG has (close to) one bit of entropy per data bit.

The first test referenced in section 7.1.1 performs the following operations to provide the SP800-90B minimum entropy estimate:

1. Gathering of the raw entropy data of the time stamps for both, the lower and upper boundary.

2. Obtaining the four least significant bits of each time delta and concatenate them to form a bit stream. Two bit streams are gathered, one for the upper and one for the lower boundary.

3. The bit stream is processed with the SP800-90B entropy testing tool to gather the minimum entropy. Two minimum entropy values are obtained, one for the lower and one for the upper boundary.

For example, on an Intel Core i7 Whiskey Lake system, the SP800-90B tool shows the following minimum entropy values when multiplying the SP800-90B tool bit-wise minimum entropy by four since four bits are processed:

- Using the 4 least significant bits of the upper boundary time deltas: 3.55952

- Using the 4 least significant bits of the lower boundary time deltas: 2.784276

7.2.4 SP800-90B Section 3.1.4

For the restart tests, the raw entropy data is collected for 1,000 Jitter RNG instances allocated sequentially. That means, for one collection of raw entropy, one Jitter RNG instance is allocated. After the conclusion of the data gathering it is deallocated and a new Jitter RNG instance is allocated for the next restart test round.

Each restart test round stores its lower and upper boundary time deltas in an individual file.

After all raw entropy data is gathered, one matrix for the lower and one for the upper boundary is generated where each line in the matrix lists the time deltas of one restart test round. The first column of the matrix, for example, therefore contains the first time delta after initializing the Jitter RNG instance for each restart test round.

The SP800-90B minimum entropy values column and row-wise is calculated the same way as outlined above:

1. Gathering of the raw restart entropy data of the time deltas for both, the lower and upper boundary.

2. Obtaining the four least significant bits of each time delta either row-wise or column-wise and concatenate them to form a bit stream. There are 1,000 bit streams row-wise upper boundary, 1,000 bit streams row-wise lower boundary, 1,000 bit streams column-wise upper boundary and 1,000 bit streams column-wise lower boundary generated.

3. The bit streams are processed with the SP800-90B entropy testing tool to gather the minimum entropy.

In a following step, the sanity check outlined in SP800-90B section 3.1.4.3 is applied to the restart test results. The steps given in 3.1.4.3 are applied.

For example, on an Intel Core i7 Whiskey Lake system, the SP800-90B tool shows the following minimum entropy values when multiplying the SP800-90B tool bit-wise minimum entropy by four since four bits are processed:

- Using the 4 least significant bits of the upper boundary time deltas in column-wise assessment – lowest entropy value of all 1,000 column entries: 2.074924

- Using the 4 least significant bits of the lower boundary time deltas in column-wise assessment – lowest entropy value of all 1,000 column entries: 2.145488

- Using the 4 least significant bits of the upper boundary time deltas in row-wise assessment – lowest entropy value of all 1,000 column entries: 2.196152

- Using the 4 least significant bits of the lower boundary time deltas in row-wise assessment – lowest entropy value of all 1,000 column entries: 2.029292

- Sanity check of upper boundary 1,000 x 1,000 matrix passes with value of one

71

- Sanity check of lower boundary 1,000 x 1,000 matrix passes with value of one

With the shown values, the restart test validation passes according to SP800-90B section 3.1.4.

7.2.5 SP800-90B Section 3.1.5

The conditioning component applied to the interrupt noise source are performed at different stages as outlined in section 3.3.

Truncation The truncation operation ensures that the entropy in that data is at maximum the truncated hash. Truncation is applied to a full output block of the Jitter RNG. As outlined in the following, each Jitter RNG output block is considered to have full entropy. Thus, the concatenation returns data whose entropy is equal to the data size.

Concatenation When applying a concatenation operation, the Jitter RNG simply adds the entropy delivered with each data entry part.

Hash The input of the hash n_{in} is fixed as it processes 256 healthy time delta values together with 256 times the previous hash output.

The output of the hash n_{out} is the message digest size of SHA-3 256, i.e. 256 bits.

Approach for Calculating Entropy In order to generate one output block, 256 bits of entropy 256 healthy time delta values are required as illustrated below.

With the given combination of the hash as outlined above, the following approach for the entropy calculation is taken for each of the data processing steps outlined in section 3.3:

- Equation 3.2:

    - n_{in} equals to 384 bits by hashing the previous state of the entropy pool, the 64 bit value of the time delta and the 64 bit loop count value.
    - n_{out} is the message digest size of 256 bits.
    - nw is the message digest size in bits.

- Function 3.3 repeats the operation 3.2 256 times with a new time delta. Thus does not alter input data sizes.

- Function 3.4 repeats the operation 3.4 as often as needed to produce the requested numbers of bits. Thus does not alter input data sizes.

7.2.6 SP800-90B Section 3.1.5.2

The Jitter RNG uses SHA-3 256 as hash operation as outlined above and is considered to be a vetted conditioning component. Thus the entropy rate of the hash output is calculated as follows using the aforementioned variables for the hash function. In addition, the following consideration applies:

- The entropy content of the input h_{in}: The input entropy of the hash is equal to the entropy provided by the processed time delta and the entropy already present in the entropy pool considering that both data components are hashed at the same time to form a new entropy pool state. Of course, the entropy held in the entropy pool will never be larger than the digest size of the used hash which is compliant to [Turan et al., 2018] section 3.1.5.1.1 table 1.

Function 3.2 Output_Entropy To perform a calculation of the Output_Entropy of a conditioning component, the input entropy must be considered. The heuristic input entropy awarded for one time stamp processed by the Jitter RNG is given in equation 7.2. Due to the concatenation operation of time stamps, the entropy of multiple time stamps can be added.

The following calculation applies to the case when one new time delta is added to the SHA3-256 state. Thus, the calculation is performed for the formula $h_{out_{SHA3-256}} = Output\_Entropy_{SHA3-256}(384, 256, 256, 1)$

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

$$P_{low} = \frac{(1 - 2^{-1})}{2^{384} - 1} \approx 2^{-385}$$

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

$$\psi = 2^{384-256} \cdot 2^{-385} + 2^{-1} = 2^{-257} + 2^{-1} \approx 2^{-1}$$
$$U = 2^{384-256} + \sqrt{2 \cdot 256 \cdot (2^{384-256}) \cdot ln(2)} = 2^{128} + \sqrt{2^{137} \cdot ln(2)} \approx 2^{128}$$
$$\omega = 2^{128} \times 2^{-385} = 2^{-257}$$
$$-log_2(max(2^{-257}, 2^{-1})) = 1$$

For the function 3.2, the entropy when 256 time deltas are received (and thus 384·256 bits are processed) is $h_{out_{SHA3-256}} = Output\_Entropy_{SHA3-256}(98304, 256, 256, 256)$ following [Turan et al., 2018] section 3.1.5.1.2. Therefore, the following calculation is applicable:

$$P_{high} = 2^{-256}$$

$$P_{low} = \frac{(1 - 2^{-256})}{2^{98304} - 1} \approx 2^{-98304}$$

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

$$\psi = 2^{98304-256} \cdot 2^{-98304} + 2^{-256} = 2^{-256} + 2^{-256} = 2^{-255}$$

$$U = 2^{98304-256} + \sqrt{2 \cdot 256 \cdot (2^{98304-256}) \cdot ln(2)} = 2^{98048} + \sqrt{2^{98057} \cdot ln(2)} \approx 2^{98048}$$

$$\omega = 2^{98048} \times 2^{-98304} = 2^{-256}$$

$$-log_2(max(2^{-255}, 2^{-256})) = 255$$

That calculation states that the 256 bits of entropy which went into the conditioning component cause 255 bits of entropy to be extracted. However, SP800-90B section 3.1.5.1.2 states that vetted conditioning components can claim full entropy outputs. That means that for the remainder of this document, the output is considered to contain 256 bits of entropy if 256 bits of entropy went in.

Furthermore, if an oversampling factor of, say, 4, is applied, $4 \cdot 256 = 1024$ time deltas of 64 bits each are to be received for generating 256 bits of entropy. Thus, the formula $h_{out_{SHA3-256}} = Output\_Entropy_{SHA3-256}(393216, 256, 256, 256)$ must be applied:

$$P_{high} = 2^{-256}$$

$$P_{low} = \frac{(1 - 2^{-256})}{2^{393216} - 1} \approx 2^{-393216}$$

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

$$\psi = 2^{393216-256} \cdot 2^{-393216} + 2^{-256} = 2^{-256} + 2^{-256} = 2^{-255}$$

$$U = 2^{393216-256} + \sqrt{2 \cdot 256 \cdot (2^{393216-256}) \cdot ln(2)} = 2^{392960} + \sqrt{2^{392969} \cdot ln(2)} \approx 2^{392960}$$

$$\omega = 2^{392960} \times 2^{-393216} = 2^{-256}$$

$$-log_2(max(2^{-255}, 2^{-256})) = 255$$

Just as mentioned above, the statement in SP800-90B is applied that allows to claim full entropy.

Comparing the results of all calculations for the SHA3-256 conditioning component, the general conclusion can be drawn that underlies the entire entropy assessment and therefore data entropy management applied by the Jitter RNG for all vetted conditioning operations:

$$h_{out_{vetted}} = Output\_Entropy_{vetted} = min(h_{in}, n_{out_{vetted}}) \qquad (7.1)$$

Conclusions for Output_Entropy As stated in [Turan et al., 2018] section 3.1.5.1.2, vetted conditioning components are allowed to claim full entropy. In case of full entropy, the following is applied which matches exactly analysis and conclusion of equation 7.1:

- $h_{out_{SHA3-256}} = nw_{SHA3-256} = n_{out_{SHA3-256}} = 256$.

Based on that conclusion, the entropy rate for each processing step given in section 3.3 can be illustrated in the following.

The entropy for the individual time deltas is defined with the following equation that is applicable irrespective when using a hardware-provided time stamp or the Jitter RNG software time stamp outlined in section 3.8.1 – the variable

osr denominates the oversampling rate applied during the initialization of the Jitter RNG:

$$h_t = \frac{1}{osr} \tag{7.2}$$

The entropy resulting from one hash operation defined by equation 3.2 is calculated as the sum of the entropy present in the entropy pool and the entropy in the new time delta capped by the output size of the used hash operation. Note that when a new random number is to be generated, the entropy pool is considered to contain no entropy, i.e. $h_{pool_0} = 0$:

$$h_{pool_n} = min(h_t + h_{pool_{n-1}}, n_{out}) \tag{7.3}$$

To generate one output block of the Jitter RNG, 256 time stamps are obtained and inserted into the entropy pool via the hash operation as outlined with equation 3.3. By inserting a new time delta into the entropy pool, the entropy content is increased by adding the entropy content of one time delta to the entropy pool, capped by the message digest size. By defining the initial entropy in the entropy pool to be zero, i.e. $h_{pool_0} = 0$, as specified above, the entropy content of the entropy pool at state k is therefore:

$$h_{pool_k} = min(\sum_{n=1}^{256} h_{t_n}, n_{out}) \tag{7.4}$$

The result of the formulas show that the entropy in the Jitter RNG output block is simply a sum of the entropy of all processed time delta values capped to the message digest size of the used hash operation.

In case h_t is determined to have a value lower to 1 (bit of entropy), the CPU Jitter RNG needs to be instructed to collect respective more time deltas to generate one 256 bit output block. This can be requested with the oversampling factor using the API `jent_entropy_collector_alloc`.

The concatenation of the individual entropy pool states k to form the requested amount of random numbers applied with equation 3.4 allows the addition of the entropy content provided with each individual entropy pool state k. The truncation of a CPU Jitter RNG output block to satisfy non-aligned requests for random numbers simply reduces the amount of entropy in the truncated block to its data size.

7.2.7 SP800-90B Section 3.1.6

The Jitter RNG uses one basic noise source: the timing variances over computation operations and memory accesses. Thus, the requirements in this section are trivially met.

The discussions above may refer to the hash operation and memory access noise source independently. However, in terms of SP800-90B the noise source is the one measurement of the execution time of a set of instructions. This set of instructions is separated into the hash component and the memory access component. The execution time of both components is measured in one step causing these two components to operate as one noise source in terms of SP800-90B.

7.2.8 SP800-90B Section 3.2.1 Requirement 1

This entire document is intended to provide the required analysis.

7.2.9 SP800-90B Section 3.2.1 Requirement 2

This entire document in general and chapter 7 in particular is intended to provide the required analysis.

7.2.10 SP800-90B Section 3.2.1 Requirement 3

There is no specific operating condition other than what is needed for the operating system to run since the noise source is a complete software-based noise source.

The only dependency the noise source has is a high-resolution timer either provided by the underlying hardware or maintained by the Jitter RNG internally which does not change depending on the environmental conditions.

7.2.11 SP800-90B Section 3.2.1 Requirement 4

This document explains the architectural security boundary.

The boundary of the implementation is the source code files provided as part of the software delivery. This source code contains API calls which are to be used by entities using the Jitter RNG.

7.2.12 SP800-90B Section 3.2.1 Requirement 5

The output of SHA-3 256 is the output of the Jitter RNG. I.e. the entropy pool maintained by the hash holds the data that is given to the caller when requesting a random number.

The noise source output without the hash operation is accessed with specific tools which add interfaces that are not present and thus not usable when employing the Jitter RNG in production mode. These additional interfaces are used for gathering the data used for the analysis documented in section 7.2.3. These interfaces perform the following operation:

1. Generate a time stamp.

2. Invoke the memory access operation

3. Invoke the hash operation

4. Generate a time stamp

5. Calculate the time delta using the two time stamps

These operations are used in the regular Jitter RNG operation as well. Additional operations like health tests and others are not performed as part of testing. Therefore, the testing interface invoke the heart of the Jitter RNG which delivers entropy.

7.2.13   SP800-90B Section 3.2.1 Requirement 6

The test tools generating the raw entropy for assessment documented in section 7.2.3 can and shall be executed on the same environment that executes the assessed Jitter RNG. Thus, the raw entropy gathering uses the same operational conditions also used for the Jitter RNG.

The Jitter RNG measures the execution time of certain operations. It rests on the fact that all operations exhibit some form of execution time jitter. That means that not only the heart of the Jitter RNG – the SHA-3 and the memory access operations – but also the auxiliary operations like the health tests and the processing of the user request add to the execution time jitter. As the raw entropy collection solely measures the hash, memory access operations and the health tests, the measured entropy is always lower than what the Jitter RNG really exhibits. Therefore, the raw entropy measurements applies a worst case scenario.

7.2.14   SP800-90B Section 3.2.1 Requirement 7

See section 7.2.4 for a description of the restart test.

7.2.15   SP800-90B Section 3.2.2 Requirement 1

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

7.2.16   SP800-90B Section 3.2.2 Requirement 2

The noise source is based on the duration of the execution time of the conditioning operation. The conditioning operation is pure software and thus does not change during the runtime of the Jitter RNG. The entropy rate only is a function of the duration of the execution time and the timer and does not depend on any other system property such as physical characteristics (e.g. temperature variations or voltage/current variations). This finding is consistent with the fact that the noise source is a pure software-based noise source which relies on the presence of a high-resolution timer. Note, the used timer is a cycle counter that increments with a given rate.

7.2.17   SP800-90B Section 3.2.2 Requirement 3

See section 7.2.6 for a discussion of the entropy provided by the Jitter RNG.
A stochastic model is not provided.

7.2.18   SP800-90B Section 3.2.2 Requirement 4

The noise source is expected to execute in the address space of the process consuming the random data generated by the Jitter RNG. This implies that the operating system process isolation and memory separation guarantees that adversaries cannot gain knowledge about the Jitter RNG operation.

7.2.19   SP800-90B Section 3.2.2 Requirement 5

The output of the noise source is non-IID as it rests on the execution time of a fixed set of CPU operations and instructions.

7.2.20 SP800-90B Section 3.2.2 Requirement 6

The raw noise generates time deltas with 256 bits size.

7.2.21 SP800-90B Section 3.2.2 Requirement 7

N/A as no additional noise source is implemented with the Jitter RNG.

7.2.22 SP800-90B Section 3.2.3 Requirement 1

The conditioning component is SHA-3 256. See section 7.2.6 for a discussion of the input and output sizes.

7.2.23 SP800-90B Section 3.2.3 Requirement 2

The Jitter RNG uses SHA-3 256 as conditioning component. This implementation can be tested with the ACVP Parser which contains a backend to cover the Jitter RNG SHA-3 256 implementation.

7.2.24 SP800-90B Section 3.2.3 Requirement 3

N/A – the hash does not use any keys.

7.2.25 SP800-90B Section 3.2.3 Requirement 4

N/A – the hash does not use any keys.

7.2.26 SP800-90B Section 3.2.3 Requirement 5

N/A - the Jitter RNG uses a vetted conditioning component only.

7.2.27 SP800-90B Section 3.2.4 Requirement 1

Test tools for measuring raw entropy are provided at the Jitter RNG web page. These tools can be used by everybody without further knowledge of the Jitter RNG.

7.2.28 SP800-90B Section 3.2.4 Requirement 2

The operation of the test tools for gathering raw data are discussed in section 7.2.3. This explanation shows that the raw unconditioned data is obtained.

7.2.29 SP800-90B Section 3.2.4 Requirement 3

The provided tools for gathering raw entropy contains exact steps how to perform the tests. These steps do not require any knowledge of the noise source.

7.2.30 SP800-90B Section 3.2.4 Requirement 4

The raw entropy tools can be executed on the same environment that hosts the Jitter RNG. Thus, the data is generated under normal operating conditions.

7.2.31 SP800-90B Section 3.2.4 Requirement 5

The raw entropy tools can be executed on the same environment that hosts the Jitter RNG. Thus, the data is generated on the same hardware and operating system that executes the Jitter RNG.

7.2.32 SP800-90B Section 3.2.4 Requirement 6

The test tools are publicly available at Jitter RNG web page allowing the replication of any raw entropy measurements.

7.2.33 SP800-90B Section 3.2.4 Requirement 7

The test invokes the raw noise components of the hash and the memory access in a tight loop to develop a worst case scenario. The regular Jitter RNG operation adds additional entropy by the processing of the hash and memory access results. Therefore, the test is considered to show the lower boundary of the entropy measurements.

7.2.34 SP800-90B Section 4.3 Requirement 1

The implemented health tests comply with SP800-90B sections 4.4 as described in section 7.2.43.

7.2.35 SP800-90B Section 4.3 Requirement 2

When either health test fails, the API call to generate random numbers `jent_read_entropy(3)` informs the caller about the failure with error codes.

Both health test failures are considered permanent failures. If one is triggered, the current instance of the Jitter RNG will always remain in error state. The documentation of the API call `jent_read_entropy(3)` explains that the caller can only clear this error state by deallocating the Jitter RNG instance followed by an allocation of a new Jitter RNG instance to reset the noise source.

When a health test failure occurs, the Jitter RNG block causing the failure is not returned to the caller.

7.2.36 SP800-90B Section 4.3 Requirement 3

The following false positive probability rates are applied:

- RCT: The false positive rate is $\alpha = 2^{-30}$ and therefore complies with the recommended false positive probability.

- APT: The cut-off value is set to 325 compliant to SP800-90B section 4.4.2 for non-binary data at a significance level of $\alpha = 2^{-30}$ with time stamp is assumed to at least provide one bit of entropy, i.e. $H = 1$[11].

---

[11]Note, the referenced Excel function seems to be imprecise when calculating the value. The data has been obtained using R-Project with the formula of $1 + qbinom(1 - 2^{-30}, 512, 2^{-1})$.

7.2.37 SP800-90B Section 4.3 Requirement 4

The Jitter RNG applies a startup health test of 1,024 noise source samples. Additional tests are applied. The collected noise source samples are not re-used for the generation of random numbers.

7.2.38 SP800-90B Section 4.3 Requirement 5

The noise source supports on-demand testing in the sense that the caller is allowed to deallocate and reallocate a new Jitter RNG handle. During the reallocation, the startup health tests are re-executed.

7.2.39 SP800-90B Section 4.3 Requirement 6

The health tests are applied to the raw, unconditioned time delta data directly obtained from the noise source before they are processed with the hash conditioning component.

7.2.40 SP800-90B Section 4.3 Requirement 7

The health tests are documented with section 3.4.
   The tests are executed as follows:

- During startup, the RCT and the APT are applied to 1,024 samples. The startup test can be triggered again when the caller allocates a new Jitter RNG handle.

- At runtime, the RCT is applied to each received time delta. The APT collects the data from 512 samples. The APT health test is calculated once all time deltas are recorded. The passing results of both tests shall be confirmed before the generated Jitter RNG block is returned to the caller.

7.2.41 SP800-90B Section 4.3 Requirement 8

There are no currently known suspected noise source failure modes.

7.2.42 SP800-90B Section 4.3 Requirement 9

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

7.2.43 SP800-90B Sections 4.4 and 4.5

The health tests described in section 3.4 are applicable to cover the requirements of SP800-90B health tests.
   The SP800-90B compliant health tests are implemented with the following rationale:

RCT The Repetition Count Test implemented by the Jitter RNG compares two back-to-back time deltas to verify that they are not identical. If the number of identical back-to-back time deltas reaches the cut-off value of 30, the RCT test raises a failure that is reported to the caller mandating

the caller to reset the Jitter RNG. The RCT uses the a cut-off value that is based on the following: $\alpha = 2^{-30}$ compliant to FIPS 140-2 IG 9.8 and compliant to SP800-90B which mandates this value to be in the range $2^{-20} \leq \alpha \leq 2^{-40}$. In addition, one time delta is assumed to at least provide one bit of entropy, i.e. $H = 1$. When applying these values to the formula given in SP800-90B section 4.4.1, the cut-off value of 31 is calculated.

When the RCT passes, the counter is set to zero for the next time delta to arrive. In mathematical terms, the verification of back-to-back values being not identical is the calculation of the first discrete derivative of the time deltas (or second discrete derivative of time stamps) to show that it is not zero. In addition, the Jitter RNG enhances the RCT by calculating also the first and third discrete derivative of the time stamp to be injected into the entropy pool by the SHA-3. With that, up to 8 consecutive time stamp values are assessed. All derivatives must always be non-zero in order to pass the RCT. If one discrete derivative shows a zero, the RCT counter is increased. Thus, the addition of the first and third derivative of the time stamp makes the RCT even more conservative. Hence, the first discrete derivative is considered to be identical to the "approved" RCT specified in SP800-90B section 4.4. In addition, linear and exponential patterns are identified with the first and third discrete derivative, respectively. As the additional pattern recognition do not invalidate the mandatory pattern recognition, this RCT approach therefore is considered to be an enhanced version of the "approved" RCT and thus meets the requirement (a) of SP800-90B section 4.5.

APT The Jitter RNG implements the Adaptive Proportion Test as defined in SP800-90B section 4.4.2. As explained in other parts of the document, one time delta value is assumed to have (at least) one bit of entropy. Thus, the cut-off value for the APT is 325 compliant to SP800-90B section 4.4.2 for non-binary data at a significance level of $\alpha = 2^{-30}$. The APT is calculated using the four least significant bits of the time delta. During initialization of the APT, a time delta is set as a base. All subsequent time deltas are compared to the base time delta. If both values are identical, the APT counter is increased by one. The window size for the APT is 512 time deltas. The implementation therefore provides an "approved" APT.

7.3 NIST Clarification Requests

In addition to complying with the requirements of FIPS 140-2 and SP800-90B, NIST requests the clarification of the following questions.

7.3.1 Sensitivity of Jitter Measurements

The question that needs to be answered is whether the logic that measures the Jitter is sensitive enough to pick up the Jitter phenomenon exhibited by the CPU.

Section 2.2 explains that on contemporary CPUs, the time stamps have a very high resolution. This resolution is so high that variances appear when simply taking two times after each other and compare the delta. This implies

that the resolution of the time stamp of contemporary CPUs measures the execution time jitter phenomenon already without any additional instructions in-between. Thus, the time stamp is sensitive enough to pick up the execution time jitter.

The Jitter RNG is intended to be usable on different platforms. To ensure that the particular platform has a time stamp mechanism that is sensitive enough for picking up the execution time jitter, the start-up health tests of the Jitter RNG collection 1,024 time deltas from the noise source operation. If only one of these time deltas would show a zero value – i.e. the time stamp mechanism of the CPU is to coarse to pick up the execution time Jitter – the start-up health test will fail and an error is returned to the caller.

7.3.2 Dependency Between Jitter Measurements

Another question that is raised by NIST asks for a rationale why there are no dependencies between individual Jitter measurements.

Sections 6.1 and 6.2 provide a dissection of the noise source. Various analyses are provided demonstrating that the execution time jitter is present in different circumstances. In addition, by identifying a case where the execution time jitter can be eliminated for one instruction, hints to the true source of the execution time jitter are given: The complexity of contemporary CPUs require the introduction of wait states between the CPU components to implement a CPU instruction. Depending on the CPU state of the CPU with many different components requiring such synchronization using wait states, the wait states introduce an uncertainty of the execution time of one particular instruction. These uncertainties are measured and picked up by the Jitter RNG. This uncertainty of the number of wait states is a function of the complexity of contemporary CPUs and do not show any dependencies of the execution time of successive instructions.

Dependencies between Jitter measurements imply that some form of patterns should be detectable.

Section 5.1.1 provides several different analyses on the noise data. One of the applied analysis is a Fast-Fourier-Transformation of the raw noise data. An FFT allows to detect patterns and dependencies between individual raw noise samples. The result of the FFT shows that no patterns can be detected which supports the conclusion that no dependencies are present.

In addition, a tool is developed to execute the Jitter RNG on bare metal as documented in section 6.3. This tool boots without an operating system and does not have device drivers that would interrupt the Jitter RNG operation. Yet, execution time Jitter is measured with this tool. This implies that the execution time jitter is not a function of the operating system, but rests solely in the CPU and its memory access, i.e. another form of dependencies between Jitter measurements can be disregarded.

7.4 Reuse of SP800-90B Analysis

The SP800-90B compliance of the Jitter RNG was reviewed by NIST where all received comments were addressed. Though, an official approval is only given when the Jitter RNG is used as part of a real FIPS 140-3 validation. In order to apply the Jitter RNG to a particular environment and to claim that this Jitter

RNG usage satisfies all SP800-90B requirements, the following steps must be performed:

1. Obtain raw noise data when executing the Jitter RNG on the intended target platform as explained in section 7.2.3. The obtained raw noise data must be processed by the SP800-90B tool to obtain an entropy rate which must be above 0.333 bit of entropy per time delta.

2. Obtain the restart noise data when executing the Jitter RNG on the intended target platform as explained in section 7.2.4. The obtained raw noise data must be processed by the SP800-90B tool to verify:

   (a) the sanity test to apply to the noise restart data must pass, and

   (b) the minimum of the row-wise and column-wise entropy rate must not be less than half of the entropy rate from measurement (1) and the entropy assessment of the noise source based on the restart data must be at least 0.333 bit of entropy per time delta.

If these steps are successfully mastered the user would now satisfy all SP800-90B criteria and thus does not need to prepare his own SP800-90B analysis since the document we discuss here covers all other aspects of the SP800-90B analysis.

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

7.5 SP800-90C Considerations

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

In general, SP800-90C defines the construction method to join entropy sources, like the Jitter RNG, with deterministic random number generators like an SP800-90A DRBG. This means that SP800-90C does not really apply to the entropy sources at all. Yet, one requirement is present in SP800-90C that must be covered by the Jitter RNG: the oversampling of 64 additional bits of entropy to claim full entropy output of the SHA3-256 conditioning component. This oversampling is achieved when the FIPS mode is active. The specification of SP800-90C defines construction methods to design non- deterministic as well as deterministic RNGs. As the specification is currently in draft form, the latest available draft from January 21, 2021 is applied. This means that with FIPS mode active, the Jitter RNG obtains the following number of time stamps:

$$Count_T = (SHA3 - 256_{digest\,size} + 64) \cdot OSR$$

For an oversampling rate of 3, the Jitter RNG collects

$$Count_T = (256 + 64) \cdot 3 = 960$$

time stamps in FIPS mode for generating one output block.

When the FIPS mode is not active and SP800-90C compliance is not sought, the Jitter RNG collects

$$Count_T = 256 \cdot 3 = 768$$

time stamps for generating one output block.

8 Conclusion

For the conclusion, we need to get back to chapter 1 and consider the initial goals we have set out.

First, let us have a look at the general statistical and entropy requirements. Chapter 4 concludes that the statistical properties of the random number bit stream generated by the CPU Jitter random number generator meets all expectations. Chapter 5 explains the entropy behavior and concludes that the collected entropy by the CPU execution time jitter is much larger than the entropy pool. In addition, that section determines that the way data is mixed into the entropy pool does not diminish the gathered entropy. Therefore, this chapter concludes that one bit of output of the CPU Jitter random number generator holds one bit of information theoretical entropy.

It is noteworthy that the distribution of the sole time deltas without any processing in kernel space was measured to be spaced in increments of three in figure 2.2. This property, however, did not show any impact on the distribution of time deltas resulting from the processing of the CPU Jitter random number generator, particularly the root entropy discussed in section 5.1.

In addition to these general goals, chapter 1 lists a number of special goals. These goals are covered in the following list where the list number equals to the list number in chapter 1.

1. On demand operation is ensured by the fact that the entropy collection loop is triggered when the caller requests data. Multiple loops are initiated sequentially if the requested bit string is larger than 256 bits.

2. When compiled as a user space application, the CPU Jitter random number generator returns roughly 10 kBytes per second on an Intel Core i7 2nd generation with 2.7 GHz. In kernel space, the speed is roughly the same on the same test system. An Intel Atom Z530 system with 1.6 GHz produces output at about 2kBytes per second.

3. The design description explains that the CPU Jitter random number generator always returns entropy. The entropy is generated when the request for it is received.

4. In virtualized environments, the fundamental property of CPU execution time jitter is still present. Moreover, reading high-resolution timing information from the CPU is typically allowed by the virtualization environment and is not subject to virtualization itself[12]. Thus, a virtual environment can execute the CPU Jitter random number generator and deliver equal entropy. Due to the additional processing that typically is present to support reading the CPU time stamp, the CPU execution time jitter is expected to be higher and thus supportive to the CPU Jitter random number generator.

5. The design description explains that the CPU Jitter random number generator always generates fresh entropy.

---

[12]Although the discussion of virtualizing the CPU time stamp – e.g. the `RDTSC` x86 processor instruction – is conducted once in a while, no attempts to implement such virtualization has been made. The goal of such virtualization would only be to hide the existence of a hypervisor to a guest. But current virtualization environments do not pursue that goal.

6. The reference implementation work in both kernel and user space.

7. The CPU Jitter random number generator requires access to either a hardware high resolution timer or the POSIX threading support to implement a software-based timer. In Linux, that is granted to unprivileged processes. Therefore, every process that is in need of entropy can instantiate its own copy of the CPU Jitter random number generator.

8. The CPU Jitter random number generator generates a fresh 256 bit random number for each request. Even though it reuses the contents of the entropy pool that was used for previous random numbers, it makes no assumption whether entropy is present or not. Moreover, the data fed into the entropy pool is not deterministic. Thus, perfect forward and backward secrecy is considered to be maintained. The question is: How do we interpret the case when an observer gets access to the entropy pool when the first few iterations of one entropy collection loop operation are performed. Let us assume that an observer accesses the entropy pool after the first iteration of the loop completes. That iteration added some but very little entropy. The observer can "brute-force" the previous random number[13] by guessing the few added numbers. Therefore, how do we interpret forward and backward secrecy here? The definition we apply is the following: Perfect forward and backward secrecy is ensured when an observer gains access to one random number generated by the CPU Jitter random number generator. When an observer gains access to the entropy pool while a random number is generated, the perfect forward and backward secrecy applies to all random numbers before the last generated random number and after the currently generated random number, excluding the last and currently generated random number. To make sure that the last generated random number is not lingering in memory for too long, the code invokes the entropy collection loop one more time after completing the calling application's request which changes the entropy pool such that in an event an observer can get access to the entropy pool, he gains nothing – this code is executed if the environment where the CPU Jitter random number generator is embedded into does not implement a secure memory handling.

8.1 Threat Scenario

After explaining the functionality of the CPU Jitter random number generator, the statistical properties and the assessment of the entropy, let us try to attack the mechanism. When attacking, we have to first determine our goals. The following list enumerates the goals:

- Direct readout of random number or the internal state of the CPU Jitter random number generator: This approach can be immediately refuted as the random number generator relies on the process separation and memory isolation offered by contemporary operating systems.

- Interleaving with the time stamp collection of the victim process.

---

[13]The same applies for the new random number when the observer would access the entropy pool in the last few iterations before completion of entropy collection loop.

8.1.1 Interleaving of Time Stamp Collection

The interleaving attack is illustrated with figure 8.1.

Jitter Collector Process	Victim Process	Attacker Process	Entropy Source
Time Stamp	Time Stamp		
Time Stamp		Time Stamp	Jitter
Time Stamp	Time Stamp		Jitter
Time Stamp		Time Stamp	Jitter
Time Stamp	Time Stamp		Jitter
Time Stamp		Time Stamp	Jitter
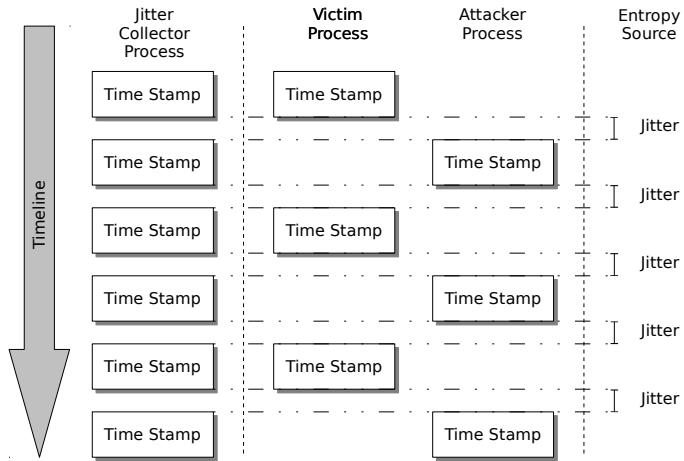
Figure 8.1: Attack process interleaving with victim process

The left process marked as "Jitter Collector Process" would be a process implementing the CPU Jitter random number generator and would run unobserved – i.e. the case when no attack is staged.

Now we can conceive a scenario where the victim process executing the entropy collection loop to gather entropy. An attacker process tries somehow to gain knowledge about the time stamps obtained by the victim process during the entropy collection loop. The attacker process may try to read also the time stamps of the system.

The worst case would be, if the attacker would be able to stage his time stamp readings such that he interleaves one-to-one with the victim process time stamp collection on the same CPU core. When executing the attacker on another CPU core, the interleaving mechanism would not work. That means that the very next time stamp gathering operation that is technically possible after the victim gathered one time stamp is read by the attacker. Then, the next time stamp is again read by the victim, and so forth. Figure 8.1 illustrates the time stamp collection of the victim and attacker in the middle part.

Now, the attacker tries to deduct the victim's time stamps from his time stamp readings. To a large degree, he is able to determine them. But the miniscule variations between adjacent time stamp readings is the source of entropy for the CPU Jitter random number generator – marked as the time span between the time stamp read operations in figure 8.1.

Comparing the attack process readings with a fully unobserved process indicates that the attacking process can never determine the victim's time stamps more accurate than the CPU execution time jitter our random number generator is based on. An attacking process is never be able to reduce the variations of the CPU execution time jitter.

A Availability of Source Code

The source code of the CPU Jitter entropy random number generator including the documentation is available at http://www.chronox.de/jent/jitterentropy-current.tar.bz2.

The source code for the test cases and R-project files to generate the graphs is available at the same web site.

B Shared Library And Stand-Alone Daemon

The CPU Jitter random number generator can be compiled as a stand-alone shared library using the `Makefile.shared` makefile. The shared library exports the interfaces outlined in `jitterentropy(3)`. After compilation, link with the shared library using the linker option `-ljitterentropy`.

To update the entropy in the `input_pool` behind the Linux `/dev/random` and `/dev/urandom` devices, the daemon `jitterentropy-rngd` is implemented. It polls on `/dev/random`. The kernel wakes up polling processes when the entropy counter falls below a threshold. In this case, the `jitterentropy-rngd` gathers 256 bytes of entropy and injects it into the `input_pool`. In addition, `/proc/sys/kernel/random/entropy_avail` is read in 5 second steps. If the value falls below 1024, `jitterentropy-rngd` gathers 256 bytes of entropy and injects it into the `input_pool`. The reason for polling `entropy_avail` is the fact that when random numbers are extracted from `/dev/urandom`, the poll on `/dev/random` is not triggered when the entropy estimator falls.

C ~~LFSR~~Hash Loop Entropy Measurements

NOTE: This section applies to the measurement using the Jitter RNG version 2 using an LSFR as the static set of CPU instructions executing time is measured. Yet, the results equally applies to the hash conditioning operation as the execution jitter is equally visible.

The following sections show the measurements explained in section 5.1 for different CPUs. These measurements all support the conclusion in section 5.1.

Note, all measurements in this sections *only* cover the CPU execution time jitter by *disabling* memory accesses. By showing that CPU execution time jitter is already sufficient, the additional measurement of of memory accesses will not change the sufficiency of the timing variations.

Note, all these tests were executed in user space. Although the compilation of the CPU Jitter random number generator will always be performed without optimizations, tests are executed with and without optimizations. Testing with optimizations considers again the worst case. If testing with optimizations shows too little entropy, the test is repeated without optimizations.

A large number of tests on different CPUs with different operating systems were executed. The following table summarizes the tests by enumerating the upper and lower boundary of the Shannon Entropy for all test systems. The table lists:

- the CPU,

- the word size (WS) of the software,

- whether the code compiled optimized with `-O2` or not,

- the upper Shannon Entropy boundary,

- the lower Shannon Entropy boundary,

- the operating system, and

- whether the `jent_entropy_init` function would accept the CPU.

The table demonstrates that the CPU Jitter random number generator delivers high-quality entropy on:

- a large range of CPUs ranging from embedded systems of MIPS and ARM CPUs, covering desktop systems with AMD, Intel and VIA x86 32 bit and 64 bit CPUs as well as Apple PowerPC, up to server CPUs of Intel Itanium, Sparc, POWER and IBM System Z;

- a large range of operating systems: Linux (different distributions and kernel versions), OpenBSD, FreeBSD, NetBSD, AIX, OpenIndiana (OpenSolaris), z/OS, Apple MacOS, Android, Windows, and microkernel based operating systems (Genode with microkernels of NOVA, Fiasco.OC, Pistachio);

- a range of different compilers: GCC, Clang, Microsoft Visual Studio, and the z/OS C compiler.

CPU	WS	Opt	Upper	Lower	OS	Acc
Intel(R) Xeon(R) CPU X5660 @ 2.80GHz	64	y	6.01	2.78	Linux	y
Intel(R) Xeon(R) CPU E5620 @ 2.40GHz	64	y	5.99	2.67	Linux	y
Intel(R) Xeon(R) CPU E5-2470 0 @ 2.30GHz	64	y	7.09	2.58	Linux	y
AMD Generic S	64	y	3.75	1.98	Linux	y
Intel(R) Xeon(R) CPU E5-4650 0 @ 2.70GHz	64	y	6.91	2.42	Linux	y
Intel(R) Atom(TM) CPU S1240 @ 1.60GHz	64	y	6.59	2.10	Linux	y
Intel(R) Core(TM)2 Quad CPU @ 2.66GHz	64	y	6.00	2.80	Linux	y
Intel(R) Xeon(R) CPU X3470 @ 2.93GHz	64	y	5.36	1.74	Linux	y
Intel(R) Xeon(R) CPU X5650 @ 2.67GHz	64	y	6.78	3.33	Linux	y
Intel(R) Xeon(R) CPU X5472 @ 3.00GHz	64	y	5.91	2.90	Linux	y
Intel(R) Xeon(R) CPU 5150 @ 2.66GHz	64	y	7.93	4.41	Linux	y
Dual-Core AMD Opteron(tm) Processor 2218	64	y	5.19	1.95	Linux	y

CPU	WS	Opt	Upper	Lower	OS	Acc
Quad-Core AMD Opteron(tm) Processor 2356	64	y	6.62	2.65	Linux	y
Intel(R) Pentium(R) CPU 1403 @ 2.60GHz	64	y	7.21	1.89	Linux	y
Genuine Intel(R) CPU @ 2.83GHz	64	y	6.86	2.66	Linux	y
Intel(R) Xeon(R) CPU X5550 @ 2.67GHz	64	y	6.79	3.13	Linux	y
Intel(R) Xeon(R) CPU X5660 @ 2.80GHz	64	y	9.25	5.46	Linux	y
Intel(R) Xeon(R) CPU E7520 @ 1.87GHz	64	y	6.87	2.57	Linux	y
Genuine Intel(R) CPU @ 2.93GHz	64	y	5.94	3.09	Linux	y
Dual Core AMD Opteron(tm) Processor 890	64	y	5.04	1.58	Linux	y
Dual-Core AMD Opteron(tm) Processor 8218	64	y	5.54	1.94	Linux	y
Intel(R) Xeon(R) CPU E5540 @ 2.53GHz	64	y	6.97	3.42	Linux	y
Dual Core AMD Opteron(tm) Processor 870	64	y	5.50	2.05	Linux	y
Intel(R) Xeon(R) CPU L5520 @ 2.27GHz	64	y	7.51	3.49	Linux	y
QEMU Virtual CPU version 0.15.1	64	y	10.26	6.97	Linux	y
Dual Core AMD Opteron(tm) Processor 280	64	y	5.11	2.25	Linux	y
Genuine Intel(R) CPU 000 @ 2.40GHz	64	y	6.40	3.08	Linux	y
Intel(R) Core(TM) i5-4570 CPU @ 3.20GHz	64	y	6.66	2.09	Linux	y
Intel(R) Xeon(R) CPU E5504 @ 2.00GHz	64	y	5.12	1.36	Linux	y
Intel(R) Xeon(R) CPU E5-4650 0 @ 2.70GHz	64	y	7.09	2.88	Linux	y
Intel(R) Xeon(R) CPU X5670 @ 2.93GHz	64	y	6.80	3.06	Linux	y
AMD Opteron(tm) Processor 6172	64	y	5.68	2.30	Linux	y
Dual Core AMD Opteron(tm) Processor 890	64	y	5.25	2.42	Linux	y
Intel(R) Xeon(R) CPU X5376 @ 2.80GHz	64	y	6.14	3.16	Linux	y
Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz	64	y	6.01	2.25	Linux	y

CPU	WS	Opt	Upper	Lower	OS	Acc
Intel(R) Xeon(R) CPU E5-4650 0 @ 2.70GHz	64	y	7.37	2.89	Linux	y
Intel(R) Xeon(R) CPU X5376 @ 2.80GHz	64	y	6.62	3.59	Linux	y
Intel(R) Core(TM) i7 CPU X 980 @ 3.33GHz	64	y	6.55	2.31	Linux	y
Intel(R) Xeon(R) CPU L5520 @ 2.27GHz	64	y	6.59	2.74	Linux	y
Intel(R) Xeon(R) CPU X5680 @ 3.33GHz	64	y	6.29	2.60	Linux	y
Intel(R) Xeon(TM) CPU 3.40GHz	64	y	6.15	2.20	Linux	y
Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz	64	y	7.30	2.73	Linux	y
AMD Opteron(tm) Processor 6386 SE	64	y	6.35	2.23	Linux	y
Genuine Intel(R) CPU 000 @ 2.27GHz	64	y	5.92	3.20	Linux	y
AMD Opteron(tm) Processor 6172	64	y	6.60	2.89	Linux	y
Intel(R) Xeon(R) CPU X5472 @ 3.00GHz	64	y	6.09	2.90	Linux	y
AMD Generic S	64	y	7.05	2.87	Linux	y
Intel(R) Xeon(R) CPU E5-4650 0 @ 2.70GHz	64	y	6.88	2.52	Linux	y
Genuine Intel(R) CPU @ 2.40GHz	64	y	4.98	1.54	Linux	y
AMD Generic S	64	y	7.50	2.48	Linux	y
Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz	64	y	6.92	3.02	Linux	y
Genuine Intel(R) CPU @ 2.66GHz	64	y	5.38	2.96	Linux	y
Intel(R) Xeon(TM) CPU 3.73GHz	64	y	5.24	1.96	Linux	y
Genuine Intel(R) CPU @ 2.80GHz	64	y	7.31	2.65	Linux	y
Intel(R) Xeon(R) CPU X7560 @ 2.27GHz	64	y	6.78	3.24	Linux	y
AMD Generic S	64	y	3.65	0.60	Linux	y
AMD Generic S	64	n	3.74	1.62	Linux	y
Intel(R) Xeon(R) CPU W3530 @ 2.80GHz	64	y	6.40	2.28	Linux	y
Intel(R) Xeon(R) CPU X5660 @ 2.80GHz	32	y	9.35	4.16	Linux	y
Intel(R) Xeon(R) CPU E5620 @ 2.40GHz	32	y	9.19	3.91	Linux	y

CPU	WS	Opt	Upper	Lower	OS	Acc
Intel(R) Xeon(R) CPU E5-2470 0 @ 2.30GHz	32	y	9.70	4.41	Linux	y
Intel(R) Xeon(R) CPU E5-4650 0 @ 2.70GHz	32	y	9.62	4.05	Linux	y
Intel(R) Atom(TM) CPU S1240 @ 1.60GHz	32	y	8.27	3.93	Linux	y
Intel(R) Core(TM)2 Quad CPU @ 2.66GHz	32	y	7.11	3.18	Linux	y
Intel(R) Xeon(R) CPU X3470 @ 2.93GHz	32	y	8.64	3.81	Linux	y
Intel(R) Xeon(R) CPU X5650 @ 2.67GHz	32	y	9.14	4.68	Linux	y
Intel(R) Xeon(R) CPU X5472 @ 3.00GHz	32	y	8.13	3.46	Linux	y
Genuine Intel(R) CPU @ 2.83GHz	32	y	8.97	3.34	Linux	y
Intel(R) Xeon(R) CPU X5550 @ 2.67GHz	32	y	9.54	4.10	Linux	y
Intel(R) Xeon(R) CPU X5660 @ 2.80GHz	32	y	11.65	7.66	Linux	y
Intel(R) Xeon(R) CPU E7520 @ 1.87GHz	32	y	8.34	3.28	Linux	y
Genuine Intel(R) CPU @ 2.93GHz	32	y	7.93	3.29	Linux	y
Intel(R) Xeon(R) CPU E5345 @ 2.33GHz	32	y	7.33	3.14	Linux	y
Intel(R) Xeon(R) CPU E5540 @ 2.53GHz	32	y	9.19	4.14	Linux	y
QEMU Virtual CPU version 0.15.1	32	y	11.32	7.40	Linux	y
Genuine Intel(R) CPU 000 @ 2.40GHz	32	y	10.21	5.19	Linux	y
Intel(R) Xeon(R) CPU E5504 @ 2.00GHz	32	y	8.85	3.34	Linux	y
Intel(R) Core(TM) i5-4570 CPU @ 3.20GHz	32	y	8.34	2.64	Linux	y
Intel(R) Xeon(R) CPU E5504 @ 2.00GHz	32	y	8.23	3.42	Linux	y
Intel(R) Xeon(R) CPU X5670 @ 2.93GHz	32	y	9.78	5.03	Linux	y
Intel(R) Xeon(R) CPU X5472 @ 3.00GHz	32	y	8.55	4.30	Linux	y
Intel(R) Xeon(R) CPU X5376 @ 2.80GHz	32	y	7.73	3.42	Linux	y
Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz	32	y	9.14	3.76	Linux	y

CPU	WS	Opt	Upper	Lower	OS	Acc
Intel(R) Xeon(R) CPU E5-4650 0 @ 2.70GHz	32	y	9.92	4.51	Linux	y
Intel(R) Xeon(R) CPU X5376 @ 2.80GHz	32	y	8.04	3.54	Linux	y
Intel(R) Core(TM) i7 CPU X 980 @ 3.33GHz	32	y	10.18	5.53	Linux	y
Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz	32	y	9.76	3.84	Linux	y
Intel(R) Xeon(R) CPU L5520 @ 2.27GHz	32	y	9.51	4.18	Linux	y
Intel(R) Xeon(TM) CPU 3.40GHz	32	y	9.05	3.99	Linux	y
Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz	32	y	10.09	4.97	Linux	y
Genuine Intel(R) CPU 000 @ 2.27GHz	32	y	9.21	4.69	Linux	y
Intel(R) Xeon(R) CPU X5472 @ 3.00GHz	32	y	9.22	4.35	Linux	y
Intel(R) Xeon(R) CPU E5-4650 0 @ 2.70GHz	32	y	9.54	4.07	Linux	y
Genuine Intel(R) CPU @ 2.40GHz	32	y	5.96	1.82	Linux	y
Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz	32	y	9.61	3.84	Linux	y
Genuine Intel(R) CPU @ 2.66GHz	32	y	5.92	1.64	Linux	y
Intel(R) Xeon(TM) CPU 3.73GHz	32	y	5.10	1.48	Linux	y
Genuine Intel(R) CPU @ 2.80GHz	32	y	9.62	4.14	Linux	y
Intel(R) Xeon(TM) CPU 3.20GHz	32	y	7.80	2.74	Linux	y
Intel(R) Xeon(R) CPU X7560 @ 2.27GHz	32	y	9.24	3.98	Linux	y
Intel(R) Xeon(R) CPU W3530 @ 2.80GHz	32	y	9.97	4.82	Linux	y
POWER7 (architected), altivec supported - CHRP IBM,8202-E4B	64	y	11.21	7.49	Linux	y
POWER7 (architected), altivec supported - CHRP IBM,8202-E4B	64	y	10.76	6.23	Linux	y
POWER7 (architected), altivec supported - CHRP IBM,8231-E2B	64	y	8.26	3.87	Linux	y
POWER5+ (gs) - CHRP IBM,9113-550	64	y	7.81	3.10	Linux	y

CPU	WS	Opt	Upper	Lower	OS	Acc
POWER5+ (gs) - CHRP IBM,9113-550	64	y	7.80	3.10	Linux	y
POWER5+ (gs) - CHRP IBM,9117-570	64	y	6.63	2.54	Linux	y
POWER7 (architected), altivec supported - CHRP IBM,8231-E2B	64	y	8.31	4.05	Linux	y
POWER7 (architected), altivec supported - CHRP IBM,8231-E2D	64	y	8.95	5.79	Linux	y
POWER7 (architected), altivec supported - CHRP IBM,8231-E2D	64	y	10.66	6.38	Linux	y
POWER7 (architected), altivec supported - CHRP IBM,8231-E2D	64	y	8.24	4.94	Linux	y
POWER5+ (gs) - CHRP IBM,9131-52A	64	y	10.99	6.68	Linux	y
POWER5 (gr) - CHRP IBM,9123-705	64	y	5.97	1.60	Linux	y
POWER7 (architected), altivec supported - CHRP IBM,8202-E4B	32	y	11.59	7.51	Linux	y
POWER7 (architected), altivec supported - CHRP IBM,8202-E4B	32	y	10.82	5.66	Linux	y
POWER7 (architected), altivec supported - CHRP IBM,8231-E2B	32	y	9.32	4.12	Linux	y
POWER5+ (gs) - CHRP IBM,9113-550	32	y	9.02	4.13	Linux	y
POWER5+ (gs) - CHRP IBM,9113-550	32	y	9.54	5.06	Linux	y
POWER5+ (gs) - CHRP IBM,9117-570	32	y	10.84	6.10	Linux	y
POWER7 (architected), altivec supported - CHRP IBM,8231-E2B	32	y	9.73	4.71	Linux	y
POWER7 (architected), altivec supported - CHRP IBM,8231-E2D	32	y	11.30	6.75	Linux	y
POWER7 (architected), altivec supported - CHRP IBM,8231-E2D	32	y	9.51	4.77	Linux	y
POWER7 (architected), altivec supported - CHRP IBM,8231-E2D	32	y	9.19	4.21	Linux	y

CPU	WS	Opt	Upper	Lower	OS	Acc
POWER5+ (gs) - CHRP IBM,9131-52A	32	y	11.91	6.89	Linux	y
POWER5 (gr) - CHRP IBM,9123-705	32	y	7.27	2.60	Linux	y
IBM/S390 - version = 00, identification = 098942, machine = 2097	31	y	5.88	1.25	Linux	y
IBM/S390 - version = 00, identification = 0A8942, machine = 2097	31	y	5.48	0.85	Linux	y
IBM/S390 - version = 00, identification = 0B8942, machine = 2097	31	y	5.53	0.87	Linux	y
IBM/S390 - version = 00, identification = 0D8942, machine = 2097	31	y	5.34	0.83	Linux	y
IBM/S390 - version = 00, identification = 038942, machine = 2097	31	y	5.73	0.91	Linux	y
IBM/S390 - version = 00, identification = 158942, machine = 2097	31	y	5.40	0.83	Linux	y
IBM/S390 - version = 00, identification = 168942, machine = 2097	31	y	5.76	1.15	Linux	y
IBM/S390 - version = 00, identification = 178942, machine = 2097	31	y	5.29	0.81	Linux	y
IBM/S390 - version = FF, identification = 058942, machine = 2097	31	n	6.40	1.77	Linux	n
IBM/S390 - version = 00, identification = 098942, machine = 2097	64	y	5.94	2.04	Linux	y
IBM/S390 - version = 00, identification = 0A8942, machine = 2097	64	y	5.74	1.98	Linux	y
IBM/S390 - version = 00, identification = 0B8942, machine = 2097	64	y	5.73	1.97	Linux	y
IBM/S390 - version = 00, identification = 0D8942, machine = 2097	64	y	5.48	1.95	Linux	y
IBM/S390 - version = 00, identification = 038942, machine = 2097	64	y	5.81	1.99	Linux	y

CPU	WS	Opt	Upper	Lower	OS	Acc
IBM/S390 - version = 00, identification = 158942, machine = 2097	64	y	5.82	1.97	Linux	y
IBM/S390 - version = 00, identification = 168942, machine = 2097	64	y	5.85	2.02	Linux	y
IBM/S390 - version = 00, identification = 178942, machine = 2097	64	y	5.64	1.96	Linux	y
Intel(r) Itanium(r) Processor Family	64	y	10.30	6.47	Linux	y
Intel(R) Itanium(R) Processor 9350	64	y	6.01	1.52	Linux	y
Intel(R) Itanium(R) Processor 9350	64	y	8.90	4.82	Linux	y
Intel(r) Itanium(r) 2 Processor 1.4GHz with 12M L3 Cache for 667MHz Platforms	64	y	5.23	1.36	Linux	y
Intel(R) Itanium(R) Processor 9350	64	y	10.41	6.95	Linux	y
Intel(R) Itanium(R) Processor 9350	64	y	6.40	1.92	Linux	y
Intel(r) Itanium(r) 2 Processor 1.6GHz with 18M L3 Cache for 533MHz Platforms	64	y	6.48	2.44	Linux	y
Intel(R) Itanium(R) Processor 9350	64	y	11.57	8.77	Linux	y
Intel(r) Itanium(r) 2 Processor 1.6GHz with 18M L3 Cache for 533MHz Platforms	64	y	8.64	4.37	Linux	y
Dual-Core Intel(R) Itanium(R) Processor 9050	64	y	11.65	5.91	Linux	y
Intel(r) Itanium(r) 2 Processor 1.6GHz with 18M L3 Cache for 533MHz Platforms	64	y	9.12	4.85	Linux	y
Intel(r) Itanium(r) 2 Processor 1.6GHz with 18M L3 Cache for 533MHz Platforms	64	y	10.53	7.82	Linux	y
Dual-Core Intel(R) Itanium(R) 2 Processor 9140M	64	y	6.07	1.90	Linux	y
Madison up to 9M cache	64	y	4.41	1.22	Linux	y

CPU	WS	Opt	Upper	Lower	OS	Acc
Intel(r) Itanium(r) 2 Processor 1.6GHz with 18M L3 Cache for 533MHz Platforms	64	y	10.44	7.55	Linux	y
Dual-Core Intel(R) Itanium(R) Processor 9040	64	y	11.08	6.44	Linux	y
Intel(r) Itanium(r) 2 Processor 1.6GHz with 24M L3 Cache for 533MHz Platforms	64	y	5.86	2.20	Linux	y
Intel(r) Itanium(r) Processor Family	64	y	12.04	8.74	Linux	y
Dual-Core Intel(R) Itanium(R) 2 Processor 9140M	64	y	6.19	2.31	Linux	y
Intel(r) Itanium(r) Processor Family	64	y	4.78	2.00	Linux	y
Madison up to 9M cache	64	y	9.11	4.35	Linux	y
Intel(r) Itanium(r) 2 Processor 1.6GHz with 24M L3 Cache for 533MHz Platforms	64	y	6.29	2.62	Linux	y
Itanium 2	64	y	4.59	1.11	Linux	y
Intel(r) Itanium(r) Processor Family	64	y	5.42	1.46	Linux	y
Intel(r) Itanium(r) 2 Processor 1.6GHz with 18M L3 Cache for 533MHz Platforms	64	y	6.01	1.69	Linux	y
Dual-Core Intel(R) Itanium(R) Processor 9040	64	y	6.03	1.96	Linux	y
AMD E350	32	y	5.95	1.56	Linux	y
AMD E350	32	n	6.30	2.75	Linux	y
AMD Phenom X6-1035T	64	y	6.67	2.90	Linux	y
AMD Phenom X6-1035T	64	n	6.55	2.93	Linux	y
AMD Semperon 3GHz	32	y	5.00	0.91	Linux	y
AMD Semperon 3GHz	32	n	5.99	2.42	Linux	y
ARMv6-rev7	32	y	5.08	1.13	Android	n
ARMv6-rev7	32	n	5.81	1.58	Android	n
ARMv7-rev1 – Samsung Galaxy SII	32	y	7.52	2.86	Android	y
ARMv7-rev1 – Samsung Galaxy SII	32	n	6.56	2.71	Android	y
ARMv7 rev 2 – LG Nexus 4.2	32	n	N/A	N/A	Android	n
ARMv7 rev 1 – HTC Desire Z	32	n	N/A	N/A	Android	n

CPU	WS	Opt	Upper	Lower	OS	Acc
AMD Athlon 4850e	64	y	3.98	2.30	Linux	y
AMD Athlon 4850e	64	n	5.06	2.75	Linux	y
AMD Athlon 7550	64	y	6.65	2.75	Linux	y
AMD Athlon 7550	64	n	6.74	3.40	Linux	y
Intel Atom Z530	32	y	5.38	2.43	Linux	y
Intel Celeron	32	y	7.97	3.33	Linux	y
Intel Core2Duo T5870	32	y	6.19	2.38	Linux	y
Intel Core2 Q6600	32	y	6.28	2.71	Linux	y
Intel Core2 Q6600	32	n	5.81	2.51	Linux	y
Intel CoreDuo L2400	32	y	6.52	2.23	Linux	y
Intel Core i5-2410M	64	y	7.25	3.75	Linux	y
Intel Core i5-2410M	64	n	9.52	3.72	Linux	y
Intel Core i5-2430M	64	y	8.10	3.55	Linux	y
Intel(R) Core(TM) i7-2620M CPU @ 2.70GHz	64	y	5.68	2.35	FreeBSD 9.1	y
Intel(R) Core(TM) i7-2620M CPU @ 2.70GHz	64	y	6.32	1.47	NetBSD 6.0	y
Intel(R) Core(TM) i7-2620M CPU @ 2.70GHz	64	y	9.40	3.86	OpenIndiana 151	y
Intel(R) Core(TM) i7-2620M CPU @ 2.70GHz	64	y	6.93	2.85	Linux no X11	y
Intel(R) Core(TM) i7-2620M CPU @ 2.70GHz	64	y	6.79	2.97	Linux with X11	y
Intel(R) Core(TM) i7-2620M CPU @ 2.70GHz	64	n	8.48	3.27	Linux with X11	y
Intel(R) Core(TM) i7-2620M CPU @ 2.70GHz	64	y	2.11	1.72	Linux with X11 compiled with Clang	y
Intel(R) Core(TM) i7-2620M CPU @ 2.70GHz	64	n	9.13	3.52	Linux with X11 compiled with Clang	y
Intel Core i7-Q720	64	y	7.84	2.98	Linux	y
MIPS 24Kc v7.4	32	y	7.01	3.31	Linux	y
MIPS 24Kc v4.12	32	y	7.54	3.95	Linux	y
MIPS 24Kc v4.12	32	n	7.78	3.47	Linux	y
MIPS 4Kec V4.8	32	n	1.83	0.46	Linux	n
MIPS 4Kec V6.8	32	y	7.21	2.28	Linux	y
MIPS 4Kec V6.8	32	n	10.60	5.02	Linux	y
Pentium Celeron Mobile 733MHz	32	y	4.57	0.70	Linux	y
Pentium Celeron Mobile 733MHz	32	n	4.38	0.49	Linux	y
AMD Opteron 6128	32	y	8.03	3.90	Linux	y
Pentium 4 3GHz	32	y	8.37	3.88	Linux	y
Pentium 4 Mobile	32	y	5.40	1.56	Linux	y

CPU	WS	Opt	Upper	Lower	OS	Acc
IBM System Z z10 (S390) on z/VM	64	y	5.55	1.99	Linux	y
IBM System Z z10 (S390) on z/VM	64	n	6.37	2.11	Linux	y
UltraSparc II	?	y	12.45	7.74	OpenBSD	y
UltraSparc II	?	n	12.05	7.32	OpenBSD	y
UltraSparc IIi 440MHz	?	y	11.58	6.62	Linux	y
UltraSparc IIi 440MHz	?	n	11.82	6.86	Linux	y
UltraSparc IIIi	?	y	11.08	7.20	FreeBSD	y
UltraSparc IIIi	?	n	10.90	6.82	FreeBSD	y
VIA Nano L2200	32	y	6.35	3.12	Linux	y
Intel Xeon E5504	64	y	5.96	1.23	Linux	y
Intel Xeon E5504	64	n	5.59	0.83	Linux	y
IBM System P POWER7 using `clock_gettime`	64	y	N/A	N/A	AIX 6.1	n
IBM System P POWER5 using `read_real_time`	64	y	7.59	2.85	AIX 6.1	y
IBM System P POWER5 using `read_real_time`	64	n	11.83	6.29	AIX 6.1	y
IBM System P POWER6 using `read_real_time`	64	y	6.71	2.86	AIX 6.1	y
IBM System P POWER6 using `read_real_time`	64	n	10.71	6.83	AIX 6.1	y
IBM System P POWER7 using `read_real_time`	64	y	7.29	3.75	AIX 6.1	y
IBM System P POWER7 using `read_real_time`	64	n	11.61	7.92	AIX 6.1	y
Apple MacBook Pro Intel Core 2 Duo	32	y	6.38	2.77	Apple MacOS 10.6	y
Apple MacBook Pro Intel Core 2 Duo	32	n	5.57	1.82	Apple MacOS 10.6	y
IBM System Z z10 using `STCKE`	64	n	9.38	5.28	z/OS 1R13	y
Intel Core Duo Solo T1300	32	n	5.13	1.32	Genode with NOVA Microkernel	y
Intel Core Duo Solo T1300	32	n	5.04	2.04	Genode with Fiasco.OC Microkernel	y
Intel Core Duo Solo T1300	23	n	5.45	2.09	Genode with Pistachio Microkernel	y
ARM Exynos 5250	32	n	6.88	2.00	Genode with Fiasco.OC	y
ARM Exynos 5250	32	n	3.21	0.00	Genode with BaseHW	n

CPU	WS	Opt	Upper	Lower	OS	Acc
AMD Athlon(tm) 64 X2 Dual Core Processor 3800+	64	y	2.00	2.00	Linux	y
AMD Athlon(tm) 64 X2 Dual Core Processor 3800+	64	n	3.88	1.47	Linux	y
AMD Phenom(tm) II X4 925 Processor	64	y	1.08	1.00	Linux	y
AMD Phenom(tm) II X4 925 Processor	64	n	5.63	2.58	Linux	y
Intel Core2 Duo	32	n	8.22	3.53	Windows 7 with Visual Studio	y
Apple PPC G5 Quad Core PPC970MP	64	y	5.27	0.94	Apple MacOS X 10.5.8	y
Apple PPC G5 Quad Core PPC970MP	64	n	9.41	2.44	Apple MacOS X 10.5.8	y
Apple PPC G5 Quad Core PPC970MP	32	y	6.22	0.96	Apple MacOS X 10.5.8	y
Apple PPC G5 Quad Core PPC970MP	32	n	6.87	2.34	Apple MacOS X 10.5.8	y
Intel(R) Core(TM) i7-3537U CPU	64	n	8.76	3.81	Linux	y

The table shows that all test systems at least without optimizations have a lower boundary of more than 1 bit. This supports the quality assessment of the CPU Jitter random number generator.

In addition, the table shows an interesting yet common trend: the newer the CPU is, the more CPU execution time jitter is present.

Nonetheless, the following exceptions are visible:

- Intel Mobile Celeron 733 MHz: The test shows that this CPU has insufficient execution jitter entropy. However, as this CPU is considered very old, the code has not been changed to catch this CPU behavior.

- IBM System Z 31 bit word size (marked as S/390 in the table above): The tests with optimized code indicates that the lower boundary has way too little entropy. However, when re-performing the tests on the same system without optimization, the lower and upper boundary again show significant improvements to values way above 1 bit. Therefore, non-optimized code is required for this system and word size which is granted for the compilation of the CPU Jitter random number generator. More details on this system is given in section C.46.

- Intel Xeon E5504: Section C.7 outlines that the test result is to be considered as an outlier. There are additional tests on a different Intel Xeon E5504 listed in the table above which show appropriate results for the

99

lower boundary of the Shannon Entropy. When enabling memory access, the timing variations increase significantly above the threshold of 1 bit.

To illustrate the tests more, a subset of the tests listed in the aforementioned table are assessed with graphs in the following subsections. The reader will find a number of tests from the table above again in the graphs below. Note, to make sure the illustrations show the worst case, the graphs truncate the outliers in the test results with a cutoff value. Thus, the values for the Shannon Entropy in the graphs below are all slightly lower than outlined in the table above. That cutoff value is chosen to focus on the values that occur the most. That usually discards higher values due to cache or TLB misses to illustrate a system without any load, supporting the analysis of a worst case scenario.

C.1 Intel Core i5 4200U

Distribution of time delta in userspace-foldtime-i5-4200U.O0 up to 310

Min: 285 - 1st Qu: 299 - Median: 300 - Mean: 300.63
3rd Qu: 303 - Max: 310 - Sigma: 2.8 - Var Coeff: 0.0093173 - Shannon Entropy: 3.18
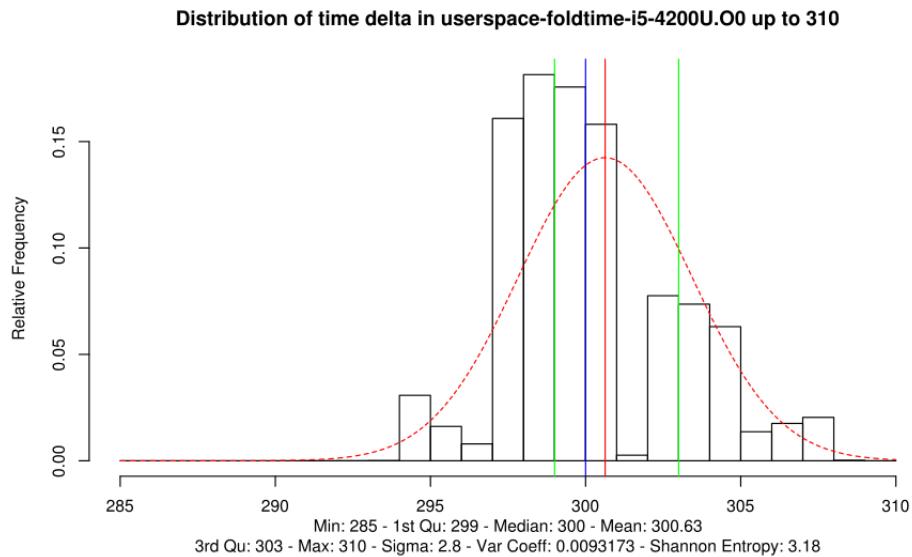
Figure C.1: ~~Lower boundary of e~~ntropy over ~~LFSR~~Hash loop in user space on Intel Core i7 3537U

100

Distribution of time delta in userspace-foldtime-i5-4200U.O0 up to 3400

Min: 287 - 1st Qu: 975 - Median: 1851 - Mean: 1831.53
3rd Qu: 2713 - Max: 3400 - Sigma: 934.99 - Var Coeff: 0.5104969 - Shannon Entropy: 9.25
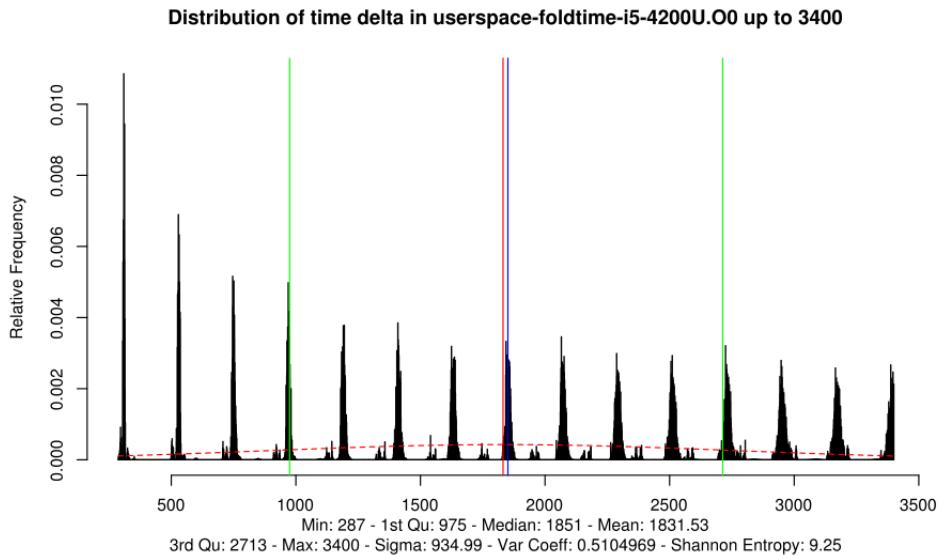
Figure C.2: ~~Upper boundary of entropy over LFSR loop in user space on Intel Core i7 3537U~~

C.2 Intel Core i7 3537U

Distribution of time delta in userspace-foldtime-i7-3537U.O0 up to 300

Min: 233 - 1st Qu: 252 - Median: 255 - Mean: 259.87
3rd Qu: 271 - Max: 300 - Sigma: 8.95 - Var Coeff: 0.0344336 - Shannon Entropy: 3.81
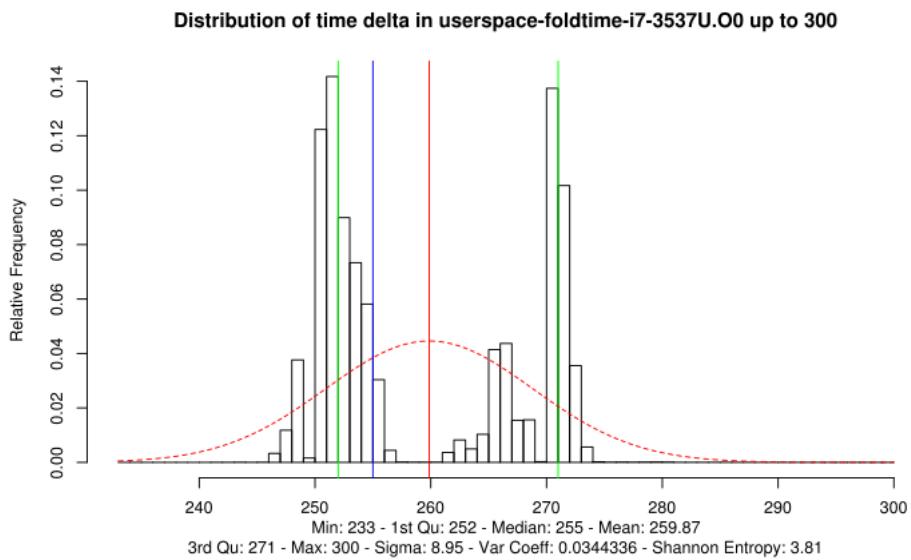
Figure C.3: ~~Lower boundary of e~~Entropy over ~~LFSR~~hash loop in user space on Intel Core i7 3537U

Distribution of time delta in userspace-foldtime-i7-3537U.O0 up to 3500

Min: 243 - 1st Qu: 1001 - Median: 1677 - Mean: 1703.56
3rd Qu: 2476 - Max: 3500 - Sigma: 883.04 - Var Coeff: 0.5183491 - Shannon Entropy: 8.76
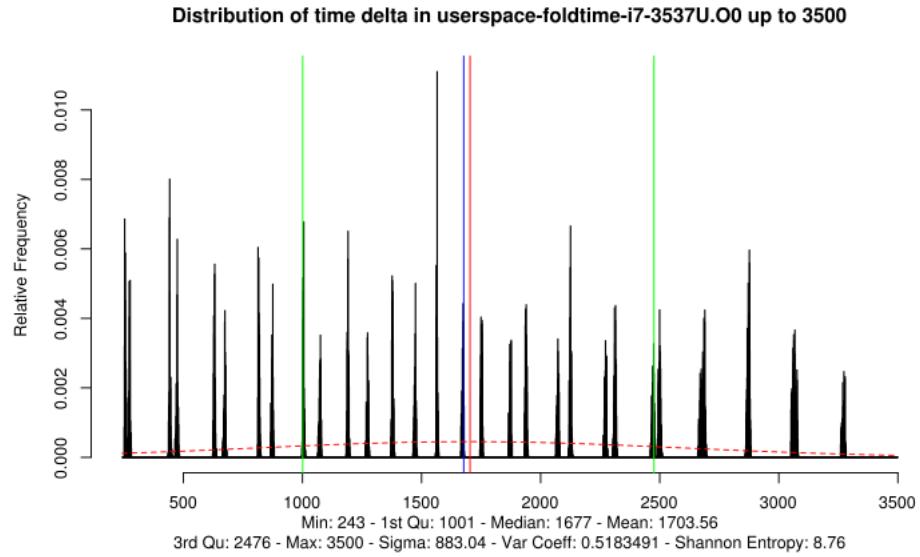
Figure C.4: ~~Upper boundary of entropy over LFSR loop in user space on Intel Core i7 3537U~~

C.3 Intel Core i7 2620M compiled with Clang

The graphs shown in section 5.1 are based on the test cases compiled with GCC and executed on an Intel Core i7 2620M. Now, the same tests on user space executed in the same environment and compiled with the Clang compiler shows interesting results. Especially, the optimizations achieved with Clang are astounding. Yet, these optimizations are not of interest, as the CPU Jitter random number generator shall be compiled without optimizations as specified in the various Makefiles and in section **??**. The non-optimized compilation is in line with the expected results.

Distribution of time delta in userspace-foldtime-i7-2620M-ubuntu-clang up to 175

Min: 36 - 1st Qu: 39 - Median: 39 - Mean: 39.9
3rd Qu: 40 - Max: 175 - Sigma: 6.62 - Var Coeff: 0.1659172 - Shannon Entropy: 1.71
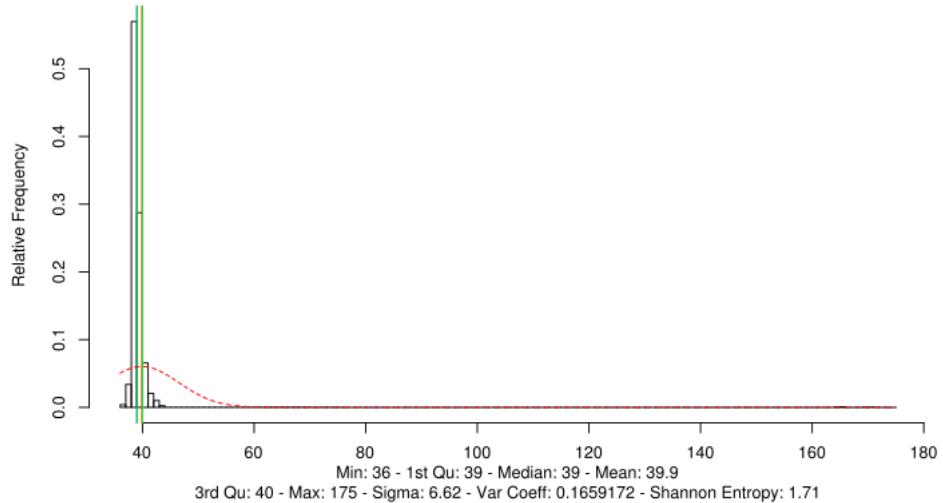
Figure C.5: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Intel Core i7 2620M – with optimizations

Distribution of time delta in userspace-foldtime-i7-2620M-ubuntu-clang up to 200

Min: 36 - 1st Qu: 39 - Median: 39 - Mean: 39.78
3rd Qu: 40 - Max: 200 - Sigma: 6.73 - Var Coeff: 0.169204 - Shannon Entropy: 2.10
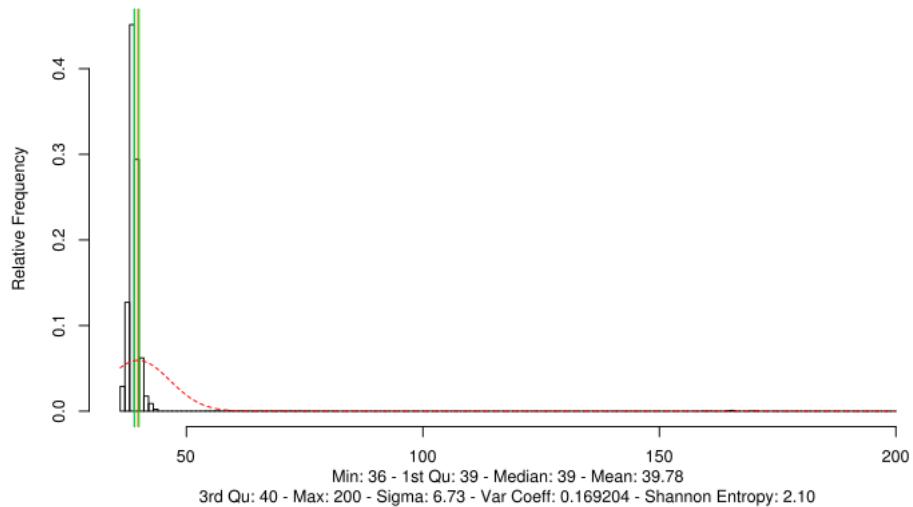
Figure C.6: ~~Upper boundary of entropy over LFSR loop in user space on Intel Core i7 2620M – with optimizations~~

Distribution of time delta in userspace-foldtime-i7-2620M-ubuntu-clang.O0 up to 300

Min: 242 - 1st Qu: 247 - Median: 249 - Mean: 249.53
3rd Qu: 250 - Max: 300 - Sigma: 4.86 - Var Coeff: 0.0194723 - Shannon Entropy: 3.43
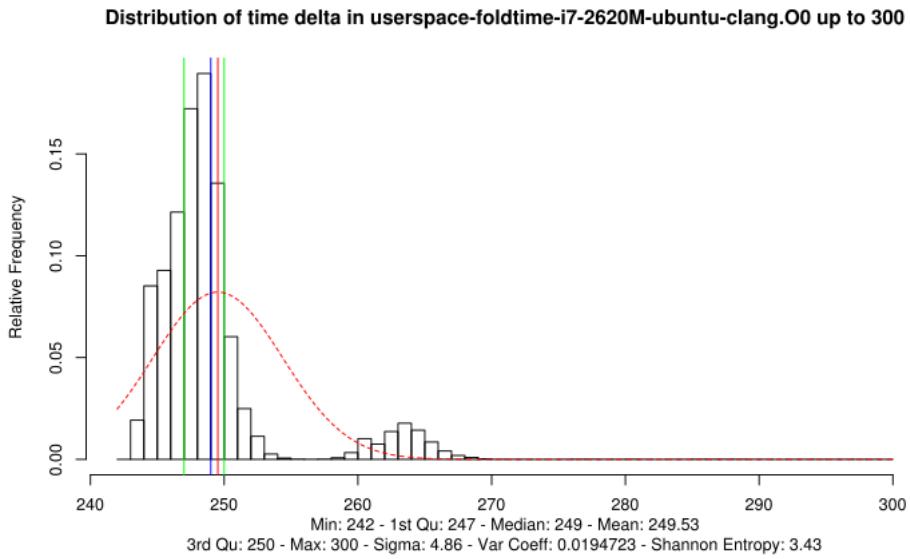
Figure C.7: ~~Entropy~~Lower boundary of entropy over ~~LFSR~~hash loop in user space on Intel Core i7 2620M – without optimizations

Distribution of time delta in userspace-foldtime-i7-2620M-ubuntu-clang.O0 up to 3500

Min: 246 - 1st Qu: 1034 - Median: 1721 - Mean: 1725.88
3rd Qu: 2429 - Max: 3500 - Sigma: 895.78 - Var Coeff: 0.5190263 - Shannon Entropy: 9.07
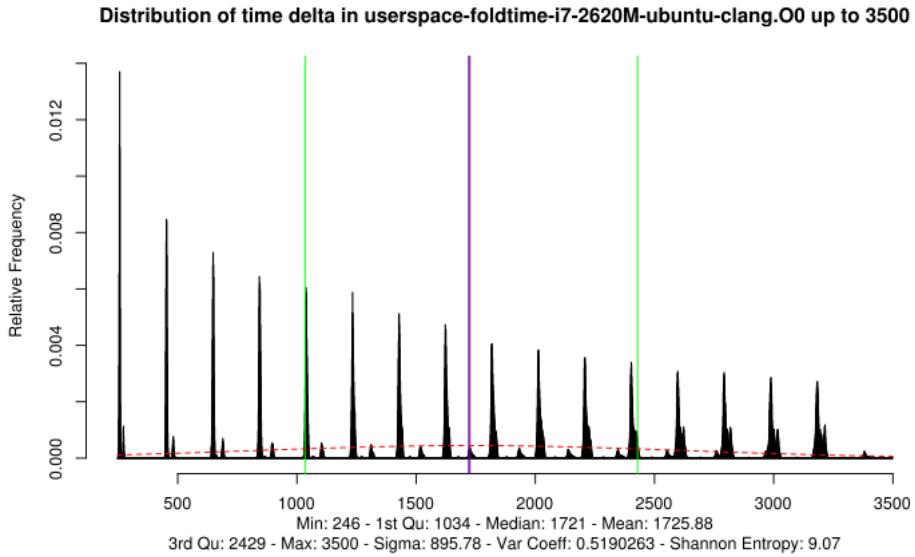
Figure C.8: ~~Upper boundary of entropy over LFSR loop in user space on Intel Core i7 2620M – without optimizations~~

C.4 Intel Core i5 2430M

Distribution of time delta in userspace-foldtime-i5-2430M up to 410

Min: 377 - 1st Qu: 397 - Median: 398 - Mean: 398.9
3rd Qu: 399 - Max: 410 - Sigma: 3.47 - Var Coeff: 0.0086975 - Shannon Entropy: 3.39
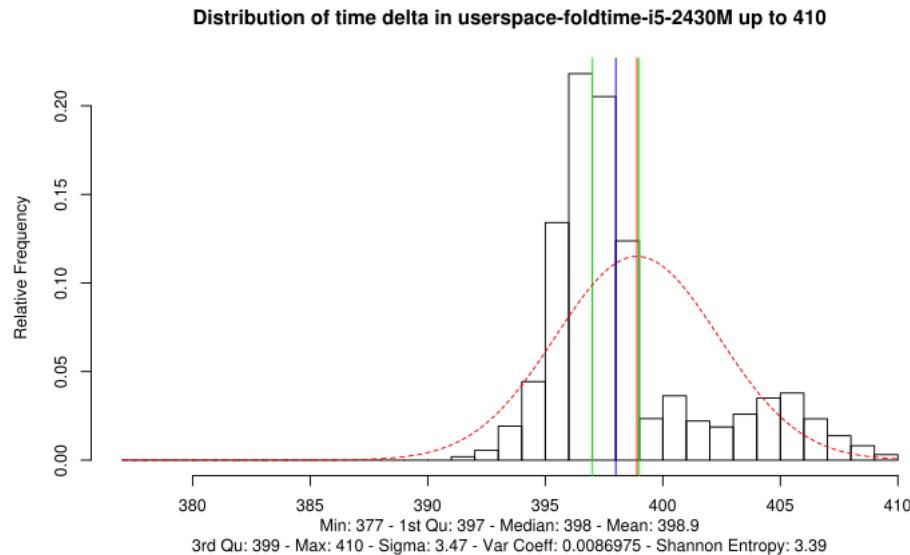
Figure C.9: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Intel Core i5 2430M

Distribution of time delta in userspace-foldtime-i5-2430M up to 3000

Min: 381 - 1st Qu: 1001 - Median: 1583 - Mean: 1536.64
3rd Qu: 2167 - Max: 2999 - Sigma: 688.54 - Var Coeff: 0.4480806 - Shannon Entropy: 8.08
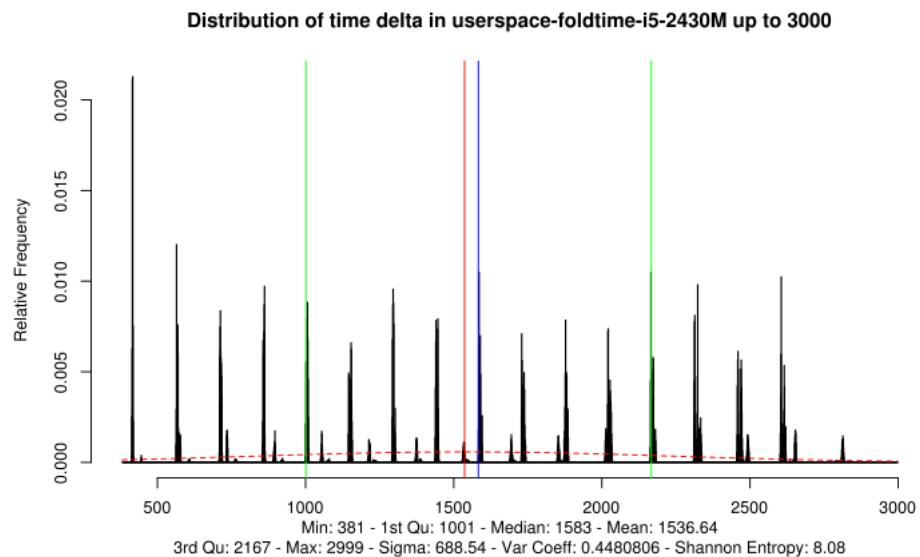
Figure C.10: ~~Upper boundary of entropy over LFSR loop in user space on Intel Core i5 2430M~~

C.5 Intel Core i5 2410M

The test system executes a Linux system with 32 bit word size even though the CPU is capable of executing 64 bit. This shall show that the word size has no impact on the observed CPU execution time jitter.
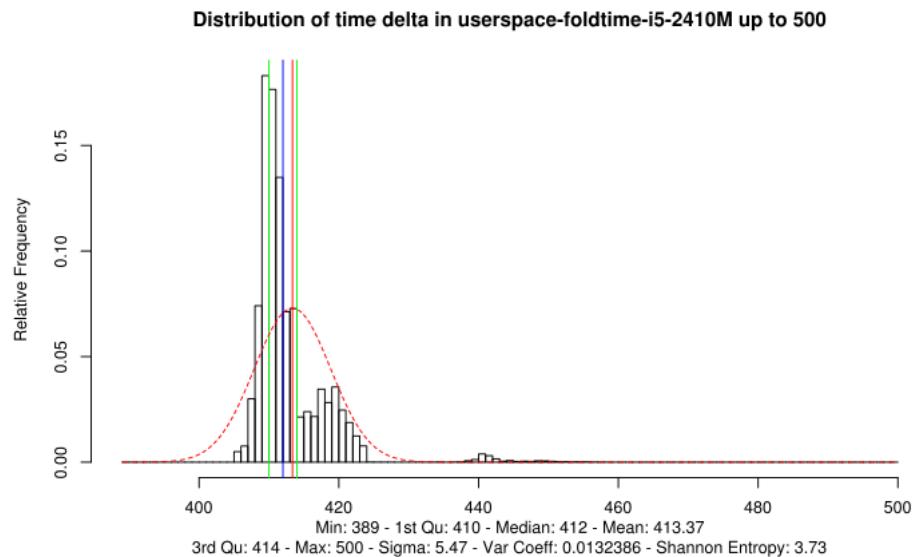
Figure C.11: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Intel Core i5 2410M

Distribution of time delta in userspace-foldtime-i5-2410M up to 3000

Min: 397 - 1st Qu: 1031 - Median: 1635 - Mean: 1586.26
3rd Qu: 2091 - Max: 3000 - Sigma: 678.51 - Var Coeff: 0.4277422 - Shannon Entropy: 7.22
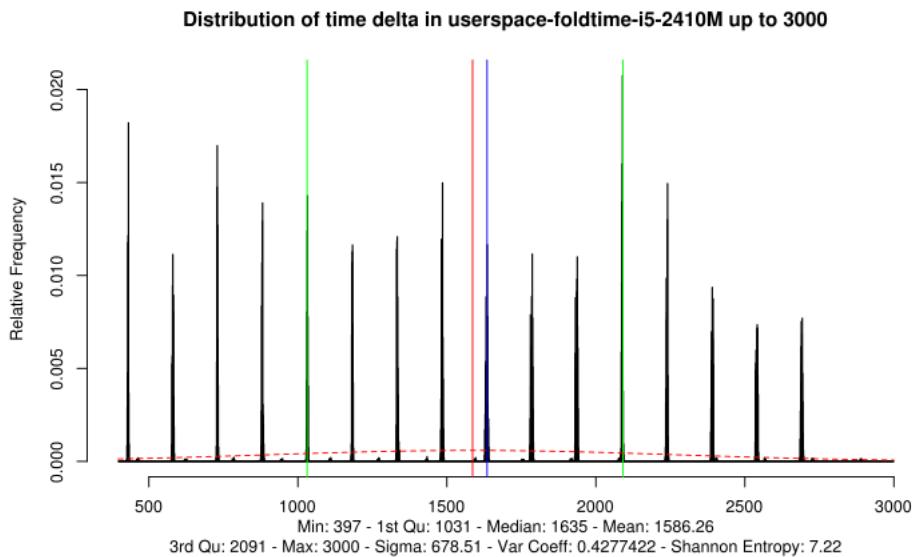
Figure C.12: ~~Upper boundary of entropy over LFSR loop in user space on Intel Core i5 2410M~~

C.6 Intel Core i7 Q720

Distribution of time delta in userspace-foldtime-i7q720 up to 200

Min: 142 - 1st Qu: 154 - Median: 154 - Mean: 158.7
3rd Qu: 158 - Max: 200 - Sigma: 10.95 - Var Coeff: 0.0689976 - Shannon Entropy: 2.60
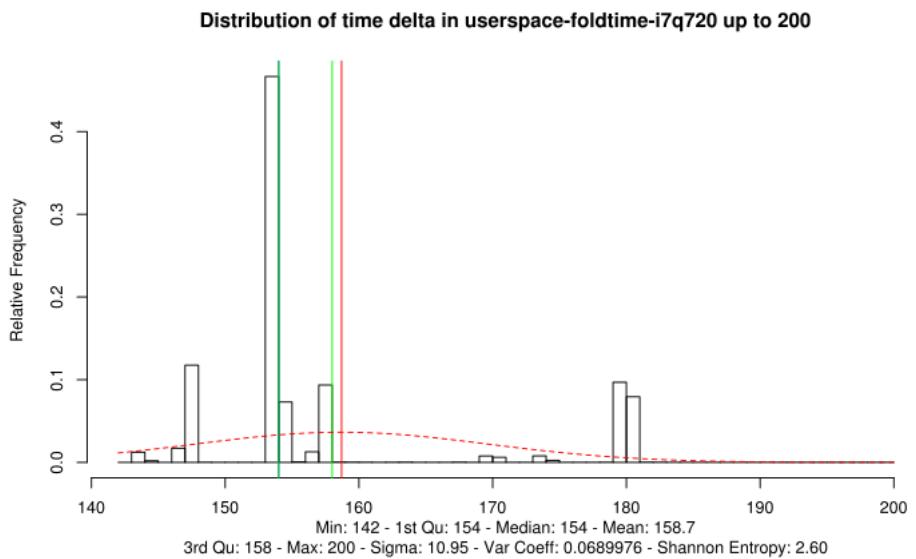
Figure C.13: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Intel Core i7 Q720

Distribution of time delta in userspace-foldtime-i7q720 up to 3000

Min: 141 - 1st Qu: 441 - Median: 947 - Mean: 911.38
3rd Qu: 1368 - Max: 3000 - Sigma: 506.96 - Var Coeff: 0.5562547 - Shannon Entropy: 7.78
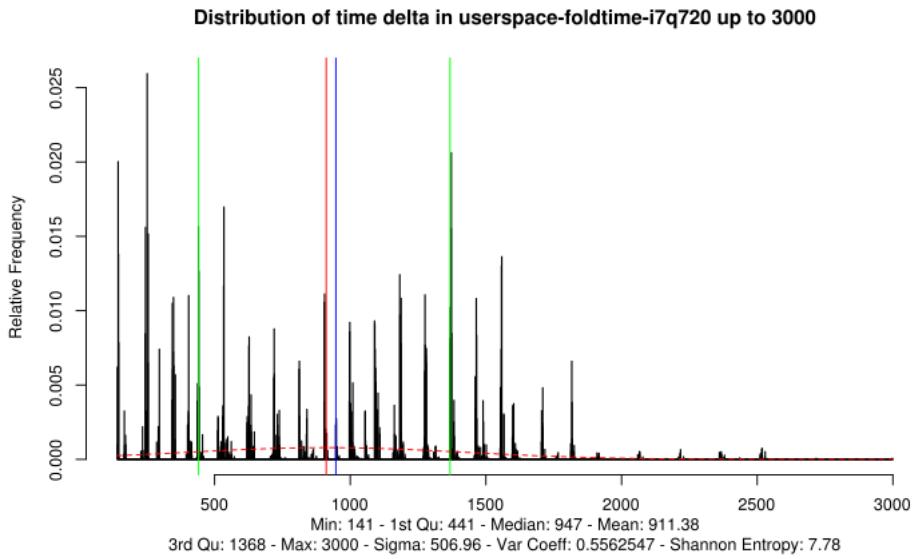
Figure C.14: ~~Upper boundary of entropy over LFSR loop in user space on Intel Core i7 Q720~~

C.7 Intel Xeon E5504

Distribution of time delta in userspace-foldtime-xeon-e5504 up to 250

Min: 220 - 1st Qu: 236 - Median: 236 - Mean: 235.01
3rd Qu: 236 - Max: 250 - Sigma: 4.26 - Var Coeff: 0.0181371 - Shannon Entropy: 1.22
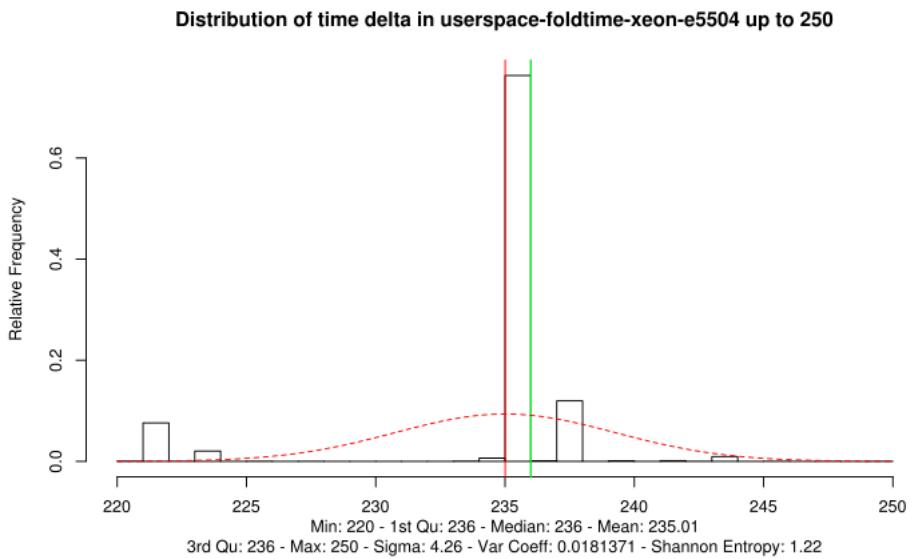
Figure C.15: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Intel Xeon E5504 – with optimizations

Distribution of time delta in userspace-foldtime-xeon-e5504 up to 2500

Min: 219 - 1st Qu: 792 - Median: 1290 - Mean: 1260.89
3rd Qu: 1808 - Max: 2488 - Sigma: 575.13 - Var Coeff: 0.456134 - Shannon Entropy: 5.95
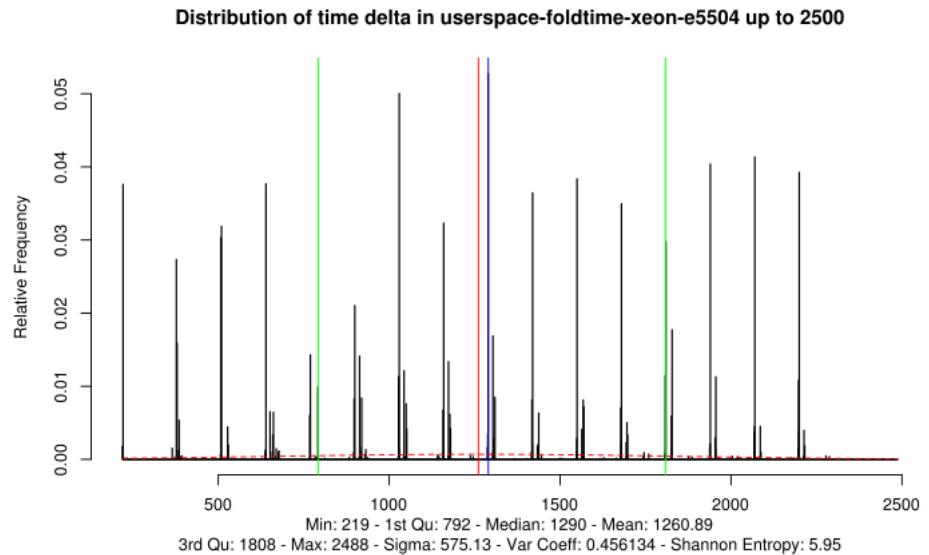
Figure C.16: Upper boundary of entropy over LFSR loop in user space on Intel Xeon E5504 – with optimizations

As the lower boundary is already close to the one bit limit, the same tests without optimizations are performed. The following graphs, however, show even a deterioration of the entropy measurement. The reader should bear in mind that the gathering of the data took less than 20 seconds. Therefore, a short-lived skew may have been observed.

Distribution of time delta in userspace-foldtime-xeon-e5504.O0 up to 540

Min: 521 - 1st Qu: 524 - Median: 524 - Mean: 524.22
3rd Qu: 524 - Max: 540 - Sigma: 0.82 - Var Coeff: 0.00157 - Shannon Entropy: 0.80
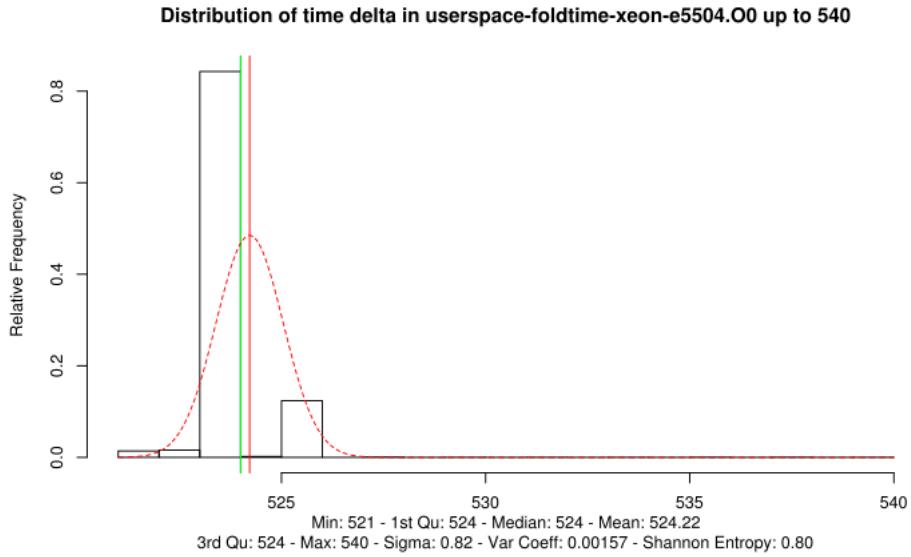
Figure C.17: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Intel Xeon E5504 – without optimizations

Distribution of time delta in userspace-foldtime-xeon-e5504.O0 up to 7000

Min: 520 - 1st Qu: 2124 - Median: 3712 - Mean: 3850.77
3rd Qu: 6092 - Max: 6998 - Sigma: 1984.72 - Var Coeff: 0.5154086 - Shannon Entropy: 5.56
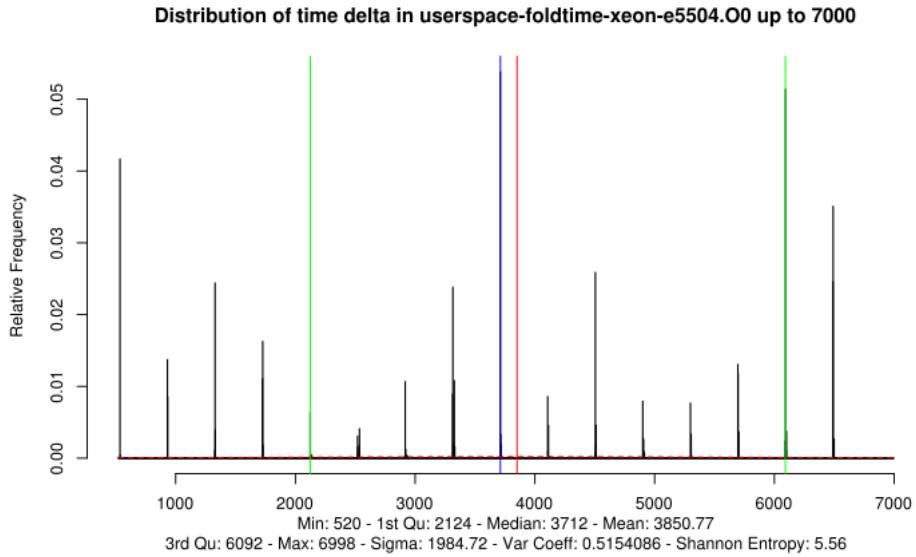
Figure C.18: ~~Upper boundary of entropy over LFSR loop in user space on Intel Xeon E5504 – without optimizations~~

After re-performing the tests, the lower boundary Shannon entropy fluctuates around 1 bit. Therefore, an additional statistical test is performed on an otherwise quiet system to see whether the entropy is still above one bit, i.e.

closer to the upper boundary of the Shannon entropy:

```
# byte wise
$ ./ent random.out
Entropy = 7.999992 bits per byte.

Optimum compression would reduce the size
of this 22188032 byte file by 0 percent.

Chi square distribution for 22188032 samples is 260.80, and randomly
would exceed this value 50.00 percent of the times.

Arithmetic mean value of data bytes is 127.5139 (127.5 = random).
Monte Carlo value for Pi is 3.141290507 (error 0.01 percent).
Serial correlation coefficient is 0.000043 (totally uncorrelated = 0.0).

# bit wise
$ ./ent -b random.out
Entropy = 1.000000 bits per bit.

Optimum compression would reduce the size
of this 178159616 bit file by 0 percent.

Chi square distribution for 178159616 samples is 1.39, and randomly
would exceed this value 50.00 percent of the times.

Arithmetic mean value of data bits is 0.5000 (0.5 = random).
Monte Carlo value for Pi is 3.141272175 (error 0.01 percent).
Serial correlation coefficient is -0.000187 (totally uncorrelated = 0.0).
```

The statistical tests shows that still no patterns are visible. Hence, the CPU is to be considered appropriate for entropy harvesting.

C.8   Intel Core 2 Quad Q6600

The tests were executed with OpenSUSE 12.3. The tests were executed without a graphical interface.
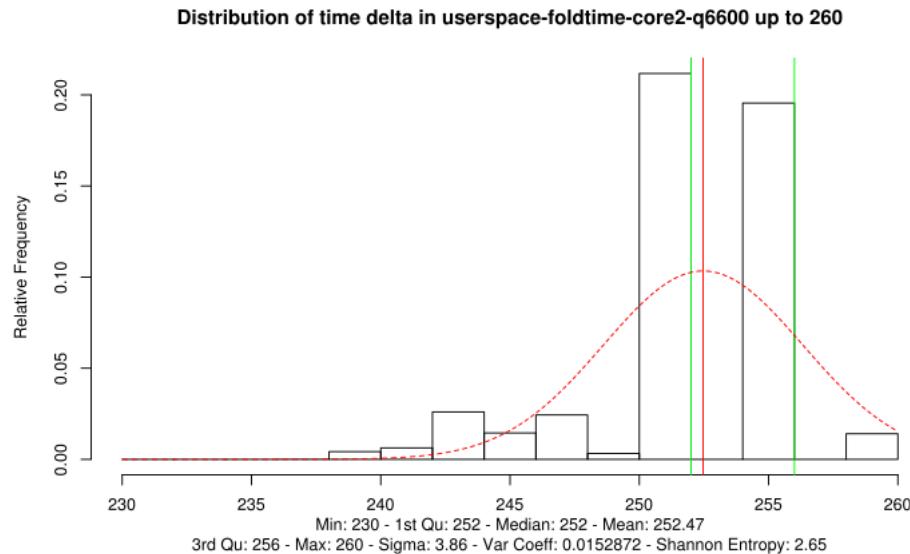
Figure C.19: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Intel Core 2 Quad Q6600 – with optimizations

Distribution of time delta in userspace-foldtime-core2-q6600 up to 2100

Min: 233 - 1st Qu: 621 - Median: 1083 - Mean: 1142.62
3rd Qu: 1553 - Max: 2098 - Sigma: 536.6 - Var Coeff: 0.4696209 - Shannon Entropy: 6.26
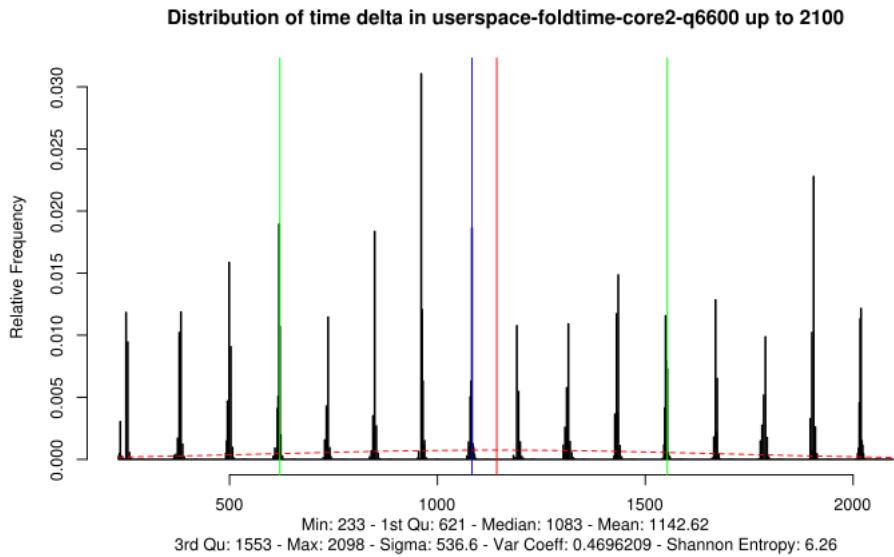
Figure C.20: ~~Upper boundary of entropy over LFSR loop in user space on Intel Core 2 Quad Q6600 – with optimizations~~

The same test compiled without optimizations is shown in the following graphs.

Distribution of time delta in userspace-foldtime-core2-q6600.O0 up to 730

Min: 462 - 1st Qu: 477 - Median: 477 - Mean: 476.31
3rd Qu: 478 - Max: 729 - Sigma: 5.46 - Var Coeff: 0.0114721 - Shannon Entropy: 2.50
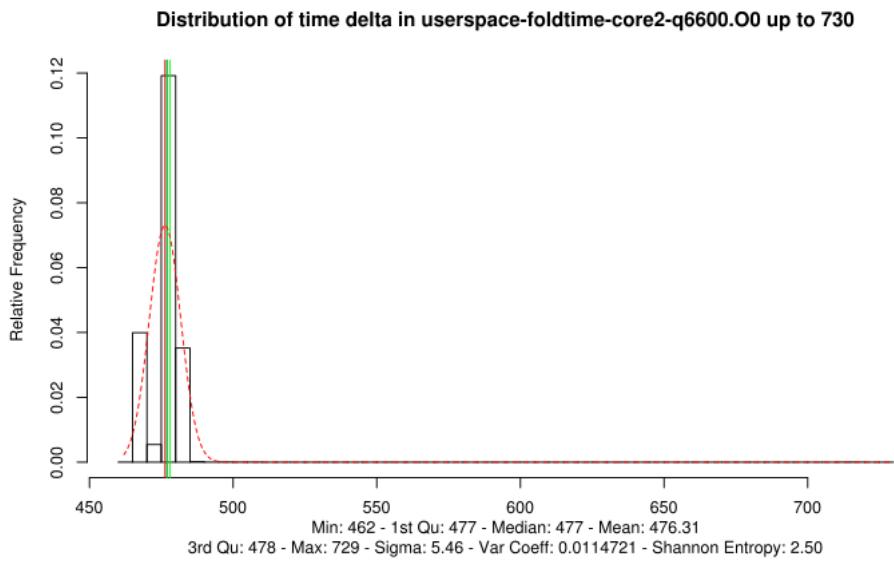
Figure C.21: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Intel Core 2 Quad Q6600 – without optimizations

Distribution of time delta in userspace-foldtime-core2-q6600.O0 up to 6000

Min: 466 - 1st Qu: 1169 - Median: 2492 - Mean: 2844.69
3rd Qu: 4458 - Max: 6000 - Sigma: 1627.48 - Var Coeff: 0.5721097 - Shannon Entropy: 5.78
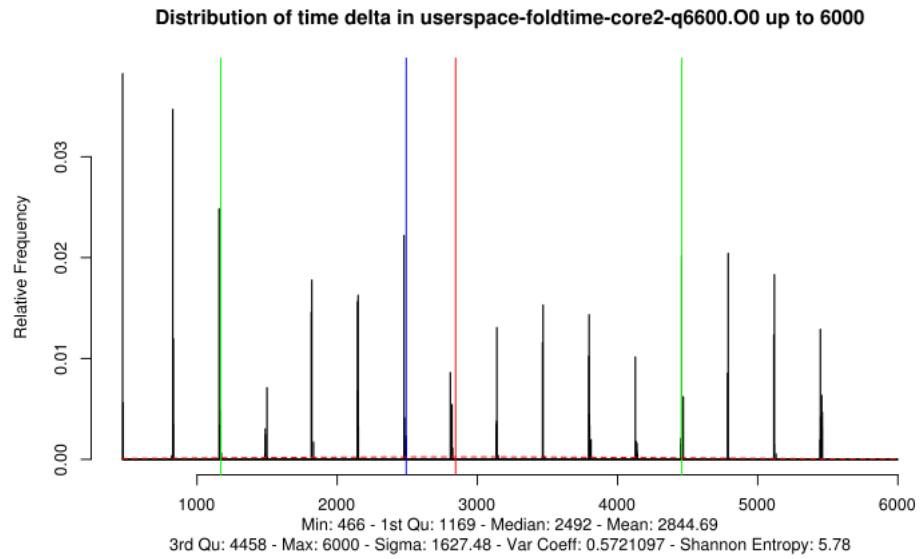
Figure C.22: ~~Upper boundary of entropy over LFSR loop in user space on Intel Core 2 Quad Q6600 – without optimizations~~

C.9 Intel Core 2 Duo T5870

Distribution of time delta in userspace-foldtime-core2duot5870 up to 3000

Min: 1606 - 1st Qu: 2584 - Median: 2584 - Mean: 2596.47
3rd Qu: 2654 - Max: 2934 - Sigma: 88.68 - Var Coeff: 0.0341542 - Shannon Entropy: 2.36
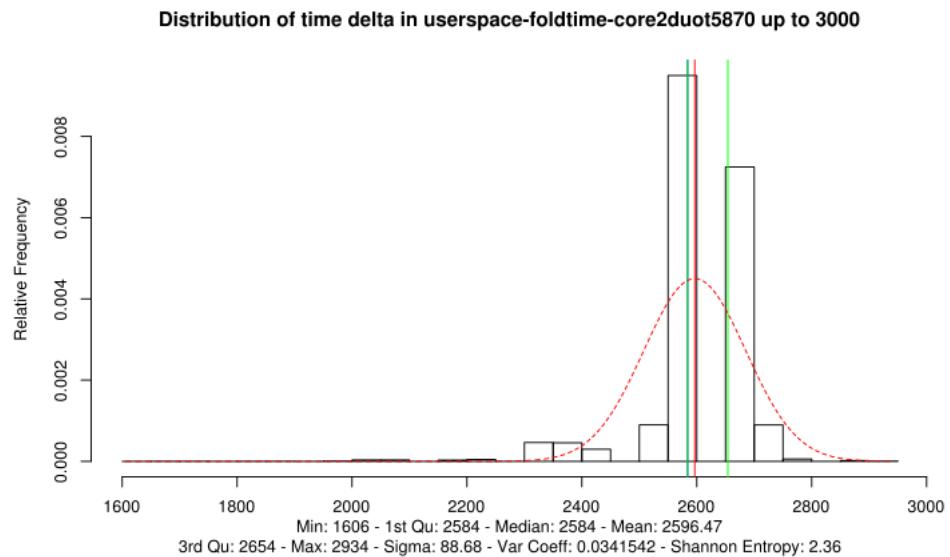
Figure C.23: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Intel Core 2 Duo T5870

Distribution of time delta in userspace-foldtime-core2duot5870 up to 6000

Min: 1606 - 1st Qu: 3352 - Median: 4121 - Mean: 4100.83
3rd Qu: 4889 - Max: 5937 - Sigma: 917.54 - Var Coeff: 0.2237442 - Shannon Entropy: 6.16
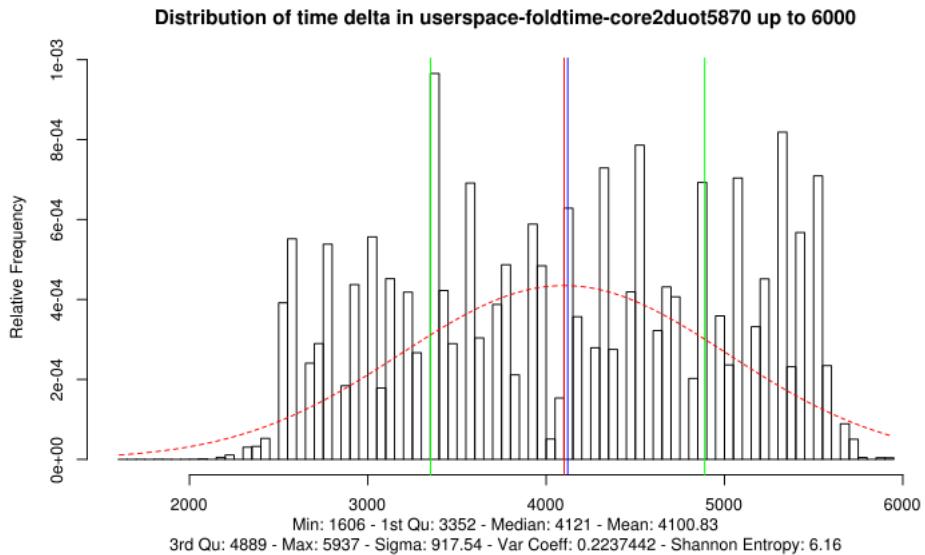
Figure C.24: ~~Upper boundary of entropy over LFSR loop in user space on Intel Core 2 Duo T5870~~

C.10   Intel Core 2 Duo With Windows 7

Distribution of time delta in userspace-foldtime-core2duo-win7 up to 1450

Min: 1333 - 1st Qu: 1365 - Median: 1386 - Mean: 1380.87
3rd Qu: 1397 - Max: 1449 - Sigma: 21.77 - Var Coeff: 0.0157666 - Shannon Entropy: 3.49
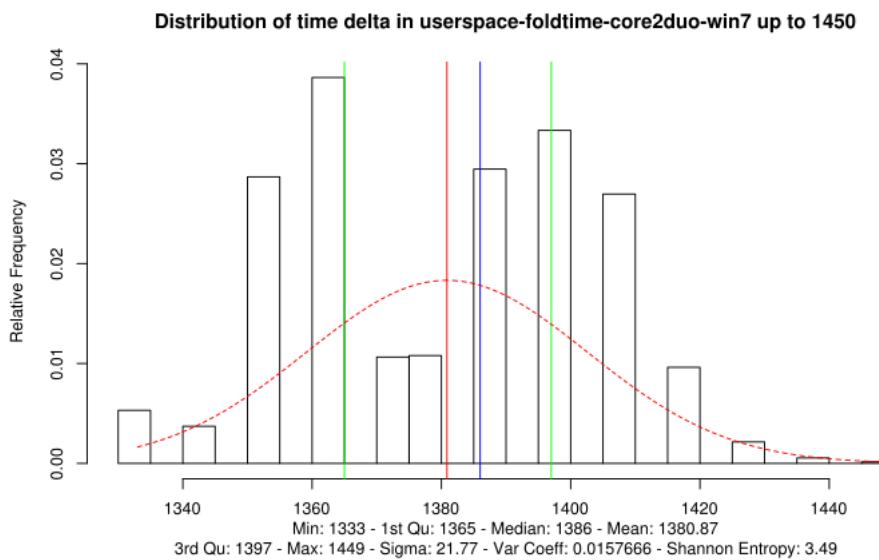
Figure C.25: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Intel Core 2 Duo with Windows – without optimizations

114

Distribution of time delta in userspace-foldtime-core2duo-win7 up to 20000

Min: 1333 - 1st Qu: 6185 - Median: 10941 - Mean: 10466.74
3rd Qu: 15708 - Max: 19992 - Sigma: 5525.16 - Var Coeff: 0.5278784 - Shannon Entropy: 8.19
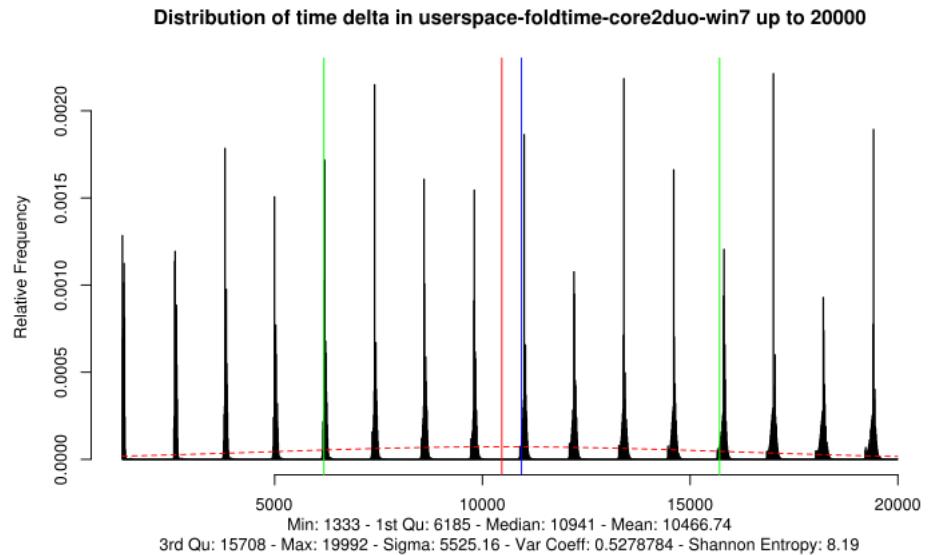
Figure C.26: ~~Upper boundary of entropy over LFSR loop in user space on Intel Core 2 Duo with Windows – without optimizations~~

C.11   Intel Core Duo L2400

Distribution of time delta in userspace-foldtime-coreduol2400 up to 3500

Min: 2444 - 1st Qu: 3143 - Median: 3143 - Mean: 3137.21
3rd Qu: 3143 - Max: 3493 - Sigma: 63.46 - Var Coeff: 0.0202288 - Shannon Entropy: 2.11
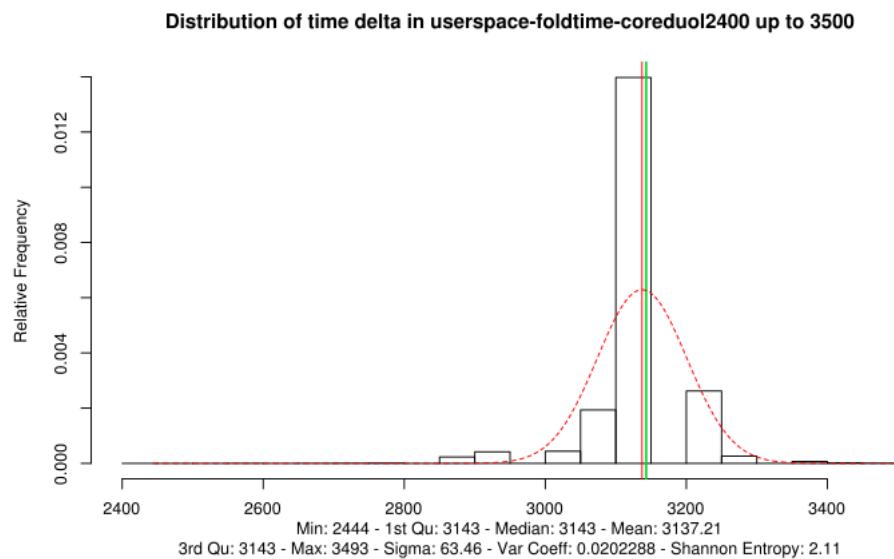
Figure C.27: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Intel Core Duo L2400

Distribution of time delta in userspace-foldtime-coreduol2400 up to 9000

Min: 2514 - 1st Qu: 4261 - Median: 5727 - Mean: 5809.26
3rd Qu: 7264 - Max: 8940 - Sigma: 1717.37 - Var Coeff: 0.2956269 - Shannon Entropy: 6.45
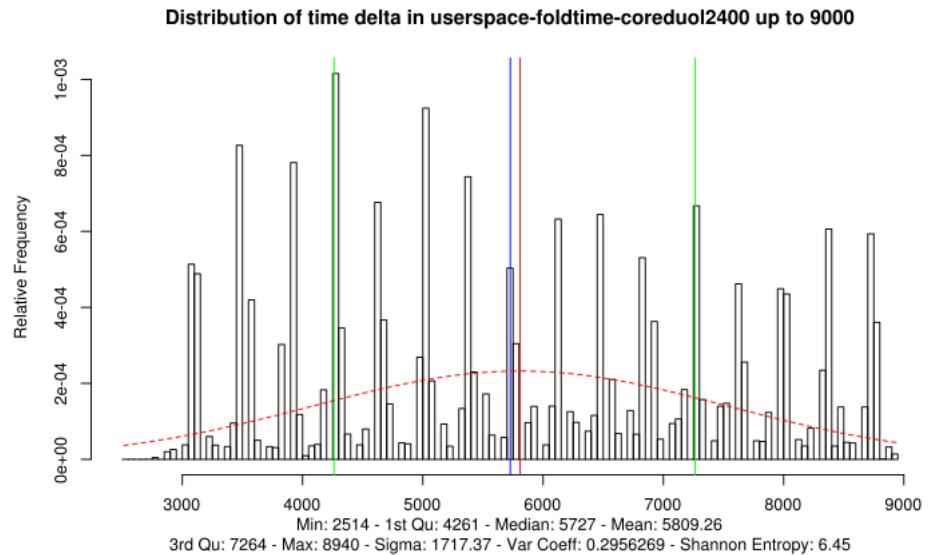
Figure C.28: ~~Upper boundary of entropy over LFSR loop in user space on Intel Core Duo L2400~~

C.12 Intel Core Duo Solo T1300 With NOVA Microkernel

Distribution of time delta in userspace-foldtime-nova-T60-T1300 up to 2550

Min: 2360 - 1st Qu: 2460 - Median: 2460 - Mean: 2468.75
3rd Qu: 2460 - Max: 2550 - Sigma: 21.27 - Var Coeff: 0.0086142 - Shannon Entropy: 1.29
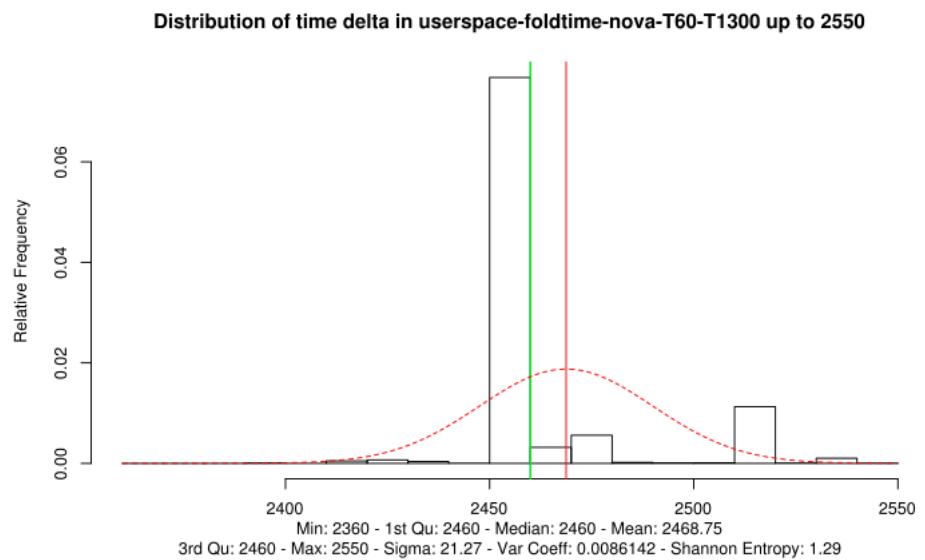
Figure C.29: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Intel Core Duo Solo T1300 with Nova Microkernel

Distribution of time delta in userspace-foldtime-nova-T60-T1300 up to 30000

Min: 2 - 1st Qu: 10320 - Median: 18060 - Mean: 16450.33
3rd Qu: 25810 - Max: 30000 - Sigma: 8976.9 - Var Coeff: 0.5456972 - Shannon Entropy: 5.05
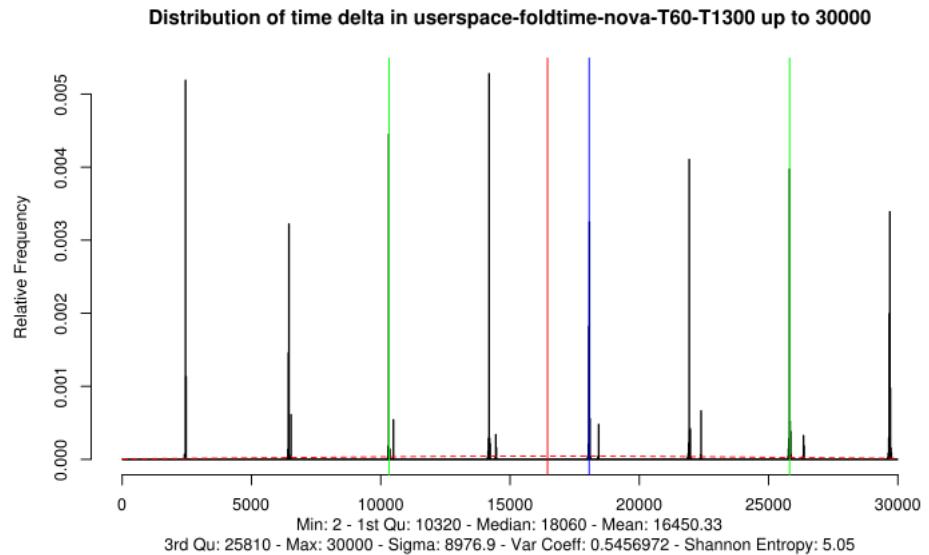
Figure C.30: ~~Upper boundary of entropy over LFSR loop in user space on Intel Core Duo Solo T1300 with Nova Microkernel~~

C.13 Intel Core Duo Solo T1300 With Fiasco.OC Microkernel

Distribution of time delta in userspace-foldtime-foc_x86_32_T60_T1300 up to 1550

Min: 1440 - 1st Qu: 1470 - Median: 1480 - Mean: 1482.78
3rd Qu: 1490 - Max: 1550 - Sigma: 13.74 - Var Coeff: 0.0092645 - Shannon Entropy: 2.01
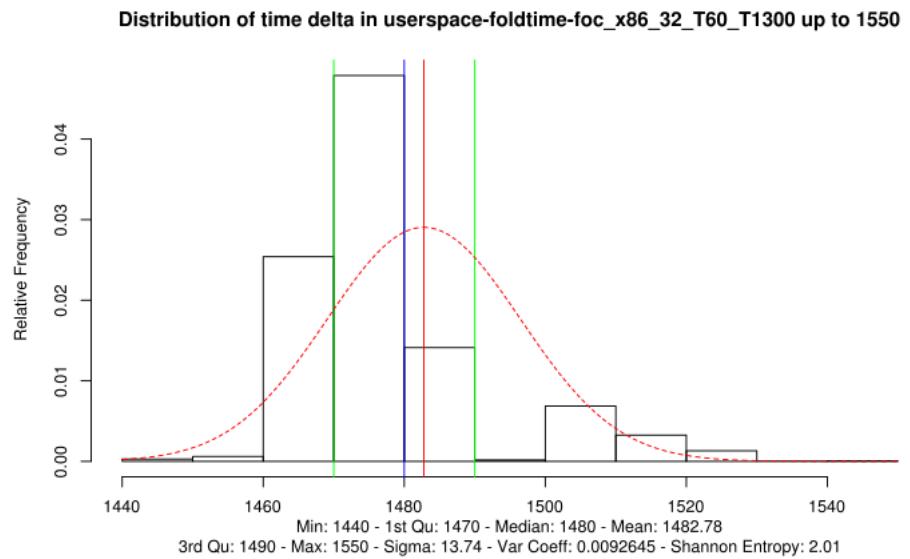
Figure C.31: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Intel Core Duo Solo T1300 with Fiasco.OC Microkernel

117

Distribution of time delta in userspace-foldtime-foc_x86_32_T60_T1300 up to 18000

Min: 1430 - 1st Qu: 3920 - Median: 8670 - Mean: 9333.33
3rd Qu: 13430 - Max: 18000 - Sigma: 5172.45 - Var Coeff: 0.5541905 - Shannon Entropy: 4.91
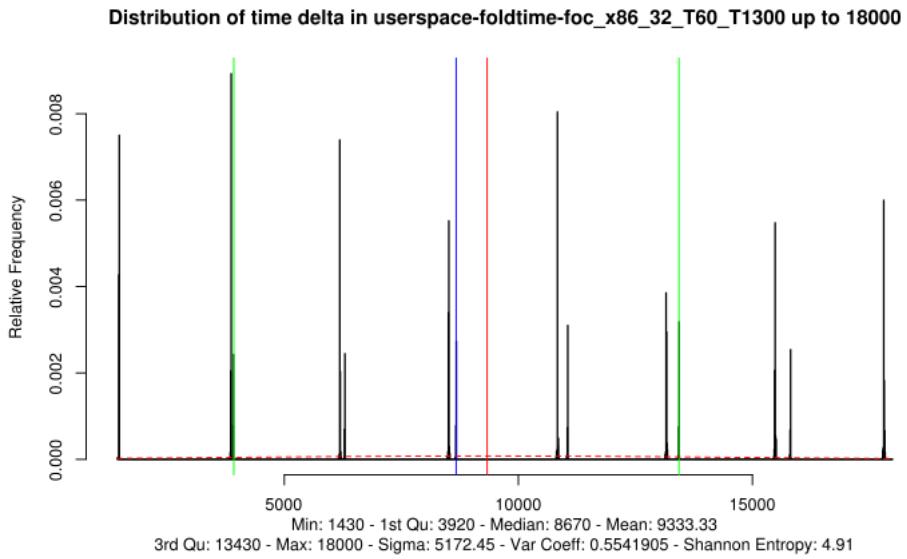
Figure C.32: ~~Upper boundary of entropy over LFSR loop in user space on Intel Core Duo Solo T1300 with Fiasco.OC Microkernel~~

C.14 Intel Core Duo Solo T1300 With Pistachio Micro-kernel

Distribution of time delta in userspace-foldtime-pistachio_x86_32_T60_T1300 up to 2550

Min: 2360 - 1st Qu: 2460 - Median: 2470 - Mean: 2474.97
3rd Qu: 2480 - Max: 2550 - Sigma: 22.2 - Var Coeff: 0.008971 - Shannon Entropy: 2.06
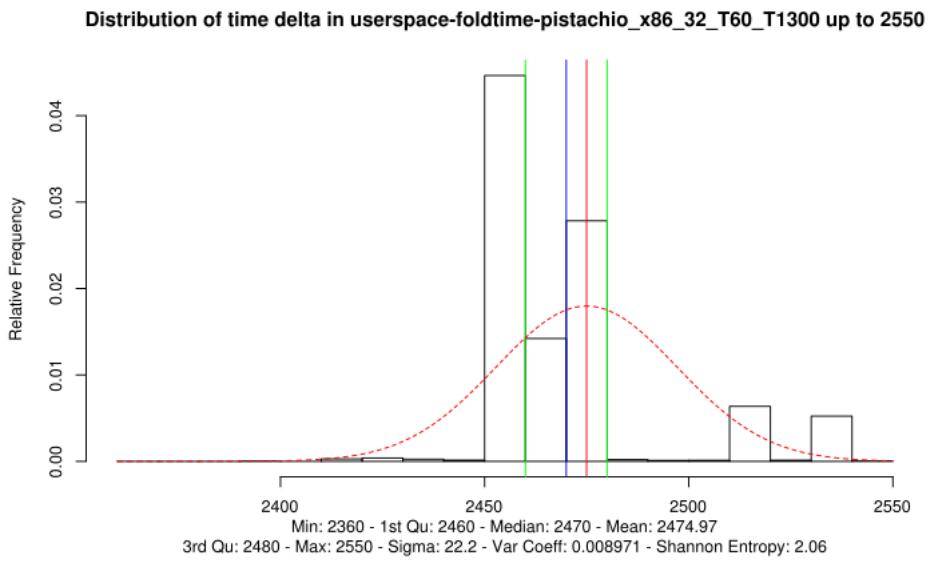
Figure C.33: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Intel Core Duo Solo T1300 with Pistachio Microkernel

Distribution of time delta in userspace-foldtime-pistachio_x86_32_T60_T1300 up to 30000

Min: 2380 - 1st Qu: 6530 - Median: 14470 - Mean: 15719.33
3rd Qu: 22400 - Max: 30000 - Sigma: 8594.28 - Var Coeff: 0.5467335 - Shannon Entropy: 5.28
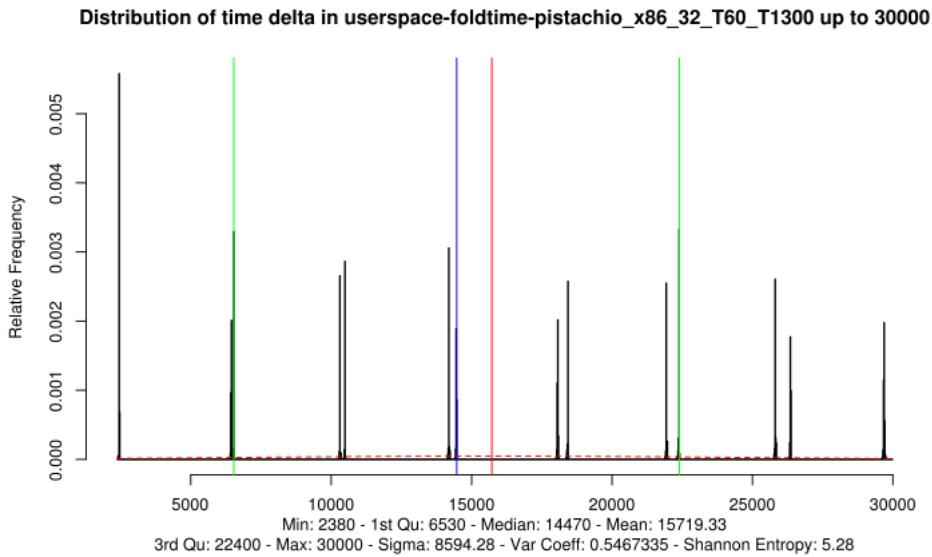
Figure C.34: ~~Upper boundary of entropy over LFSR loop in user space on Intel Core Duo Solo T1300 with Pistachio Microkernel~~

C.15 Intel Atom Z530

Distribution of time delta in userspace-foldtime-atomz530 up to 4700

Min: 3212 - 1st Qu: 3422 - Median: 3422 - Mean: 3526.61
3rd Qu: 3423 - Max: 4680 - Sigma: 277.98 - Var Coeff: 0.0788225 - Shannon Entropy: 2.38
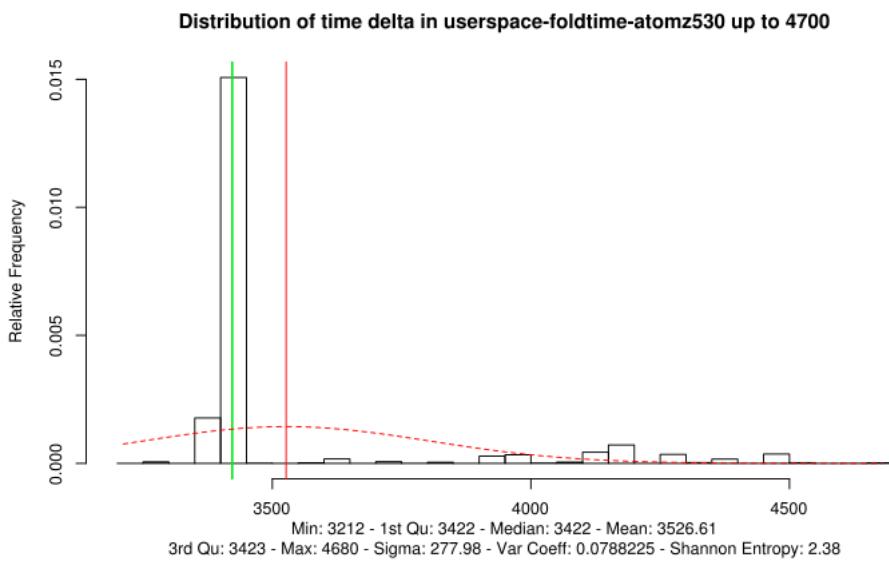
Figure C.35: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Intel Atom Z530

Distribution of time delta in userspace-foldtime-atomz530 up to 21000

Min: 3212 - 1st Qu: 6984 - Median: 11453 - Mean: 9941.03
3rd Qu: 13479 - Max: 20954 - Sigma: 4013.62 - Var Coeff: 0.4037426 - Shannon Entropy: 5.33
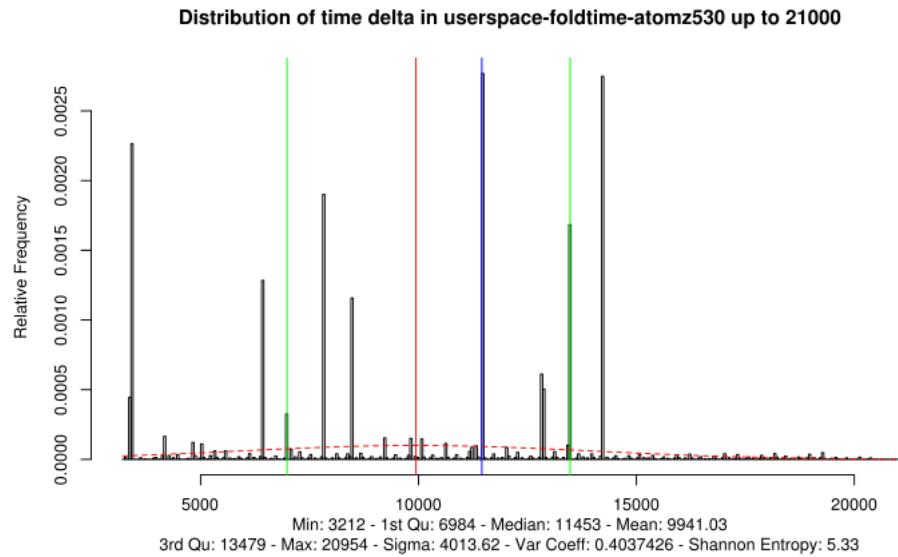
Figure C.36: ~~Upper boundary of entropy over LFSR loop in user space on Intel Atom Z530~~

C.16 Intel Core 2 Duo on Apple MacBook Pro

The following test was executed on an Apple MacBook Pro executing MacOS X 10.8. The compilation was done using GCC.

Distribution of time delta in userspace-foldtime-osx10.8 up to 300

Min: 204 - 1st Qu: 222 - Median: 223 - Mean: 224.03
3rd Qu: 228 - Max: 300 - Sigma: 5.55 - Var Coeff: 0.0247693 - Shannon Entropy: 2.73
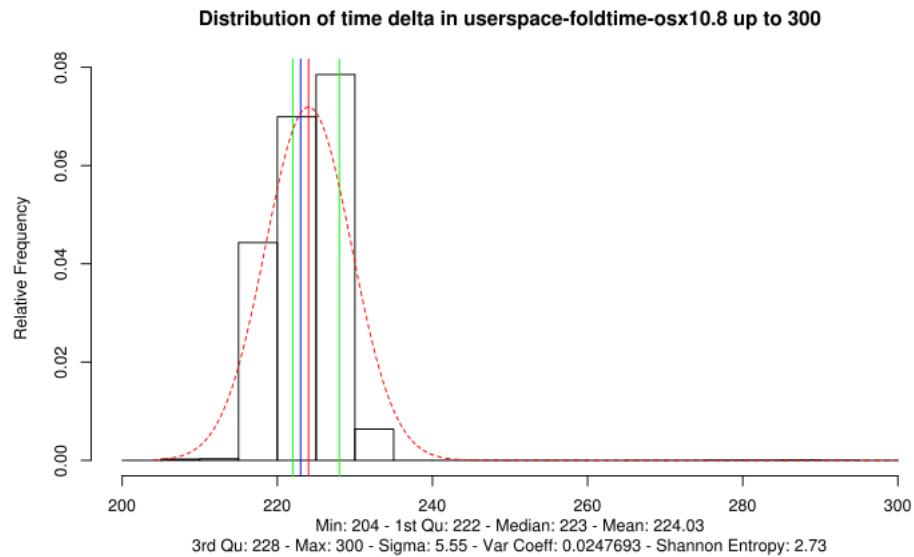
Figure C.37: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Apple MacBook Pro – with optimizations

120

Distribution of time delta in userspace-foldtime-osx10.8 up to 3000

Min: 204 - 1st Qu: 661 - Median: 1101 - Mean: 1087.81
3rd Qu: 1540 - Max: 2996 - Sigma: 506.76 - Var Coeff: 0.4658565 - Shannon Entropy: 6.37
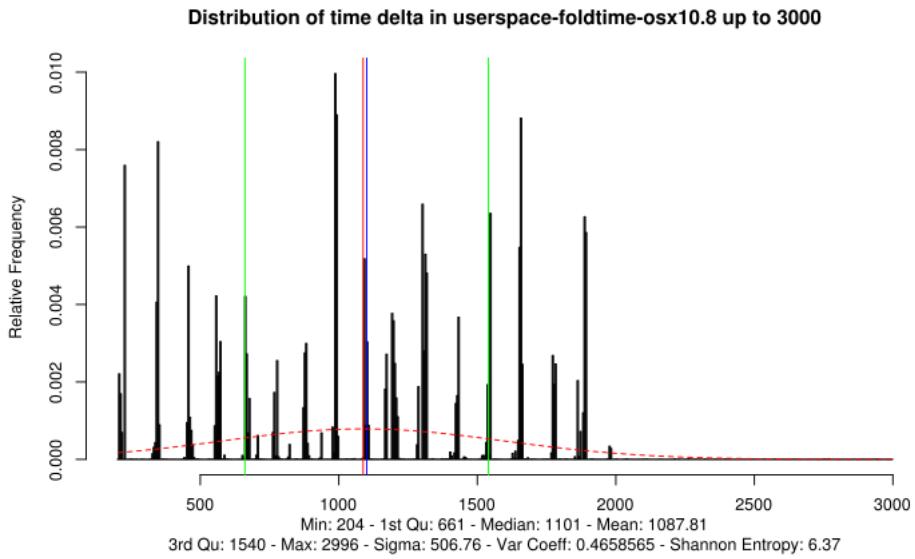
Figure C.38: ~~Upper boundary of entropy over LFSR loop in user space on Apple MacBook Pro – with optimizations~~

C.17   Intel Celeron

Distribution of time delta in userspace-foldtime-celeron up to 1370

Min: 1319 - 1st Qu: 1353 - Median: 1357 - Mean: 1354.98
3rd Qu: 1358 - Max: 1369 - Sigma: 6.47 - Var Coeff: 0.0047721 - Shannon Entropy: 3.22
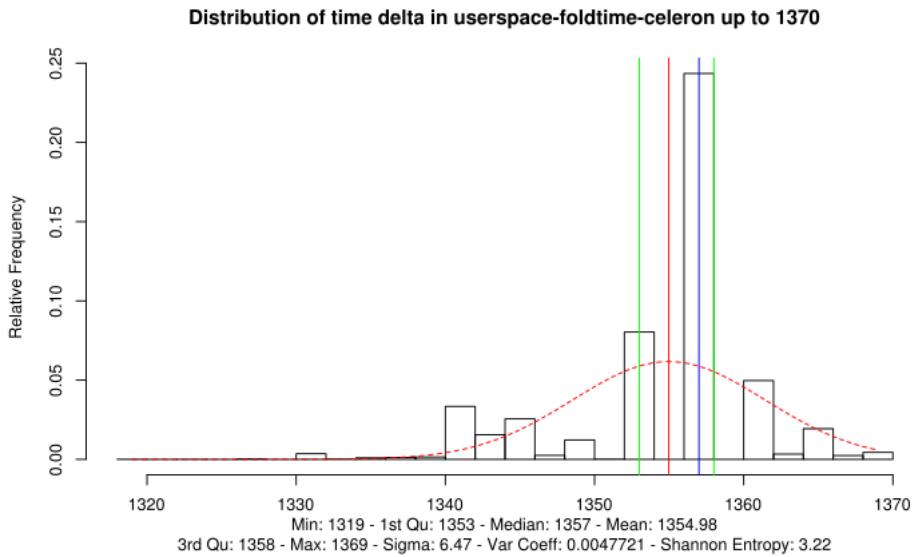
Figure C.39: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Intel Celeron

Distribution of time delta in userspace-foldtime-celeron up to 6000

Relative Frequency

Min: 1312 - 1st Qu: 2418 - Median: 3238 - Mean: 3371.14
3rd Qu: 4305 - Max: 5998 - Sigma: 1222.95 - Var Coeff: 0.3627713 - Shannon Entropy: 7.95
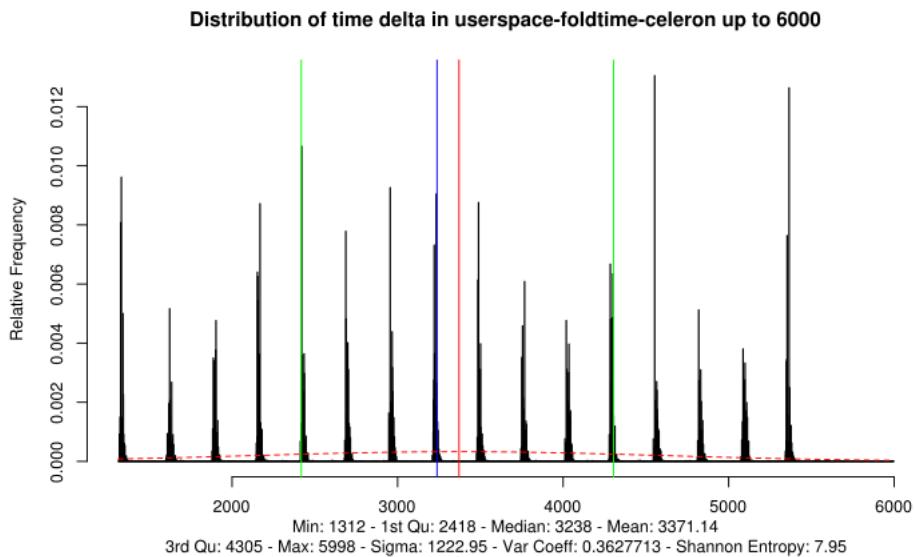
Figure C.40: ~~Upper boundary of entropy over LFSR loop in user space on Intel Celeron~~

C.18 Intel Mobile Celeron 733 MHz

The test was compiled without optimizations.

Distribution of time delta in userspace-foldtime-mobileceleron.O0 up to 4100

Relative Frequency

Min: 3974 - 1st Qu: 3991 - Median: 3991 - Mean: 3993.24
3rd Qu: 3991 - Max: 4100 - Sigma: 13.7 - Var Coeff: 0.0034315 - Shannon Entropy: 0.46
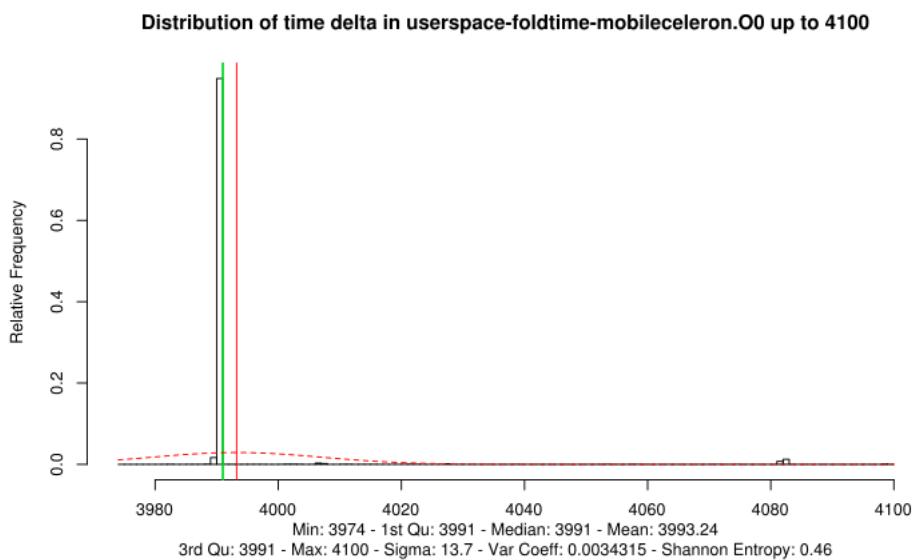
Figure C.41: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Intel Mobile Celeron

Distribution of time delta in userspace-foldtime-mobileceleron.O0 up to 36000

Min: 4029 - 1st Qu: 14476 - Median: 16554 - Mean: 19928.86
3rd Qu: 26945 - Max: 36000 - Sigma: 6843.54 - Var Coeff: 0.3433985 - Shannon Entropy: 4.35
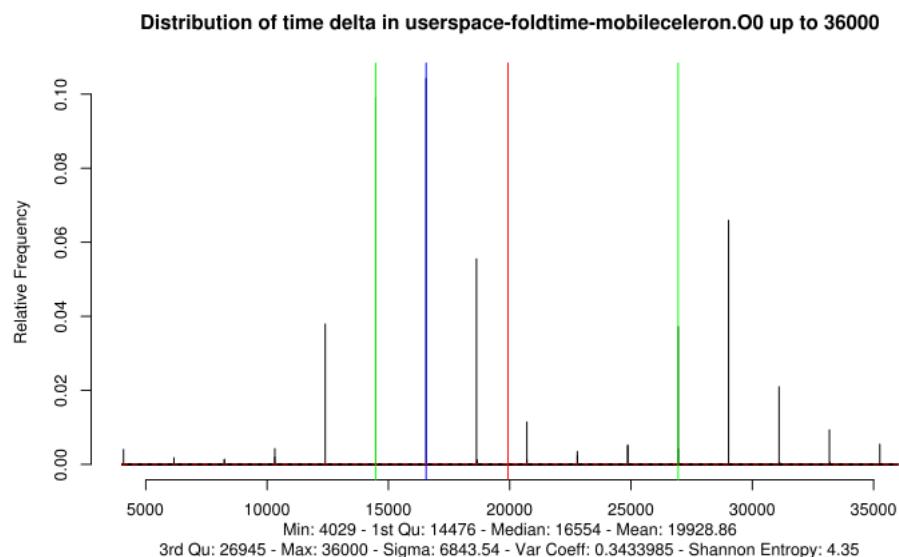
Figure C.42: Upper boundary of entropy over LFSR loop in user space on Intel Mobile Celeron

The tests results indicate that the CPU execution time jitter is insufficient for entropy collection. However, as this CPU is considered so old, the code has not been changed to catch this CPU behavior.

C.19 Intel Pentium P4 3GHz

Distribution of time delta in userspace-foldtime-p4-3GHz up to 950

Min: 893 - 1st Qu: 920 - Median: 923 - Mean: 922.36
3rd Qu: 923 - Max: 950 - Sigma: 6.87 - Var Coeff: 0.0074497 - Shannon Entropy: 3.41
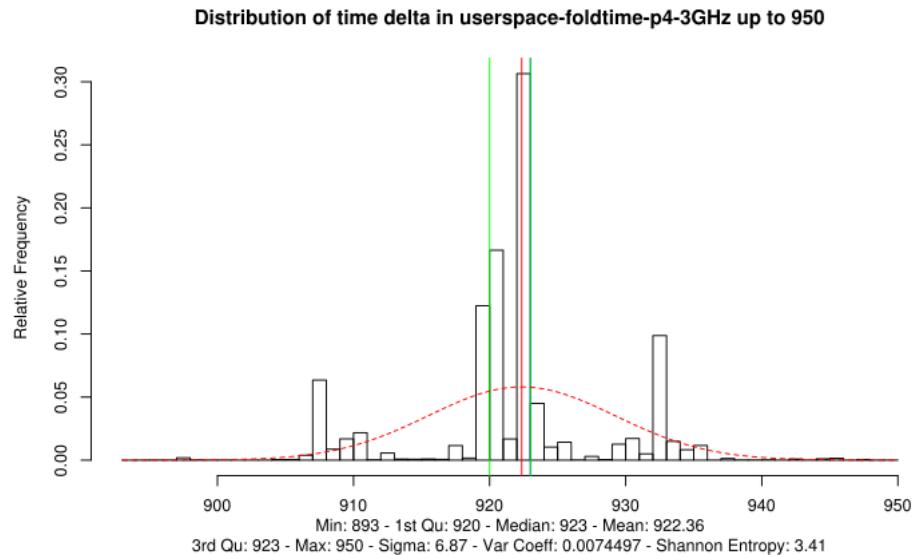
Figure C.43: EntropyLower boundary of entropy over LFSRhash loop in user space on Intel Pentium P4 3GHz

Distribution of time delta in userspace-foldtime-p4-3GHz up to 5000

Min: 892 - 1st Qu: 1848 - Median: 2707 - Mean: 2798.29
3rd Qu: 3697 - Max: 5000 - Sigma: 1145.06 - Var Coeff: 0.4091988 - Shannon Entropy: 8.13
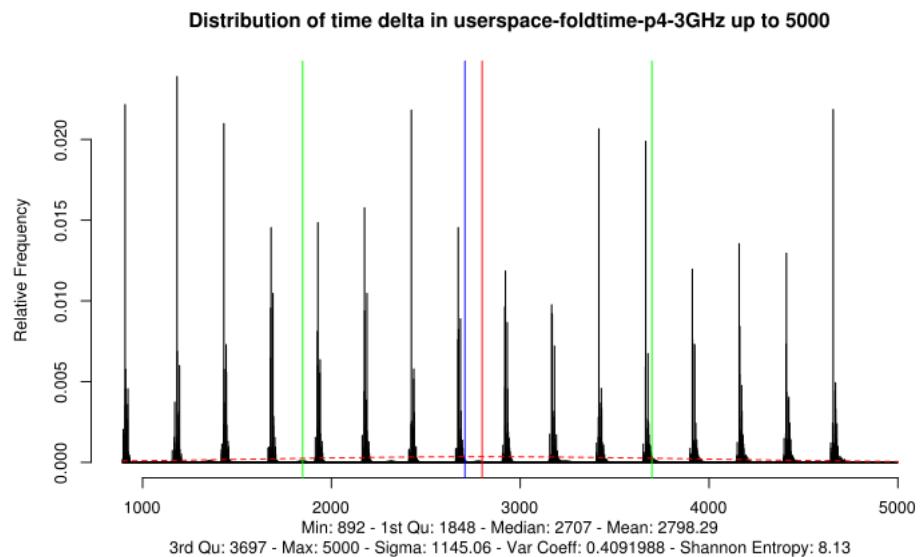
Figure C.44: Upper boundary of entropy over LFSR loop in user space on Intel Pentium P4 3GHz

C.20   Intel Pentium P4 Mobile

Distribution of time delta in userspace-foldtime-p4 up to 7300

Min: 5028 - 1st Qu: 5029 - Median: 5308 - Mean: 5225.1
3rd Qu: 5308 - Max: 7264 - Sigma: 176.69 - Var Coeff: 0.0338161 - Shannon Entropy: 1.52
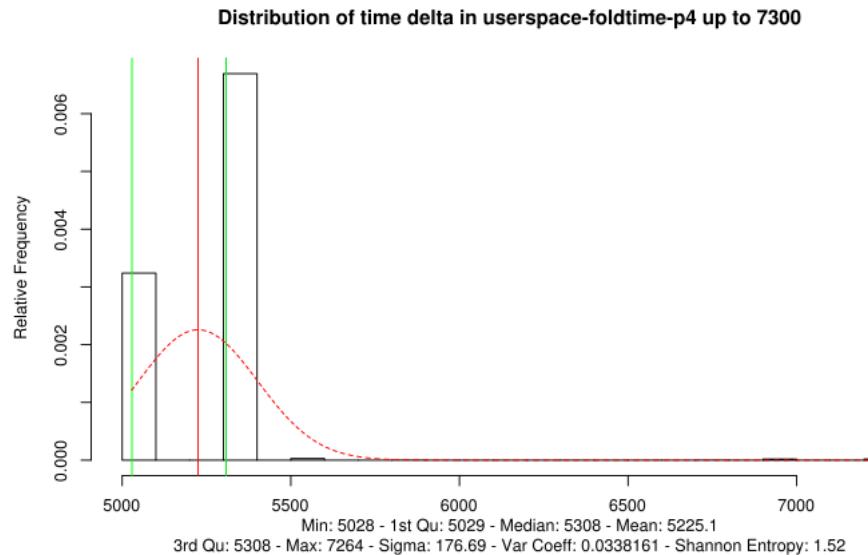
Figure C.45: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Intel Pentium P4 Mobile

Distribution of time delta in userspace-foldtime-p4mobile up to 25000

Min: 5028 - 1st Qu: 7543 - Median: 9499 - Mean: 10470.28
3rd Qu: 13689 - Max: 24865 - Sigma: 3450.92 - Var Coeff: 0.3295923 - Shannon Entropy: 5.35
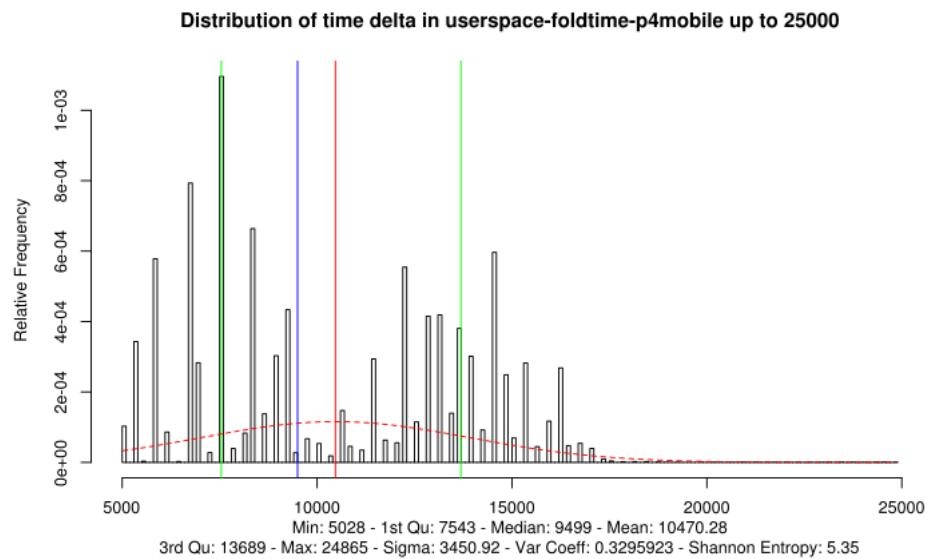
Figure C.46: ~~Upper boundary of entropy over LFSR loop in user space on Intel Pentium P4 Mobile~~

As the Shannon entropy values and the distribution may suggest that patterns are present, the following statistical test is executed, showing that no patterns are visible. Therefore, the CPU execution time jitter is considered to be appropriate for harvesting entropy.

```
# byte-wise
$ ent random-p4mobile.data
Entropy = 7.999998 bits per byte.

Optimum compression would reduce the size
of this 108351488 byte file by 0 percent.

Chi square distribution for 108351488 samples is 230.51, and randomly
would exceed this value 75.00 percent of the times.

Arithmetic mean value of data bytes is 127.5061 (127.5 = random).
Monte Carlo value for Pi is 3.141212037 (error 0.01 percent).
Serial correlation coefficient is 0.000075 (totally uncorrelated = 0.0).

# bit-wise
$ ent -b random-p4mobile.data
Entropy = 1.000000 bits per bit.

Optimum compression would reduce the size
of this 866811904 bit file by 0 percent.

Chi square distribution for 866811904 samples is 0.12, and randomly
would exceed this value 50.00 percent of the times.

Arithmetic mean value of data bits is 0.5000 (0.5 = random).
Monte Carlo value for Pi is 3.141212037 (error 0.01 percent).
Serial correlation coefficient is 0.000023 (totally uncorrelated = 0.0).
```

C.21   AMD Opteron 6128
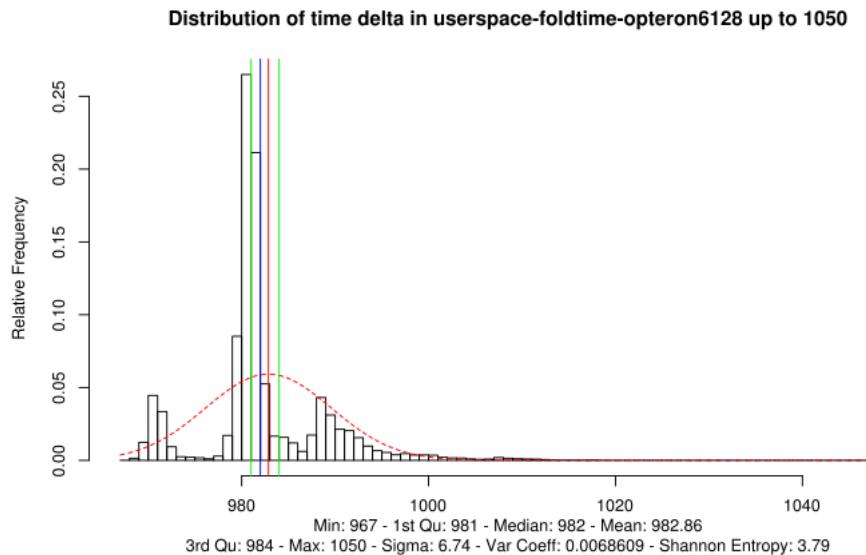
Figure C.47: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on AMD Opteron 6128

Distribution of time delta in userspace-foldtime-opteron6128 up to 6000

Min: 967 - 1st Qu: 1919 - Median: 3427 - Mean: 3280.47
3rd Qu: 4648 - Max: 6000 - Sigma: 1428.71 - Var Coeff: 0.4355196 - Shannon Entropy: 7.97
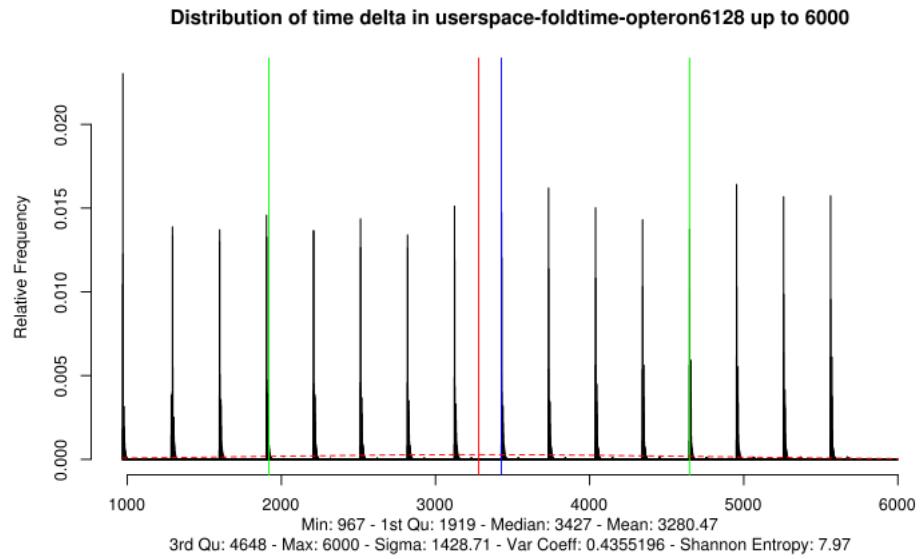
Figure C.48: ~~Upper boundary of entropy over LFSR loop in user space on AMD Opteron 6128~~

C.22   AMD Phenom II X6 1035T

Distribution of time delta in userspace-foldtime-amdphenom-X6-1035T up to 240

Min: 184 - 1st Qu: 191 - Median: 198 - Mean: 195.37
3rd Qu: 199 - Max: 240 - Sigma: 4.81 - Var Coeff: 0.0246124 - Shannon Entropy: 2.89
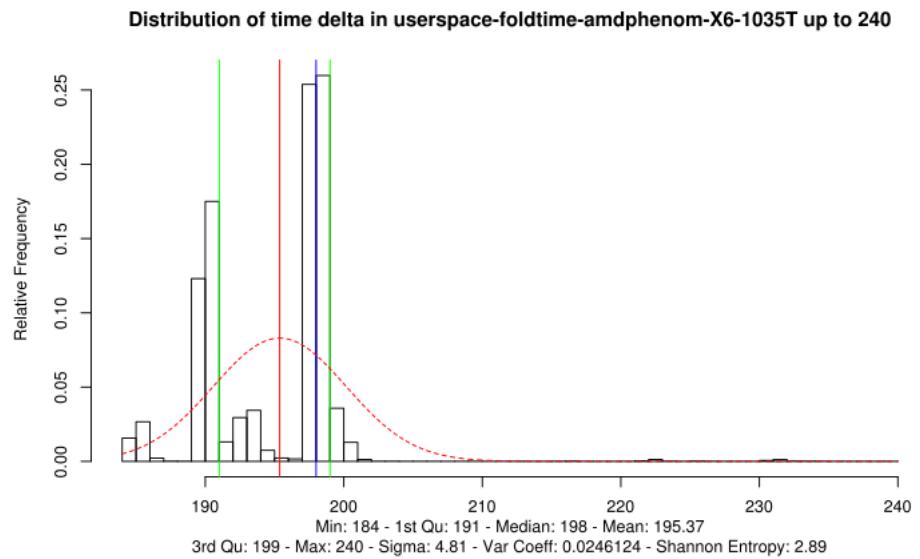
Figure C.49: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on AMD Phenom II X6 1035T

Distribution of time delta in userspace-foldtime-amdphenom-X6-1035T up to 1900

Min: 184 - 1st Qu: 546 - Median: 900 - Mean: 872.6
3rd Qu: 1254 - Max: 1886 - Sigma: 406.56 - Var Coeff: 0.4659156 - Shannon Entropy: 6.66
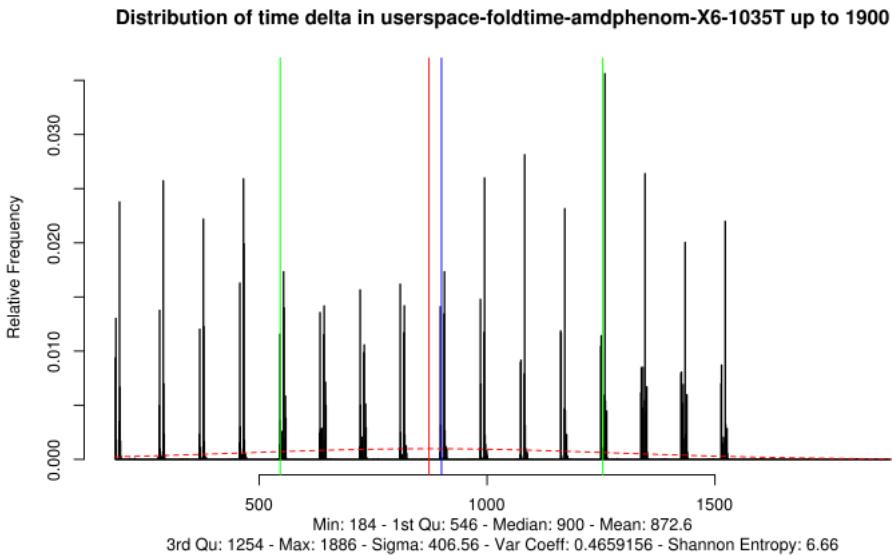
Figure C.50: ~~Upper boundary of entropy over LFSR loop in user space on AMD Phenom II X6 1035T~~

C.23 AMD Athlon 7550

Distribution of time delta in userspace-foldtime-athlon-7550 up to 670

Min: 652 - 1st Qu: 664 - Median: 665 - Mean: 664.18
3rd Qu: 666 - Max: 670 - Sigma: 3.17 - Var Coeff: 0.0047747 - Shannon Entropy: 2.72
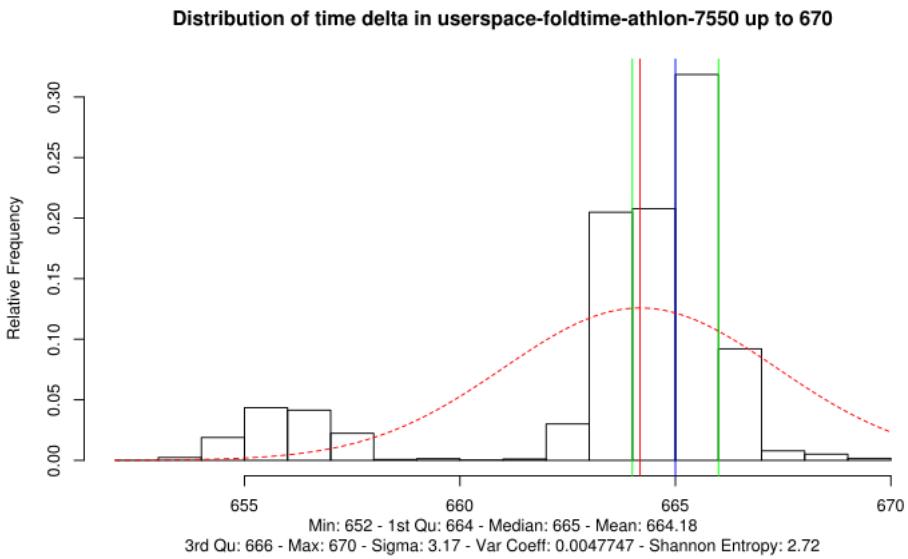
Figure C.51: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on AMD Athlon 7550 – with optimizations

Distribution of time delta in userspace-foldtime-athlon-7550 up to 4500

Relative Frequency

Min: 653 - 1st Qu: 1416 - Median: 2374 - Mean: 2486.98
3rd Qu: 3584 - Max: 4500 - Sigma: 1132.22 - Var Coeff: 0.4552592 - Shannon Entropy: 6.64
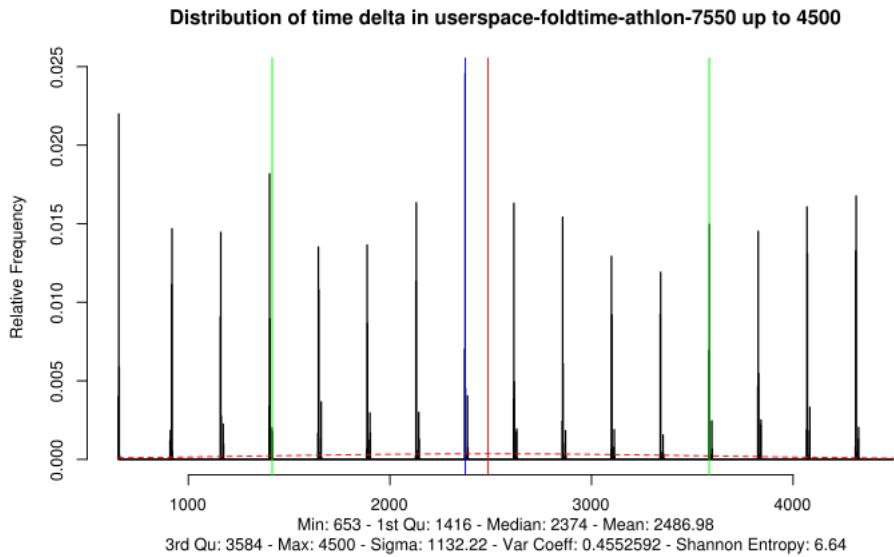
Figure C.52: ~~Upper boundary of entropy over LFSR loop in user space on AMD Athlon 7550 – with optimizations~~

The same tests without optimizations show the following results:

Distribution of time delta in userspace-foldtime-athlon-7550.O0 up to 940

Relative Frequency

Min: 907 - 1st Qu: 927 - Median: 928 - Mean: 927.18
3rd Qu: 929 - Max: 940 - Sigma: 4.16 - Var Coeff: 0.0044892 - Shannon Entropy: 3.29
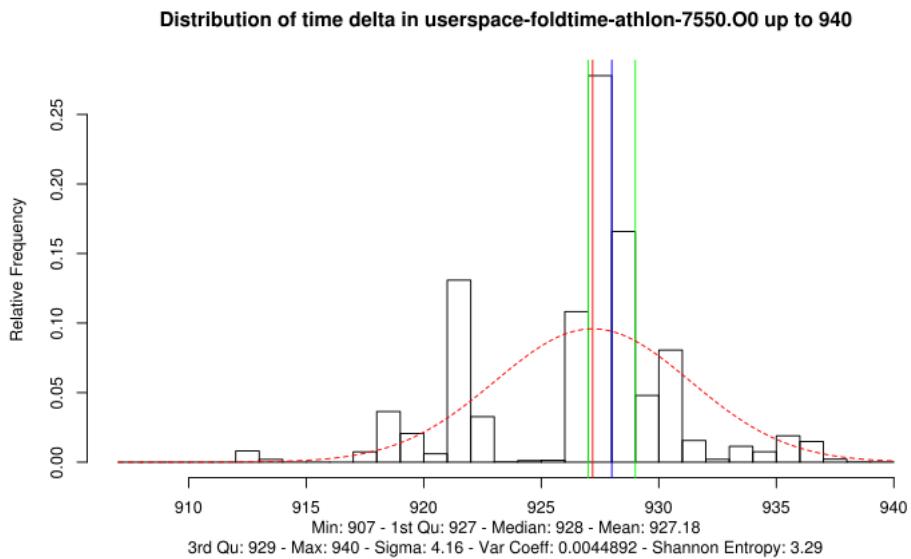
Figure C.53: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on AMD Athlon 7550 – without optimizations

**Distribution of time delta in userspace-foldtime-athlon-7550.O0 up to 8500**

Min: 904 - 1st Qu: 2389 - Median: 4332 - Mean: 4587.49
3rd Qu: 6745 - Max: 8500 - Sigma: 2296.76 - Var Coeff: 0.5006574 - Shannon Entropy: 6.73
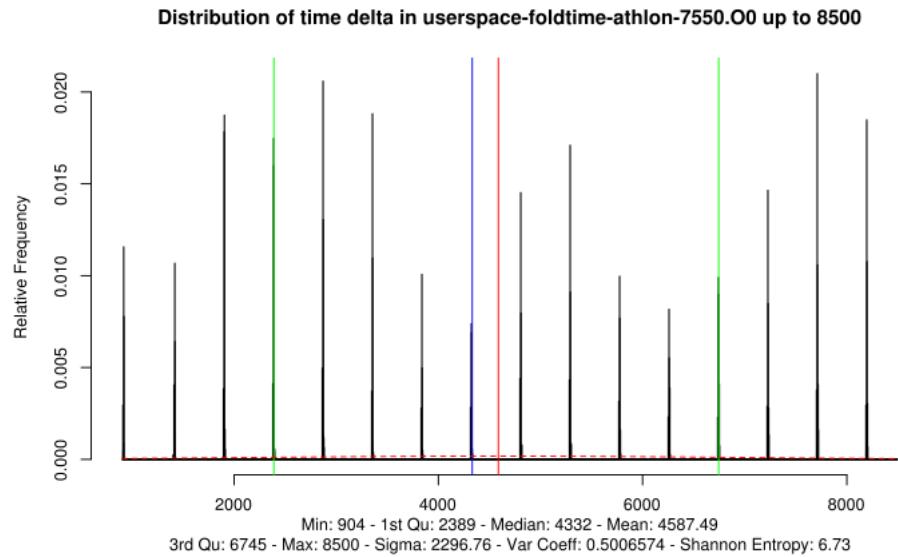
Figure C.54: ~~Upper boundary of entropy over LFSR loop in user space on AMD Athlon 7550 – without optimizations~~

C.24 AMD Athlon 4850e

Distribution of time delta in userspace-foldtime-athlon-4850e up to 3650

Min: 3352 - 1st Qu: 3631 - Median: 3632 - Mean: 3601.82
3rd Qu: 3632 - Max: 3632 - Sigma: 86.47 - Var Coeff: 0.0240079 - Shannon Entropy: 1.23
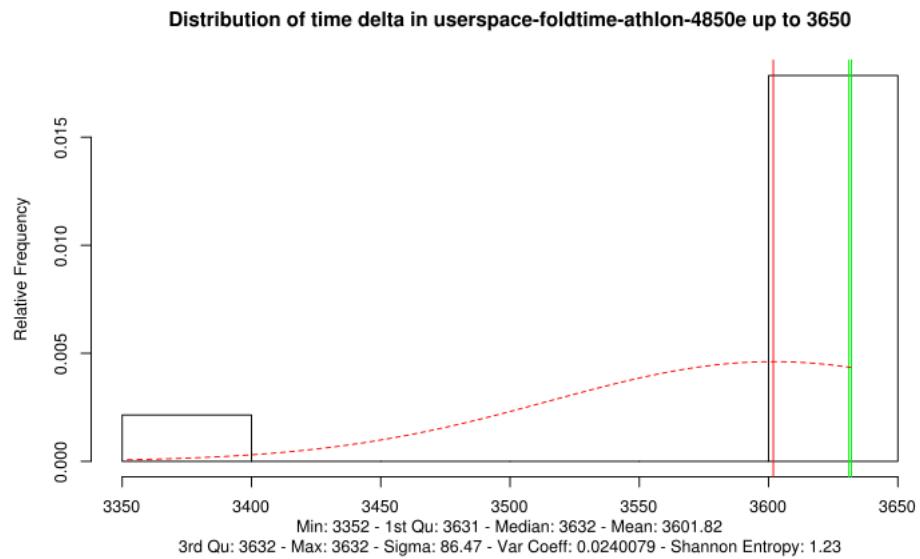
Figure C.55: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on AMD Athlon 4850e – with optimizations

**Distribution of time delta in userspace-foldtime-athlon-4850e up to 6500**

Min: 3352 - 1st Qu: 4190 - Median: 4470 - Mean: 4587.12
3rd Qu: 5029 - Max: 6426 - Sigma: 677.09 - Var Coeff: 0.147606 - Shannon Entropy: 3.84
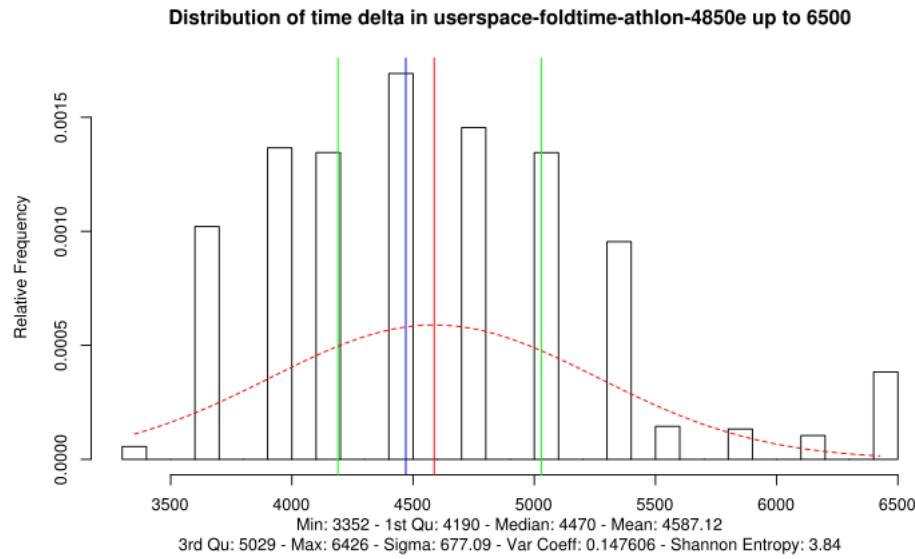
Figure C.56: ~~Upper boundary of entropy over LFSR loop in user space on AMD Athlon 4850e – with optimizations~~

The optimized tests show very low variations, albeit the graphs are slightly misleading as one histogram bar contain up to three consecutive values. The same tests without optimizations show the following results:

Distribution of time delta in userspace-foldtime-athlon-4850e.O0 up to 6500

Min: 3631 - 1st Qu: 3911 - Median: 3911 - Mean: 4137.98
3rd Qu: 3912 - Max: 6426 - Sigma: 676.54 - Var Coeff: 0.1634953 - Shannon Entropy: 2.64
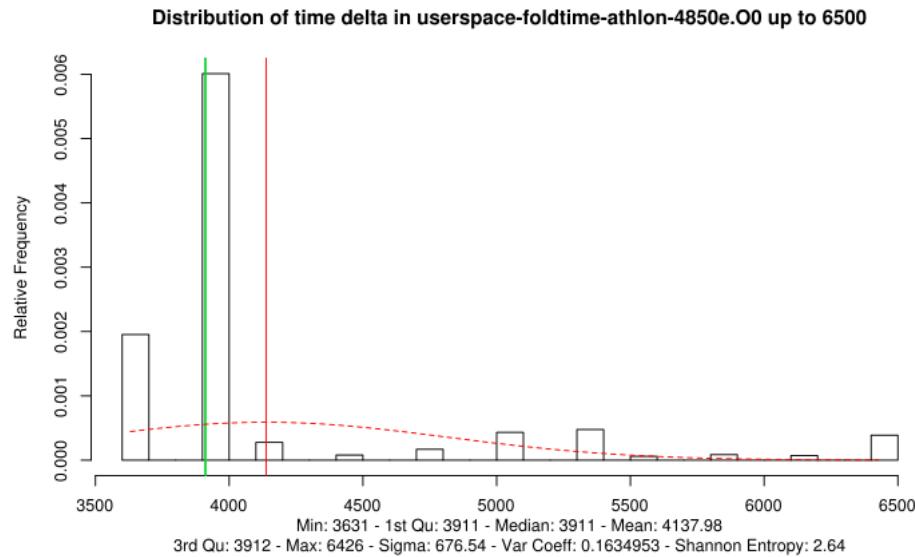
Figure C.57: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on AMD Athlon 4850e – without optimizations

131

Distribution of time delta in userspace-foldtime-athlon-4850e.O0 up to 15000

Min: 3631 - 1st Qu: 5029 - Median: 6426 - Mean: 6416.85
3rd Qu: 7822 - Max: 14807 - Sigma: 1638.04 - Var Coeff: 0.2552724 - Shannon Entropy: 5.00
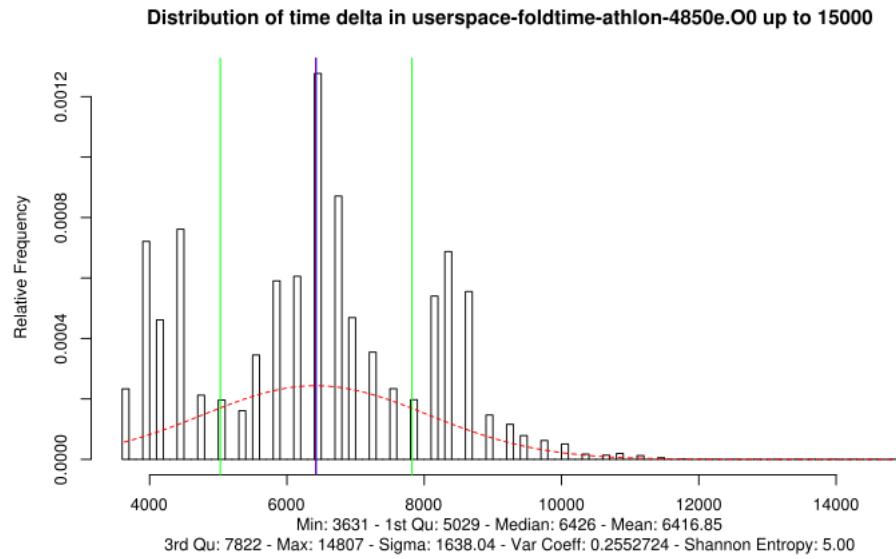
Figure C.58: ~~Upper boundary of entropy over LFSR loop in user space on AMD Athlon 4850e – without optimizations~~

C.25 AMD E350

Distribution of time delta in userspace-foldtime-amde350 up to 520

Min: 495 - 1st Qu: 510 - Median: 510 - Mean: 509.3
3rd Qu: 510 - Max: 520 - Sigma: 3.58 - Var Coeff: 0.0070219 - Shannon Entropy: 1.47
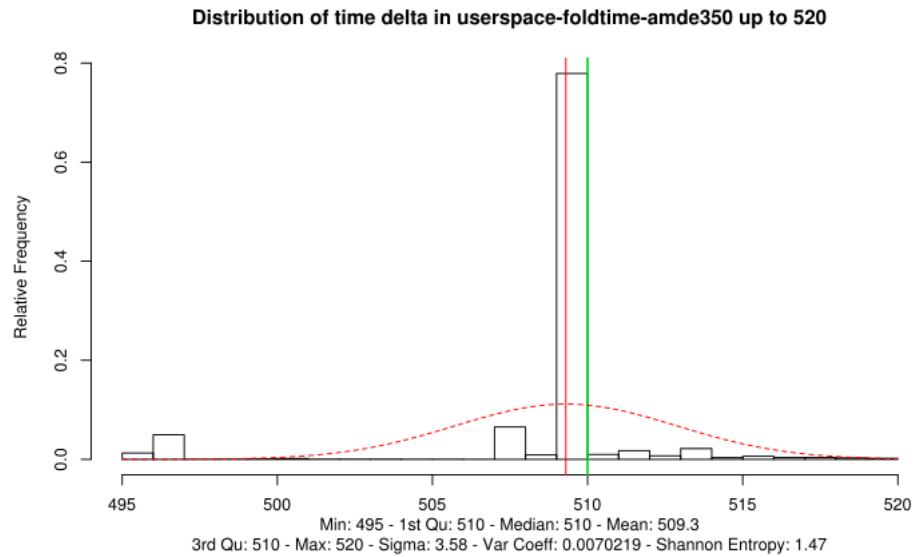
Figure C.59: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on AMD E350 – with optimizations

Distribution of time delta in userspace-foldtime-amde350 up to 4500

Min: 493 - 1st Qu: 1260 - Median: 2015 - Mean: 2159
3rd Qu: 3270 - Max: 4500 - Sigma: 1143.64 - Var Coeff: 0.5297106 - Shannon Entropy: 5.94
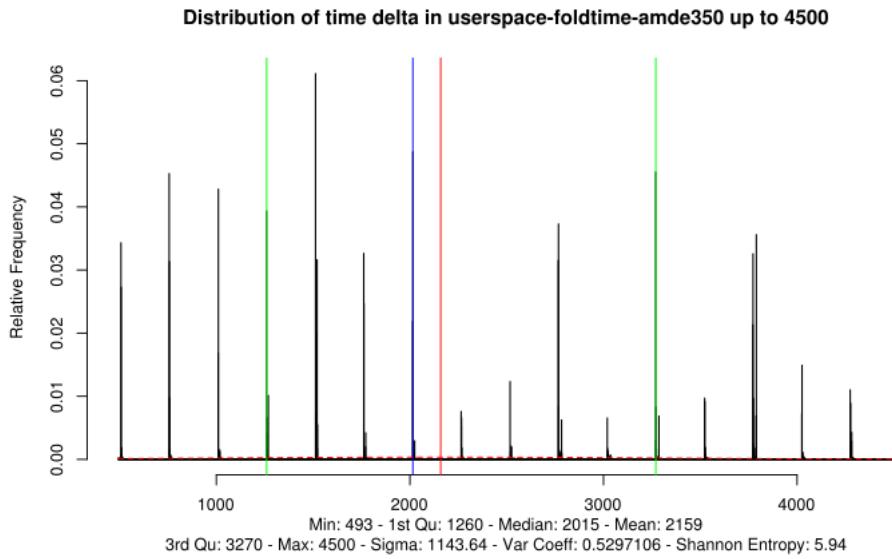
Figure C.60: ~~Upper boundary of entropy over LFSR loop in user space on AMD E350 – with optimizations~~

The same tests without optimizations show the following results:

Distribution of time delta in userspace-foldtime-amde350.O0 up to 1150

Min: 810 - 1st Qu: 1121 - Median: 1134 - Mean: 1126.12
3rd Qu: 1134 - Max: 1150 - Sigma: 16.1 - Var Coeff: 0.0143002 - Shannon Entropy: 2.71
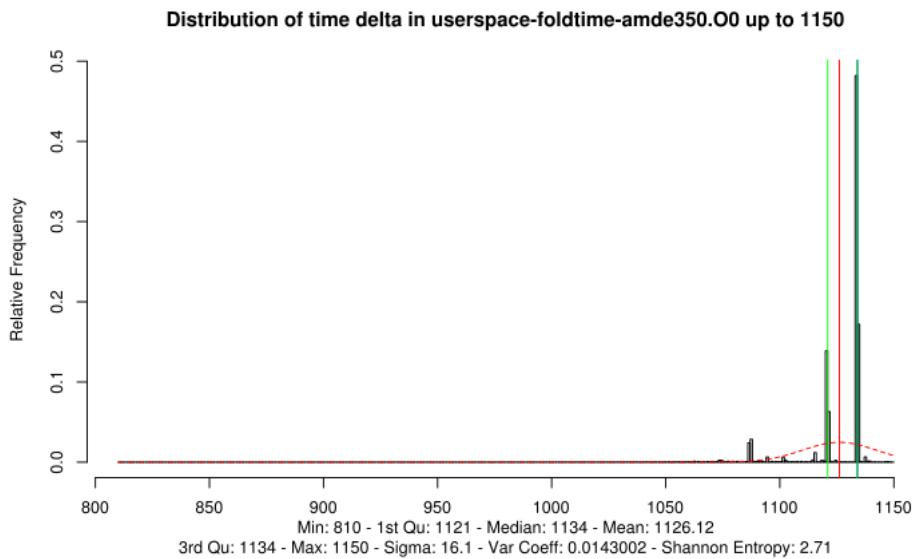
Figure C.61: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on AMD E350 – without optimizations

Distribution of time delta in userspace-foldtime-amde350.O0 up to 13000

Min: 800 - 1st Qu: 3503 - Median: 6542 - Mean: 7076.05
3rd Qu: 10440 - Max: 13000 - Sigma: 3660.21 - Var Coeff: 0.517268 - Shannon Entropy: 6.27
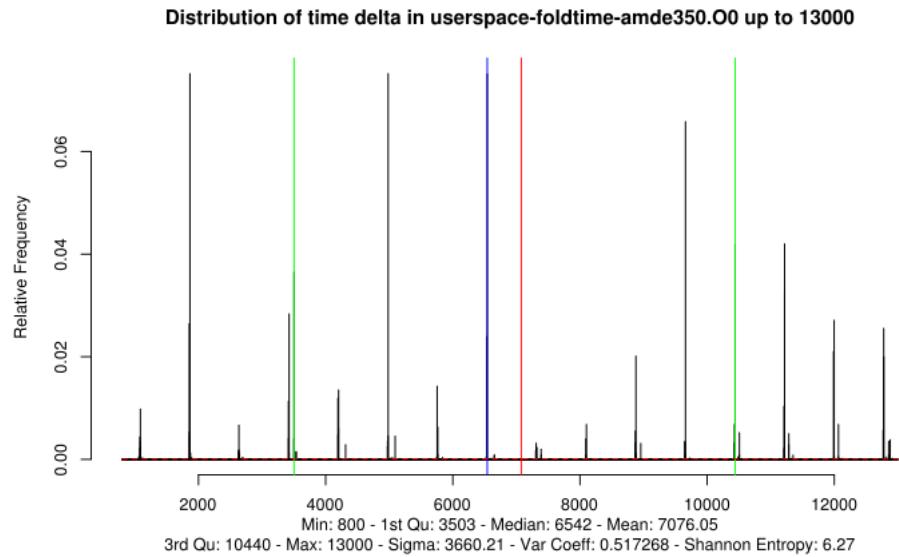
Figure C.62: ~~Upper boundary of entropy over LFSR loop in user space on AMD E350 – without optimizations~~

C.26   AMD Semperon 3GHz

Distribution of time delta in userspace-foldtime-amdsemperon-3GHz up to 770

Min: 744 - 1st Qu: 755 - Median: 755 - Mean: 753.9
3rd Qu: 755 - Max: 770 - Sigma: 2.8 - Var Coeff: 0.0037153 - Shannon Entropy: 0.91
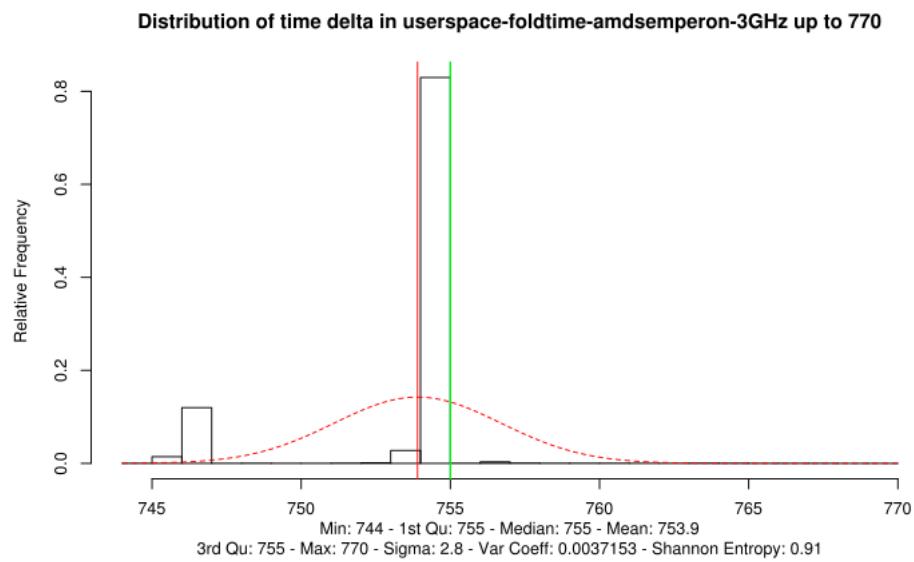
Figure C.63: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on AMD Semperon 3GHz – with optimizations

Distribution of time delta in userspace-foldtime-amdsemperon-3GHz up to 6000

Min: 743 - 1st Qu: 1781 - Median: 3810 - Mean: 3437.01
3rd Qu: 4826 - Max: 6000 - Sigma: 1686.88 - Var Coeff: 0.4907996 - Shannon Entropy: 4.98
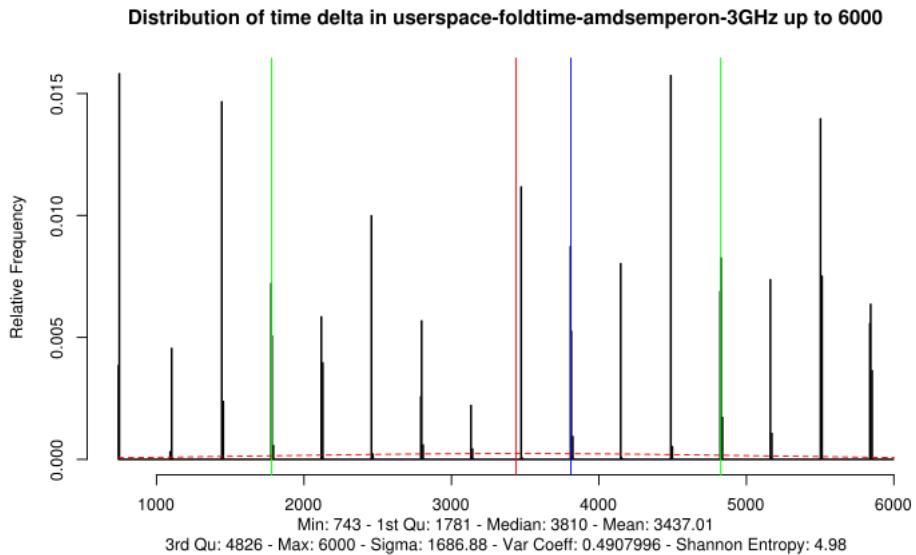
Figure C.64: ~~Upper boundary of entropy over LFSR loop in user space on AMD Semperon 3GHz – with optimizations~~

The graphs show that ~~lower boundary of~~ the CPU timing jitter over the ~~LFSR~~hash operation contains less than 1 bit of entropy. This statement has the potential to significantly weaken the quality of this random number generator. However, as outlined at the beginning, the tests are performed with optimized code. Optimization streamline the code such that the resulting binary does not fully follow the strict C code sequences, but the compiler tries to ensure that the result is always the same. As the quality of the CPU Jitter random number generator depends on the timing behavior and not so much on the result of computations, optimizations are not important for the random number generator.

To ensure that optimizations are the problem of the insufficient execution jitter as the execution time is made too fast on fast, but less complex CPUs, the same test without optimizations is invoked again. To compile code without optimizations, either use no special flags or `-O0`.

Distribution of time delta in userspace-foldtime-amdsemperon-3GHz.O0 up to 1140

Min: 1095 - 1st Qu: 1122 - Median: 1123 - Mean: 1122.37
3rd Qu: 1123 - Max: 1140 - Sigma: 5.29 - Var Coeff: 0.0047128 - Shannon Entropy: 2.40
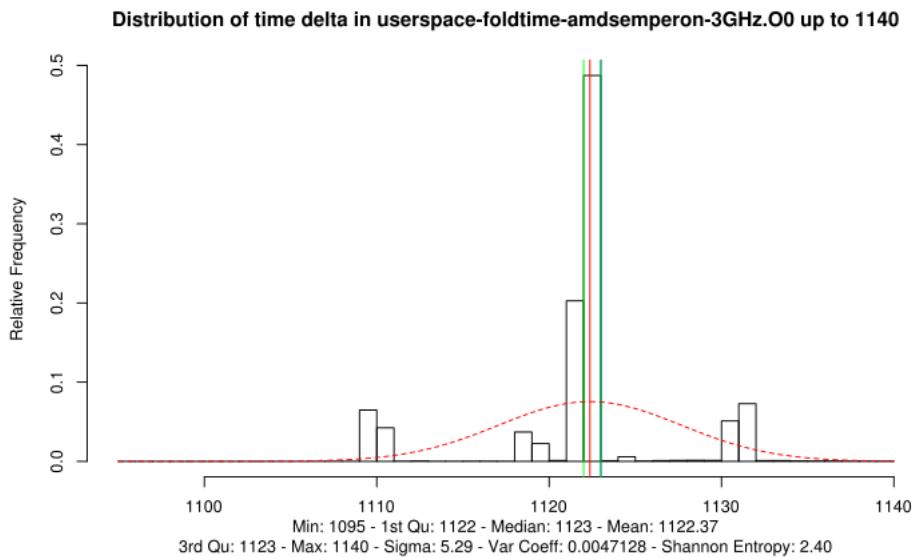
Figure C.65: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on AMD Semperon 3GHz – without optimizations

Distribution of time delta in userspace-foldtime-amdsemperon-3GHz.O0 up to 11500

Min: 1090 - 1st Qu: 3123 - Median: 5779 - Mean: 5817.55
3rd Qu: 9098 - Max: 11500 - Sigma: 3065.26 - Var Coeff: 0.5268986 - Shannon Entropy: 5.97
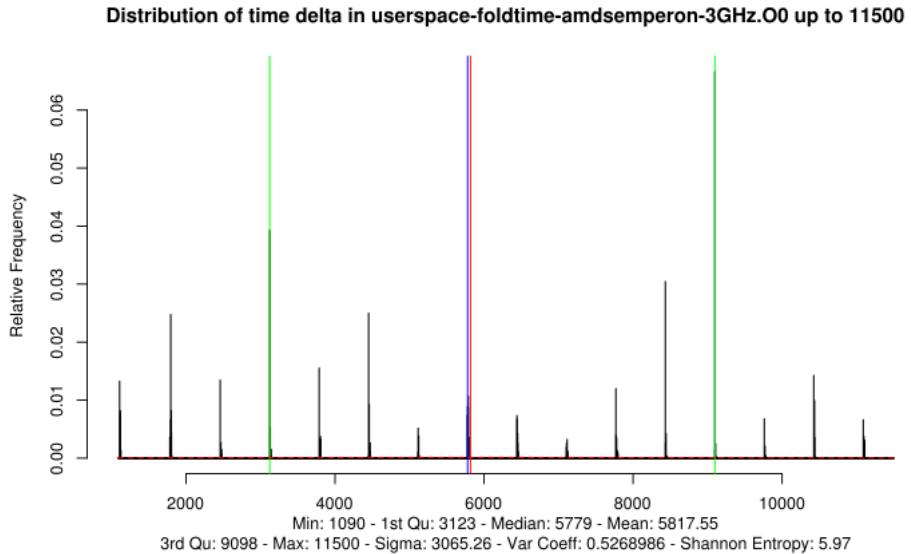
Figure C.66: ~~Upper boundary of entropy over LFSR loop in user space on AMD Semperon 3GHz – without optimizations~~

Looking at the Shannon Entropy value a conclusion can be drawn that without optimizations, the required CPU execution timing jitter is present with a sufficient rate.

136

To support the conclusion that the compilation of non-optimized code on an AMD Semperon still produces high-quality random numbers, the statistical testing with `ent` is performed:

Listing 5: Statistical Properties of Non-Optimized Code on AMD Semperon

```
# byte-wise
$ ent entropy.amdsemperon.00
Entropy = 7.999968 bits per byte.

Optimum compression would reduce the size
of this 5525504 byte file by 0 percent.

Chi square distribution for 5525504 samples is 247.18, and randomly
would exceed this value 50.00 percent of the times.

Arithmetic mean value of data bytes is 127.5252 (127.5 = random).
Monte Carlo value for Pi is 3.142346161 (error 0.02 percent).
Serial correlation coefficient is -0.000274 (totally uncorrelated = 0.0).

# bit-wise
$ ent -b entropy.amdsemperon.00
Entropy = 1.000000 bits per bit.

Optimum compression would reduce the size
of this 44204032 bit file by 0 percent.

Chi square distribution for 44204032 samples is 2.54, and randomly
would exceed this value 25.00 percent of the times.

Arithmetic mean value of data bits is 0.5001 (0.5 = random).
Monte Carlo value for Pi is 3.142346161 (error 0.02 percent).
Serial correlation coefficient is 0.000115 (totally uncorrelated = 0.0).
```

C.27   VIA Nano L2200

Distribution of time delta in userspace-foldtime-viananol2200 up to 1910

Min: 1871 - 1st Qu: 1887 - Median: 1891 - Mean: 1890.08
3rd Qu: 1892 - Max: 1907 - Sigma: 6.67 - Var Coeff: 0.0035312 - Shannon Entropy: 3.04
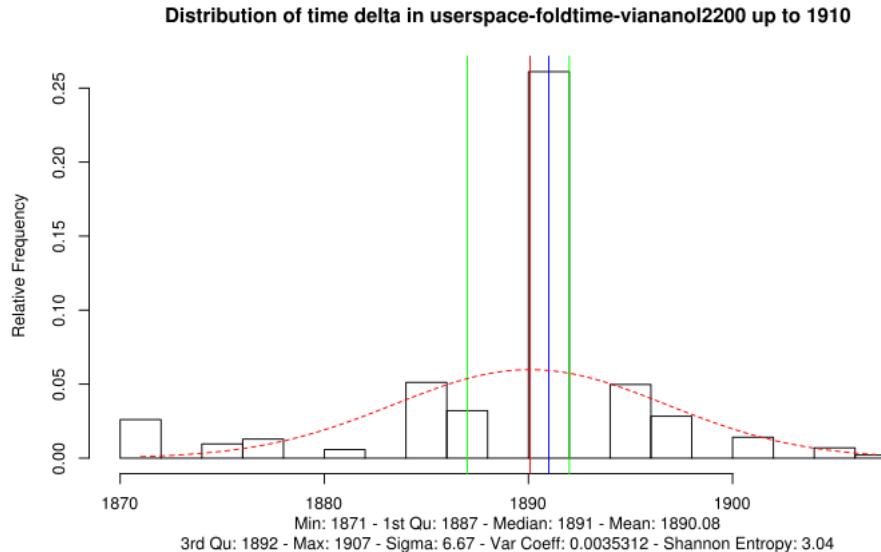
Figure C.67: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on VIA Nano L2200

137

Distribution of time delta in userspace-foldtime-viananol2200 up to 11000

Min: 1866 - 1st Qu: 4258 - Median: 6610 - Mean: 6519.34
3rd Qu: 8368 - Max: 10997 - Sigma: 2561.52 - Var Coeff: 0.3929117 - Shannon Entropy: 6.31
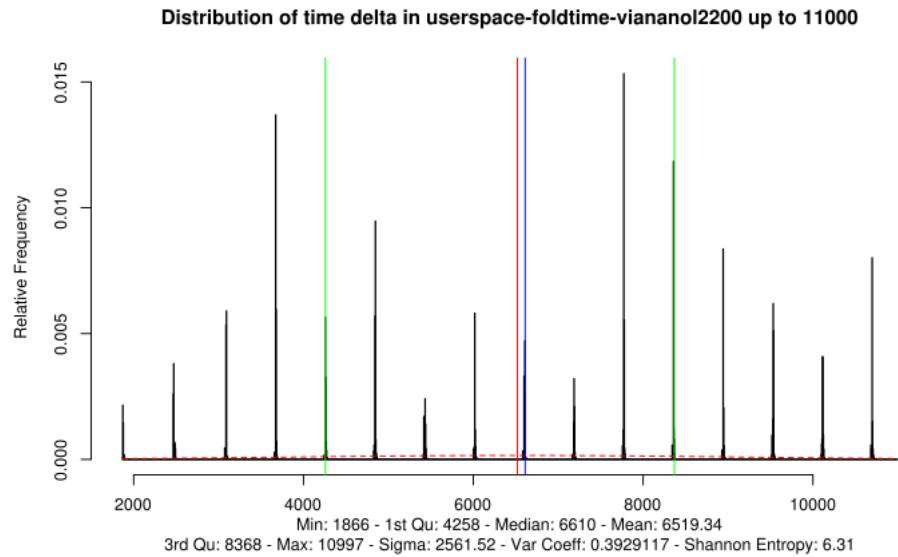
Figure C.68: ~~Upper boundary of entropy over LFSR loop in user space on VIA Nano L2200~~

C.28   MIPS 24KC v7.4

This test was executed on a Ubiquiti NanoStation M5 providing a Freifunk router. The OS on that system is modified with OpenWRT.

Distribution of time delta in userspace-foldtime-mips24Kc-v7.4 up to 4600

Min: 4476 - 1st Qu: 4507 - Median: 4533 - Mean: 4523.18
3rd Qu: 4543 - Max: 4600 - Sigma: 18.4 - Var Coeff: 0.004068 - Shannon Entropy: 3.17
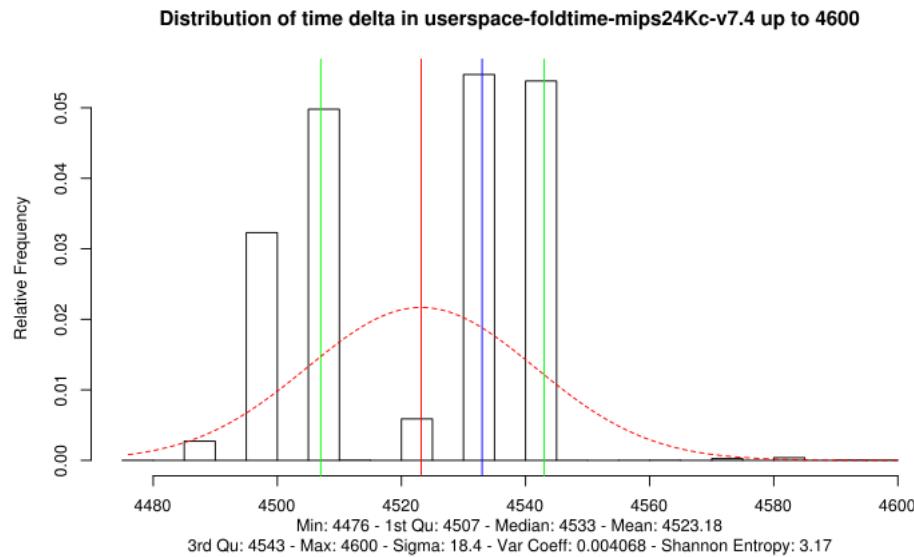
Figure C.69: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on MIPS 24KC v7.4

Distribution of time delta in userspace-foldtime-mips24Kc-v7.4 up to 38000

Min: 4533 - 1st Qu: 6969 - Median: 18795 - Mean: 18999.88
3rd Qu: 28303 - Max: 38000 - Sigma: 11243.43 - Var Coeff: 0.5917632 - Shannon Entropy: 6.83
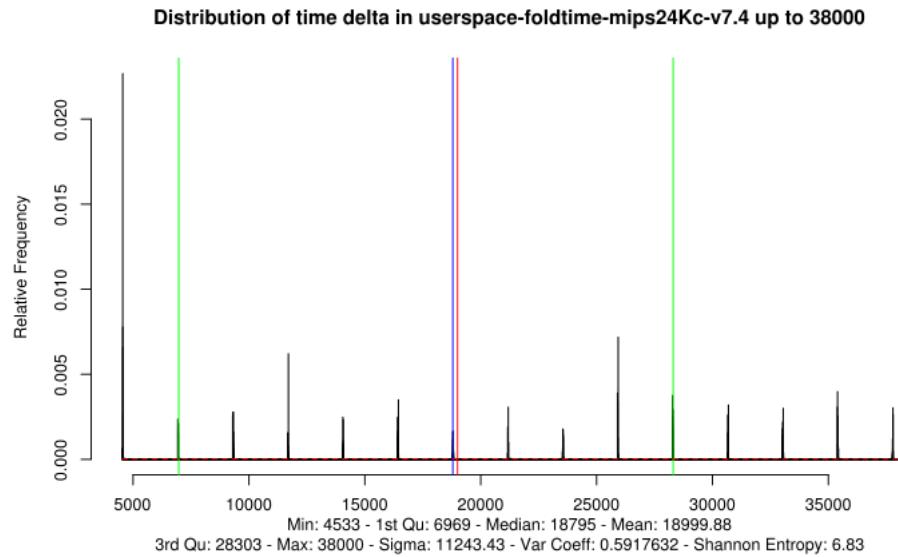
Figure C.70: ~~Upper boundary of entropy over LFSR loop in user space on MIPS 24KC v7.4~~

C.29   MIPS 24KC v4.12 Ikanos Fusiv Core

This test without optimization was executed on a Fritz Box 7390 providing a home router. The OS on that Fritz Box is modified with Freetz.

Distribution of time delta in userspace-foldtime-MIPS24KcV4.12.O0 up to 7500

Min: 7399 - 1st Qu: 7435 - Median: 7436 - Mean: 7434.21
3rd Qu: 7440 - Max: 7500 - Sigma: 15.72 - Var Coeff: 0.0021151 - Shannon Entropy: 3.20
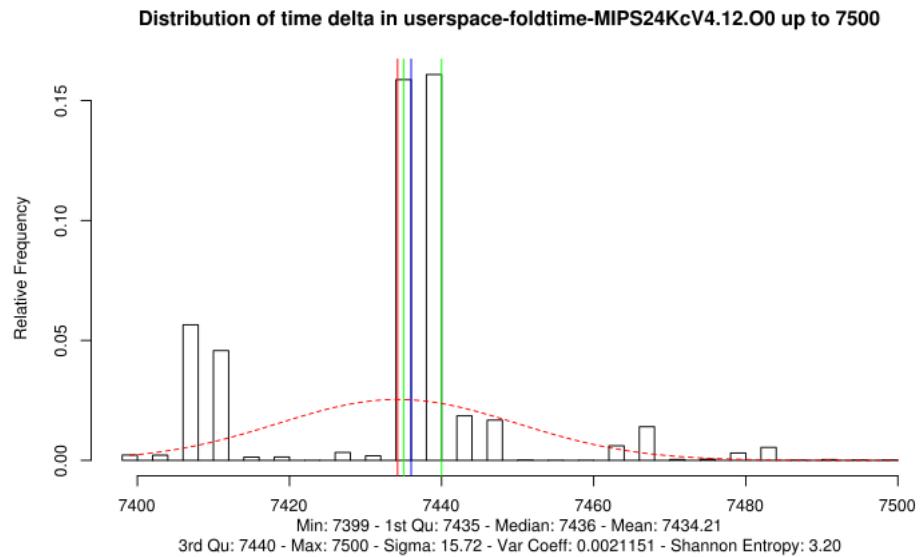
Figure C.71: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space without optimization on MIPS 24KC v4.2

139

Distribution of time delta in userspace-foldtime-MIPS24KcV4.12.O0 up to 85000

Min: 7391 - 1st Qu: 34075 - Median: 55511 - Mean: 52338.7
3rd Qu: 71439 - Max: 84999 - Sigma: 23079.2 - Var Coeff: 0.4409586 - Shannon Entropy: 6.98
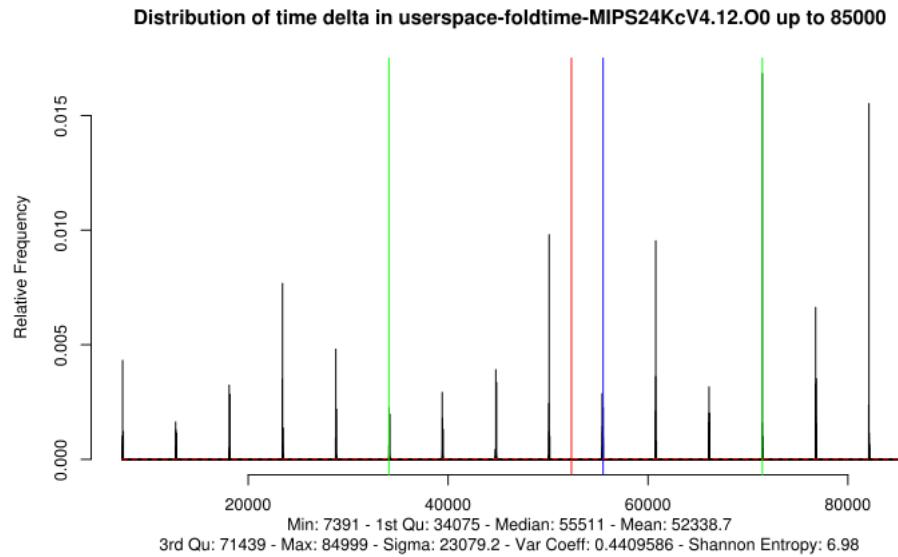
Figure C.72: ~~Upper boundary of entropy over LFSR loop in user space without optimization on MIPS 24KC v7.4~~

C.30 MIPS 4KEc V6.8

This test was executed on a Fritz Box 7270 providing a home router. The OS on that Fritz Box is modified with Freetz.

**Distribution of time delta in userspace-foldtime-mips4Kec-V6.8 up to 5100**

Min: 4999 - 1st Qu: 5000 - Median: 5000 - Mean: 5027.22
3rd Qu: 5067 - Max: 5100 - Sigma: 33.3 - Var Coeff: 0.0066239 - Shannon Entropy: 1.64
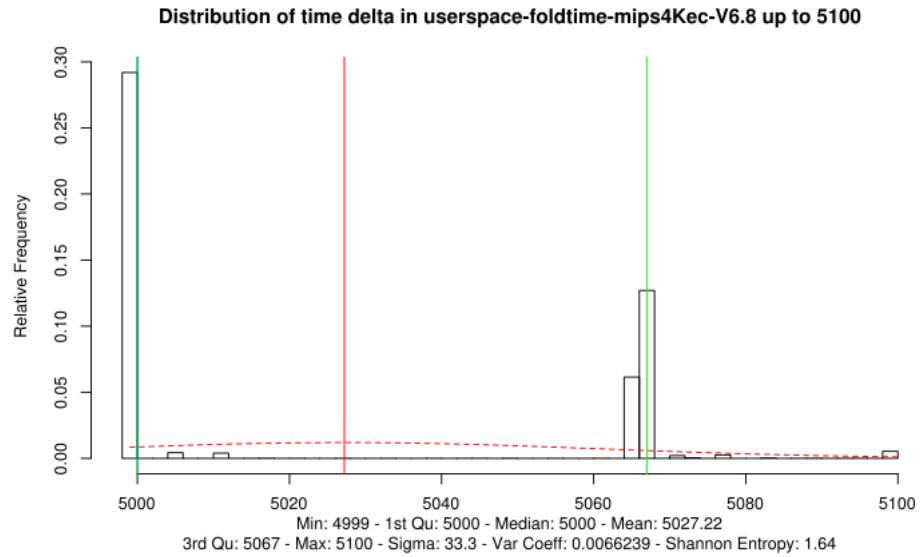
Figure C.73: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on MIPS 4KEc V6.8 – with optimizations

Distribution of time delta in userspace-foldtime-mips4Kec-V6.8 up to 40100

Min: 4999 - 1st Qu: 12162 - Median: 21427 - Mean: 22200.99
3rd Qu: 30767 - Max: 40100 - Sigma: 10447.81 - Var Coeff: 0.4706013 - Shannon Entropy: 6.96
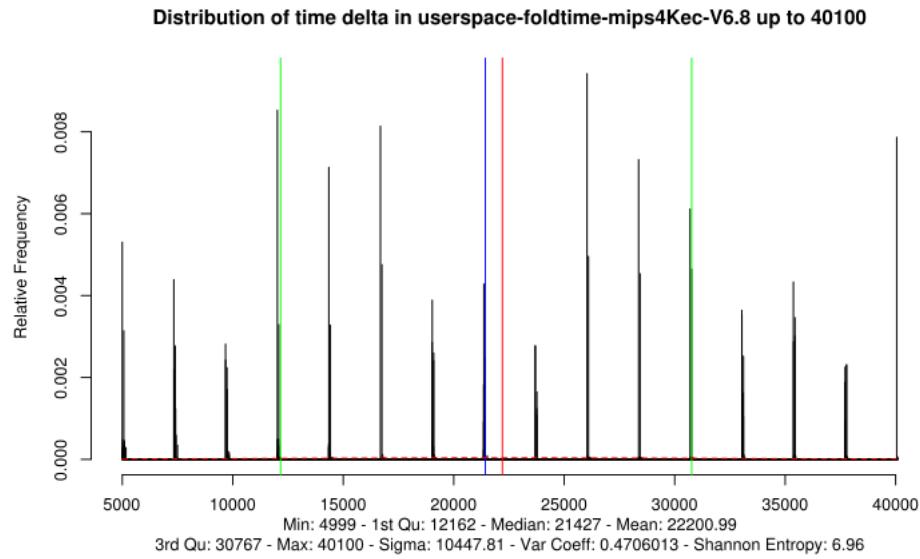
Figure C.74: ~~Upper boundary of entropy over LFSR loop in user space on MIPS 4KEc V6.8 – with optimizations~~

Just to give the reader an impression how the optimization changes the measurement, here is the same CPU measurement without optimization.

Distribution of time delta in userspace-foldtime-mips4Kec-V6.8.O0 up to 12000

Min: 11527 - 1st Qu: 11633 - Median: 11673 - Mean: 11671.63
3rd Qu: 11700 - Max: 12000 - Sigma: 74.57 - Var Coeff: 0.0063887 - Shannon Entropy: 4.16
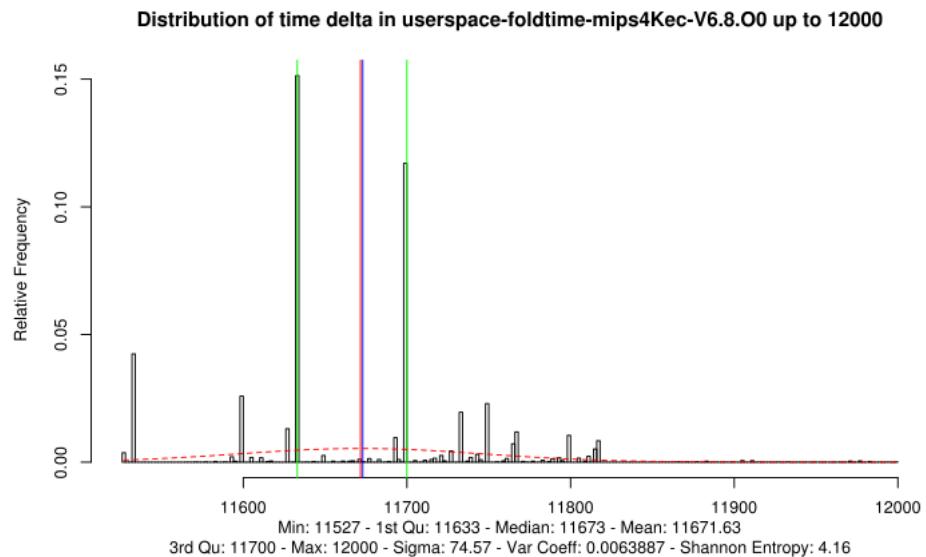
Figure C.75: Entropy ~~Lower boundary of entropy~~ over LFSR ~~loop~~ hash in user space on MIPS 4KEc V6.8 – without optimizations

Distribution of time delta in userspace-foldtime-mips4Kec-V6.8.O0 up to 140000

Min: 11527 - 1st Qu: 37100 - Median: 71133 - Mean: 72527.26
3rd Qu: 104627 - Max: 140000 - Sigma: 37614.48 - Var Coeff: 0.5186254 - Shannon Entropy: 10.20
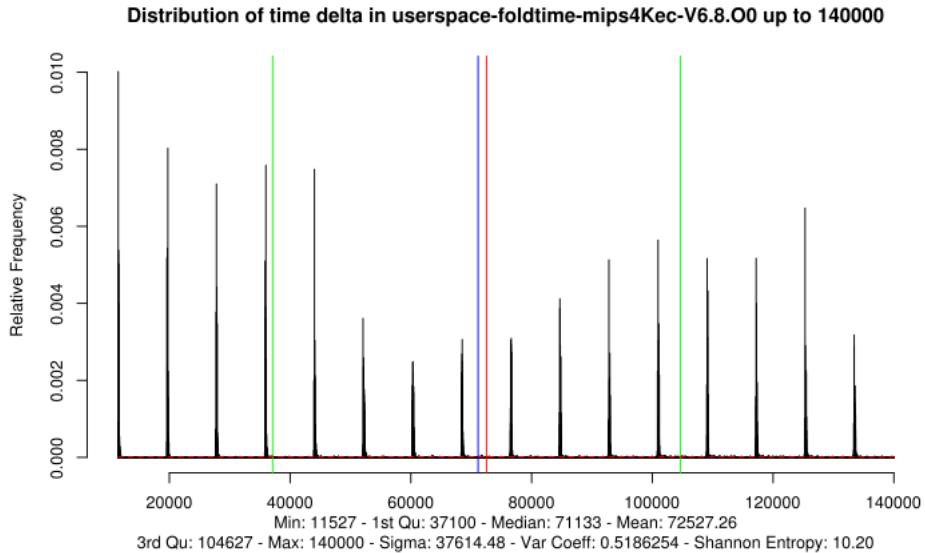
Figure C.76: ~~Upper boundary of entropy over LFSR loop in user space on MIPS 4KEc V6.8 – without optimizations~~

To support the conclusion that this CPU is an appropriate source for entropy, the following statistical analysis was performed. This analysis shows the suitability of the gathered data:

```
# byte-wise
$ ent random.fritz7270.out
Entropy = 7.999997 bits per byte.

Optimum compression would reduce the size
of this 58580496 byte file by 0 percent.

Chi square distribution for 58580496 samples is 241.25, and randomly
would exceed this value 50.00 percent of the times.

Arithmetic mean value of data bytes is 127.4989 (127.5 = random).
Monte Carlo value for Pi is 3.141300135 (error 0.01 percent).
Serial correlation coefficient is 0.000129 (totally uncorrelated = 0.0).

# bit-wise
$ ent -b random.fritz7270.out
Entropy = 1.000000 bits per bit.

Optimum compression would reduce the size
of this 468643968 bit file by 0 percent.

Chi square distribution for 468643968 samples is 0.01, and randomly
would exceed this value 75.00 percent of the times.

Arithmetic mean value of data bits is 0.5000 (0.5 = random).
Monte Carlo value for Pi is 3.141300135 (error 0.01 percent).
Serial correlation coefficient is 0.000042 (totally uncorrelated = 0.0).
```

C.31 MIPS 4KEc V4.8

This test was executed on a T-Com Speedport W701V providing a home router. The OS on that Speedport router is modified with Freetz.

142

The following measurements were conducted without optimizations.
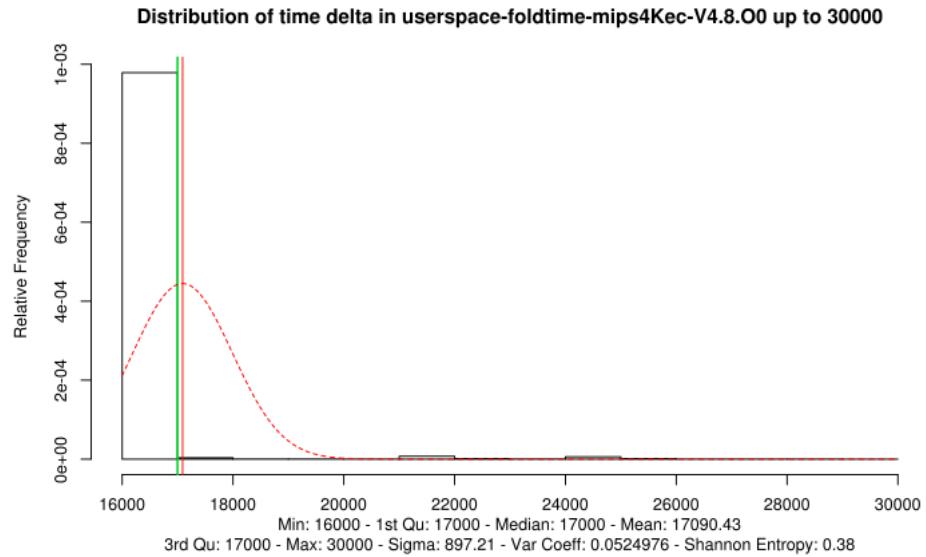
Figure C.77: Entropy Lower boundary of entropy over LFSR hash loop in user space on MIPS 4KEc V4.8 – without optimizations
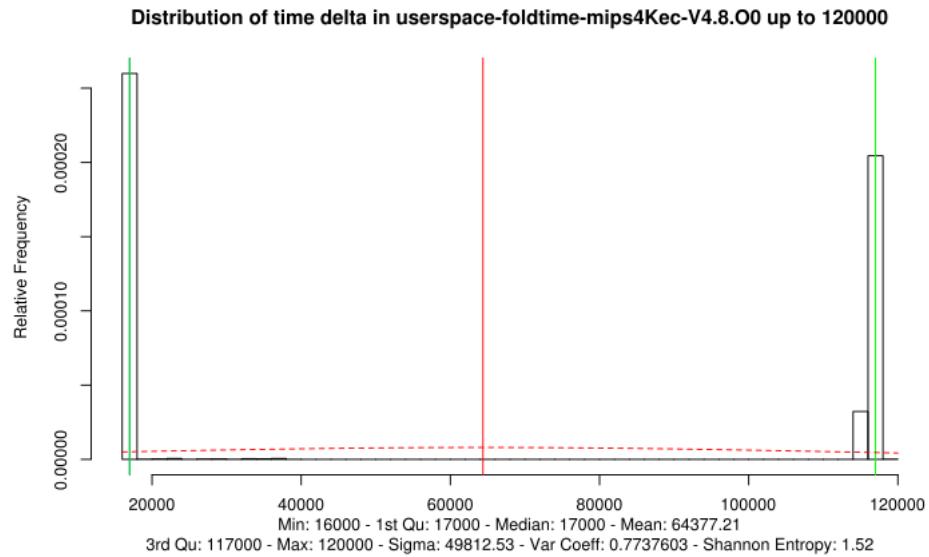

Figure C.78: Upper boundary of entropy over LFSR loop in user space on MIPS 4KEc V4.8 – without optimizations

The graph indicates and the measurement of the Shannon Entropy concludes

that the CPU execution time jitter on this CPU is too small. The reason for that is the coarse counter which increments in multiples of 1,000. However, the good news is that on this CPU, the `jent_entropy_init(3)` call would fail, informing the caller about to not use the CPU Jitter random number generator.

C.32 ARM Exynos 5250 with Fiasco.OC Microkernel

The following measurements were conducted without optimizations.
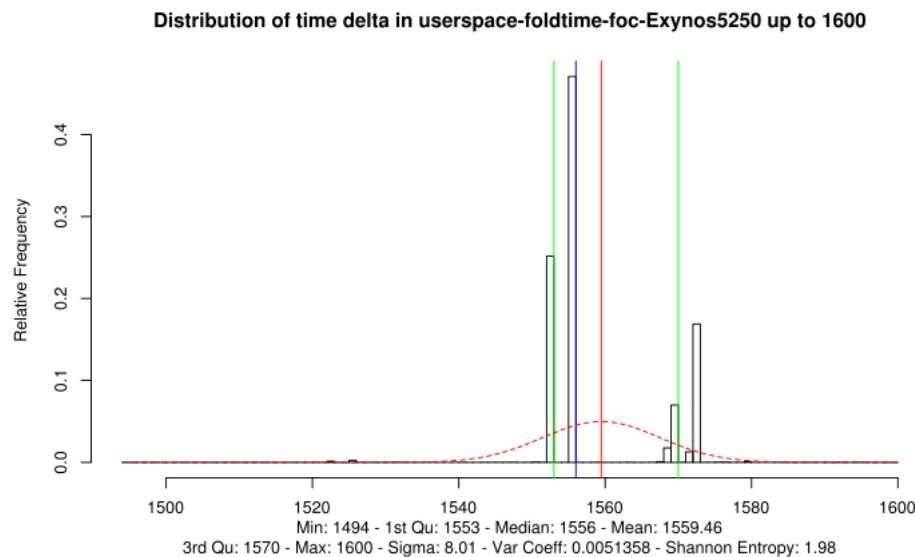
Figure C.79: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Exynos 5250 with Fiasco.OC – without optimizations

Distribution of time delta in userspace-foldtime-foc-Exynos5250 up to 22100

Min: 1502 - 1st Qu: 6868 - Median: 12471 - Mean: 11838.18
3rd Qu: 17943 - Max: 22100 - Sigma: 6317.57 - Var Coeff: 0.5336604 - Shannon Entropy: 6.85
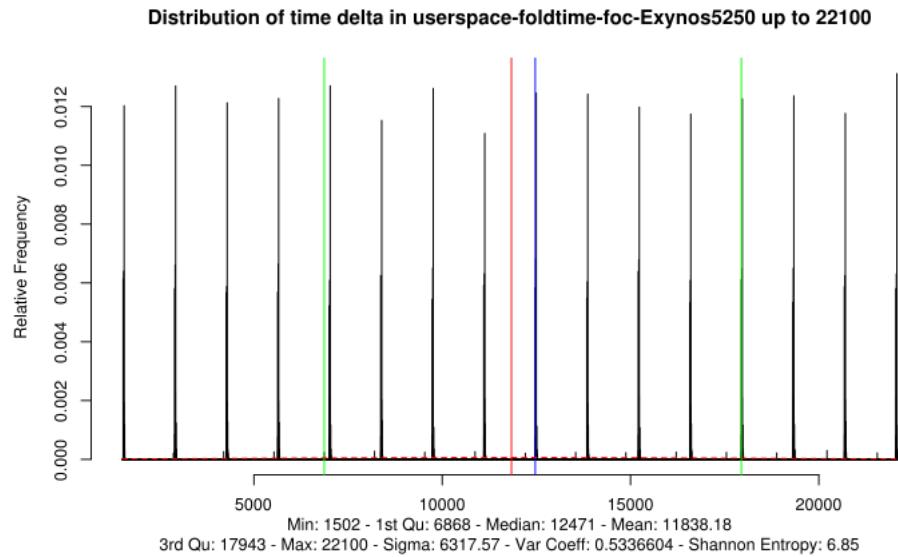
Figure C.80: ~~Upper boundary of entropy over LFSR loop in user space on Exynos 5250 with Fiasco.OC – without optimizations~~

C.33 ARMv7 rev 1 – Samsung Galaxy S2

This test was executed on a Samsung Galaxy S2 with CyanogenMod 9 (Android 4.1). The clocksource (which is the backend to the `clock_gettime(CLOCK_REALTIME)` system call) is:

```
$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
mct-frc
$ cat /sys/devices/system/clocksource/clocksource0/current_clocksource
mct-frc
```

Distribution of time delta in userspace-foldtime-armv7-rev1 up to 3500

Min: 2375 - 1st Qu: 2917 - Median: 2958 - Mean: 2886.38
3rd Qu: 2959 - Max: 3500 - Sigma: 156.64 - Var Coeff: 0.0542687 - Shannon Entropy: 2.79
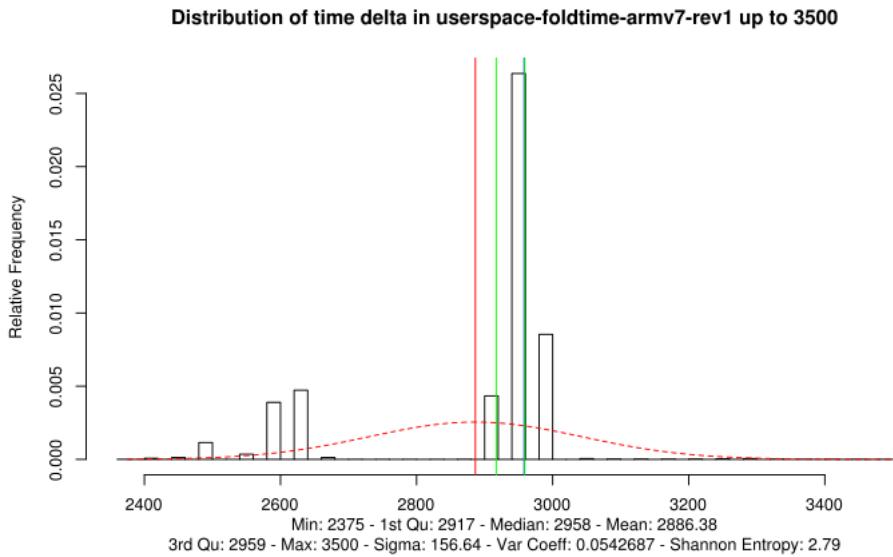
Figure C.81: EntropyLower boundary of entropy over LFSRhash loop in user space on ARMv7 rev 1 – with optimizations

Distribution of time delta in userspace-foldtime-armv7-rev1 up to 23000

Min: 2375 - 1st Qu: 5292 - Median: 8417 - Mean: 8531.41
3rd Qu: 11500 - Max: 23000 - Sigma: 3584.62 - Var Coeff: 0.4201671 - Shannon Entropy: 7.49
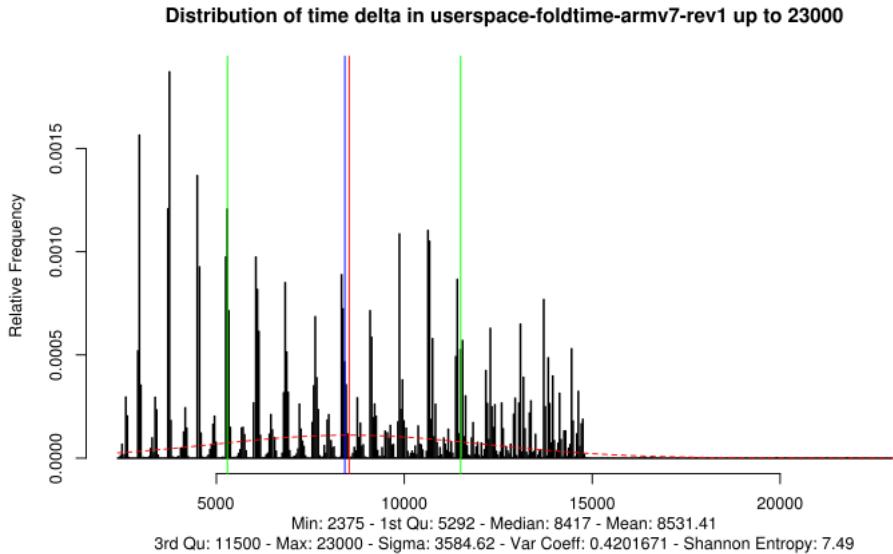
Figure C.82: Upper boundary of entropy over LFSR loop in user space on ARMv7 rev 1 – with optimizations

Although the tests with optimizations already indicate sufficient entropy, the same test without optimizations is conducted with the following results just to illustrate the appropriateness of the entropy source.

Distribution of time delta in userspace-foldtime-armv7-rev1.O0 up to 5200

Min: 3500 - 1st Qu: 4042 - Median: 4083 - Mean: 4045.14
3rd Qu: 4084 - Max: 5167 - Sigma: 116.44 - Var Coeff: 0.0287842 - Shannon Entropy: 2.67
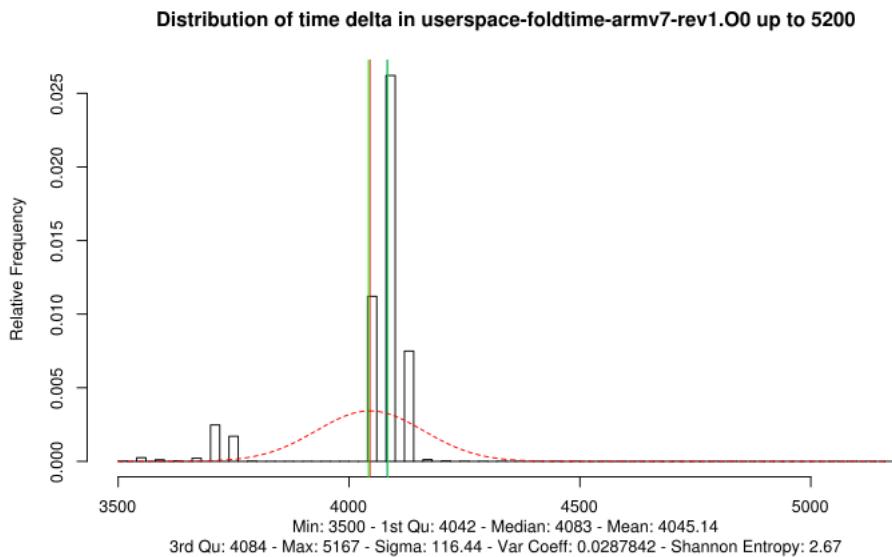
Figure C.83: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on ARMv7 rev 1 – without optimizations

Distribution of time delta in userspace-foldtime-armv7-rev1.O0 up to 45000

Min: 3500 - 1st Qu: 9042 - Median: 17834 - Mean: 16626.98
3rd Qu: 23459 - Max: 45000 - Sigma: 8235.32 - Var Coeff: 0.4952988 - Shannon Entropy: 6.54
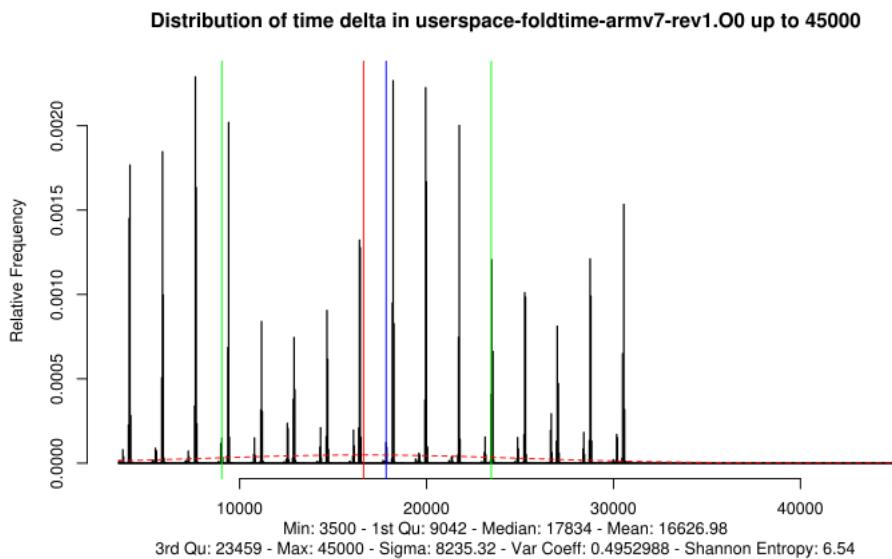
Figure C.84: ~~Upper boundary of entropy over LFSR loop in user space on ARMv7 rev 1 – without optimizations~~

147

C.34 ARMv7 rev 2 – LG Nexus 4.2

The tests on a LG Nexus 4 with a ARMv7 rev 2 CPU and a Linux kernel 3.4
yielded in the following result: most of the time delta values were zero. This
implies that the `jent_entropy_init(3)` call rejects this system.

C.35 ARMv7 rev 0 – Samsung Galaxy S4

The tests on a Samsung Galaxy S4 with a ARMv7 rev 0 CPU and a Linux
kernel 3.4 yielded in the following result: most of the time delta values were
zero. This implies that the `jent_entropy_init(3)` call rejects this system.

The clocksources tested are: gp_timer. When enabling the dg_timer clock-
source, the system reboots after 10 seconds and can therefore not be used.

C.36 ARMv7 rev 1 – HTC Desire Z

The tests on a HTC Desire Z with a ARMv7 rev 1 CPU and a Linux kernel
2.6.32 shows the following results: most of the time delta values were zero. This
implies that the `jent_entropy_init(3)` call rejects this system.

It is unclear whether the coarse timing values is due to an old hardware timer
or whether the Linux system does not support the readout of the high-resolution
timer. The Linux kernel up to version 3.2 did not implement a callback for
`random_get_entropy` on an ARM platform. Therefore it is possible that the
old Android version on the smartphone did not implement access to a potentially
available high-resolution timer.

C.37 ARMv6 rev 7

This test was executed on a Raspberry Pi with Linux kernel 3.6.

Distribution of time delta in userspace-foldtime-ARMv6-rev7 up to 5000

Min: 1999 - 1st Qu: 3000 - Median: 3000 - Mean: 2824.89
3rd Qu: 3000 - Max: 5000 - Sigma: 429.97 - Var Coeff: 0.1522071 - Shannon Entropy: 1.11
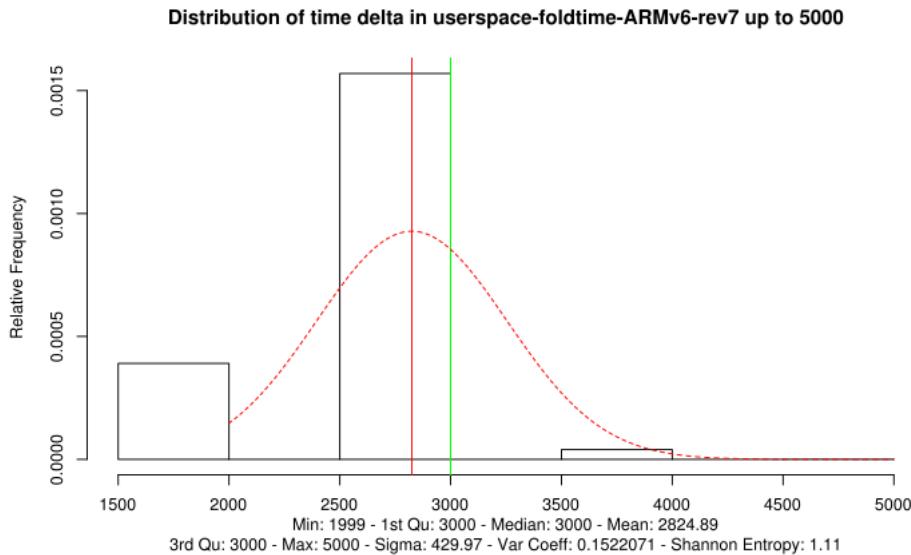
Figure C.85: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on ARMv6 rev 7 – with optimizations

Distribution of time delta in userspace-foldtime-ARMv6-rev7 up to 26000

Min: 1999 - 1st Qu: 6000 - Median: 12000 - Mean: 12009.81
3rd Qu: 17000 - Max: 26000 - Sigma: 6251.62 - Var Coeff: 0.5205426 - Shannon Entropy: 5.03
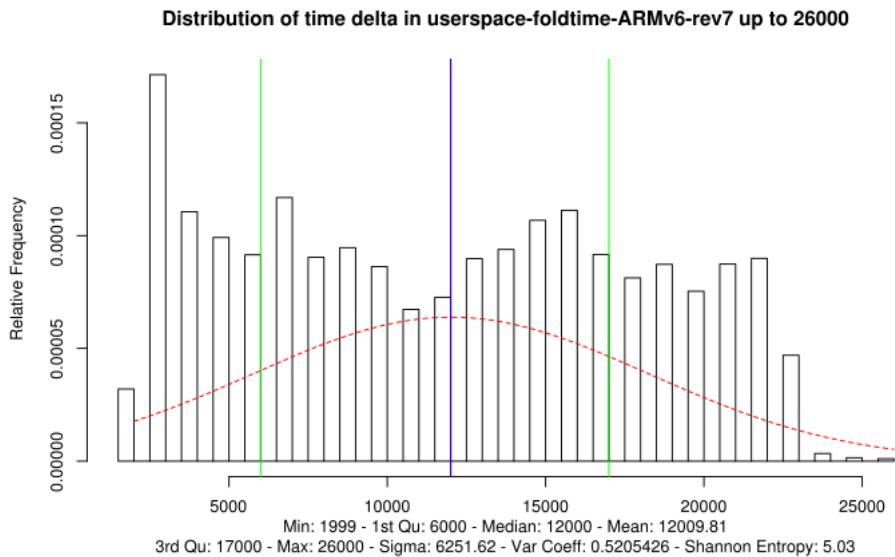
Figure C.86: ~~Upper boundary of entropy over LFSR loop in user space on ARMv6 rev 7 – with optimizations~~

Just to give the reader an impression how the optimization changes the measurement and to demonstrate that without optimizations the entropy is higher, here is the same CPU measurement without optimization.

149

Distribution of time delta in userspace-foldtime-ARMv6-rev7.O0 up to 9000

Min: 5999 - 1st Qu: 6000 - Median: 6999 - Mean: 6535.91
3rd Qu: 7000 - Max: 9000 - Sigma: 521.92 - Var Coeff: 0.0798546 - Shannon Entropy: 1.54
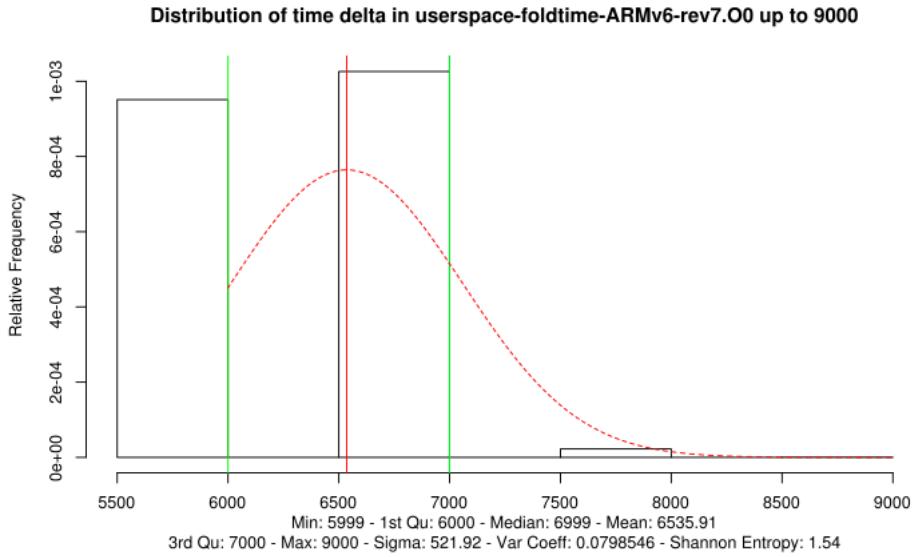
Figure C.87: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on ARMv6 rev 7 – without optimizations

Distribution of time delta in userspace-foldtime-ARMv6-rev7.O0 up to 90000

Min: 5999 - 1st Qu: 19999 - Median: 40999 - Mean: 40373.15
3rd Qu: 58999 - Max: 90000 - Sigma: 21882.58 - Var Coeff: 0.5420083 - Shannon Entropy: 5.76
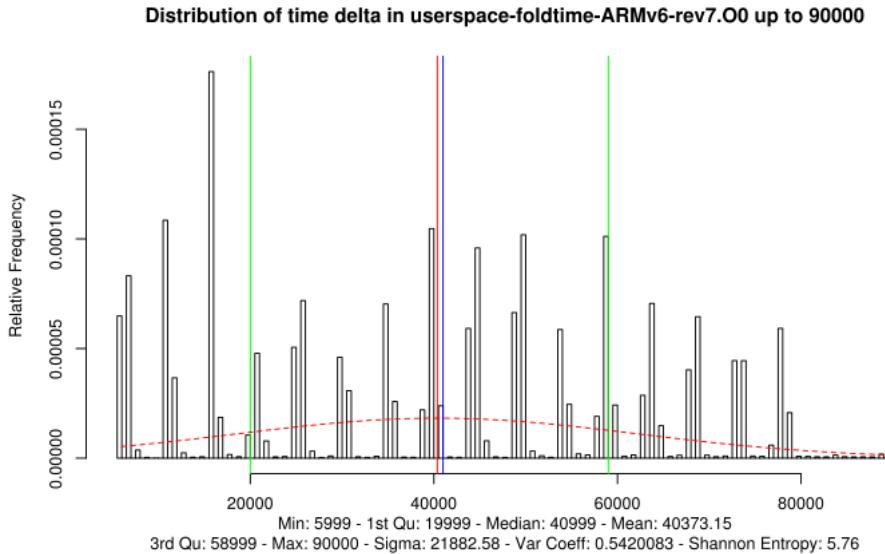
Figure C.88: ~~Upper boundary of entropy over LFSR loop in user space on ARMv6 rev 7 – without optimizations~~

Even though the Shannon Entropy would allow the CPU execution jitter to be used, the timer is too coarse and `jent_entropy_init` does not pass this CPU, because the timer is too coarse as it increments in steps of 1,000. As it is

150

visible in the graphs with the lower boundaries, the majority of entropy comes from two values for the time delta.

C.38 IBM POWER7 with AIX 6.1

The tests with AIX 6.1 executing within an IBM LPAR on an IBM POWER7 CPU and the code obtaining the timer with the POSIX function call of `clock_gettime` yielded the following result: the time delta values were all divisible by 1,000. This implies that the `jent_entropy_init(3)` call rejects this system.

However, AIX provides a second function to read a high-resolution timer: `read_real_time`. When using this function – which is the case as defined in `jitterentropy-base-user.h` – returns a time stamp that is fine grained with the following graphs.


Distribution of time delta in userspace-foldtime-p7-aix61 up to 115

Min: 96 - 1st Qu: 105 - Median: 108 - Mean: 107.82
3rd Qu: 111 - Max: 115 - Sigma: 3.74 - Var Coeff: 0.0347013 - Shannon Entropy: 3.66
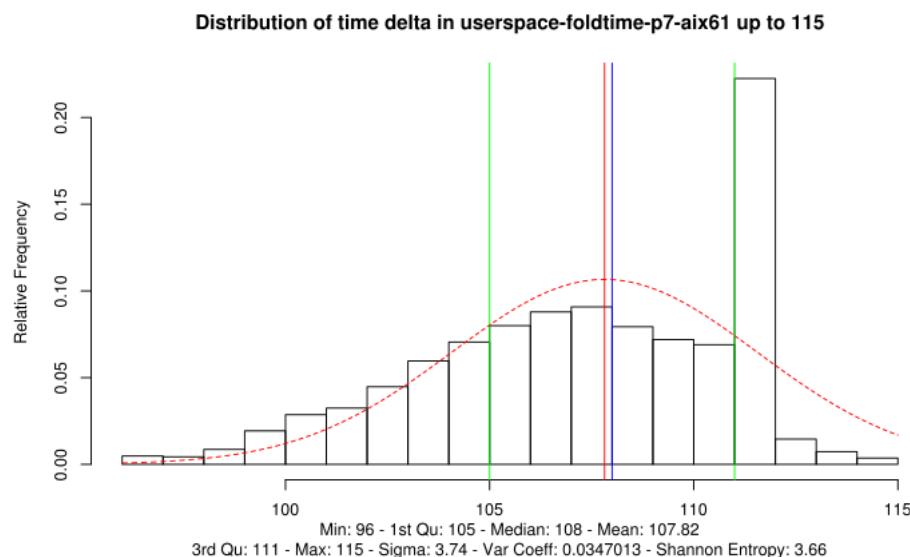
Figure C.89: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on AIX 6.1 and POWER7 – with optimizations

151

Distribution of time delta in userspace-foldtime-p7-aix61 up to 1300

Min: 96 - 1st Qu: 185 - Median: 263 - Mean: 469.12
3rd Qu: 761 - Max: 1300 - Sigma: 345.15 - Var Coeff: 0.7357309 - Shannon Entropy: 7.26
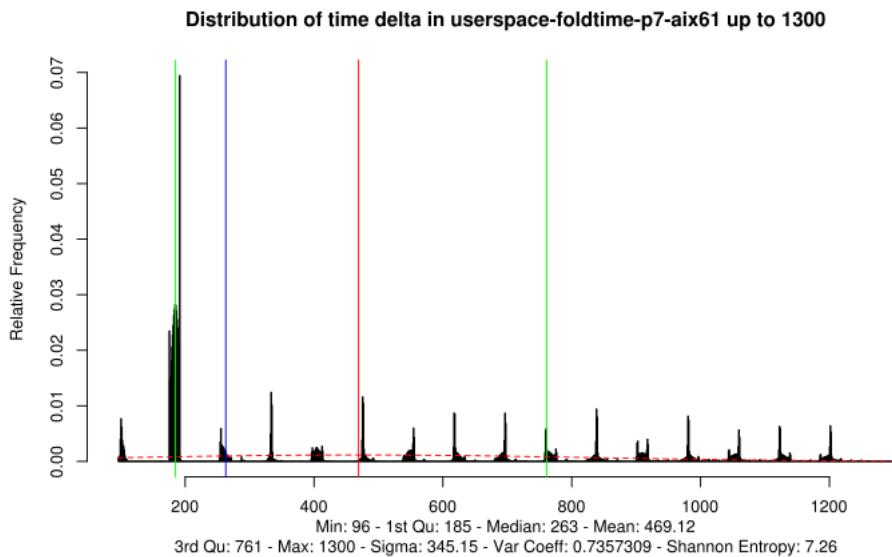
Figure C.90: ~~Upper boundary of entropy over LFSR loop in user space on AIX 6.1 and POWER7 – with optimizations~~

Just to give the reader an impression how the optimization changes the measurement and to demonstrate that without optimizations the entropy is higher, here is the same CPU measurement without optimization.


Distribution of time delta in userspace-foldtime-p7-aix61.O0 up to 1160

Min: 560 - 1st Qu: 982 - Median: 1003 - Mean: 996.23
3rd Qu: 1066 - Max: 1160 - Sigma: 103.11 - Var Coeff: 0.1034971 - Shannon Entropy: 7.77
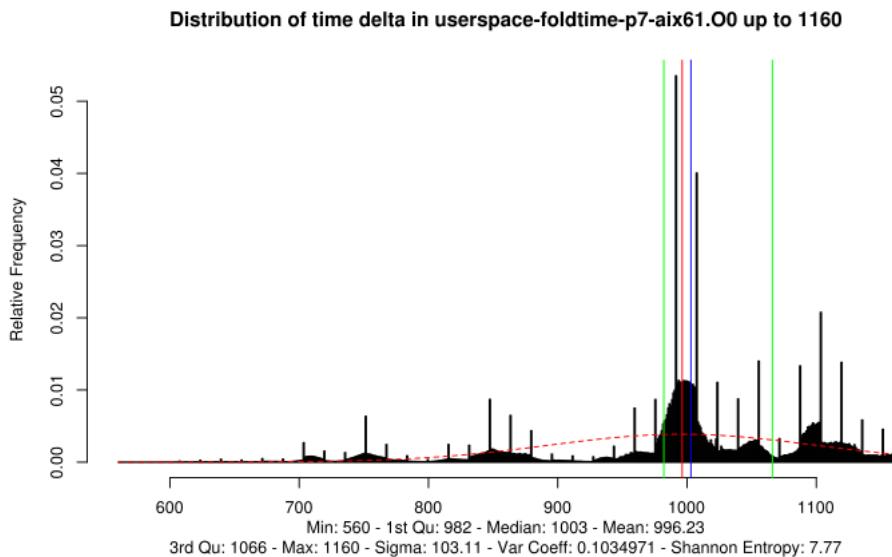
Figure C.91: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on AIX 6.1 and POWER7 – without optimizations

152

Distribution of time delta in userspace-foldtime-p7-aix61.O0 up to 15000

Min: 606 - 1st Qu: 1910 - Median: 2082 - Mean: 4833.93
3rd Qu: 7755 - Max: 15000 - Sigma: 4045.29 - Var Coeff: 0.8368543 - Shannon Entropy: 11.45
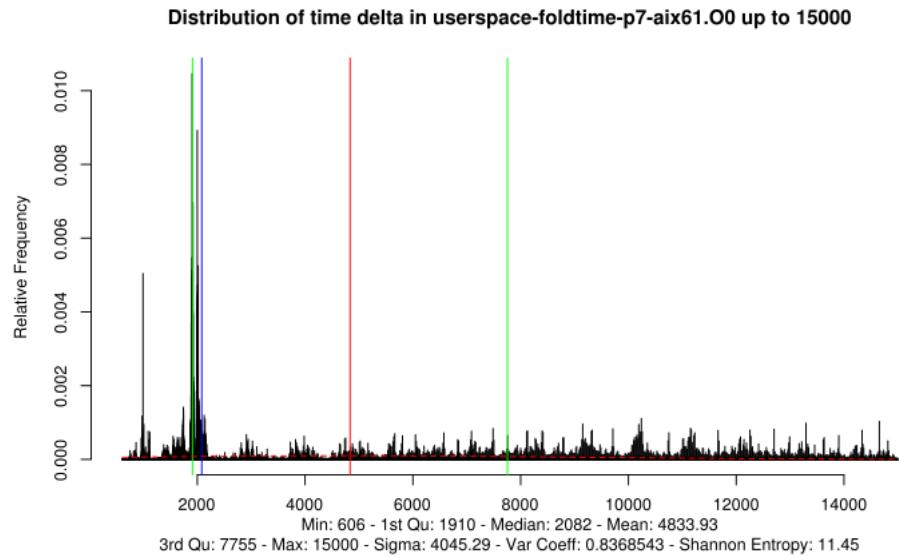
Figure C.92: ~~Upper boundary of entropy over LFSR loop in user space on AIX 6.1 and POWER7 – without optimizations~~

C.39   IBM POWER7 with Linux

Distribution of time delta in userspace-foldtime-power7 up to 180

Min: 122 - 1st Qu: 133 - Median: 141 - Mean: 140.81
3rd Qu: 148 - Max: 180 - Sigma: 9.25 - Var Coeff: 0.0656952 - Shannon Entropy: 4.89
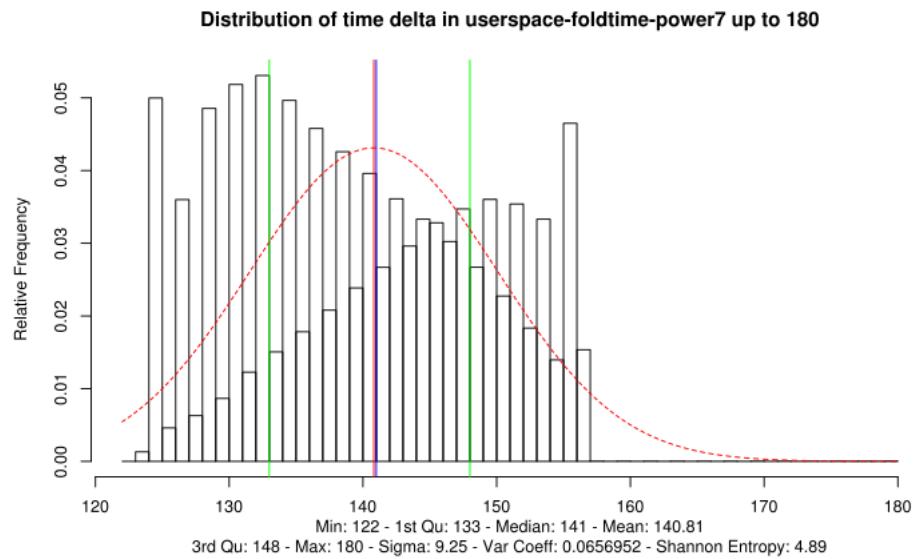
Figure C.93: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on IBM POWER7

Distribution of time delta in userspace-foldtime-power7 up to 1900

Min: 122 - 1st Qu: 420 - Median: 715 - Mean: 691.25
3rd Qu: 974 - Max: 1900 - Sigma: 323.9 - Var Coeff: 0.4685774 - Shannon Entropy: 8.23
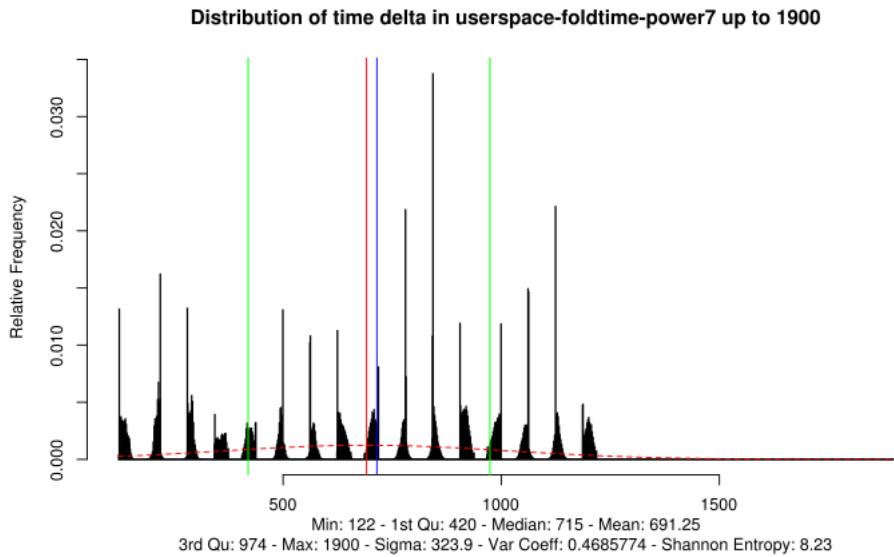
Figure C.94: ~~Upper boundary of entropy over LFSR loop in user space on IBM POWER7~~

C.40 IBM POWER5 with Linux

**Distribution of time delta in userspace-foldtime-power5 up to 370**

Min: 353 - 1st Qu: 363 - Median: 363 - Mean: 364.36
3rd Qu: 367 - Max: 369 - Sigma: 2.15 - Var Coeff: 0.0058944 - Shannon Entropy: 1.43
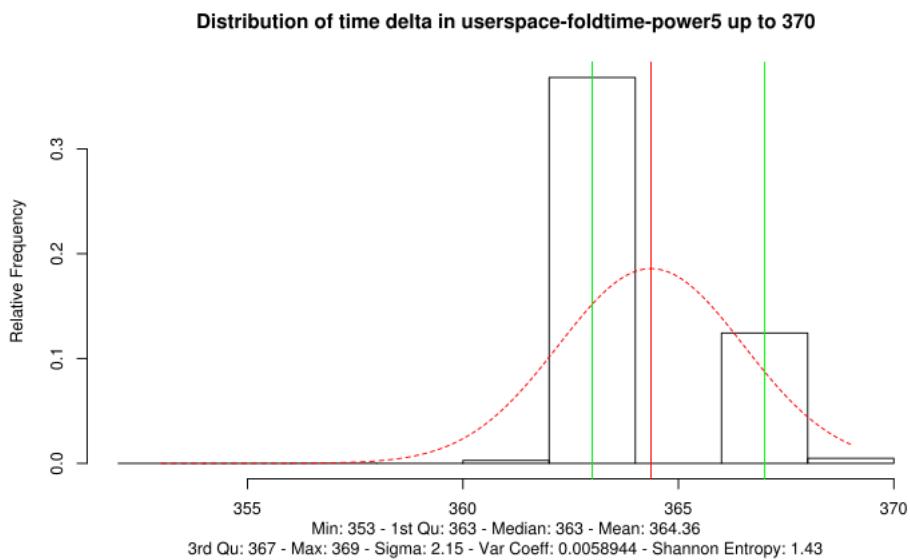
Figure C.95: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on IBM POWER5

154

Distribution of time delta in userspace-foldtime-power5 up to 3000

Min: 354 - 1st Qu: 973 - Median: 1607 - Mean: 1661.87
3rd Qu: 2353 - Max: 2997 - Sigma: 798.73 - Var Coeff: 0.4806193 - Shannon Entropy: 5.73
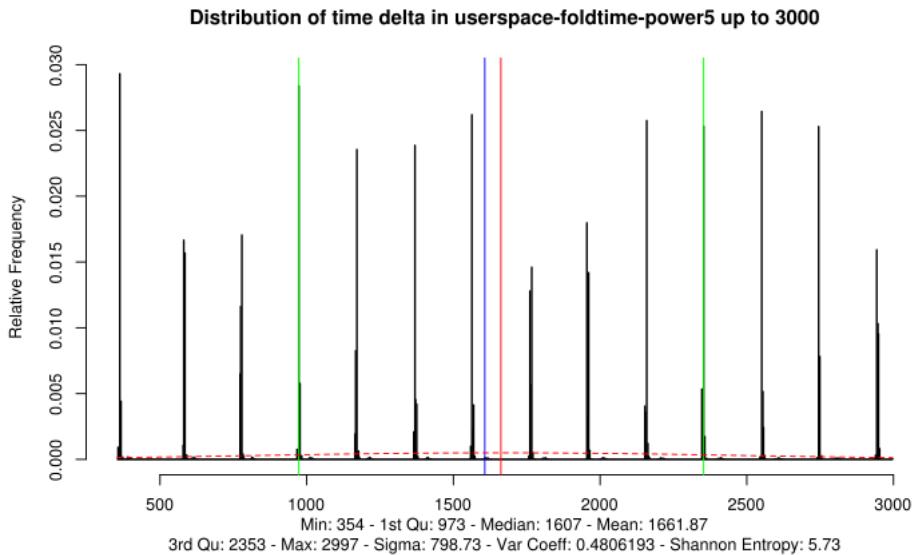
Figure C.96: ~~Upper boundary of entropy over LFSR loop in user space on IBM POWER5~~

C.41 Apple G5 QuadCore PPC 970MP

The following tests were executed on an Apple G5 Quad-Core with PowerPC 970MP CPU and Apple MacOS X 10.5.8. The word size of the tests is 32 bit. However, the 64 bit word size show similar results as indicated in the table at the beginning of this appendix.

The tests show that the optimized compilation contain insufficient jitter as the Shannon Entropy of the lower boundary is less than 1 bit, whereas the non-optimized compilation shows a sufficient jitter.

155

Distribution of time delta in userspace-foldtime-ppc-G5-macos-10.5.8 up to 20

Min: 16 - 1st Qu: 17 - Median: 18 - Mean: 17.74
3rd Qu: 18 - Max: 20 - Sigma: 0.47 - Var Coeff: 0.0266687 - Shannon Entropy: 0.96
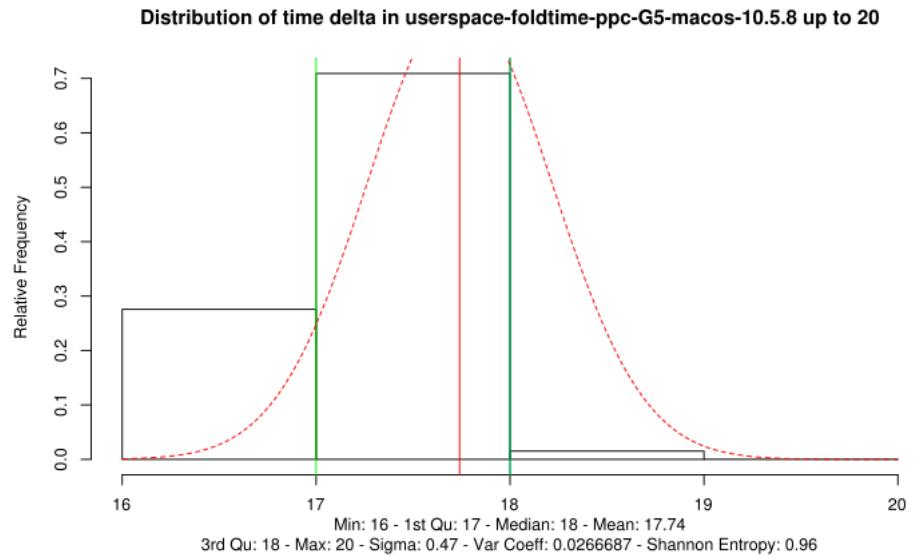
Figure C.97: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Apple G5 with optimizations

**Distribution of time delta in userspace-foldtime-ppc-G5-macos-10.5.8 up to 300**

Min: 17 - 1st Qu: 82 - Median: 147 - Mean: 141.3
3rd Qu: 211 - Max: 300 - Sigma: 75.66 - Var Coeff: 0.5354515 - Shannon Entropy: 6.20
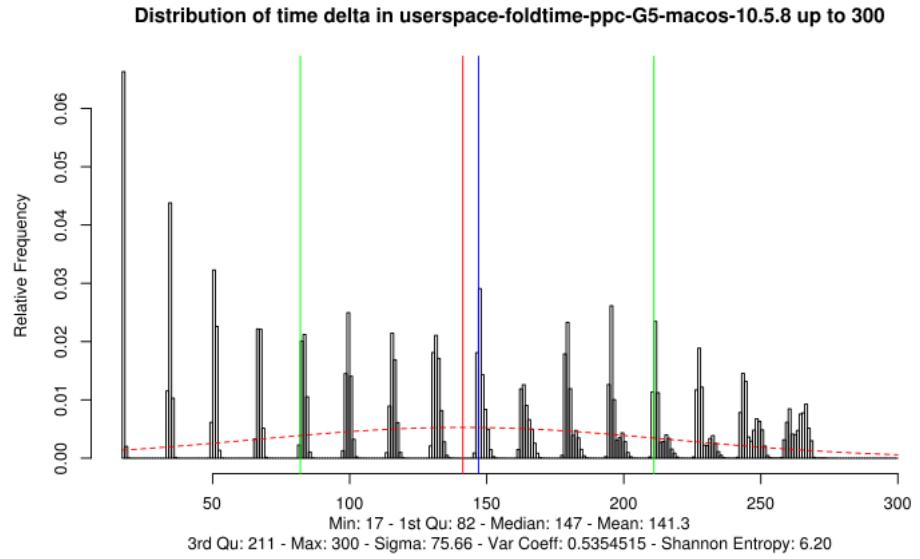
Figure C.98: ~~Upper boundary of entropy over LFSR loop in user space on on Apple G5 with optimizations~~

156

Distribution of time delta in userspace-foldtime-ppc-G5-macos-10.5.8.O0 up to 165

Min: 143 - 1st Qu: 159 - Median: 160 - Mean: 160.47
3rd Qu: 162 - Max: 165 - Sigma: 1.49 - Var Coeff: 0.0092899 - Shannon Entropy: 2.25
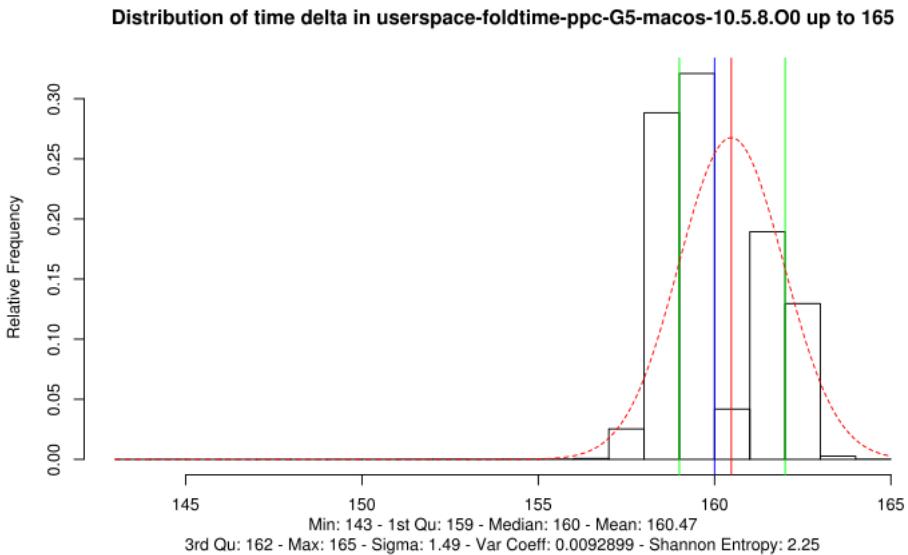
Figure C.99: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Apple G5 without optimizations

Distribution of time delta in userspace-foldtime-ppc-G5-macos-10.5.8.O0 up to 2500

Min: 145 - 1st Qu: 611 - Median: 1237 - Mean: 1228.86
3rd Qu: 1815 - Max: 2500 - Sigma: 677.79 - Var Coeff: 0.5515569 - Shannon Entropy: 6.80
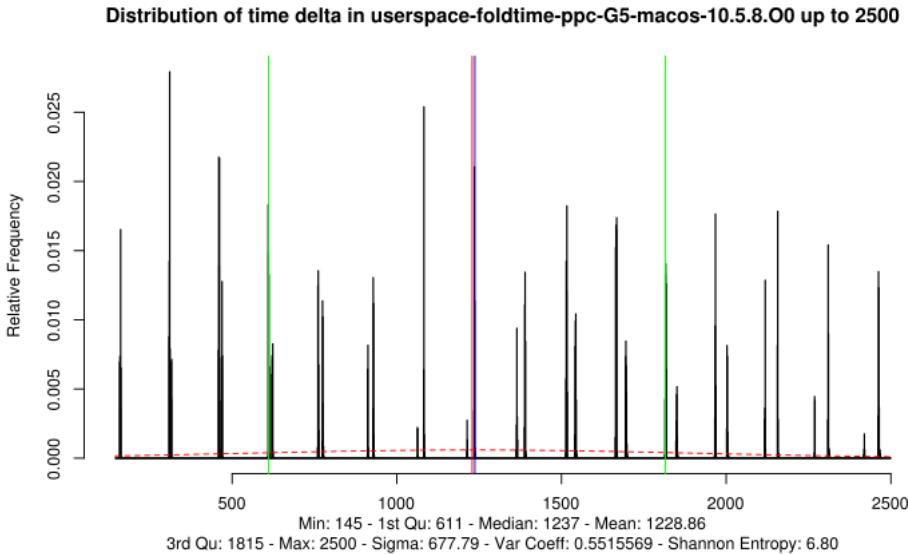
Figure C.100: ~~Upper boundary of entropy over LFSR loop in user space on on Apple G5 with optimizations~~

C.42 SUN UltraSparc IIIi

This test was executed on a SUN UltraSparc-IIIi with FreeBSD 9.1.

The graph for the lower boundary is impressive: it looks like a normal distribution!

Distribution of time delta in userspace-foldtime-ultrasparcIIIi up to 2800

Min: 2614 - 1st Qu: 2672 - Median: 2683 - Mean: 2683.11
3rd Qu: 2693 - Max: 2800 - Sigma: 15.68 - Var Coeff: 0.0058435 - Shannon Entropy: 6.00
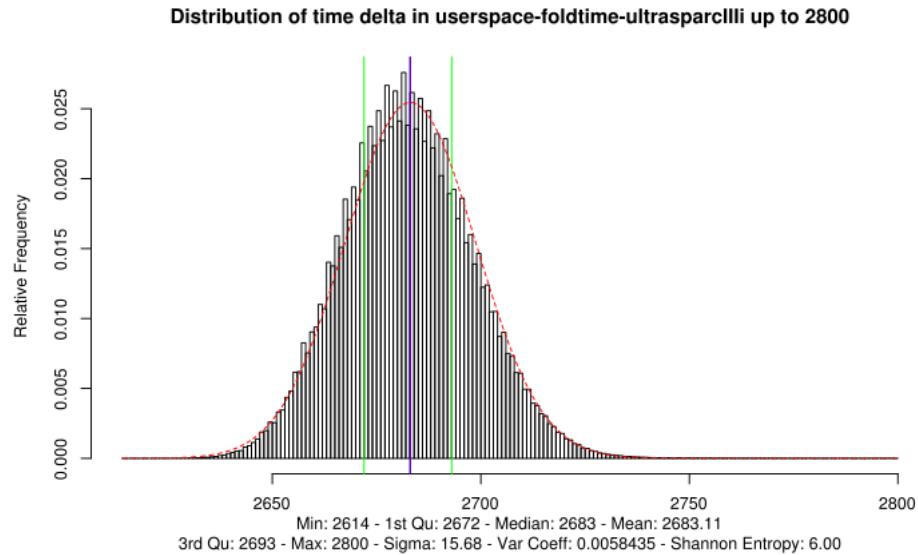
Figure C.101: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on SUN UltraSparc IIIi – with optimizations

Distribution of time delta in userspace-foldtime-ultrasparcIIIi up to 9000

Min: 2629 - 1st Qu: 4232 - Median: 5372 - Mean: 5405.08
3rd Qu: 6533 - Max: 9000 - Sigma: 1548.27 - Var Coeff: 0.2864481 - Shannon Entropy: 11.03
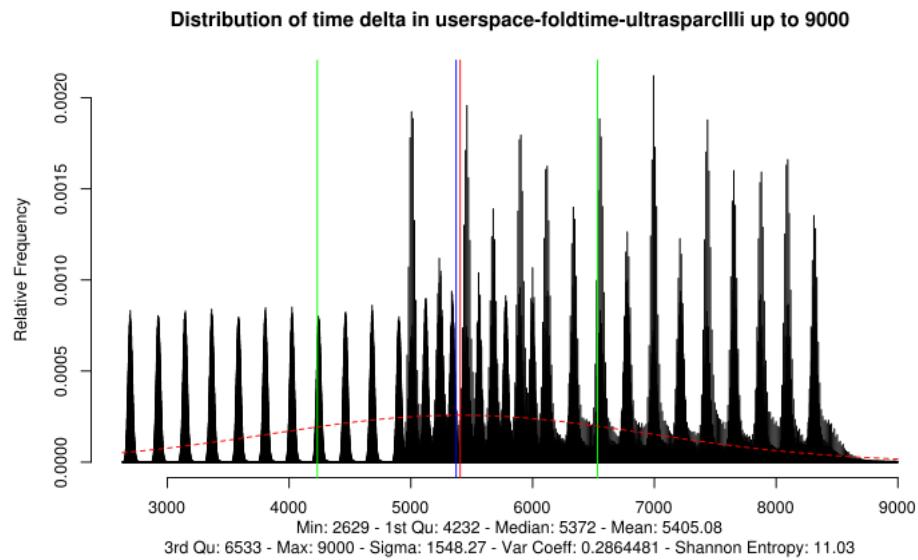
Figure C.102: ~~Upper boundary of entropy over LFSR loop in user space on SUN UltraSparc IIIi – with optimizations~~

Just to give the reader an impression how the optimization changes the measurement and to demonstrate that without optimizations the entropy is higher, here is the same CPU measurement without optimization.

Distribution of time delta in userspace-foldtime-ultrasparcIIIi.O0 up to 4500

Min: 4326 - 1st Qu: 4401 - Median: 4413 - Mean: 4412.87
3rd Qu: 4425 - Max: 4500 - Sigma: 17.78 - Var Coeff: 0.0040302 - Shannon Entropy: 6.20
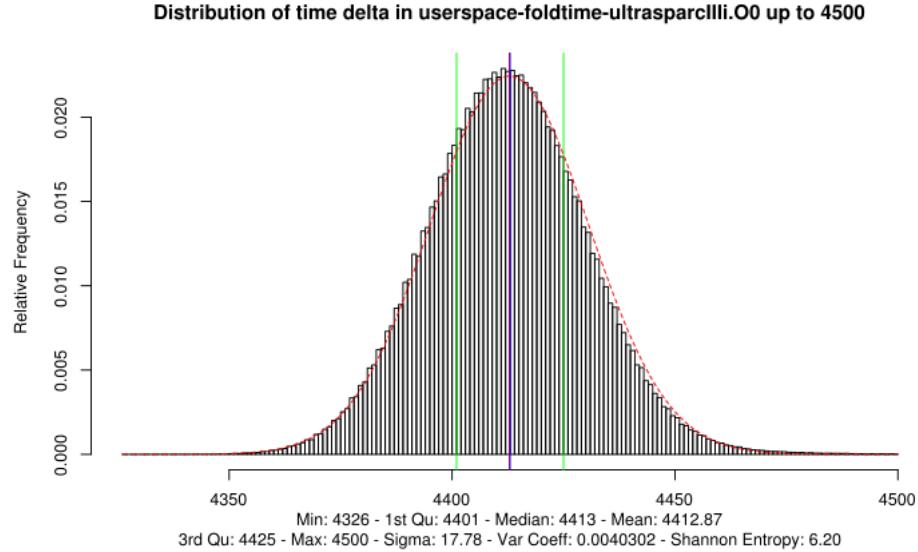
Figure C.103: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on SUN UltraSparc IIIi – without optimizations

Distribution of time delta in userspace-foldtime-ultrasparcIIIi.O0 up to 30000

Min: 4337 - 1st Qu: 11545 - Median: 17868 - Mean: 17388.11
3rd Qu: 23503 - Max: 30000 - Sigma: 7074.66 - Var Coeff: 0.4068679 - Shannon Entropy: 10.81
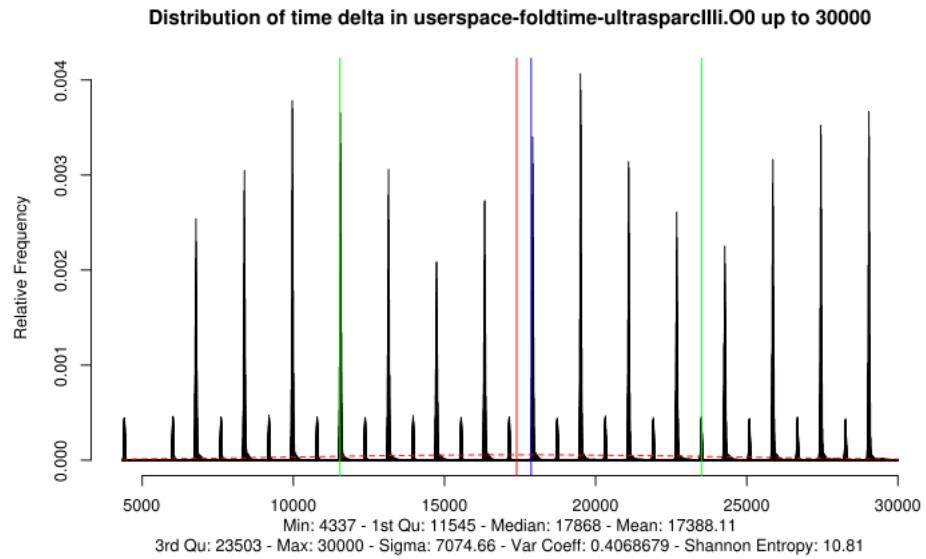
Figure C.104: ~~Upper boundary of entropy over LFSR loop in user space on SUN UltraSparc IIIi – without optimizations~~

C.43   SUN UltraSparc II

This test was executed on a SUN UltraSparc-II with OpenBSD 5.3.

Distribution of time delta in userspace-foldtime-ultrasparcII up to 8200

Min: 7682 - 1st Qu: 7975 - Median: 8045 - Mean: 8041.2
3rd Qu: 8115 - Max: 8200 - Sigma: 88.9 - Var Coeff: 0.0110553 - Shannon Entropy: 7.33
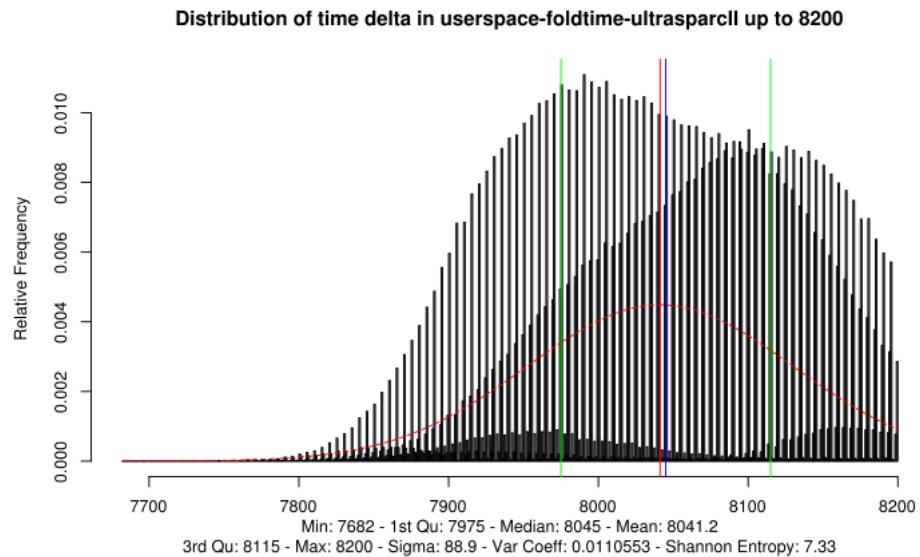
Figure C.105: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on SUN UltraSparc II – with optimizations

Distribution of time delta in userspace-foldtime-ultrasparcII up to 21000

Min: 7846 - 1st Qu: 11382 - Median: 14609 - Mean: 14424.78
3rd Qu: 17748 - Max: 21000 - Sigma: 3769.5 - Var Coeff: 0.2613212 - Shannon Entropy: 12.42
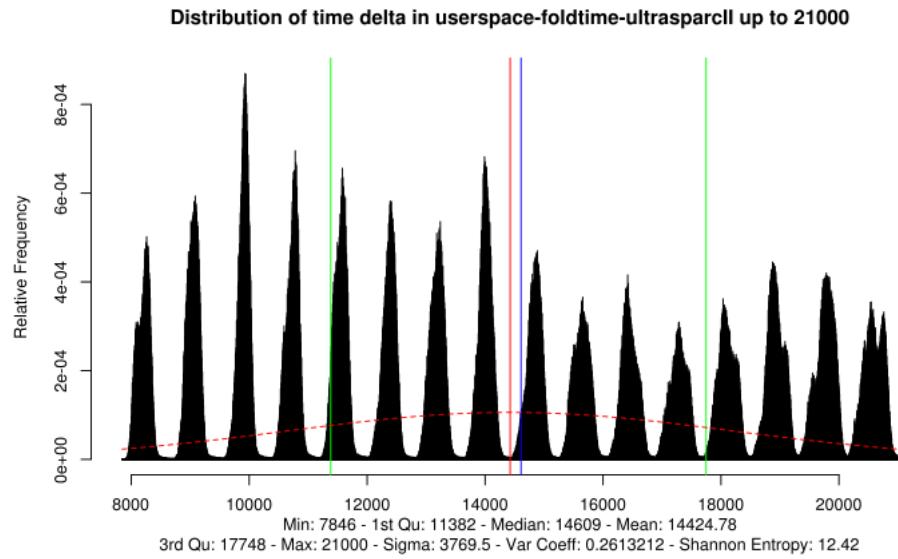
Figure C.106: ~~Upper boundary of entropy over LFSR loop in user space on SUN UltraSparc II – with optimizations~~

160

The same tests were executed without optimizations. No material differences in the distribution are present.

C.44 SUN UltraSparc IIi (Sabre)

This test was executed on a SUN UltraSparc-IIi with 440MHz executing Gentoo 2.1. The operating system is configured without a graphical interface.

Distribution of time delta in userspace-foldtime-ultrasparc-IIi-440MHz-gentoo up to 6700

Min: 6318 - 1st Qu: 6518 - Median: 6559 - Mean: 6558.89
3rd Qu: 6600 - Max: 6700 - Sigma: 58.53 - Var Coeff: 0.0089236 - Shannon Entropy: 6.48
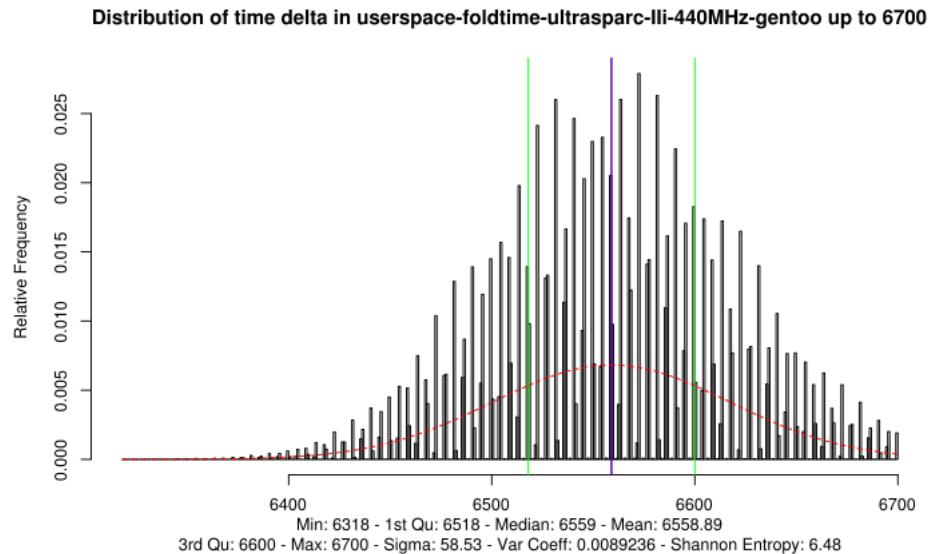
Figure C.107: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on SUN UltraSparc IIi – with optimizations

**Distribution of time delta in userspace-foldtime-ultrasparc-IIi-440MHz-gentoo up to 25000**

Min: 6327 - 1st Qu: 11123 - Median: 15709 - Mean: 15468.82
3rd Qu: 20246 - Max: 25000 - Sigma: 5452.27 - Var Coeff: 0.3524684 - Shannon Entropy: 11.54
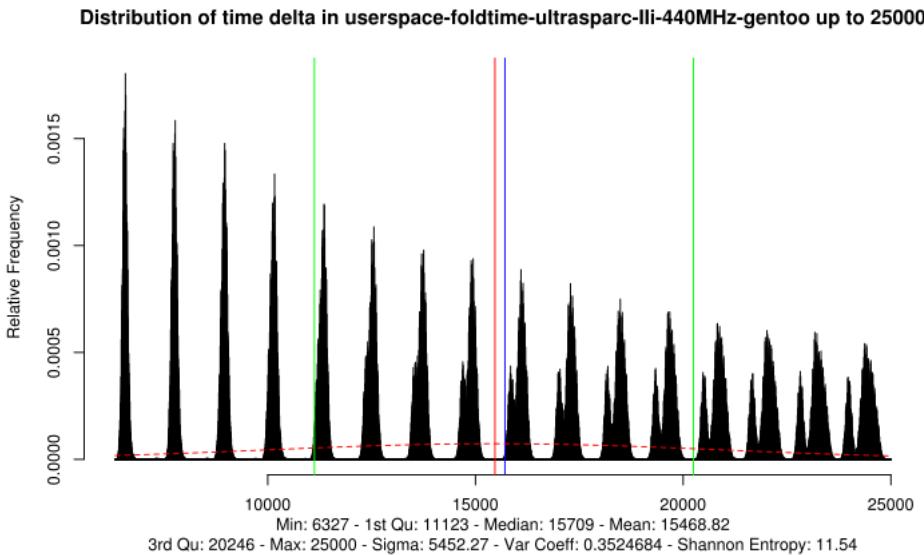
Figure C.108: ~~Upper boundary of entropy over LFSR loop in user space on SUN UltraSparc-IIi – with optimizations~~

Just to give the reader an impression how the optimization changes the measurement and to demonstrate that without optimizations the entropy is higher, here is the same CPU measurement without optimization.

Distribution of time delta in userspace-foldtime-ultrasparc-IIi-440MHz-gentoo up to 13700

Min: 13228 - 1st Qu: 13524 - Median: 13573 - Mean: 13569.4
3rd Qu: 13618 - Max: 13700 - Sigma: 64.81 - Var Coeff: 0.0047762 - Shannon Entropy: 6.58
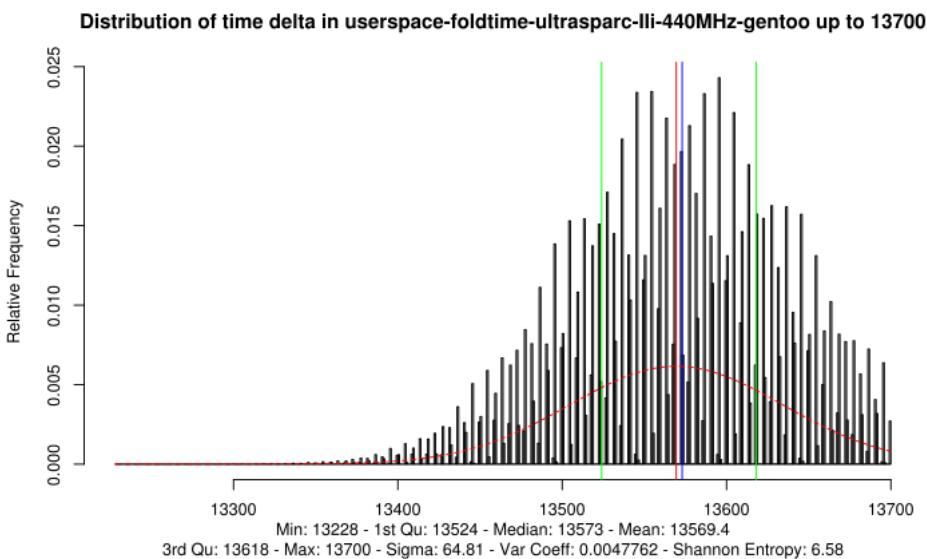
Figure C.109: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on SUN UltraSparc IIi – without optimizations

162

Distribution of time delta in userspace-foldtime-ultrasparc-IIi-440MHz-gentoo up to 130000

Min: 13282 - 1st Qu: 35974 - Median: 65607 - Mean: 68968.93
3rd Qu: 95322 - Max: 129999 - Sigma: 34141.26 - Var Coeff: 0.4950238 - Shannon Entropy: 11.77
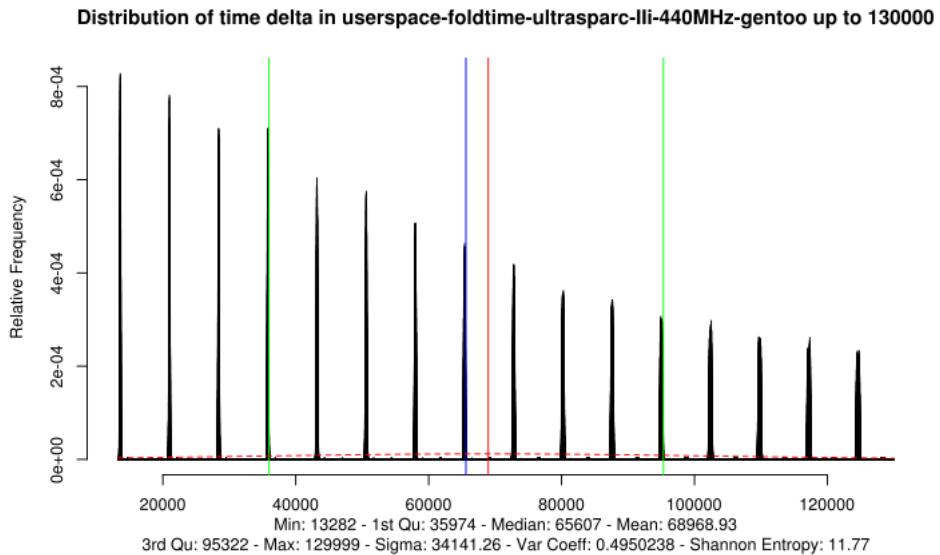
Figure C.110: ~~Upper boundary of entropy over LFSR loop in user space on SUN UltraSparc IIi – without optimizations~~

C.45 IBM System Z z10

This test was executed on an IBM System Z EC12 offering a z/VM virtual machine with one CPU to a z/OS 1R13.

Distribution of time delta in userspace-foldtime-zos.O0 up to 1100000000000000

Min: 910395627798528 - 1st Qu: 978565348720640 - Median: 1002754604531712 - Mean: 995652431149932
3rd Qu: 1015948744065024 - Max: 1099511627776000 - Sigma: 24700840377958.2 - Var Coeff: 0.0248087 - Shannon Entropy:
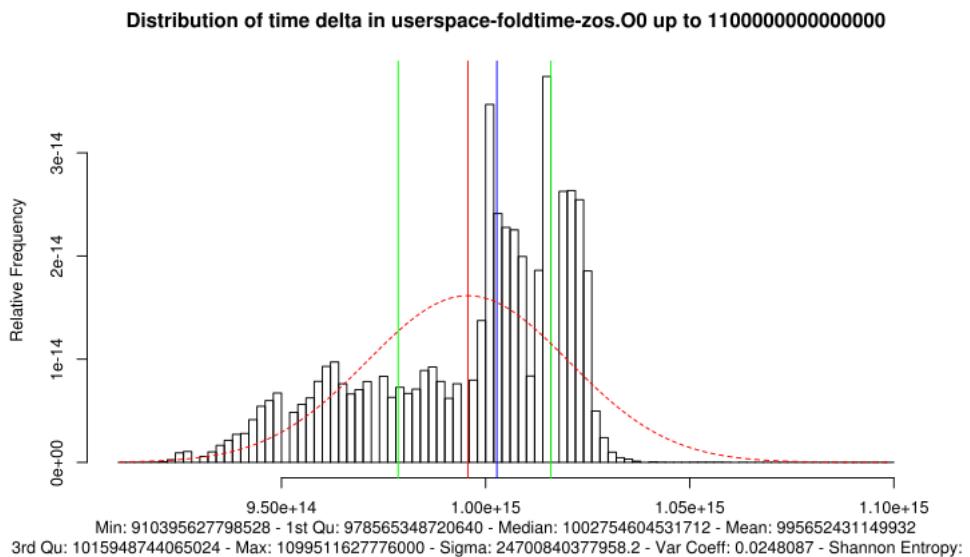
Figure C.111: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on z/OS – without optimizations

163

Due to the values being so large, the value for the Shannon Entropy is truncated in the graph above. That value is the same as printed in the able at the beginning of this appendix: 5.28 bits.

Distribution of time delta in userspace-foldtime-zos.O0 up to 15000000000000000

Min: 936783906865152 - 1st Qu: 1825189302108160 - Median: 4217726604148736 - Mean: 4072165697693716
3rd Qu: 5849401859768320 - Max: 14999537626120192 - Sigma: 2452580310497134 - Var Coeff: 0.6022791 - Shannon Entropy
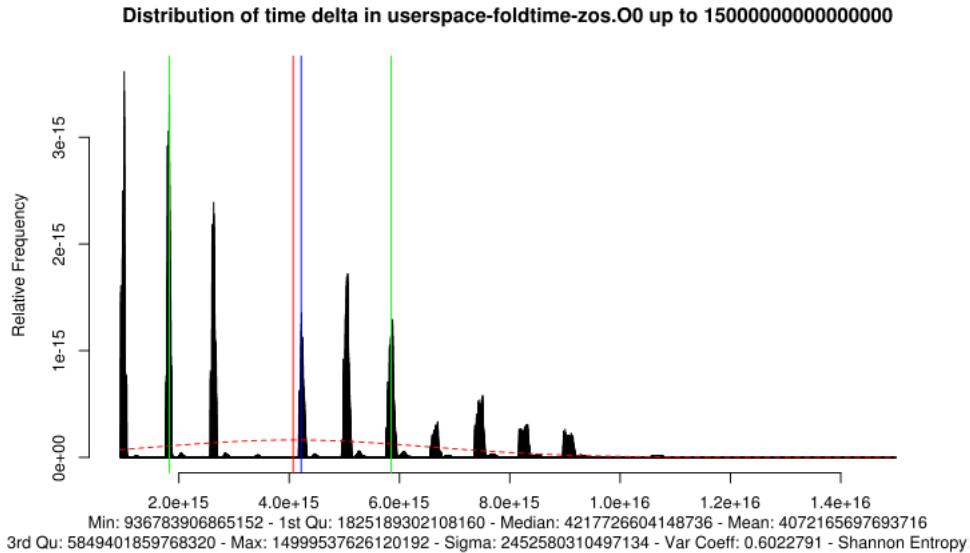
Figure C.112: ~~Upper boundary of entropy over LFSR loop in user space on zOS – without optimizations~~

~~Due to the values being so large, the value for the Shannon Entropy is truncated in the graph above. That value is the same as printed in the able at the beginning of this appendix: 9.38 bits.~~

The graphs show a similar pattern to other systems in other sections. However, the timer values have much larger numbers than for any other test. The reason is the way how the timer is read and the fact that System Z mainframes have a timer that has a much higher resolution. The timer is read with the `STCKE` processor instruction such that the moving 64 low bits of the 128 bit value returned by `STCKE` are returned. That means that the lowest 7 bits are cut off which do not contain timer information as specified in the processor manual.

C.46 IBM System Z z10

This test was executed on an IBM System Z z10 offering a z/VM virtual machine with one CPU to a SLES11 SP2.

C.46.1 64 bit Word Size

The processor is identified as "version = FF, identification = 058942, machine = 2097".

Distribution of time delta in userspace-foldtime-s390-zvm up to 230

Min: 187 - 1st Qu: 187 - Median: 188 - Mean: 201.47
3rd Qu: 219 - Max: 219 - Sigma: 15.54 - Var Coeff: 0.0771485 - Shannon Entropy: 1.91
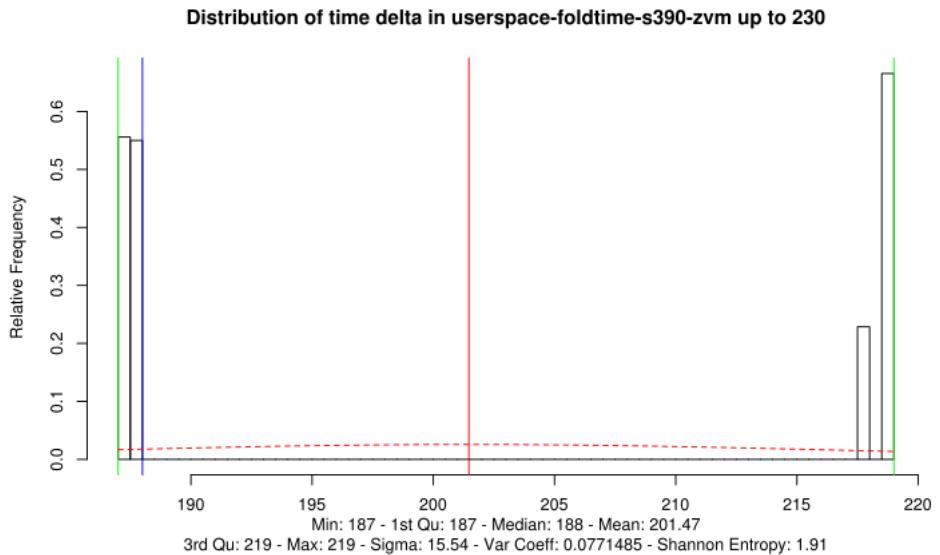
Figure C.113: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on IBM System Z z10 – 64 bit with optimizations


Distribution of time delta in userspace-foldtime-s390-zvm up to 2200

Min: 187 - 1st Qu: 687 - Median: 1187 - Mean: 1180.13
3rd Qu: 1688 - Max: 2188 - Sigma: 594.61 - Var Coeff: 0.5038505 - Shannon Entropy: 5.54
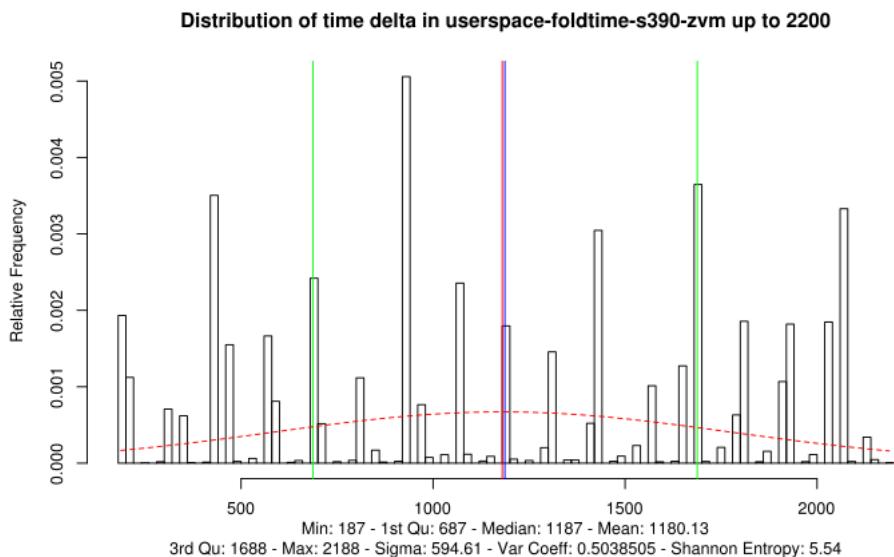
Figure C.114: ~~Upper boundary of entropy over LFSR loop in user space on IBM System Z z10 – 64 bit with optimizations~~

Just to give the reader an impression how the optimization changes the measurement and to demonstrate that without optimizations the entropy is higher, here is the same CPU measurement without optimization.

Distribution of time delta in userspace-foldtime-s390-zvm.O0 up to 730

Min: 656 - 1st Qu: 687 - Median: 688 - Mean: 699.15
3rd Qu: 719 - Max: 720 - Sigma: 15.12 - Var Coeff: 0.0216213 - Shannon Entropy: 1.90
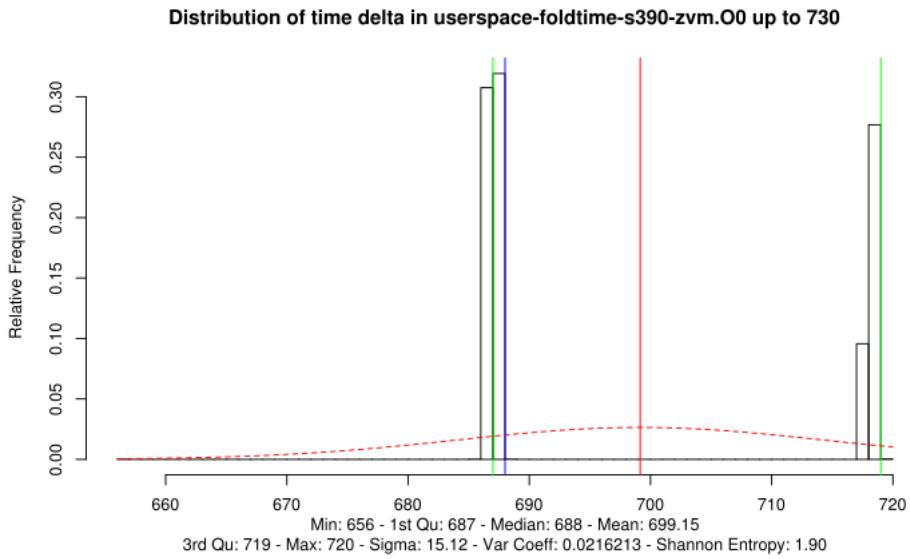
Figure C.115: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on IBM System Z z10 – 64 bit without optimizations

Distribution of time delta in userspace-foldtime-s390-zvm.O0 up to 9300

Min: 655 - 1st Qu: 2406 - Median: 4656 - Mean: 4781.34
3rd Qu: 6907 - Max: 9291 - Sigma: 2532.29 - Var Coeff: 0.5296182 - Shannon Entropy: 6.32
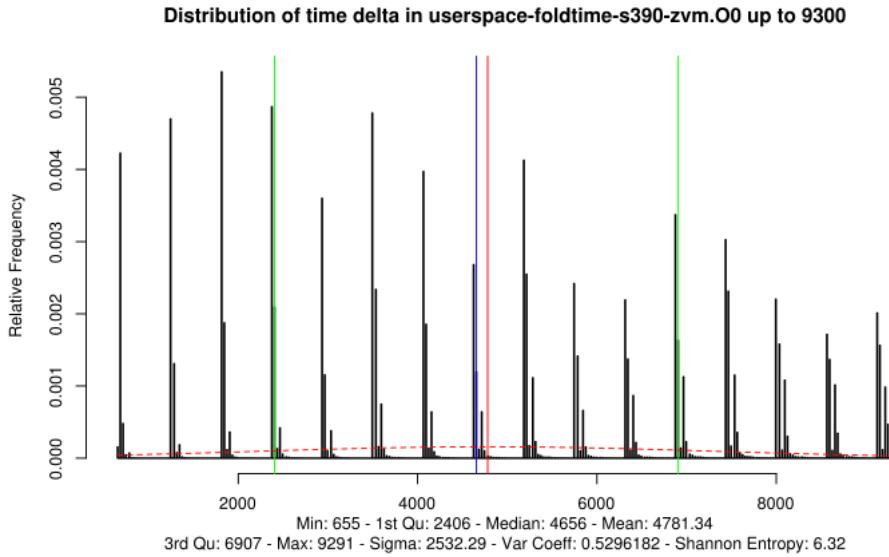
Figure C.116: ~~Upper boundary of entropy over LFSR loop in user space on IBM System Z z10 – 64 bit without optimizations~~

C.46.2 31 bit Word Size

The same hardware system is tested when compiling the test with 31 bit word size.

Distribution of time delta in userspace-foldtime-s390-zvm-31bit up to 260

Min: 218 - 1st Qu: 250 - Median: 250 - Mean: 244.51
3rd Qu: 250 - Max: 251 - Sigma: 11.92 - Var Coeff: 0.0487493 - Shannon Entropy: 0.86
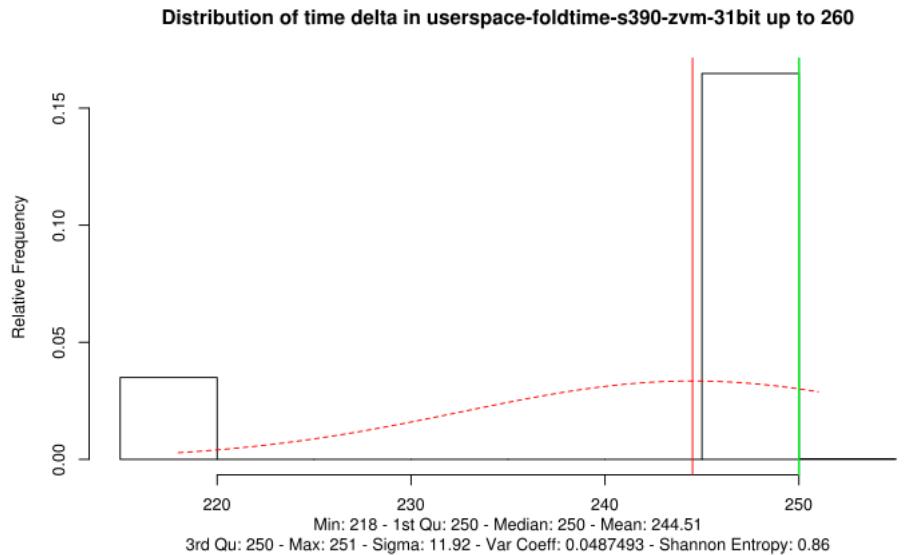
Figure C.117: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on IBM System Z z10 – 31 bit with optimizations

Distribution of time delta in userspace-foldtime-s390-zvm-31bit up to 2600

Min: 218 - 1st Qu: 562 - Median: 1000 - Mean: 1126.89
3rd Qu: 1625 - Max: 2599 - Sigma: 695.89 - Var Coeff: 0.6175302 - Shannon Entropy: 5.20
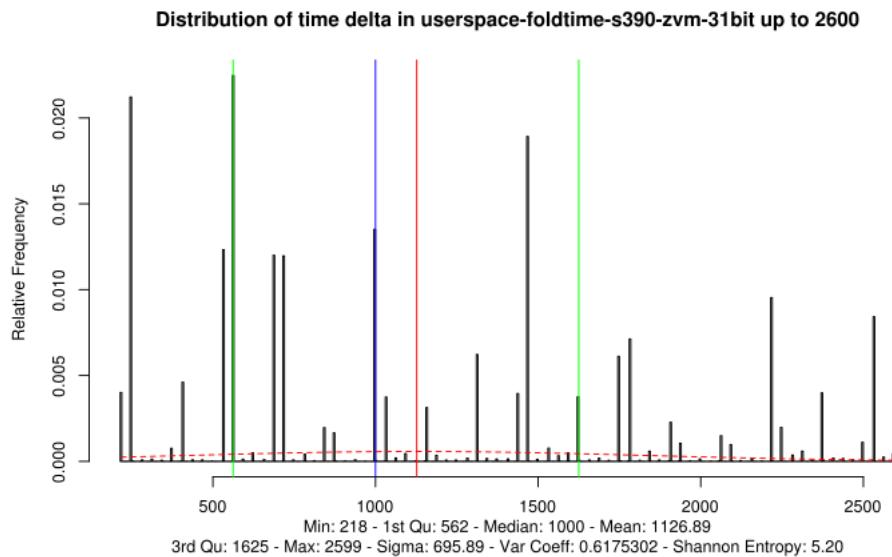
Figure C.118: ~~Upper boundary of entropy over LFSR loop in user space on IBM System Z z10 – 31 bit with optimizations~~

167

As already indicated in the table at the beginning of Appendix C the measurements of 31 bit compilations on an IBM System Z with optimizations show way to little entropy at the lower boundary. This is visible in the graphs above. Therefore, the tests are re-performed without optimizations, i.e. the compilation of a regular CPU Jitter random number generator. These new measurements are given in the graphs below. They show a significant improvement over the optimized code. The test without optimizations show a sufficient entropy that is significantly higher than the optimized code. Therefore, when using the non-optimized code, which is the case for the regular runtime of the RNG, the 31 bit word size on IBM System Z is considered appropriate.

Distribution of time delta in userspace-foldtime-s390-zvm-31bit.O0 up to 1100

Min: 937 - 1st Qu: 969 - Median: 969 - Mean: 984.05
3rd Qu: 1000 - Max: 1095 - Sigma: 20.73 - Var Coeff: 0.0210633 - Shannon Entropy: 1.72
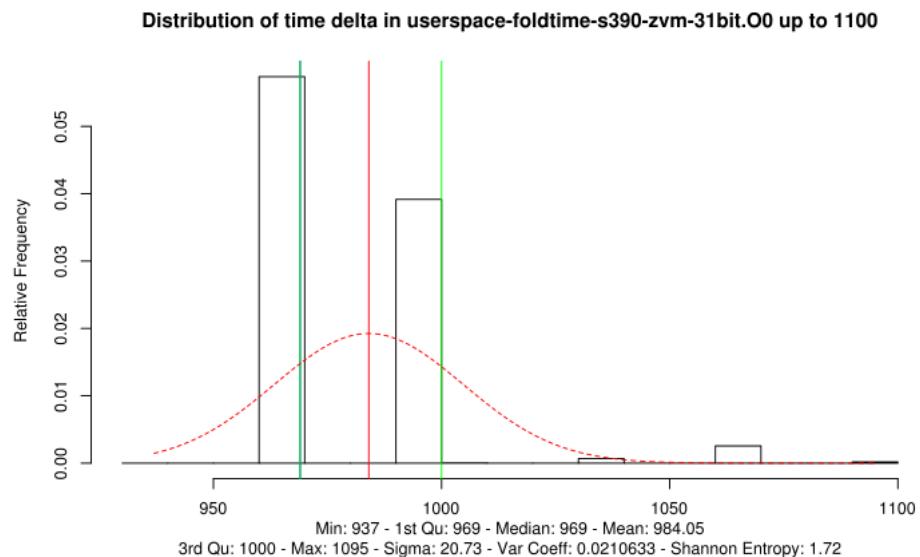
Figure C.119: Entropy~~Lower boundary of entropy~~ over LFSR ~~loop~~hash in user space on IBM System Z z10 – 31 bit without optimizations

168
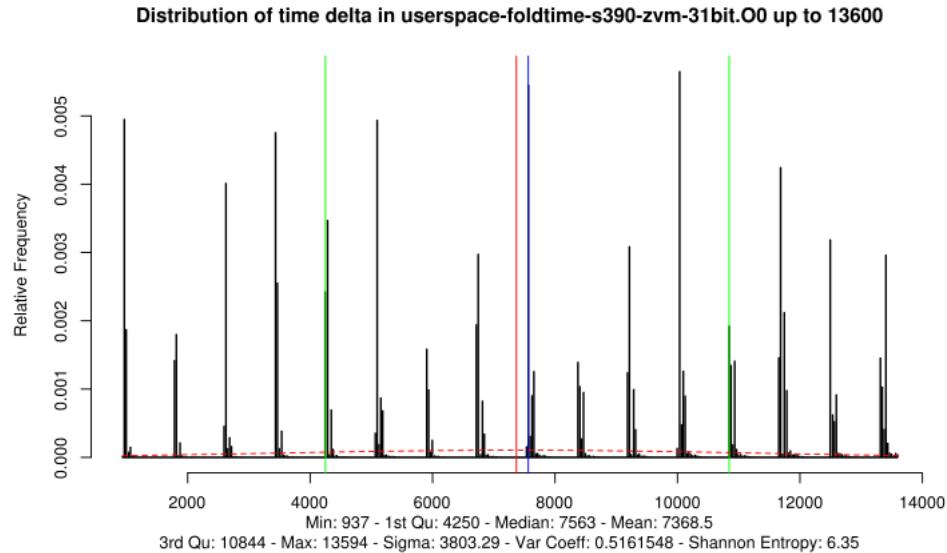
Figure C.120: Upper boundary of entropy over LFSR loop in user space on IBM System Z z10 – 31 bit without optimizations

C.47 Intel Core i7 2620M With RDTSC Instruction

The following tests were executed on the same hardware, but with different operating systems to allow analyzing the impact of the operating system on the CPU execution time jitter.

To avoid any interference from context switches and similar, the time stamp is gathered by fetching it directly from the CPU with the following assembler code:

```
/* taken from Linux kernel */
#define DECLARE_ARGS(val, low, high)    unsigned low, high
#define EAX_EDX_VAL(val, low, high)     ((low) | ((__u64)(high) << 32))
#define EAX_EDX_RET(val, low, high)     "=a" (low), "=d" (high)

static inline void jent_get_nstime(__u64 *out)
{
        DECLARE_ARGS(val, low, high);
        asm volatile("rdtsc" : EAX_EDX_RET(val, low, high));
        *out = EAX_EDX_VAL(val, low, high);
}
```

With this change, the CPU Jitter random number generator only uses the `malloc` and `free` functions during startup and termination from the operating systems *and no other mechanism*! The header file that provides this code is found in `arch/jitterentropy-base-x86.h` and is a drop-in replacement of `jitterentropy-base-user.h`.

As different compilers are used to generate the binaries for the different operating systems, all tests were compiled without optimization.

When comparing the different graphs, the following findings can be drawn:

- The user space of the operating system has an impact on the measurements. If a large number of user space applications are executing, includ-

169

ing X11, the CPU execution time jitter is significantly larger compared to systems where hardly any user space application is running in parallel with the measurements.

- The size of the kernel has *no* significant impact on the CPU execution time jitter. Although the mean values of the tests on the BSD systems and the Linux system without X11 differ significantly, the jitter size itself represented by the number of different time delta values[14] does not differ significantly.

Still, the final conclusion is that regardless of the used operating system, the CPU execution time jitter measurements indicate that the random number generator can be used on all systems.

C.47.1   Ubuntu Linux 13.04 with KDE

The following test was executed on an Ubuntu 13.04 with the graphical environment of KDE was running.

Distribution of time delta in userspace-foldtime-i7.O0 up to 650

Min: 561 - 1st Qu: 582 - Median: 588 - Mean: 587.45
3rd Qu: 591 - Max: 648 - Sigma: 9.25 - Var Coeff: 0.0157501 - Shannon Entropy: 3.24
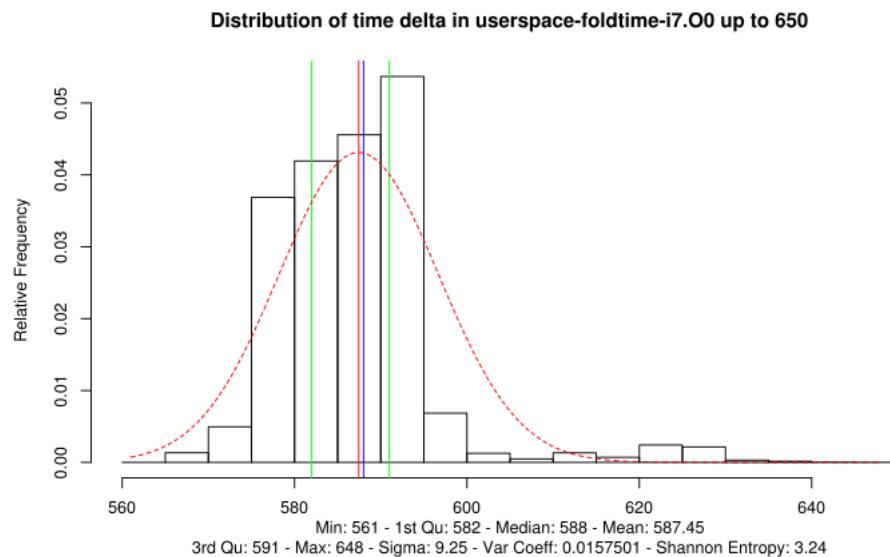
Figure C.121: Entropy~~Lower boundary of entropy~~ over LFSR ~~loop~~hash in user space on Ubuntu Linux 13.04 with KDE and Intel Core i7 2620M

[14]I.e. the number of bars in the histogram.

Distribution of time delta in userspace-foldtime-i7.O0 up to 8500

Min: 573 - 1st Qu: 2157 - Median: 4203 - Mean: 4398.83
3rd Qu: 6264 - Max: 8499 - Sigma: 2363.96 - Var Coeff: 0.5374078 - Shannon Entropy: 8.44
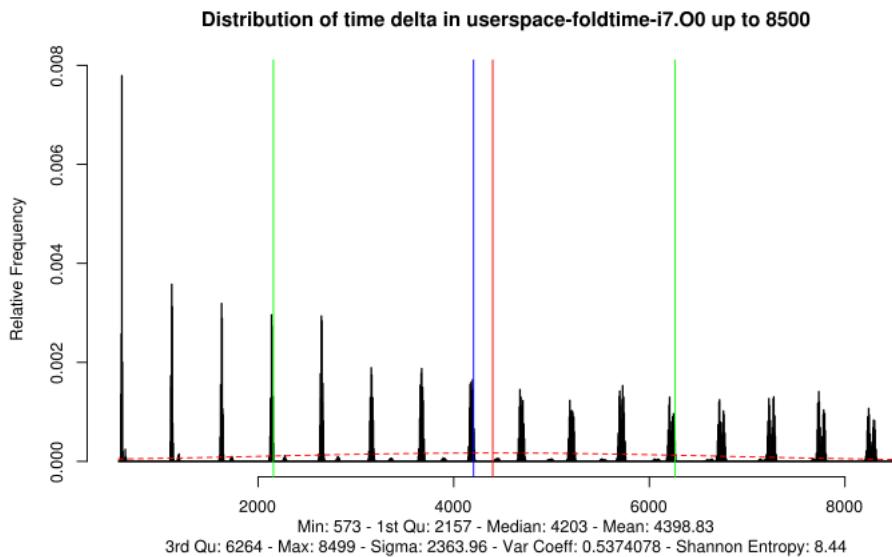
Figure C.122: ~~Upper boundary of entropy over LFSR loop in user space on Ubuntu Linux 13.04 with KDE and Intel Core i7 2620M~~

C.47.2 Ubuntu Linux 13.04 without X11

The following test was executed on an Ubuntu 13.04 that was booted with the kernel command line option of `init=/bin/bash`. This option implies that no user space processes besides `init` and `bash` were running. Especially, no X11 windowing system was executing.

Distribution of time delta in userspace-foldtime-i7-2620M-ubuntu-noX up to 770

Min: 567 - 1st Qu: 732 - Median: 740 - Mean: 737.58
3rd Qu: 744 - Max: 768 - Sigma: 7.74 - Var Coeff: 0.010496 - Shannon Entropy: 2.85
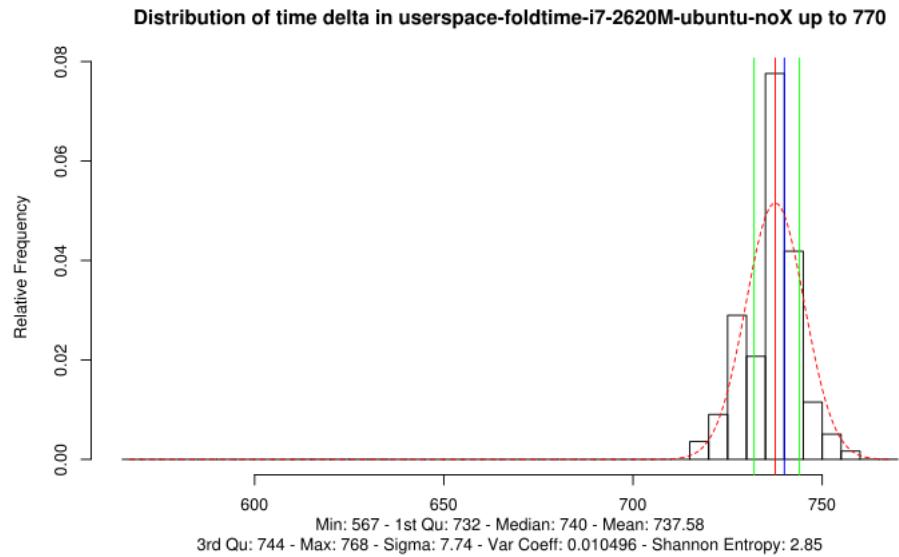
Figure C.123: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on Ubuntu Linux 13.04 without X11 and Intel Core i7 2620M

Distribution of time delta in userspace-foldtime-i7-2620M-ubuntu-noX up to 10000

Min: 597 - 1st Qu: 2692 - Median: 5256 - Mean: 5198.39
3rd Qu: 7816 - Max: 10000 - Sigma: 2864.35 - Var Coeff: 0.5510081 - Shannon Entropy: 6.85
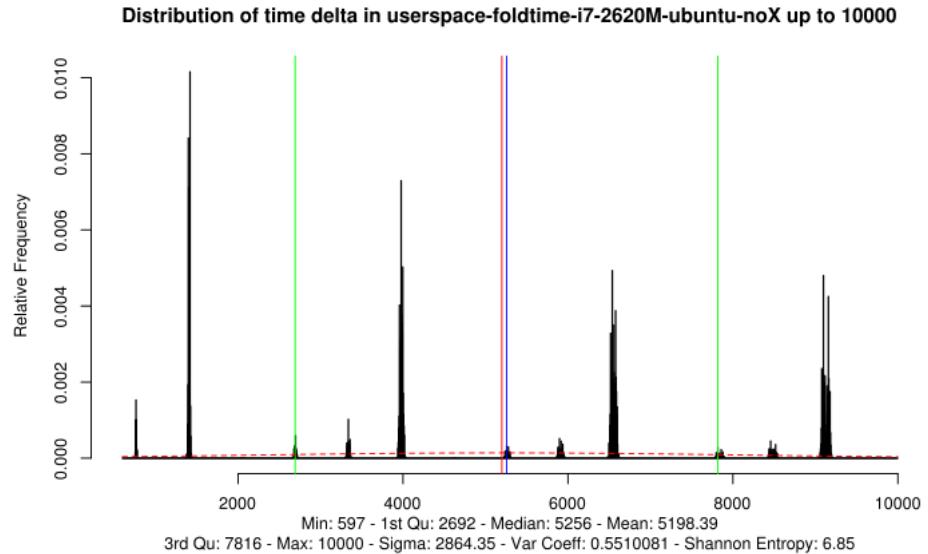
Figure C.124: ~~Upper boundary of entropy over LFSR loop in user space on Ubuntu Linux 13.04 without X11 and Intel Core i7 2620M~~

C.47.3 OpenIndiana 151a7

The desktop version of OpenIndiana was installed which implied that X11 with
Gnome was up and running.

Distribution of time delta in userspace-foldtime-i7-2620M-openindiana151 up to 870

Min: 780 - 1st Qu: 831 - Median: 837 - Mean: 837.92
3rd Qu: 843 - Max: 870 - Sigma: 8.88 - Var Coeff: 0.010601 - Shannon Entropy: 3.57
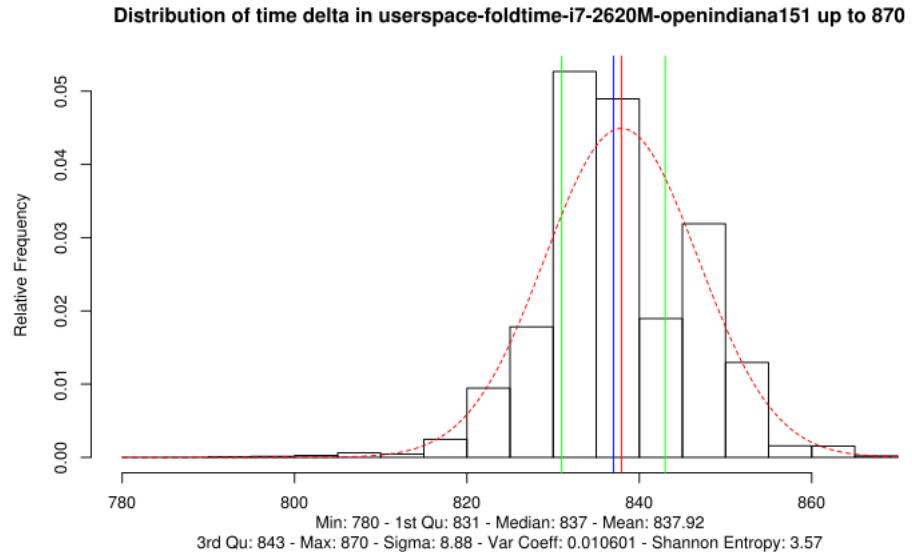
Figure C.125: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user
space on OpenIndiana 151a7 and Intel Core i7 2620M

Distribution of time delta in userspace-foldtime-i7-2620M-openindiana151 up to 12000

Min: 786 - 1st Qu: 3711 - Median: 6636 - Mean: 6342.8
3rd Qu: 9420 - Max: 12000 - Sigma: 3358.7 - Var Coeff: 0.5295299 - Shannon Entropy: 9.36
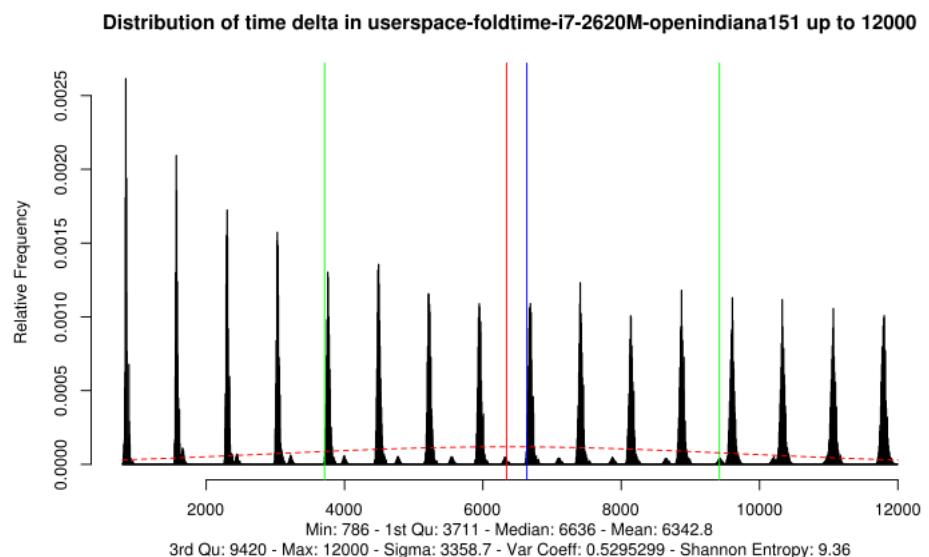
Figure C.126: ~~Upper boundary of entropy over LFSR loop in user space on
OpenIndiana 151a7 and Intel Core i7 2620M~~

173

C.47.4 NetBSD 6.0

The LiveCD image for NetBSD was used that did not execute X11 and hardly any other user space applications.
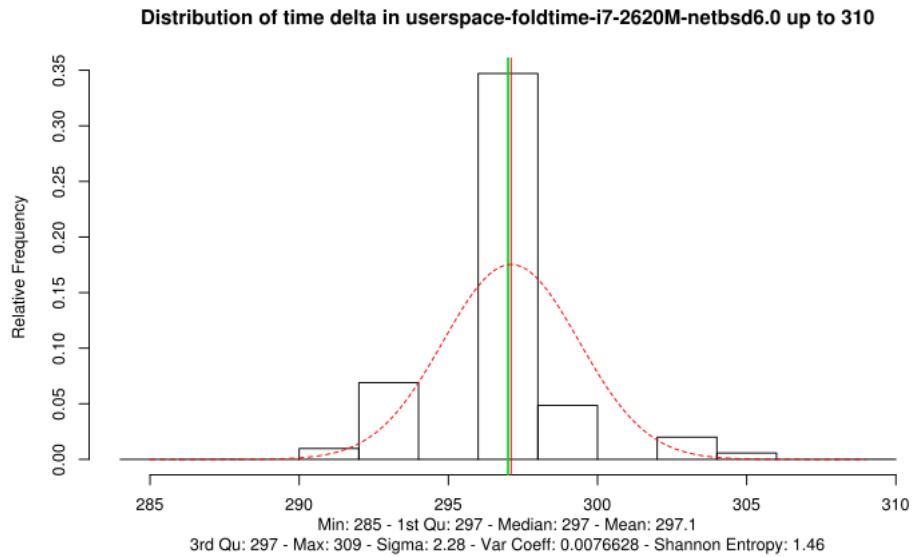
Figure C.127: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on NetBSD 6.0 and Intel Core i7 2620M
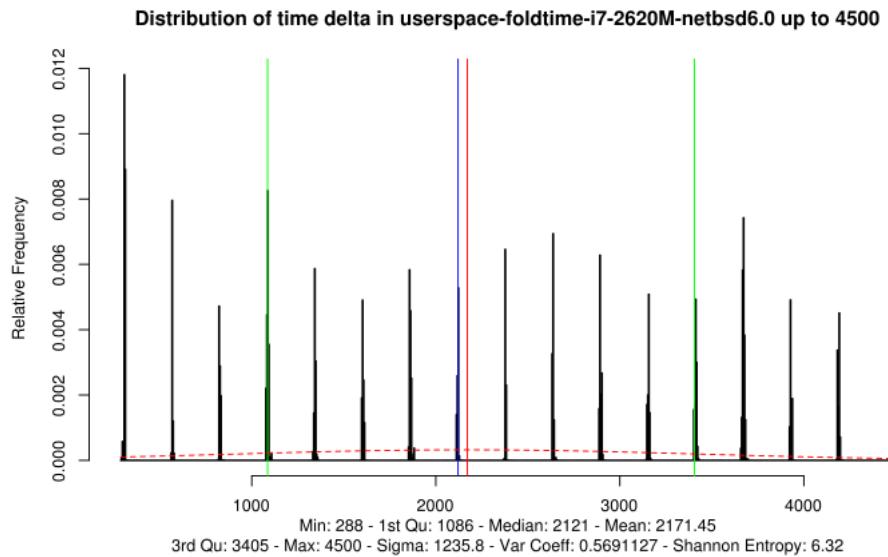
Figure C.128: ~~Upper boundary of entropy over LFSR loop in user space on NetBSD 6.0 and Intel Core i7 2620M~~

C.47.5 FreeBSD 9.1

The LiveCD image for FreeBSD was used that did not execute X11 and hardly any other user space applications.
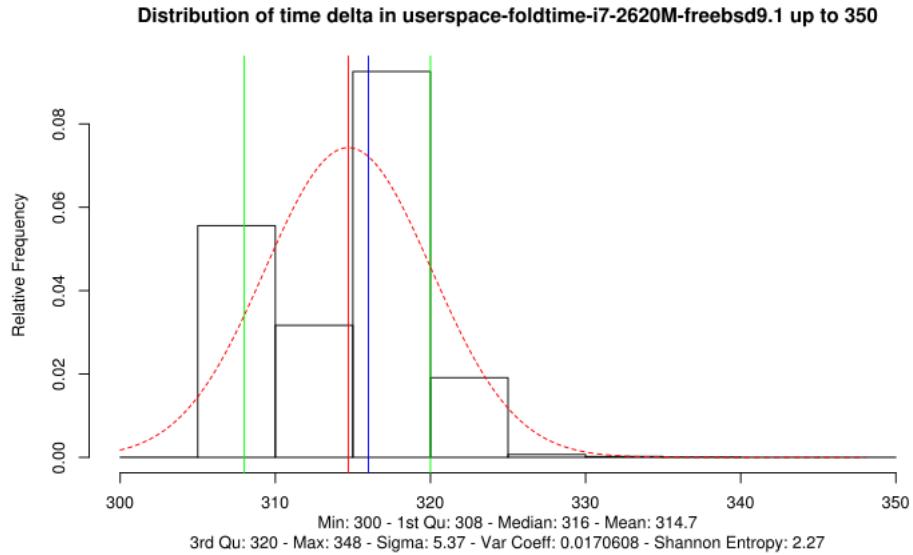
Figure C.129: Entropy~~Lower boundary of entropy~~ over ~~LFSR~~hash loop in user space on FreeBSD 9.1 and Intel Core i7 2620M
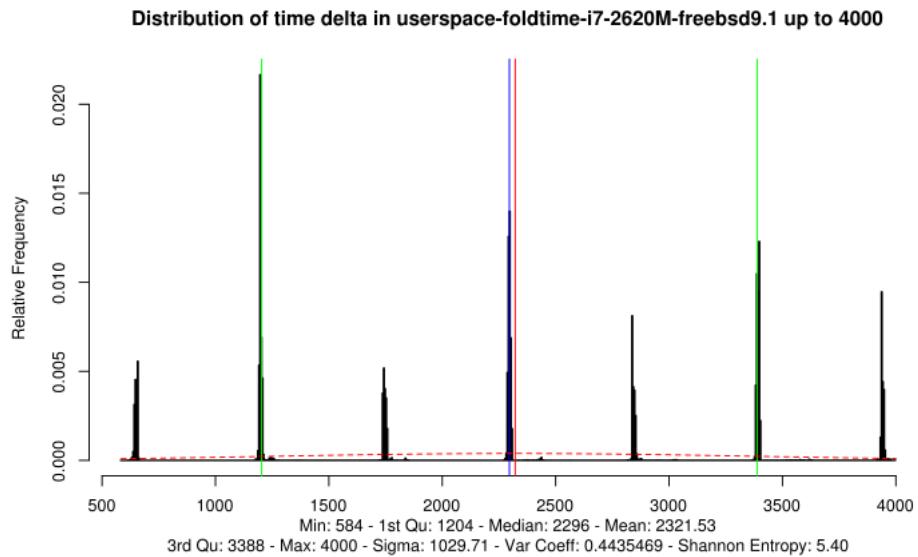
Figure C.130: ~~Upper boundary of entropy over LFSR loop in user space on FreeBSD 9.1 and Intel Core i7 2620M~~

D BSI AIS 20 / 31 NTG.1 Properties

The CPU Jitter random number generator is believed to comply with the BSI AIS 20 / 31 NTG.1 properties as follows:

NTG.1.1 See comments in `jent_entropy_init` which show that statistical defects in the noise source, i.e. the timing jitter are identified. *Covered*

NTG.1.2 The `jent_read_entropy` function will only read from an entropy pool when the entire entropy collection loop is completed. As this loop fills the entropy pool with full entropy as described in chapter 5 supported by section 5.1, even in the worst case the CPU Jitter random number generator ensures to provide the requested entropy to the caller. *Covered*

NTG.1.3 The timing jitter is an unpredictable noise source as shown in chapter 2. The entropy in the noise source is magnified by mixing it into an entropy pool by retaining its entropy and ensuring that the entropy is conveyed to the caller by delivering a random bit string. Section 8 bullet 9 outlines the perfect forward and backward secrecy. *Covered*

NTG.1.4 To generate 128 bits from the RNG, we pull twice from entropy pool and concatenate the two random values – as it is done in `jent_read_entropy`. Let us assume the birthday paradox where a collision of 2^{64} random values of size 128 bits occurs with the probability of

$$P(collisions\,after\,2^{64}\,random\,values) = 0.3935$$

The number of n pariwise different bit strings with a length of 128 bits is

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

The probability therefore is:

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

Using the estimation
$$A > (2^{128} - n + 1)^n$$

we have a lower boundary for P. Thus, we can calculate, for example, that 2^{55} bit strings of length 128 bits following each other are pairwise different with probability of P>0.999996. Thus, when using k > 2^{55} bit string with length 128 bits, and these strings show no collision with probability of P > 1 - e, with e 3.8e-6, the bit strings are pairwise different. *This result satisfies even AVA_VAN.5.* The analysis rests on the assumption that the bit stream follow an rectangular distribution, i.e. the bit stream is a white noise. The discussion of the statistical properties of the bit stream in chapter 4 demonstrates the property of a white noise. *Covered*

NTG.1.5 All of the following statistical tests pass even though the seed source is not processed with a cryptographic whitening function!

- BSI Test procedure A is passed.
- BSI Test suite A is passed

- `dieharder` test passed on kernel and user space generator

- `ent` test passed

Covered

NTG.1.6 Using `ent` we get 7.9999... bits of entropy per byte. That value is the Shannon entropy of the input data. This value implies that we have more than 99.7% (Shannon) entropy in the output. In addition, the discussion in chapter 5 supports the statement that more than 99.7% entropy is present in the output. *Covered*

E   Bibliographic Reference

References

Meltem Sönmez Turan, Elaine Barker, John Kelsey, Kerry A. McKay, Mary L. Baish, and Mike Boyle. *NIST Special Publication 800-90B Recommendation for the Entropy Sources Uses for Random Bit Generation.* 2018.

F   Thanks

Special thanks for providing input as well as mathematical support goes to:

- Yi Mao

- Steve Weingart

- Jeremy Powell

- Gerald Krummeck

- Michael Robrecht

- Roman Drahtmüller

- Louis Losee

- Josef Söntgen

- Sandy Harris

Also, special thanks go to all people executed the test applications to gather the results in appendix C.

G   License

The implementation of the CPU Jitter random number generator, all support mechanisms, the test cases and the documentation are subject to the following license.

Copyright Stephan Müller <smueller@chronox.de>, 2013.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

H   Change Log

Date	JENT Version	Change
2021-03-30	3.0.2	Add comments received from NIST
		Incorporate changes from updates applied to 3.0.2
2021-07-25	3.1.0	Incorporate Lag Predictor Test
2021-10-26	3.3.1	Documentation updates for random memory access
		Fix calculation in 7.2.6 based on comments from
		Lisa Rabe and Agustín Antonio Sánchez Bosque
		Add SP800-90C consideration
2021-12-19	3.4.0	Correct entropy rate in section 7.4
		Update section 3.3 due to updated SHA-3 integration
2022-07-01	3.4.0	Clarification of the entropy rate from memory accesses and instruction variations in chapter 6
		Remove stale content regarding LFSR