CPU Time Jitter Based Non-Physical True Random Number Generator

Stephan Müller
atsec information security
Stephan.Mueller@atsec.com

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 entropy, 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.

The complete version of the analysis together with large amounts of test results is provided at www.chronox.de.

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.
- Virtualized environments remove an operating system from direct hardware access.
- 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.
- 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.
- On Linux, the majority of the entropy for the `input_pool` behind `/dev/random` is gathered from the `get_cycles` 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 entropy with a speed that satisfies today's requirement for entropy. The random number generator shall be able to be used synchronously with the entropy 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 entropy 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 entropy 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 Section 2
- Presentation of CPU Jitter random number generator design in Section 3
- Discussion of the statistical properties of the random number generator output in Section 4
- Assessment of the entropy behavior in the random number generator in Section 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?

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 timer tick and its processing which alters the caches.
- Cache coherency strategies of the CPU with its cores add variances to instruction execution time as the cache controlling logic must check other caches for their information before an instruction or memory access is fetched from the local cache.
- 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.

In addition to the hardware nondeterminism, the following operating system caused system usage adds to the non-deterministic execution time 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;
- 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 entropy. The following assumptions apply:

- Attacker having hardware level privileges 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 entropy-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 the entropy collection. Again, with such an attacker, the entropy collection is the least of our worries in this case.
- The CPU Jitter random number generator is always executed on CPUs connected to peripherals. When the CPU has no peripherals, including no access to RAM or any busses, special software can be expected to execute on the CPU fully deterministically. However, as this scenario requires a highly specialized environment that does not allow general purpose computing, this scenario is not applicable.

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 miniscule 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. For the sample code sequence given in Figure 1, the variation in time is shown in Figure 2.

The contents of the variable `delta` is not identical between the individual loop iterations. When running the

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

void main(void)
{
...
        jent_get_nstime(&time);
        jent_get_nstime(&time2);
        delta = time2 - time;
...
}
```

Figure 1: Sample code for time variance observation

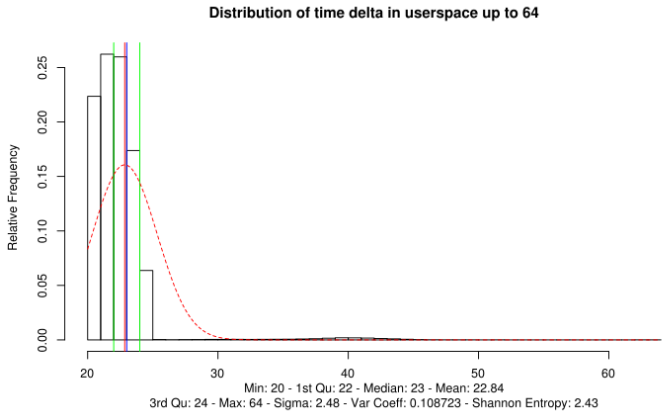

Figure 2: Distribution of time variances in user space over 1.000.000 loops

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.

Please note that the actual results of the aforementioned code contains a few exceptionally large deltas as an operating system can never be fully quiesced. 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 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 5.2 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. For an initial test, the time stamp variance collection is invoked 30.000.000 times. The generation of the given number of time deltas is very fast, typically less than 10 seconds. 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.

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 Section 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 "magnify" 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 entropy 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.

The following section presents the design of the random number generator. 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.

The random number generator maintains a 64 bit unsigned integer variable, the entropy pool, that is indicated with the gray shaded boxes in Figure 3 which identify the entropy pool at two different times in the processing.

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

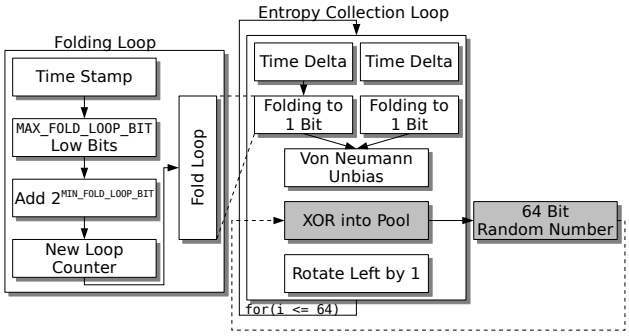

Figure 3: Entropy Collection Operation

1. fetches a time stamp to calculate a delta to the time stamp of the previous loop iteration,
2. folds the time delta value into one bit,
3. processes this value with a Von-Neumann unbias operation,
4. adds this value to the entropy pool using XOR,
5. rotates the pool to fill the next bit value of the pool.

The loop is executed exactly 64 times as each loop iteration generates one bit to fill all 64 bits of the entropy pool. After the loop finishes, the contents of the entropy pool is given to the caller as a 64 bit random number[1]. The following subsection discuss every step in detail.

When considering that the time delta is always computed from the delta to the previous loop iteration, and the fact that the majority of the execution time is spent in the folding loop, the central idea of the CPU Jitter Random Number Generator is to measure the execution time jitter over the execution of the folding loop.

3.1.1 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,

[1] If the caller provides an oversampling rate of greater than 1 during the allocation of the entropy collector, the loop iteration count of 64 is multiplied by this oversampling rate value. For example, an oversample rate of 3 implies that the 64 loop iterations are executed three times – i.e. 192 times.

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. In essence, all steps of the entropy collection loop are performed, except of mixing the delta into the pool and rotating the pool. 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 XOR 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.

3.1.2 Folding Operation of Time Delta

The folding operation is depicted by the left side of Figure 3. That folding operation is implemented by a loop where the loop counter is not fixed.

To calculate the new fold loop counter a new time stamp is obtained. All bits above the value `MAX_FOLD_LOOP_BITS` – which is set to 4 – are zeroed. The idea is that the fast moving bits of the time stamp value determine the size of the collection loop counter. Why is it set to 4? The 4 low bits define a value between 0 and 16. This uncertainty is used to quickly stabilize the distribution of the output of that folding operation to an equidistribution of 0 and 1, i.e. about 50% of all output is 0 and also about 50% is 1. See Section 5.2.1 for a quantitative analysis of that distribution. To ensure that the collection loop counter has a minimum value, the value 1 is added – that value is controlled with `MIN_FOLD_LOOP_BIT`. Thus, the range of the folding counter value is from 1 to (16 + 1 - 1). Now, this newly determined collection loop counter is used to perform a new fold loop as discussed in the following.

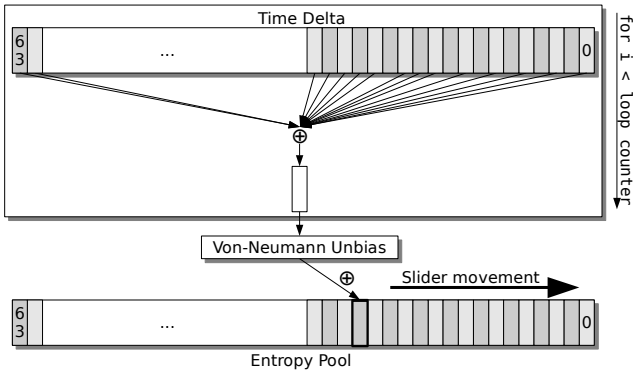

Figure 4: Folding of the time delta and mixing it into the entropy pool

Figure 4 shows the concept of the folding operation of one time delta value.

The upper 64 bit value illustrated in Figure 4 is the time delta obtained at the beginning of the current entropy collection loop iteration. Now, the time delta is partitioned into chunks of 1 bit starting at the lowest bit. The different shades of gray indicate the different 1 bit chunks. The 64 1 bit chunks of the time value are XORed with each other to form a 1 bit value. With the XORing of all 1 bit chunks with each other, any information theoretical entropy that is present in the time stamp will be preserved when folding the value into the 1 bit. But as we fold it into 1 bit, the maximum entropy the time stamp can ever add to the entropy pool is, well, 1 bit. The folding operation is done as often as specified in the loop count.

3.1.3 Von-Neumann Unbias Operation

According to RFC 1750 section 5.2.2, a Von-Neumann unbias operation can be considered to remove any potential skews that may be present in the bit stream of the noise source. The operation is used to ensure that in case skews are present, they are eliminated. The unbias operation is only applicable if the individual consecutive bits are considered independent. Chapter 5 indicates the independence of these individual bits.

To perform the Von-Neumann unbias operation, two independently generated folded bits are processed.

3.1.4 Adding Unbiased Folded Time Delta To Entropy Pool

After obtaining the 1 bit folded and unbiased time stamp, how is it mixed into the entropy pool? The lower 64 bit value in Figure 4 indicates the entropy pool. The 1 bit folded value is XORed with 1 bit from the entropy pool.

But which bit is used? The rotation to the left by 1 bit that concludes the entropy collection loop provides the answer. When the entropy collection loop perform the very first iteration, the 1 bit is XORed into bit 0 of the entropy pool. Now, that pool is rotated left by 1 bit. That means that bit 63 before the rotation becomes bit 0 after the rotation. Thus, the next round of the entropy collection loop XORes the 1 bit folded time stamp again into bit 0 which used to be bit 63 in the last entropy collection loop iteration[2].

The reason why the rotation is done with the value 1 is due to the fact that we have 1 bit we want to add to the pool. The way how the folded bit values are added to the entropy pool can be viewed differently from a mathematical standpoint when considering 64 1 bit values: instead of saying that each of the 64 1 bit value is XORed independently into the entropy pool and the pool value is then rotated, it is equivalent to state that 64 1 bit values are concatenated and then the concatenated value is XORed into the entropy pool. The reader shall keep that analogy in mind as we will need it again in Section 5.

3.2 Generation of Random Number Bit Stream

We now know how one 64 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.

[2]Note, Figure 4 illustrates that the the folded bit of the time delta is moved over the 64 bit entropy pool as indicated with the bold black box (a.k.a the "slider"). Technically, the slider stays at bit 0 and the entropy pool value rotates left. The end result of the mixing of the folded bit into the entropy pool, however, is identical, regardless whether you rotate the entropy pool left or move the slider to the right. To keep the figure illustrative, it indicates the movement of the slider.

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

1. Check whether the requested size is smaller than 64 bits. If yes, generate one 64 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 64 bit random number is generated.

2. If the requested size is larger than 64 bits, generate one random number, copy it to the target. Reduce the requested size by 64 bits and decide now whether the remaining requested bits are larger or smaller than 64 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 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)` function which validates some basic properties of the time stamp. Only if this validation succeeds, the CPU Jitter random number generator can be used.

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 zeroing 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 FIPS 140-2 continuous test is primed by copying the random number generated in step 3 into the comparing value and again triggering the entropy collection loop for a fresh random number.

3.4 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 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.

3.5 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.

3.6 FIPS 140-2 Continuous Self Test

If the consuming application enables a FIPS 140-2 compliant mode – which is observable by the CPU Jitter random number generator callback of `jent_fips_enabled` – the FIPS 140-2 mode is enabled.

This mode ensures that the continuous self test is enforced as defined by FIPS 140-2.

3.7 Intended Method of Use

The CPU Jitter random number generator must be compiled without optimizations. The discussion in Section 5.1 supported by Appendix F explains the reason.

The interface discussed in Section 3.2 is implemented such that a caller requesting an arbitrary number of bytes is satisfied. The output can be fed through a whitening function, such as a deterministic random number generator or a hash based cryptographically secure whitening function. The appendix provides various implementations of linking the CPU Jitter random number generator with deterministic random number generators.

However, the output can also be used directly, considering the statistical properties and the entropy behavior assessed in the following chapters. The question, however, is whether this is a wise course of action. Whitening shall help to protect the entropy that is in the pool against observers. This especially a concern if you have a central entropy source that is accessed by multiple users – where a user does not necessarily mean human user or application, since a user or an application may serve multiple purposes and each purpose is one "user". The CPU Jitter random number generator is designed to be instantiated multiple times without degrading the different instances. If a user employs its own private instance of the CPU Jitter random number generator, it may be questionable whether a whitening function would be necessary.

But bottom line: it is a decision that the reader or developer employing the random number generator finally has to make. The implementations offered in the appendices offer the connections to whitening functions. Still, a direct use of the CPU Jitter random number generator is offered as well.

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` must 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 other POSIX systems, the `clock_gettime` function is available for this operation.
  - Linux kernel space: In the Linux kernel, the `get_cycles` function obtains this information. The directory `arch/` contains various assembler implementations for different CPUs to avoid using an operating system service. If `get_cycles` returns 0, which is possible on several architectures, such as MIPS, 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 `get_cycles` 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_malloc` is a wrapper for the `malloc` function call to obtain memory.
- `jent_free` is a wrapper for calling the `free` function to release the memory.

Loop count	0	1	2	3	4	Bit sum	Figure
1	0	1	1	0	0	N/A	N/A
2	0	0	0	1	0	N/A	N/A
3	1	1	0	0	1	4	5
Result 1	1	2	1	1	1	6	7
Result 2	1	2	1	2	1	7	9

Table 1: Example description of tests

- `__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`

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 Section 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 the flag `CRYPTO_CPU_JITTERENTROPY_STAT`, instrumentations are added to the code that obtain the data for the following graphs and distributions. The tests can be automatically reperformed 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 5 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.

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.

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 5 contains the distribution of the bit sum for the generated random numbers in user space.

In addition, the kernel space distribution is given in Figure 6 – they are almost identical and thus show the same behavior of the CPU Jitter random number generator

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

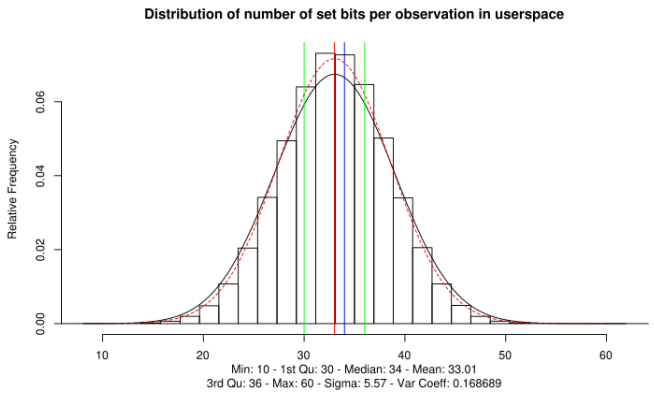

Figure 5: Bit sum of last round of entropy collection loop user space

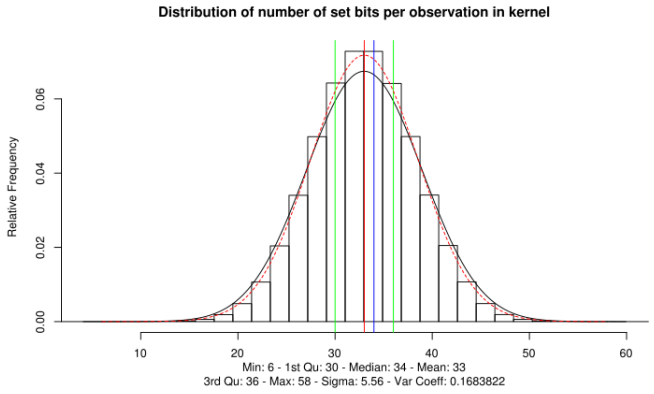

Figure 6: Bit sum of last round of entropy collection loop kernel space

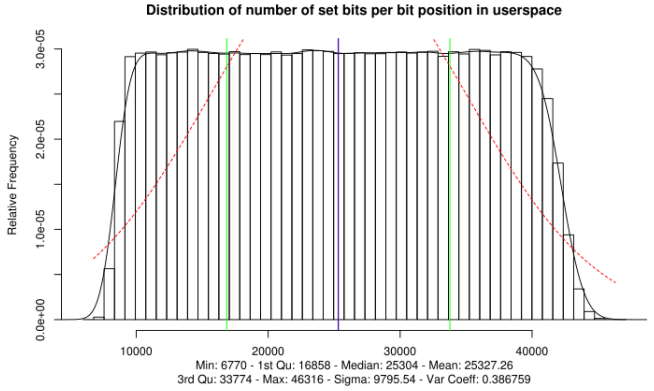

Figure 7: Bit sum of set bits per bit position in user space

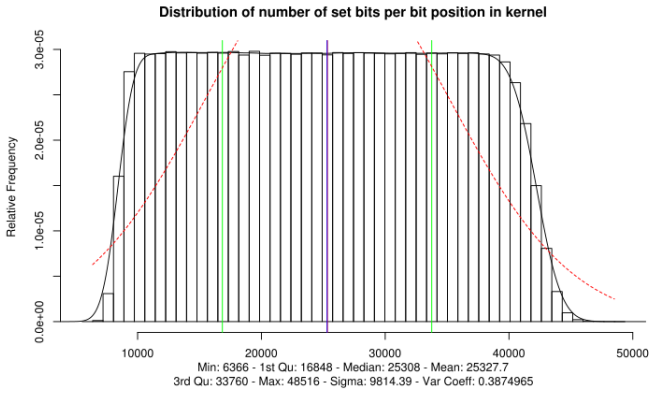

Figure 8: Bit sum of set bits per bit position in kernel space

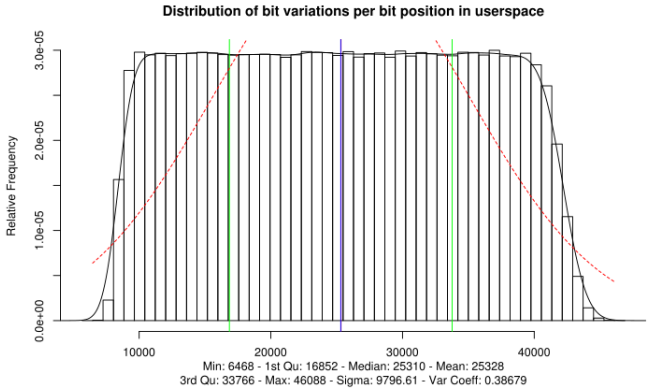

Figure 9: Bit sum of bit variations per bit position in user space

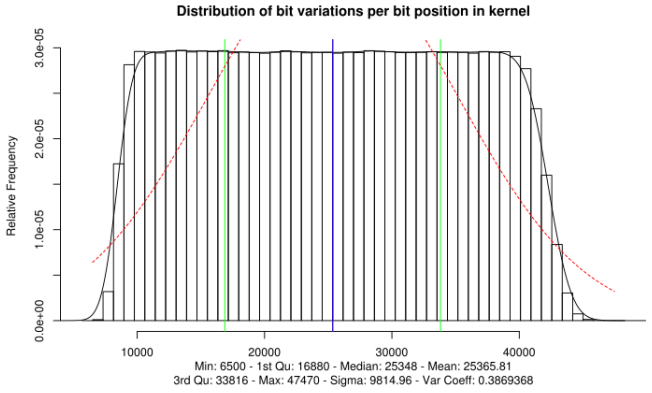

Figure 10: Bit sum of bit variations per bit position in kernel space

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 7.

Again, for the kernel we have an almost identical distribution shown in Figure 8. And again, we conclude that the behavior of the CPU Jitter random number generator in both worlds is identical.

Just like above, the plot for the kernel space is given in Figure 10.

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 9.

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 Section 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 11 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.

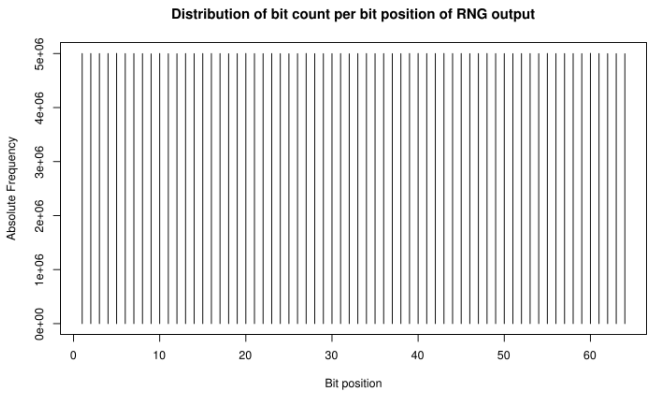

Figure 11: Distribution of bit count per bit position of RNG output

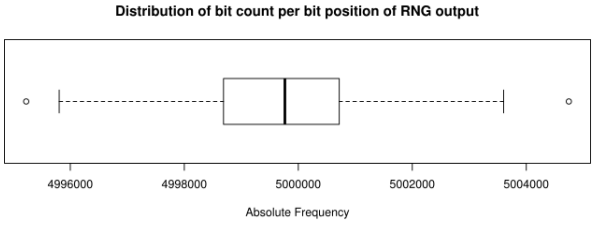

Figure 12: Box plot of variations in bit count per bit position of RNG output

Figure 11 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 12 that shows the variance when focusing on the upper end of the columns.

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 11.

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:

```
$ 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-Square test showed the test result of 50%, i.e. a perfect result – but even the shown 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.
- Fiasco.OC microkernel;

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.

Analyzing the output bit stream shows that the Chi-Square test of `ent` in both byte-wise and bit-wise output will result in the value of 0.01 / 100.00 which indicates a bit stream that is not random. This is readily clear, because the time delta calculation always returns the same value: 23.

Important remark: The mentioned test can only be conducted when the CPU Jitter random number generator initialization function of `jent_entropy_init(3)` is not called. This function implements a number of statistical tests of the time source. In case the time source would operate in static increments, the initialization function would detect this behavior and return an error.

If the CPU Jitter random number generator would be used with a cryptographic secure whitening function, the outlined "anti" test would *not* show any problems in the output stream. That means that a cryptographic whitening function would hide potential entropy source problems!

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 between 1 and 4 using the following code:

```
static unsigned int pad = 0;
static __u64 tmp = 0;
static inline void jent_get_nstime(__u64 *out)
{
        tmp += 23;
        pad++;
        *out = (tmp + (pad & 0x3));
}
```

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

Using such a pattern would again fail the `ent` test as the Chi-Square test is at 100 or 0.01 and the data stream can be compressed. Thus, such a time stamp increment would again be visible in the statistical analysis of this chapter.

In addition to the Chi-Square test, the measurements of the second derivation of the time stamp, the variations of time deltas, would present very strange patterns like, zero, or spikes, but no continuously falling graph as measured.

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 various tests.

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 Section 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 64 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. With Section 5.2, the explanation is given how the entropy that is present in the root cause, the CPU execution time jitter, is harvested, maintained through the processing of the random number generator and accumulated in the entropy pool. This section provides the information theoretical background to back up the statistical analyses given in Section 4.

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.

5.1 Base Entropy Source

As outlined in Section 3, the variations of the time delta is the source of entropy. Unlike the graphs outlined in Section 2 where two time stamps are invoked immediately after each other, the CPU Jitter random number generator places the folding loop between each time stamp gathering. That implies that the CPU jitter over the folding loop is measured and used as a basis for entropy.

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

- The result of the folding loop shall return a one bit value that has one bit of entropy.
- The delta of two time stamps before and after the folding loop is given to the folding loop to obtain the one bit value.

When viewing both findings together, we can conclude that the CPU jitter of the time deltas each folding loop shows *must* exceed 1 bit of entropy. Only this way we can ensure that the folded time delta value has one bit of entropy – see Section 5.2.1 for an explanation why the folding operation retains the entropy present in the time delta up to one bit.

Tests are implemented that measure the variations of the time delta over an invocation of the folding loop. The tests are provided with the `tests_userspace/timing/jitterentropy-foldtime.c` test case for user space, and the `stat-fold` DebugFS file for testing the kernel space. To ensure that the measurements are based on the worst-case analysis, the user space test is compiled with `-O2` optimization[3]. The kernel space test is compiled with the same optimization as the kernel itself.

The design of the folding loop in Section 3.1.2 explains that the number of folding loop iterations varies between 2^0 and 2^4 iterations. The testing of the entropy of the folding loop must identify the lower boundary and the upper boundary. The lower boundary is the minimum entropy the folding loop at least will have: this minimum entropy is the entropy observable over a fixed folding loop count. The test uses 2^0 as the fixed folding loop

[3]The CPU execution time jitter varies between optimized and non-optimized binaries. Optimitzed binaries show a smaller jitter compared to non-optimized binaries. Thus, the test applies a worst case approach with respect to the optimizations, even though the design requires the compilation without optimizations.

count. On the other hand, the upper boundary of the entropy is set by allowing the folding loop count to float freely within the above mentioned range.

It is expected that the time stamps used to calculate the folding 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 13 presents the lower boundary of the folding loop 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 14 provides the upper boundary of the folding loop. With the graph of the upper boundary, we see 16 spikes which are the spikes of the lower boundary scattered by the folding loop counter. If the folding loop counter is 1, the variation of the time delta is centered around a lower value than the variations of a folding loop counter of 2 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 two graphs use the time deltas of 10,000,000 invocations of the folding 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 lower boundary shows a Shannon Entropy above 2.9 bits and the upper boundary a Shannon Entropy above 6.7 bits.

In addition to the user space measurements, Figures 15 and 16 present the lower and upper boundary of the folding 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.

As this measurement is the basis of all entropy discussion, Appendix F shows the measurements for many

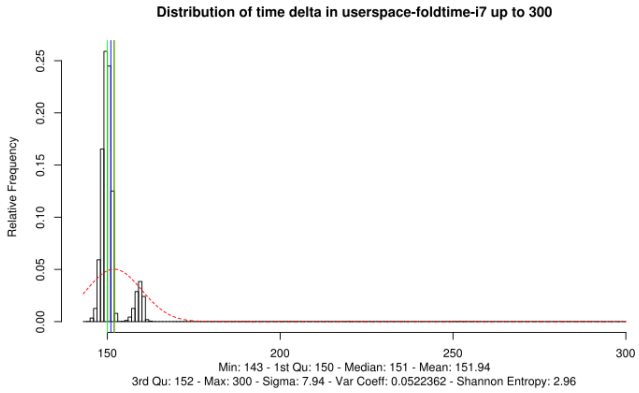

Figure 13: Lower boundary of entropy over folding loop in user space

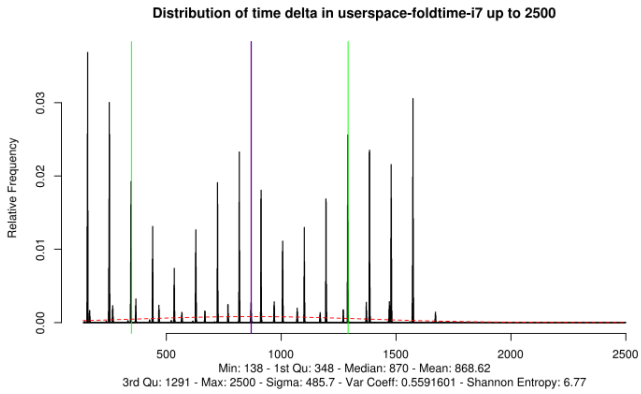

Figure 14: Upper boundary of entropy over folding loop in user space

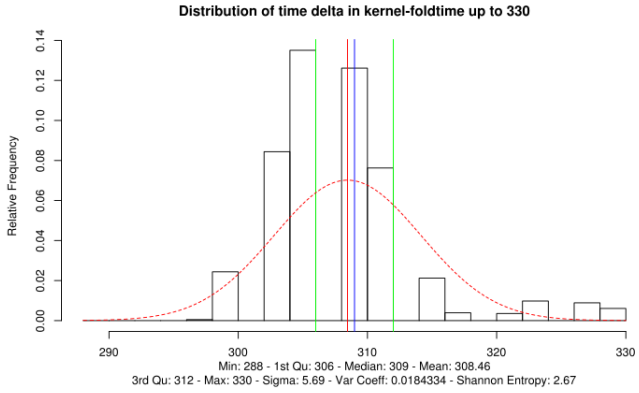

Figure 15: Lower boundary of entropy over folding loop in kernel space

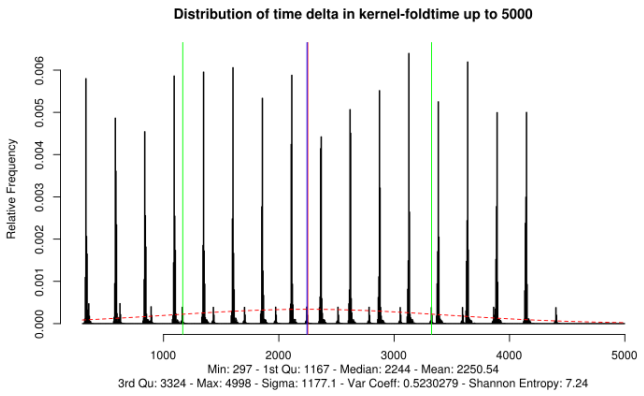

Figure 16: Upper boundary of entropy over folding loop in kernel space

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 compilation to show the difference.

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.

The reader should also consider that the measured Shannon Entropy is a conservative measurement as the test invokes the folding 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 folding 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 Section 4. If patterns would occur, some of the statistical tests would indicate problems. Specifically the Chi-Square test is very sensitive to any patterns. Moreover, the "anti" tests presented in Section 4.3 explain that patterns are easily identifiable.

5.1.1 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 folding 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[4] where all measurements of time deltas above 600 were removed to make the graph more readable (i.e. the outliers are removed). It is interesting to see that the execution time has a downward trend that stabilizes after some 60,000 folding loops. 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.

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.

The constant variations support the case that the CPU execution time jitter is agnostic of the with frequency scaling and power management levels.

[4]The measurements of the folding 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.

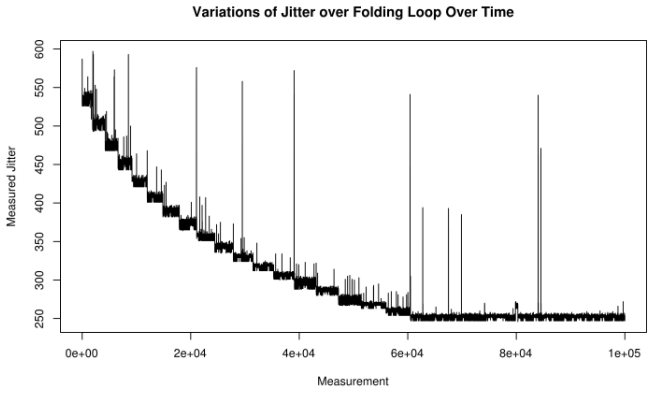

Figure 17: Variations of the execution time jitter over time when performing folding loop jitter measurements with Frequency Scaling / Power Management

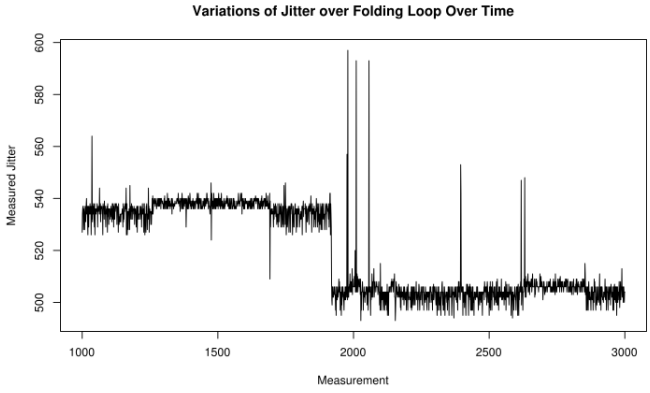

Figure 18: Variations of the execution time jitter over time when performing folding loop jitter measurements with Frequency Scaling / Power Management – "zoomed in at measurements 1,000 - 3,000"

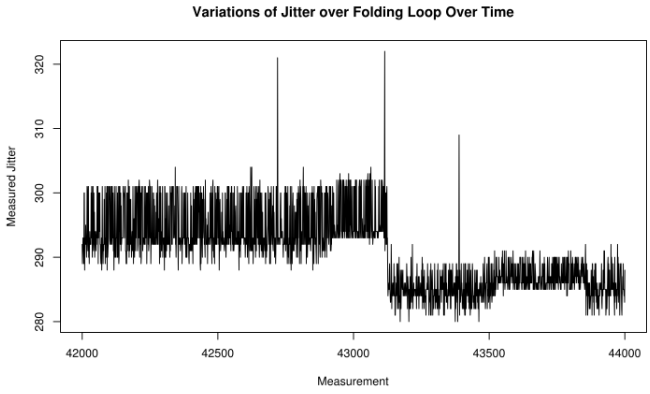

Figure 19: Variations of the execution time jitter over time when performing folding loop jitter measurements with Frequency Scaling / Power Management – "zoomed in at measurements 42,000 - 44,000"

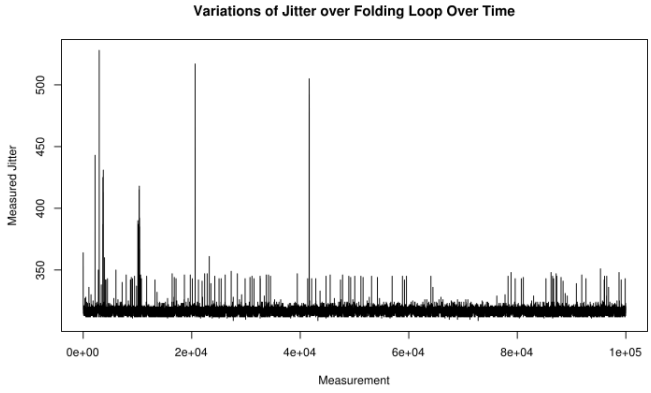

Figure 20: Variations of the execution time jitter over time when performing folding loop jitter measurements with Frequency Scaling / Power Management disabled

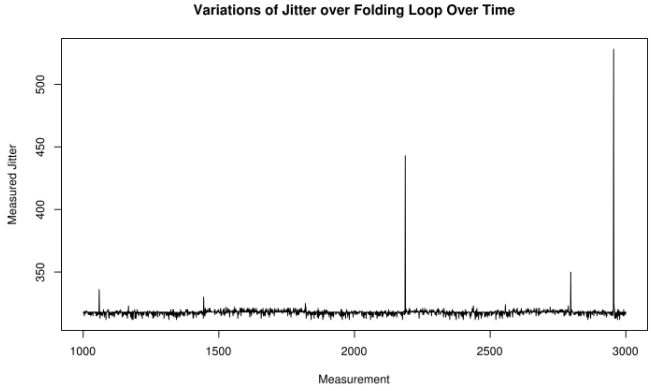

Figure 21: Variations of the execution time jitter over time when performing folding loop jitter measurements with Frequency Scaling / Power Management disabled – "zoomed in at measurements 1,000 - 3,000"

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.

5.2 Flow of Entropy

Entropy is a is a phenomenon that is typically characterized with the formula for the Shannon Entropy H

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

where N is the number of samples, and p_i is the probability of sample i. As the Shannon Entropy formula uses

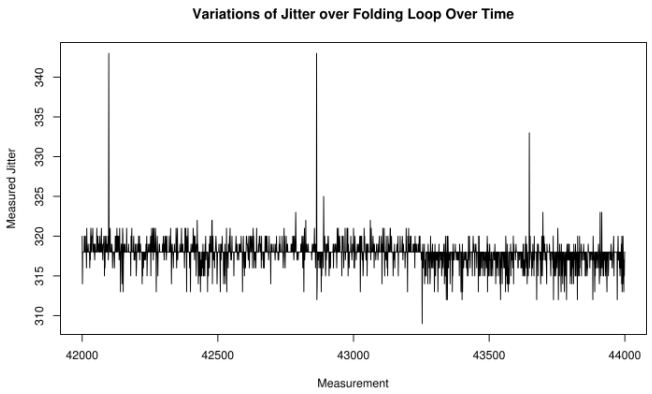

Figure 22: Variations of the execution time jitter over time when performing folding loop jitter measurements with Frequency Scaling / Power Management disabled – "zoomed in at measurements 42,000 - 44,000"

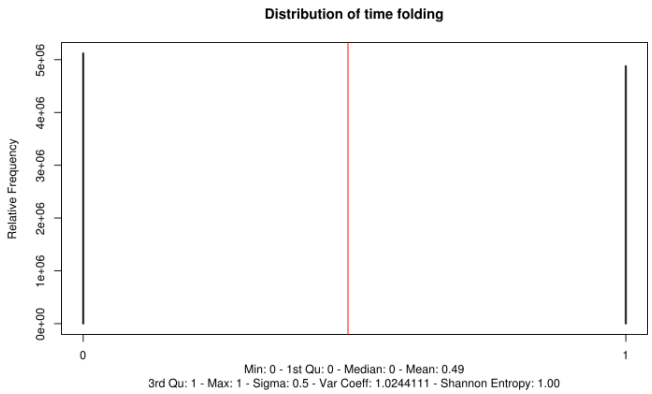

Figure 23: Measurement of time folding operation

the logarithm at base 2, that formula results in a number of bits of entropy present in an observed sample.

Considering the logarithm in the Shannon Entropy formula one has to be careful on which operations can be applied to data believed to contain entropy to not lose it. The following operations are allowed with the following properties:

- Concatenation of bit strings holding entropy implies that the combined string contains the combination of both entropies, i.e. the entropy value of both strings are added. That is only allowed when both observations are independent from each other.
- A combination of the bit strings of two *independent* observations using XOR implies that the resulting string holds the entropy equaling to larger entropy of both strings – for example XORing two strings, one string with 10 bits in size and 5 bits of entropy and another with 20 bits holding 2 bits results in a 20 bit string holding 5 bits of entropy. The key is that even a string with 0 entropy XORed with a string holding entropy will not diminish the entropy of the latter.

Any other operation, including partial overlapping concatenation of strings will diminish the entropy in the resulting string in ways that are not easily to be determined. These properties set the limit in which the CPU Jitter random number generator can process the time stamps into a random bit stream.

The graphs about the distribution of time deltas and their variations in Section 5.1 include an indication of the Shannon Entropy which is based on the observed samples using the mentioned formula for the Shannon Entropy. In each case, the Shannon Entropy is way above 1 bit – a value which is fundamental to the following discussion.

5.2.1 First Operation: Folding of Time Delta

According to the implementation illustrated with Figure 3, the first operation after the CPU Jitter random number generator obtains a time delta is the folding operation. The list of allowed operations include the XOR operation. The folding is an XOR operation of the 64 1 bit slices of the 64 bit time stamp. The XOR operation does not diminish the entropy of the overall time stamp when considered as slices. The overall time delta is expected to have more than 1 bit of entropy according to figures in Section 5.1. The string size after the folding is 1 bit and can thus not hold more than 1 bit of entropy.

To measure that entropy, the folding operation is closely analyzed with the test `tests_userspace/timing/jitterentropy-folding.c`. This test performs the folding operation as illustrated in the left hand side of Figure 3, i.e. a time delta is created which is folded. The folded value is recorded and a folding operation is performed. The distribution of the bit value – an integer ranging from 0 to 1 – resulting from the folding operation is recorded. Figure 23 shows the distribution of this test when measuring 10,000,000 invocations of that time stamp with the folding operation applied.

The distribution shows that both values have an equal chance of being selected. That implies that the Shannon Entropy is 1.0 as recorded in the legend of the diagram. We conclude that the folding operation will retain 1 bit of entropy provided that the input, i.e. the timing value holds 1 or more bits of entropy.

Note, the repetition of the folding loop is of no harm to the entropy as the same value is calculated during each folding loop execution.

5.2.2 Second Operation: Von-Neumann Unbias

The Von-Neumann unbias operation does not have an effect on the entropy of the source. The mathematical proof is given in the document A proposal for: Functionality classes for random number generators Version 2.0 by Werner Schindler section 5.4.1 issued by the German BSI.

The requirement on using the Von-Neumann unbias operation rests on the fact that the input to the unbias operation are two independent bits. The independence is established by the following facts:

1. The bit value is determined by the delta value which is affected by the CPU execution jitter. That jitter is considered independent of the CPU operation before the time delta measurement,
2. The delta value is calculated to the previous execution loop iteration. That means that two loop iterations generate deltas based on each individual loop. The delta of the first loop operation is neither part of the delta of the second loop (e.g. when the second delta would measure the time delta of both loop iterations), nor is the delta of the second loop iteration affected by the first operation based on the finding in bullet 1.

5.2.3 Third Operation: Entropy Pool Update

What is the next operation? Let us look again at Figure 3. The next step after folding and unbiasing is the mixing of the folded value into the entropy pool by XORing it into the pool and rotating the pool.

The reader now may say, these are two distinct operations. However, in Section 3.1.2 we already concluded that the XOR operation using 64 1 bit folded values together with the rotation by 1 bit of the entropy pool can mathematically be interpreted as a concatenation of 64 1 bit folded values into a 64 bit string. Thus, both operations are assessed as a concatenation of the individual folded bits into a 64 bit string followed by an XOR of that string into the entropy pool.

Going back to the above mentioned allowed operations with bit strings holding entropy, the concatenation operation adds the entropy of the individual bit strings that are concatenated. Thus, we conclude that the concatenation of 64 strings holding 1 bit of entropy will result in a bit string holding 64 bit of entropy.

When concatenating additional n 1 bit strings into the 64 bit entropy pool will not increase the entropy any more as the rotation operation rolls around the 64 bit value and starts at the beginning of that value again. When the entropy collection loop counter has a value that is not divisible by 64, the last bit string XORed into the entropy pool is less than 64 bits – for example, the counter has the value 260, the 4 last folded bits generated by the loop will form a 4 bit string that is XORed into the entropy pool. This last bit string naturally contains less than 64 bits of entropy – the maximum entropy it contains is equal to the number of bits in that string. Considering the calculation rules for entropy mentioned above, XORing a string holding less entropy with a string with more entropy will not diminish the entropy of the latter. Thus, the XORing of the last bits into the entropy pool will have no effect on the entropy of the entropy pool.

There is a catch to the calculation: the math only applies when the individual observations, i.e. the individual 1 bit folded time delta values, are independent from each other. The argument supporting the independence of the individual time deltas comes back to the fundamental property of the CPU execution time jitter which has an unpredictable variation. Supportive is the finding that one entropy collection loop iteration, which generates a 1 bit folded value, has a much wider distribution compared to Figure 2 – the reader may particularly consider the standard deviation. This variation in the execution time of the loop iteration therefore breaks any potentially present dependencies between adjacent loop counts and their time deltas. Note again, the time deltas we collect only need 1 bit of entropy. Looking at Figure 24 which depicts the distribution of the execution time of one entropy loop iteration, we see that the variation and its included Shannon Entropy is high enough

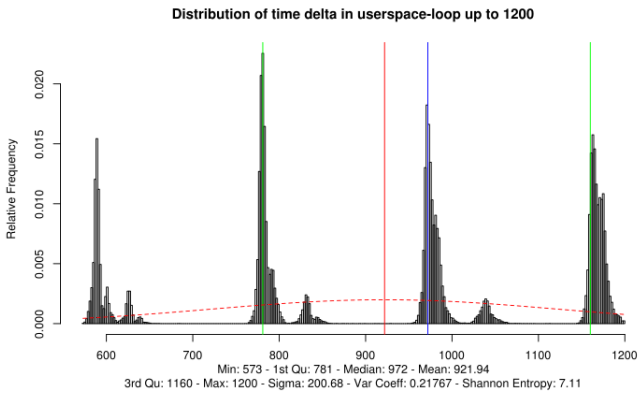

Figure 24: Distribution of execution time of one entropy collection loop iteration

to support the conclusion of an independence between time deltas of adjacent loop iterations.

Thus, we conclude that our entropy pool holds 64 bit of entropy after the conclusion of the mixing operation.

5.2.4 Fourth Operation: Generation of Output String

The fourth and last operation on the bit string holding entropy is the generation of the string of arbitrary length.

The generation of the output string is performed by concatenating random numbers of the size of 64 bit with each other until the resulting bit string matches the requested size. The individual random numbers are generated by independent invocations of the entropy collection loop.

Using concatenation and the conclusion from the preceding sections[5], the entropy in the resulting bit string is equal to the number of bits in that string.

The CPU Jitter random number generator operates on 64 bit blocks – the length of the entropy pool. When the requested bit string length is not divisible by 64 bits, the last chunk concatenated with the output bit stream is therefore less than 64 bits with the reminding bits not given to the caller – note, the caller is only able to specify the output size in bytes and thus in 8-bit chunks. Why is this operation not considered to diminish the entropy of the last chunk below its number of bits? To find the answer, let us go back how the entropy pool is constructed: one bit of folded timer value known to have one bit of entropy is added to the pool. When considering the entropy pool as 64 segments of individual bits, every individual bit still contains 1 bit of entropy, because the only operation each single bit is modified with, is XOR. Thus, every bit in the bit string of the entropy pool holds one bit of entropy. This ultimately implies that when taking a subset of the entropy pool, that subset still has as much entropy as the size of the subset in bits.

[5]The entropy pool contains 64 bit of entropy after the completion of the random number generation.

5.3 Reasons for Chosen Values

The reader now may ask why the time delta is folded into one bit and not into 2 or even 4 bits. Using larger bit strings would reduce the number of foldings and thus speed up the entropy collection. Measurements have shown that the speed of the CPU Jitter random number generator is cut by about 40% when using 4 bits versus 2 bits or 2 bits versus 1 bit. However, the entire argumentation for entropy is based on the entropy observed in the execution time jitter illustrated in Section 5.1. The figures in this section support the conclusion that the Shannon Entropy measured in Section 5.1 is the absolute worst case. To be on the save side, the lower boundary of the measured entropy shall always be significantly higher than the entropy required for the value returned by the folding operation.

Another consideration for the size of the folded time stamp is important: the implications of the last paragraph in Section 5.2.4. The arguments and conclusions in that paragraph only apply when using a size of the folded time stamp that is less or equal 8 bits, i.e. one byte.

6 Conclusion

For the conclusion, we need to get back to Section 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.

In addition to these general goals, Section 1 lists a number of special goals. These goals are considered to be covered. A detailed assessment on the coverage of these goals is given in the original document.

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 Linux Kernel Implementation

The document describes in Section 1 the goals of the CPU Jitter random number generator. One of the goals is to provide individual instances to each consumer of entropy. One of the consumers are users inside the Linux kernel.

As described above, the output of the CPU Jitter random number generator is not intended to be used directly. Instead, the output shall be used as a seed for either a whitening function or a deterministic random number generator. The Linux kernel support provided with the CPU Jitter random number generator chooses the latter approach by using the ANSI X9.31 DRNG that is provided by the Linux kernel crypto API.

Figure 25 illustrates the connection between the entropy collection and the deterministic random number generators offered by the Linux kernel support. The interfaces at the lower part of the illustration indicate the Linux kernel crypto API names of the respective deterministic random number generators and the file names within `/sys/kernel/debug`, respectively.

Every deterministic random number generator instance is seeded with its own instance of the CPU Jitter random number generator. This implementation thus uses one of the design goals outlined in Section 1, namely multiple,

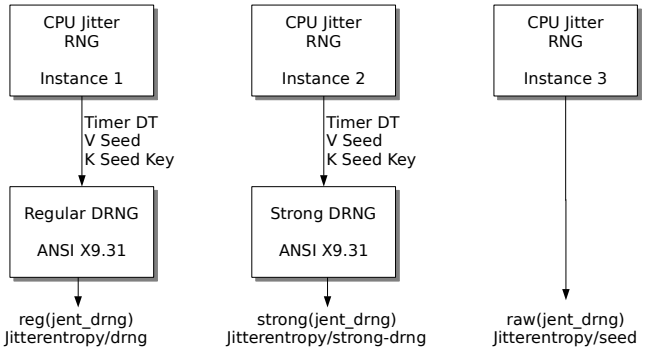

Figure 25: Using CPU Jitter RNG to seed ANSI X9.31 DRNGs

unrelated instantiations of the CPU Jitter random number generator.

The offered deterministic random number generators have the following characteristics:

- The regular deterministic random number generator is re-seeded with entropy from the CPU Jitter random number generator after obtaining `MAX_BYTES_RESEED` bytes since the last re-seed. Currently that value is set to 1 kilobytes. In addition, when reaching the limit of `MAX_BYTES_REKEY` bytes since the last re-key, the deterministic random number generator is re-keyed using entropy from the CPU Jitter random number generator. This value is currently set to 1 megabytes.

- The strong deterministic random number generator is re-seeded and re-keyed after the generator of `MAX_BYTES_STRONG_RESEED` bytes and `MAX_BYTES_STRONG_REKEY` bytes, respectively. The re-seeding value is set to 16 bytes, which is equal to the block size of the deterministic random number generator. This implies that the information theoretical entropy of one block of random number generated from the deterministic random number generator is always 16 bytes. The re-key value is set to 1 kilobytes.

- Direct access to the CPU Jitter random number generator is provided to the caller when raw entropy is requested.

Currently, the kernel crypto API only implements a full reset of the deterministic random number generators.

Therefore, the description given above is the plan after the kernel crypto API has been extended. Currently, when hitting the re-seed threshold, the deterministic random number generator is reset with 48 bytes of entropy from the CPU Jitter random number generator. The re-key value is currently not enforced.

B.1 Kernel Crypto API Interface

When compiling the source code with the configuration option `CRYPTO_CPU_JITTERENTROPY_KCAPI`, the kernel crypto API bonding code is compiled. That code registers the mentioned deterministic random number generators with the kernel crypto API. The bonding code provides a very thin wrapper around the management code for the provided random number generators.

The deterministic random number generators connected with as well as the direct access to the CPU Jitter random number generator are accessible using the following kernel crypto API names:

reg(jent_rng) Regular deterministic random number generator

strong(jent_rng) Strong deterministic random number generator

raw(jent_rng) Direct access to the CPU Jitter random number generator which returns unmodified data from the entropy collection loop.

When invoking a reset operation on one of the deterministic random number generator, the implementation performs the re-seed and re-key operations mentioned above on this deterministic random number generator irrespectively whether the thresholds are hit.

A reset on the `raw(jent_rng)` instance is a noop.

B.2 Kernel DebugFS Interface

The kernel DebugFS interface offered with the code is *only* intended for debugging and testing purposes. During regular operation, that code *shall not* be compiled as it allows access to the internals of the random number generation process.

The DebugFS interface is compiled when enabling the `CRYPTO_CPU_JITTERENTROPY_DBG` configuration option. The interface registers the following files within the directory of `/sys/kernel/debug/jitterentropy`:

stat The `stat` file offers statistical data about the regular and strong random number generators, in particular the total number of generated bytes and the number of re-seeds and re-keys.

stat-timer This file contains the statistical timer data for one entropy collection loop count: time delta, delta of time deltas and the entropy collection loop counter value. This data forms the basis of the discussion in Section 4. Reading the file will return an error if the code is not compiled with `CRYPTO_CPU_JITTERENTROPY_STAT`.

stat-bits This file contains the three tests of the bit distribution for the graphs in Section 4. Reading the file will return an error if the code is not compiled with `CRYPTO_CPU_JITTERENTROPY_STAT`.

stat-fold This file provides the information for the entropy tests of the folding loop as outlined in Section 5.1. Reading the file will return an error if the code is not compiled with `CRYPTO_CPU_JITTERENTROPY_STAT`.

drng The `drng` file offers access to the regular deterministic random number generator to pull random number bit streams of arbitrary length. Multiple applications calling at the same time are supported due to locking.

strong-rng The `strong-drng` file offers access to the strong deterministic random number generator to pull random number bit streams of arbitrary length. Multiple applications calling at the same time are supported due to locking.

seed The `seed` file allows direct access to the CPU Jitter random number generator to pull random number bit streams of arbitrary lengths. Multiple applications calling at the same time are supported due to locking.

timer The `timer` file provides access to the time stamp kernel code discussed in Section 2. Be careful when obtaining data for analysis out of this file: redirecting the output immediately into a file (even a file on a TmpFS) significantly enlarges the measurement and thus make it look having more entropy than it has.

collection_loop_count This file allows access to the entropy collection loop counter. As this counter value is considered to be a sensitive parameter, this

file will return -1 unless the entire code is compiled with the CRYPTO_CPU_JITTERENTROPY_STAT flag. *This flag is considered to be dangerous* for normal operations as it allows access to sensitive data of the entropy pool that shall not be accessible in regular operation – if an observer can access that data, the CPU Jitter random number generator must be considered to deliver much diminished entropy. Nonetheless, this flag is needed to obtain the data that forms the basis of some graphs given above.

B.3 Integration with random.c

The CPU Jitter random number generator can also be integrated with the Linux /dev/random and /dev/urandom code base to serve as a new entropy source. The provided patch instantiates an independent copy of an entropy collector for each entropy pool. Entropy from the CPU Jitter random number generator is only obtained if the entropy estimator indicates that there is no entropy left in the entropy pool.

This implies that the currently available entropy sources have precedence. But in an environment with limited entropy from the default entropy sources, the CPU Jitter random number generator provides entropy that may prevent /dev/random from blocking.

The CPU Jitter random number generator is only activated, if jent_entropy_init passes.

B.4 Test Cases

The directory tests_kernel/kcapi-testmod/ contains a kernel module that tests whether the Linux Kernel crypto API integration works. It logs its information at the kernel log.

The testing of the interfaces exported by DebugFS can be performed manually on the command line by using the tool dd with the files seed, drng, strong-drng, and timer as dd allows you to set the block size precisely (unlike cat). The other files can be read using cat.

C Libgcrypt Implementation

Support to plug the CPU Jitter random number generator into libgcrypt is provided. The approach is to add the callback to the CPU Jitter random number generator into _gcry_rndlinux_gather_random. Thus, the CPU Jitter random number generator has the ability to run every time entropy is requested. Figure 26 illustrates how the CPU Jitter random number generator hooks into the libgcrypt seeding framework.

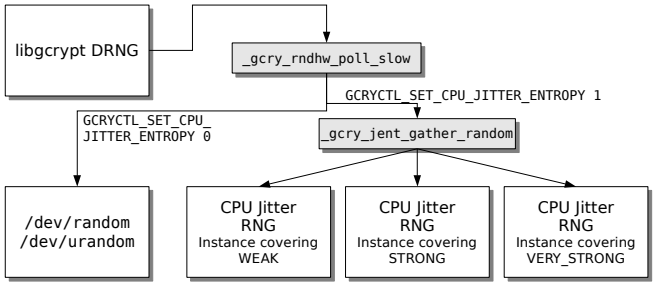

Figure 26: Use of CPU Jitter RNG by libgcrypt

The wrapper code around the CPU Jitter random number generator provided for libgcrypt holds the following instances of the random number generator. Note, the operation of the CPU Jitter random number generator is unchanged for each type. The goal of that approach shall ensure that each type of seed request is handled by a separate and independent instance of the CPU Jitter random number generator.

weak_entropy_collector Used when GCRY_WEAK_RANDOM random data is requested.

strong_entropy_collector Used when GCRY_STRONG_RANDOM random data is requested.

very_strong_entropy_collector Used when GCRY_VERY_STRONG_RANDOM random data is requested.

The CPU Jitter random number generator with its above mentioned instances is initialized when the caller uses GCRYCTL_SET_CPU_JITTER_ENTROPY with the flag 1. At this point, memory is allocated.

Only if the above mentioned instances are allocated, the wrapper code uses them! That means the callback from _gcry_rndlinux_gather_random to the CPU Jitter random number generator only returns random bytes when these instances are allocated. In turn, if they are not allocated, the normal processing of _gcry_rndlinux_gather_random is continued.

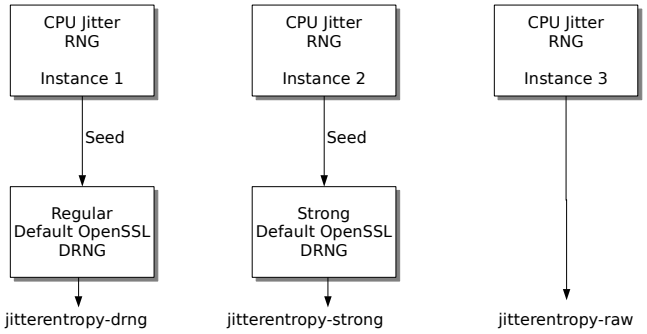

Figure 27: CPU Jitter random number generator seeding OpenSSL default DRNG

If the user wants to disable the use of the CPU Jitter random number generator, a call to `GCRYCTL_SET_CPU_JITTER_ENTROPY` with the flag 0 must be made. That call deallocates the random number generator instances.

The code is tested with the test application `tests_userspace/libgcrypt/jent_test.c`. When using `strace` on this application, one can see that after disabling the CPU Jitter random number generator, `/dev/random` is opened and data is read. That implies that the standard code for seeding is invoked.

See `patches/README` for details on how to apply the code to libgcrypt.

D OpenSSL Implementation

Code to link the CPU Jitter random number generator with OpenSSL is provided.

An implementation of the CPU Jitter random number generator encapsulated into different OpenSSL Engines is provided. The relationship of the different engines to the OpenSSL default random number generator is depicted in Figure 27.

The following OpenSSL Engines are implemented:

jitterentropy-raw The `jitterentropy-raw` engine provides direct access to the CPU Jitter random number generator.

jitterentropy-drng The `jitterentropy-drng` engine generates random numbers out of the OpenSSL default deterministic random number generator. This DRNG is seeded with 16 bytes out of CPU Jitter random number generator every 1024 bytes. After 1,048,576 bytes, the DRNG is seeded and re-keyed, if applicable, with 48 bytes after a full reset of the DRNG. When the Note, the intention of this engine implementation is that it is registered as the default OpenSSL random number generator using `ENGINE_set_default_RAND(3)`.

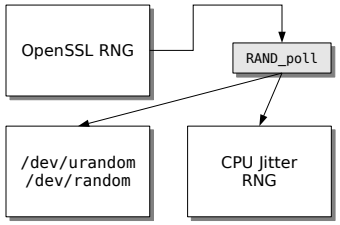

Figure 28: Linking OpenSSL with CPU Jitter RNG

jitterentropy-strong The `jitterentropy-strong` engine is very similar to `jitterentropy-drng` except that the reseeding values are 16 bytes and 1024 bytes, respectively. The goal of the reseeding is that always information theoretical entropy is present in the DRNG[6].

The different makefiles compile the different engine shared library. The test case `tests_userspace/openssl/jitterentropy-eng-test.c` shows the proper working of the respective CPU Jitter random number generator OpenSSL Engines.

In addition, a patch independent from the OpenSSL Engine support is provided that modifies the `RAND_poll` API call to seed the OpenSSL deterministic random number generator. The `RAND_poll` first tries to obtain entropy from the CPU Jitter random number generator. If that fails, e.g. the initialization call fails due to missing high-resolution timer support, the standard call procedure to open `/dev/urandom` or `/dev/random` or the EGD is performed.

Figure 28 illustrates the operation.

The code is tested with the test application `tests_userspace/openssl/jent_test.c`. When using `strace` on this application, one can see that after patching OpenSSL, `/dev/urandom` is not opened and thus

[6] For the FIPS 140-2 ANSI X9.31 DRNG, this equals to one AES block. For the default SHA-1 based DRNG with a block size of 160 bits, the reseeding occurs a bit more frequent than necessary, though.

not used. That implies that the CPU Jitter random number generator code for seeding is invoked.

See `patches/README` for details on how to apply the code to OpenSSL.

E 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.

F Folding Loop Entropy Measurements

Measurements as explained in Section 5.1 for different CPUs are executed on a large number of tests on different CPUs with different operating systems were executed. The test results demonstrate 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 and Intel x86 32 bit and 64 bit CPUs up to server CPUs of Intel Itanium, Sparc, POWER and IBM System Z;
- a large range of operating systems: Linux, OpenBSD, FreeBSD, NetBSD, AIX, OpenIndiana (OpenSolaris), AIX, z/OS, and microkernel based operating systems (Genode with microkernels of NOVA, Fiasco.OC, Pistachio);
- a range of different compilers: GCC, Clang and the z/OS C compiler.

The listing of the test results is provided at the web site offering the source code as well.

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.