What to Expect When You're Expecting

(to Assess the CPU Jitter Random Number Generator Against SP 800-90B)

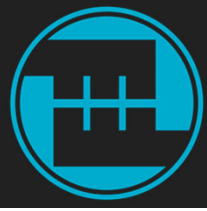
Joshua E. Hill, PhD

CMUF Entropy WG
20210727/
20210810

(Presentation Version 20210904-1)

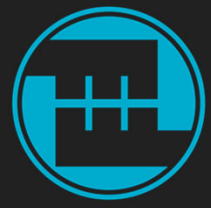
The CPU Jitter Random Number Generator (JEnt)

- Non-Physical Entropy Source
- Based on the difficulty an attacker has in guessing executing timing variations due to memory access and various CPU microarchitectural features.
  - One of several sources based on the same principle.
- Some nice characteristics that enable public discussion
  - Conceptually simple.
  - Open Source
  - Thoroughly documented by the author.
  - A good candidate for being an exemplar.
    - A reference 90B report could be released for such a design.
    - This is perhaps less interesting than it sounds, as the analysis would necessarily be somewhat hardware-specific.



The CPU Jitter Random Number Generator (JEnt)

- Widely used
  - Tightly integrated into many opensource distributions, both directly and through rngd.
  - Works on a variety of operating environments.
- JEnt has been Frequently integrated into FIPS modules.
- JEnt :
  - Has targeted SP 800-90B compliance since v2.2.0. (September 24, 2019)
  - Has been used as the basis of several SP 800-90B efforts.
  - v3.1.0 (July 25, 2021) seems to be broadly SP 800-90B compliant.
  - Earlier versions have some (correctable) problems.

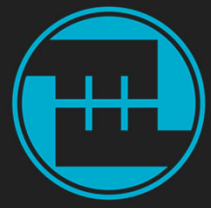


JEnt v3.1.0 SP 800-90B Validation Notes

JEnt v3.1.0 Notes

Make sure that JENT_CONF_DISABLE_LOOP_SHUFFLE is defined in jitterentropy.h.

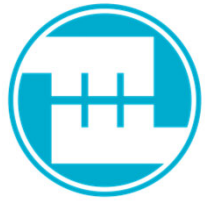
- This is currently the default (starting JEnt v3.0.2)
- This disables the pseudorandom behavior that can make assessments harder and mask failures from the health testing.



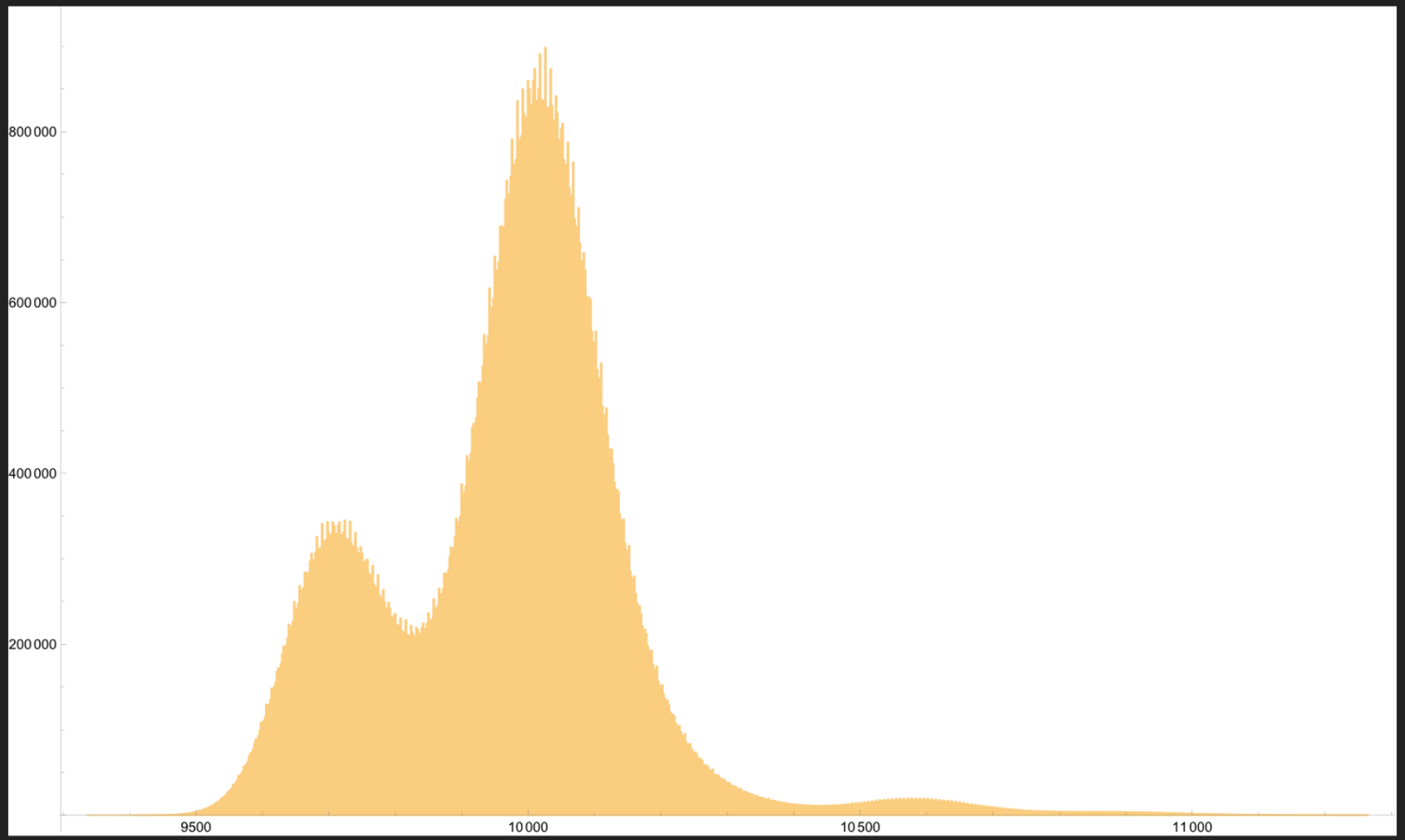
Loop Shuffle Code (Disabled)

When the macro is defined, this returns the fixed minimum value 2^{min}.

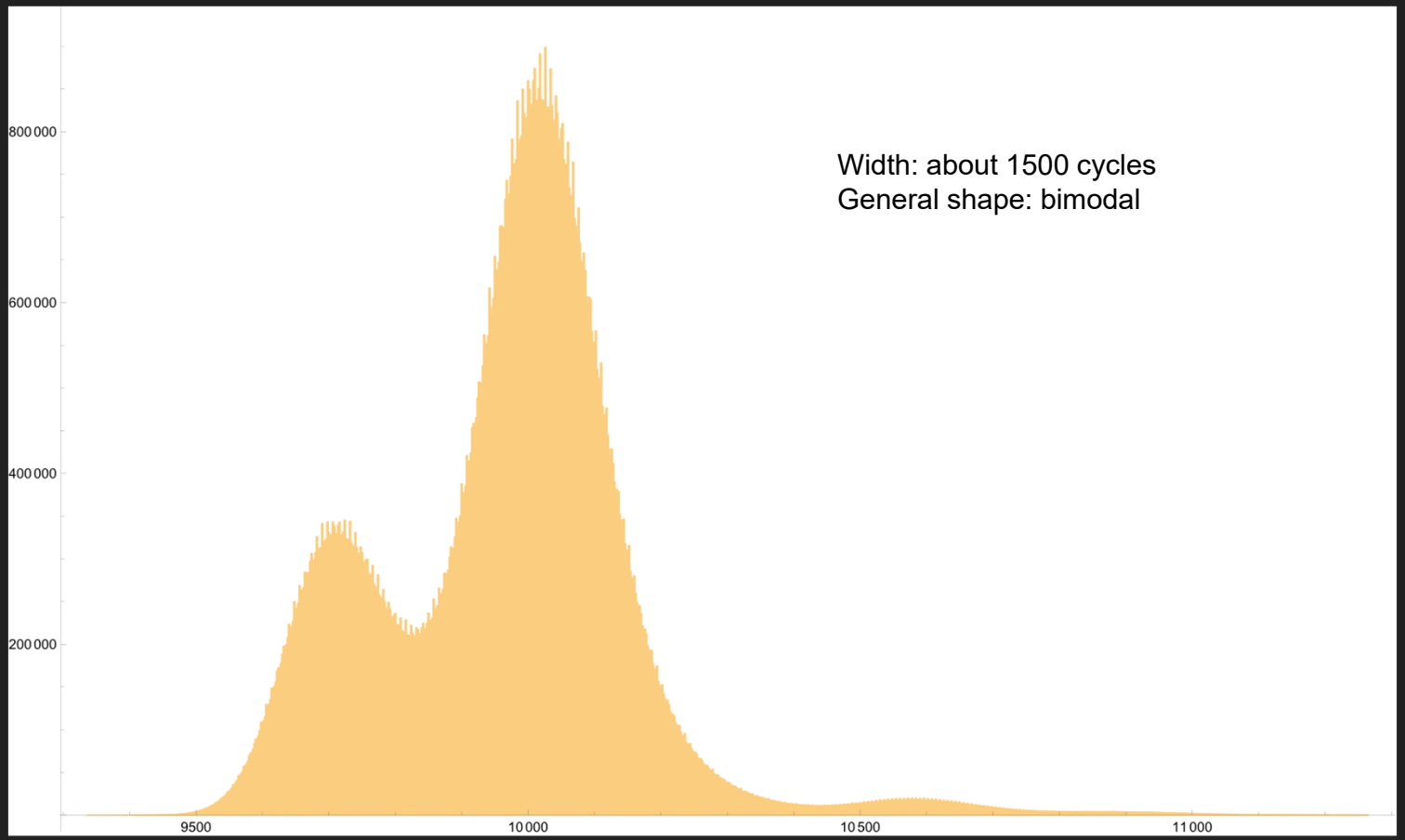
```
static uint64_t jent_loop_shuffle(struct rand_data *ec, unsigned int bits, unsigned int min)
{
#ifdef JENT_CONF_DISABLE_LOOP_SHUFFLE
…
        return (1<<min);
#else /* JENT_CONF_DISABLE_LOOP_SHUFFLE */
…
#endif /* JENT_CONF_DISABLE_LOOP_SHUFFLE */
}
```



Example Delta Distribution: Loop Shuffle Disabled



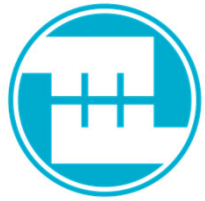
Example Delta Distribution: Loop Shuffle Disabled



Loop Shuffle Code Enabled

The current time XOR (part of the last output) is used as the basis for the varying part of the returned value.

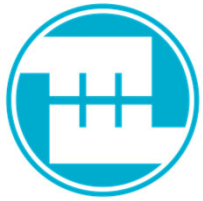
```
static uint64_t jent_loop_shuffle(struct rand_data *ec, unsigned int bits, unsigned int min)
{
#ifdef JENT_CONF_DISABLE_LOOP_SHUFFLE
...
#else /* JENT_CONF_DISABLE_LOOP_SHUFFLE */

        uint64_t time = 0;
        uint64_t shuffle = 0;
        ...
        if (ec) {
                jent_get_nstime_internal(ec, &time);
                time ^= ec->data[0];
        }

        ...XOR time down to length bits in shuffle...

        return (shuffle + (1<<min));
#endif /* JENT_CONF_DISABLE_LOOP_SHUFFLE */
}
```



Hash Count Variation

When “loop shuffle” is enabled, the hash loop count is a **pseudo-random** integer from 1 to 8 (inclusive)

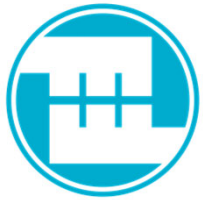
```
static void jent_hash_time(…)
{
…
#define MAX_HASH_LOOP 3
#define MIN_HASH_LOOP 0
        uint64_t hash_loop_cnt =
                    jent_loop_shuffle(ec, MAX_HASH_LOOP, MIN_HASH_LOOP);
        sha3_256_init(ctx);
…
        for (j = 0; j < hash_loop_cnt; j++) {
                sha3_update(ctx, ec->data, SHA3_256_SIZE_DIGEST);
                sha3_update(ctx, (uint8_t *)&time, sizeof(uint64_t));
                sha3_update(ctx, (uint8_t *)&j, sizeof(uint64_t));
…
                if (stuck || (j < hash_loop_cnt - 1))
                        sha3_final(ctx, itermediary);
                else
                        sha3_final(ctx, ec->data);
        }
}
```



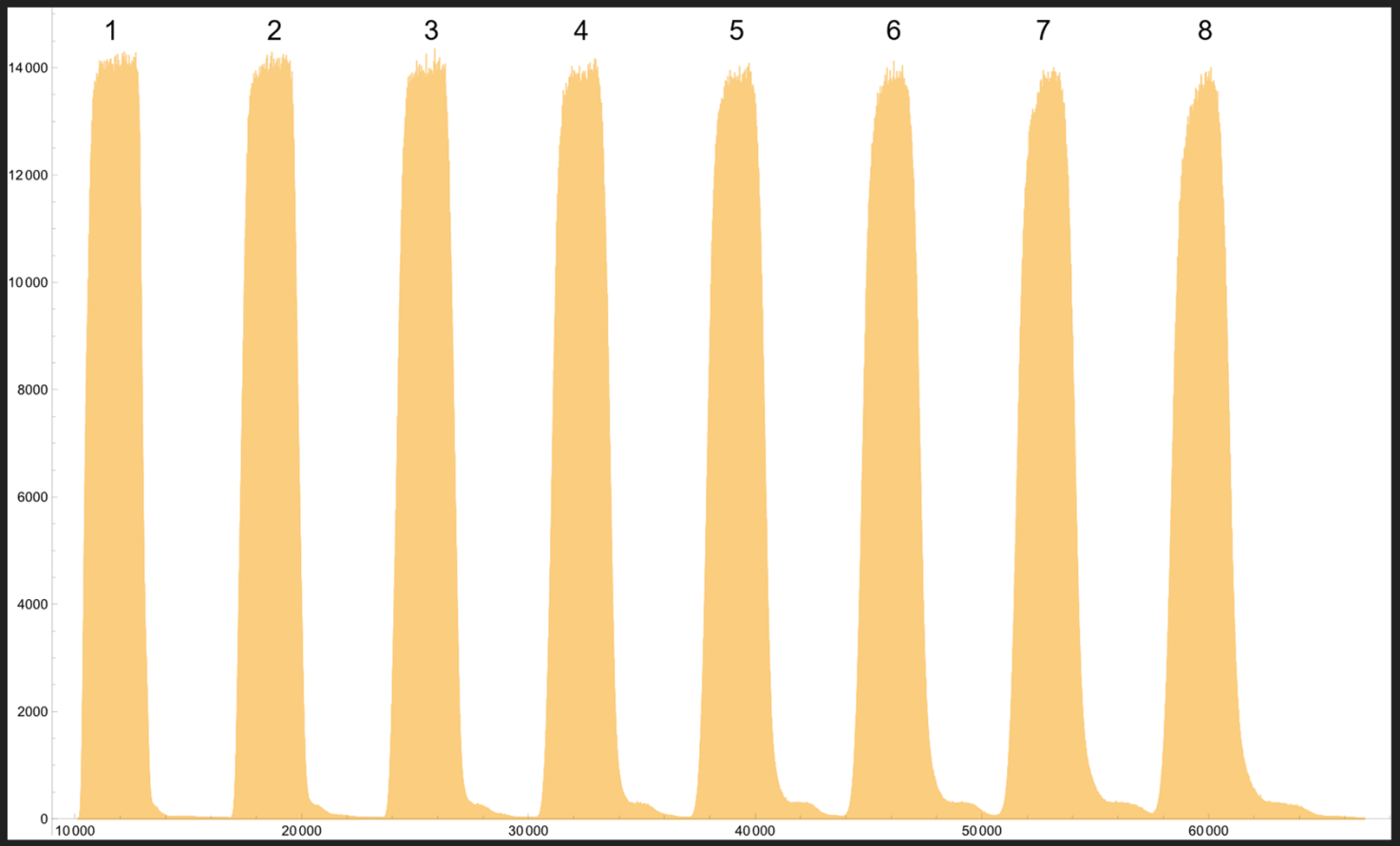
Memory Access Variation

When “loop shuffle” is enabled, the memaccess loop count is a **pseudo-random** integer from 129 to 256 (inclusive)

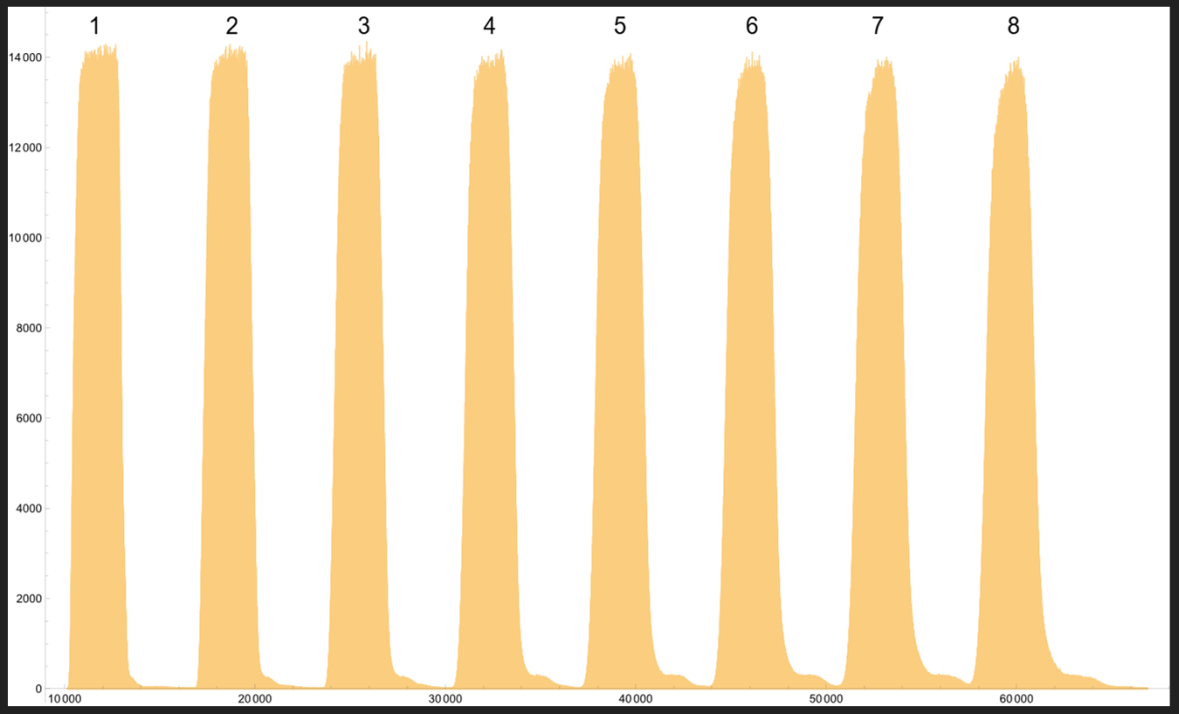
```
static void jent_memaccess(struct rand_data *ec, uint64_t loop_cnt)
{
…
#define MAX_ACC_LOOP_BIT 7
#define MIN_ACC_LOOP_BIT 0
        uint64_t acc_loop_cnt =
                jent_loop_shuffle(ec, MAX_ACC_LOOP_BIT, MIN_ACC_LOOP_BIT);
…
        for (i = 0; i < (ec->memaccessloops + acc_loop_cnt); i++) {
                …do memory access stuff…
        }
}
```

Example Delta Distribution: Loop Shuffle Enabled



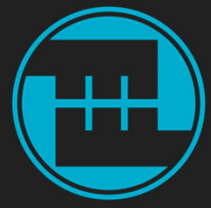
Example Delta Distribution: Loop Shuffle Enabled



Each sub-distribution is about 3400 cycles wide.

Per-mode expansion and obscured bimodal structure due to variation in memaccess rounds.

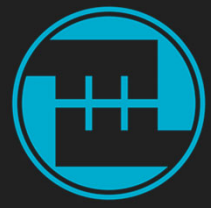
Eight Distinct modes correspond with number of SHA-3 iterations performed.



The Loop Shuffle: Why Do We Care?

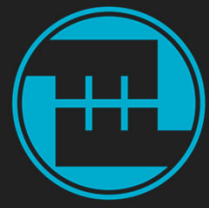
The pseudorandom “Loop Shuffle” does a few things:

- It masks failure conditions, so health testing largely isn’t meaningful.
- It artificially increases the statistically assessed entropy.
  - You get a “false” 3 bits of entropy (largely in higher-order bits) from the variable number of SHA-3 invocations.
  - You get some difficult-to-characterize additional variation in the low-order bits from the mem access loop count variation (less than 7 “false” bits of entropy).
    - This “smears” the underlying bimodal distribution and removes obvious features.



The Loop Shuffle: Historic Impact

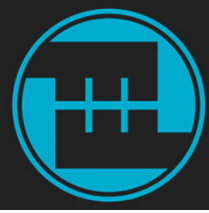
- There are platforms that historically seemed to support JEnt use (i.e., which passed start-up health testing) but whose output was actually largely pseudorandom.
- Many (particularly low-power embedded) systems never had access to fine-grained timers and/or don't possess the microarchitectural features that lead to uncertainty.
- This led to such entropy sources claiming more entropy than was available.
- After the changes in JEnt v3.0.2, this problem is no longer masked by default.
- Systems that newly experience persistent JEnt start-up failures likely never produced substantial entropy using this library.



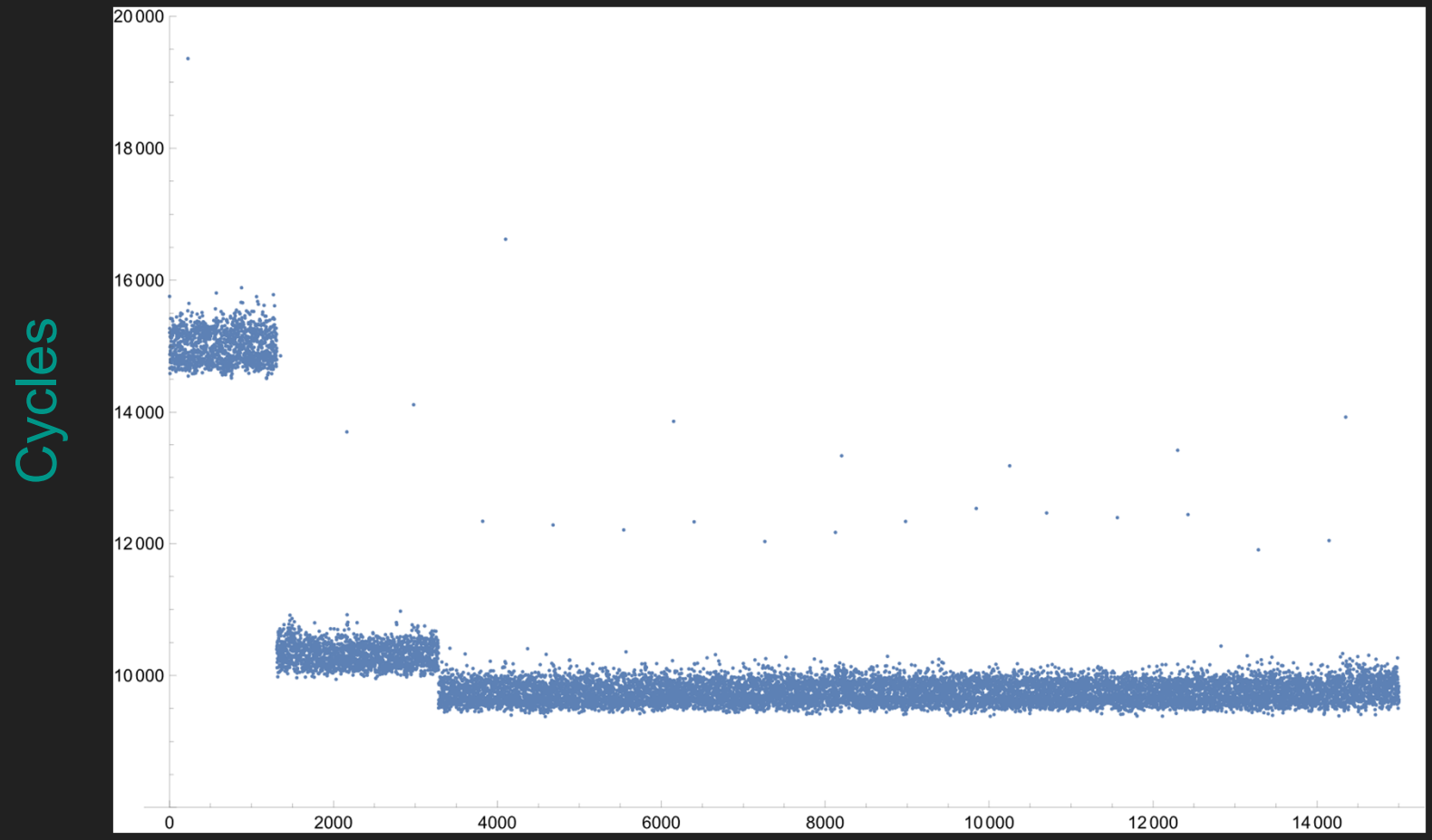
On the Cost of CPU Magic

For all testing leading to $H_{submitter}$, make sure to discard the first few thousand outputs so that the branch prediction/memory caching can initialize.

- Including both the transitory initial behavior and the long-term behavior may artificially inflate the assessed entropy, so it is useful to discard the samples that reflect this transitory behavior.
- When using jitterentropy-hashtime, the second column of the outputs can be used, as this data is generated after all the data in the first column is generated.

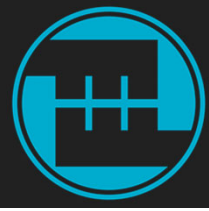


On the Cost of CPU Magic



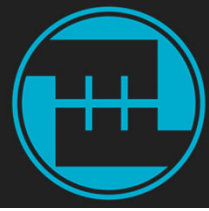
Start Me Up!

- Make sure the program using the jitter entropy library runs the jent_entropy_init function before running the jent_entropy_collector_alloc function.
  - This is not enforced by the library.
- The jent_entropy_init function runs some important tests:
  - Runs the start-up health tests,
  - Tests for the presence of a fine-grained hardware timer, and
  - Finds the timer GCD.



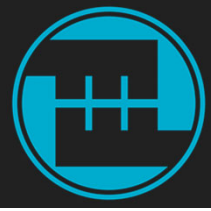
When Time Becomes a Loop

- If the internal timer is supported, then this process will set an internal global flag (jent_force_internal_timer) if the timer isn't sufficiently fine-grained to support JEnt (and the system has more than one CPU core, and has pthreads…)
- This result is likely to be consistent between invocations.
- Make sure to perform a validation using the same clock source used by the deployed entropy source.



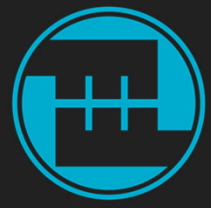
When Time Becomes a Loop

- On hardware with a fine-grained hardware timer force its use:
  - Disable the internal timer by removing the JENT_CONF_ENABLE_INTERNAL_TIMER macro, and/or
  - By consistently using the JENT_DISABLE_INTERNAL_TIMER flag when calling jent_entropy_collector_alloc.
- On hardware without a fine-grained timer (and which has multiple processors/cores and supports pthreads) force the internal timer using the JENT_FORCE_INTERNAL_TIMER flag when calling jent_entropy_collector_alloc.



When Time Becomes a Loop

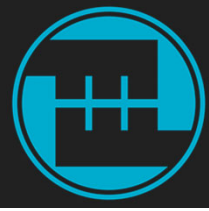
- If the entropy source doesn't force either approach, then either could be used, and the tester should independently assess the entropy available from the internal timer and the hardware timer, and the claimed entropy must be the minimum of these two values.



RCT, Let Me Be!

The RCT used is a strictly stronger version of the approved SP 800-90B health test.

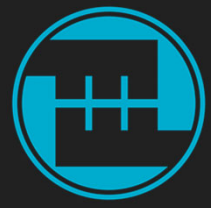
- The tester can use the same argument as used in SP 800-90B Section 4.4.1 to find an upper bound for α due (only) to repeated deltas for the modified RCT for the standard. An overall α must also account for repeated second deltas.
- The tester can use the assessed entropy to estimate a “90B nominal” value for α, and then use the entropy estimate provided by the (literal) MCV estimator to estimate a likely upper bound.
- The submitter/tester will need to produce an argument for compliance to SP 800-90B Section 4.5 Criteria a.
  - See [Müller 2021, Section 7.2.43].



Try Plugging It In…

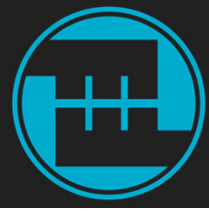
Verify that fips_enabled is set.

- Check that fips_enabled flag is set to 1 and/or
- Pass in the JENT_FORCE_FIPS flag when calling jent_entropy_collector_alloc.
- If the fips_enabled flag is not set, then health test failures are not reported and the library does not overseed as per the current SP 800-90C draft.



On the Perils of Advertising

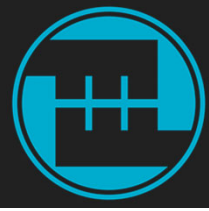
- If the submitter wants to make a full entropy claim using the new SP 800-90C draft definition, then they will need to adjust the value used as osr once the tester determines the entropy estimate for raw data, H. In order to make a claim of full entropy from the output of the conditioner, set $osr \geq [1/H]$, $\mathrm{ENTROPY\_SAFETY\_FACTOR} \geq 64$, and operate in FIPS mode.
    - Most users of this library presume that the output is (very nearly) full entropy.



Credential Me!

Get an algorithm validation on the included SHA3-256 implementation.

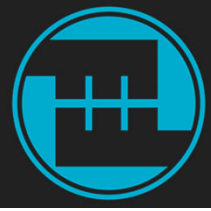
- The conditioner is a chained invocation of SHA3-256 (where each call of jent_measure_jitter contributes one additional round of chaining).
- If the function is viewed as non-vetted, then the truncation performed in jent_read_entropy is a compliance issue.
- In jent_hash_time, each round takes the output from the prior jent_measure_jitter call (supplemental data), the current input delta value (the raw data), and a loop count (supplemental data).



NO TICKET!

A summary of the SP 800-90B conditioning parameters per conditioned output (i.e., jent_random_data call) are summarized in the following table:

Parameter	**Value**
n_{in}	$((256 + \text{ENTROPY\_SAFETY\_FACTOR}) \times \text{osr} + 1) \times 64$
n_{out}	256
nw	256
h_{in}	$((256 + \text{ENTROPY\_SAFETY\_FACTOR}) \times \text{osr} + 1) \times H$

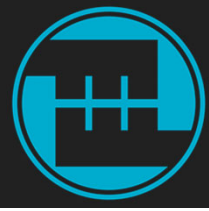


“It means just what I choose it to mean – neither more nor less”

SP 800-90B Section 4.3 Requirement #8 states:

“The submitter shall provide documentation of any known or suspected noise source failure modes (e.g., the noise source starts producing periodic outputs like 101…01), and shall include developer-defined continuous tests to detect those failures. These should include potential failure modes that might be caused by an attack on the device.”

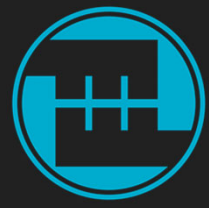
- In [Müller 2021, Section 7.2.41], the developer claims that there are no known or suspected noise source failure modes.



Not a Witch

This may be misleading on some architectures, as the jent_entropy_init function includes start-up health tests for a number of conditions:

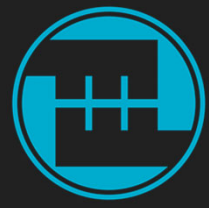
- The timer value is ever 0 (resulting in a ENOTIME return code).
- The delta value is ever 0 (resulting in a ECOARSETIME return code).
- The delta value is stuck (that is, the timer value, delta or second delta are fixed between two consecutive delta values) more than 90% of the time (resulting in a ESTUCK return code).
- The common divisor of all observed delta values is greater than or equal to 100 (resulting in a ECOARSETIME return code).
- Consecutive delta values are, on average, less than 1 apart from each other (resulting in a EMINVARVAR return code).



Not a Which?

- Repeated deterministic patterns of various lengths can emerge in some circumstances. The apparent variation in such deterministic patterns does not contribute entropy, so large numbers of such strings should be detected and an error should be raised. The lag predictor health test is intended to detect this failure mode.

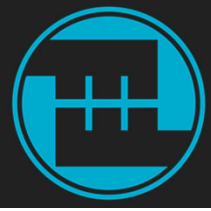
On some architectures these failure modes may be profoundly unlikely to occur, and thus one could say that on such architectures these are not anticipated failure modes for that fixed hardware, though these tests continue to exist in any case.



Which Witch?

For non-passing start-up health tests and restart tests :

- Increase the values of JENT_MEMORY_BLOCKS and JENT_MEMORY_BLOCKSIZE. This increases variation due to various memory cache microarchitectural characteristics.
  - On modern Intel systems, a value of 8192 (JENT_MEMORY_BLOCKS) and 128 (JENT_MEMORY_BLOCKSIZE) seems to work well.
  - On RISCV systems, JENT_MEMORY_BLOCKS setting of 2048 has tested well.
  - There is a script in the distribution that searches for effective values for these parameters; one can use this to establish appropriate values, or just do analogous testing.
- Increasing JENT_LAG_HISTORY_SIZE to 128 or more may also decreases the efficacy of caching, which tends to increase timing variation.

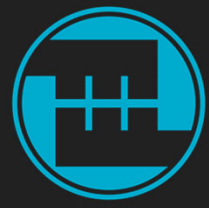


How Much Witch Could a Which Chuck Chuck…

For non-passing start-up health tests and restart tests (cont):

- Increase the return value of jent_loop_shuffle to its maximum value (namely (1U<<min)+(1U<<bits)-1). This might help in the situation where variations induced by external events are a significant contributor to the uncertainty; this change increases the amount of time where such events could occur.

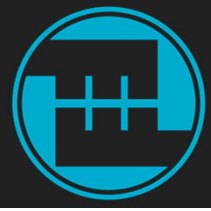
If none of these changes resolve the start-up health test and restart test failures, then this library should not be used with the tested hardware.



Restarts, and Where To Find Them

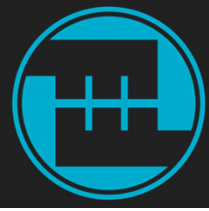
The notion of Restart varies:

- A software module may reasonably use the approach employed by Müller's test script.
- A hardware module is likely to require a full restart of the hardware module between each sampling.



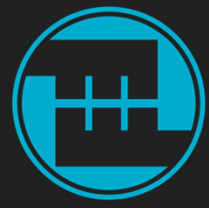
Heuristic Estimates of Entropy

- When looking at a histogram of delta values (once the pseudorandom variation has been disabled), the various sub-distributions (“modes” or “humps”) in the histogram correspond with different repeated sets of events that contribute delay between the two measurements, and the width of each sub-distribution is a consequence of the amount of variation for that fixed event set (for example, differing microarchitectural-dependent variations that occur for the same set of events).
- The tester should characterize all apparent sub-distributions in the raw results, and limit the $H_{submitter}$ claim to the most predictable (i.e., lowest assessed entropy) of the observed sub-distributions.
    - This testing makes sense under the assumption that an attacker can drive the entropy source toward certain sub-distributions.



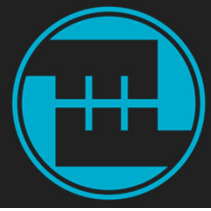
Heuristic Estimates of Entropy

- Histograms only tell part of the story.
  - The source is not expected to be IID!
- On some platforms, the noise source can lapse into largely deterministic repeating patterns.
  - Such defects are revealed by the MultiMMC Predictor, LZ78Y Predictor, t-tuple, and Lag Predictor estimators.
  - These failures are intended to be eventually detected by the Lag Predictor health test.



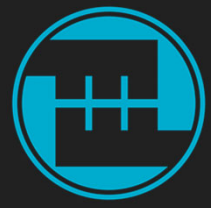
Invariant Variants

- JEnt relies on hardware-specific timing. Small departures from the described hardware may result in radically different characteristics and thus a significantly different produced entropy.
- An assessment should apply to only the specific hardware that was tested.
  - This hardware should be completely specified and testing should be re-conducted if any of the entropy-relevant hardware is changed.



Heuristic Estimates of Entropy

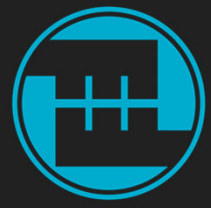
- If the dataset is large enough to support statistical assessment of each identified sub-distribution, this can provide some insight.
- It is better to try to naturalistically induce these distinct behaviors as much as possible.



Entropy-Relevant Parameters

Hardware:

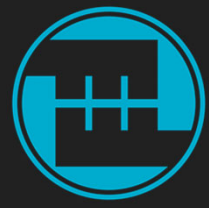
1. CPU:
    a) Model and clock rate
    b) Stepping/version
2. RAM configuration:
    a) Memory type
    b) Memory size
    c) Number of memory parts
    d) Memory buffering
    e) Memory timings
    f) Number of memory channels in use



Entropy-Relevant Parameters

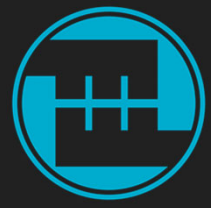
Changing certain software configuration parameters may have some impact on the SP 800-90B compliance and/or the rate of produced entropy.

- JENT_CONF_ENABLE_INTERNAL_TIMER
- JENT_CONF_DISABLE_LOOP_SHUFFLE
- JENT_MEMORY_BLOCKS
- JENT_MEMORY_BLOCKSIZE
- JENT_POWERUP_TESTLOOPCOUNT
- apt_cutoff
- JENT_APT_WINDOW_SIZE
- JENT_LAG_WINDOW_SIZE
- JENT_LAG_HISTORY_SIZE
- JENT_MIN_OSR

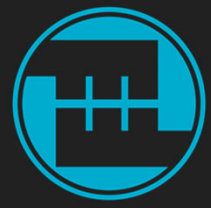


Entropy-Relevant Parameters

- osr
- fips_enabled
- mem
- enable_notime
- MAX_HASH_LOOP
- MIN_HASH_LOOP
- JENT_MEMORY_ACCESSLOOPS
- MAX_ACC_LOOP_BIT
- MIN_ACC_LOOP_BIT
- ENTROPY_SAFETY_FACTOR
- Optimizer Setting

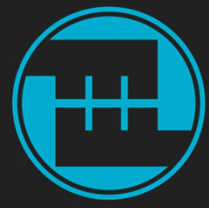


Older JEnt Version Notes



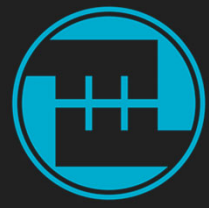
History is Complicated

- Pseudorandom Variation in raw data: v2.2.0-v3.0.1
  - This reduces the meaningfulness of health tests and statistical assessment of this data.
- Raw data extracted from a noise source analog: v2.2.0-v3.0.1
- User can’t force a specific counter source: v3.0.0-v3.0.1
- Idiomatic timer format: v2.2.0-v3.0.1
- Start up tests performed using a flawed noise source analog: v2.2.0-v3.0.2

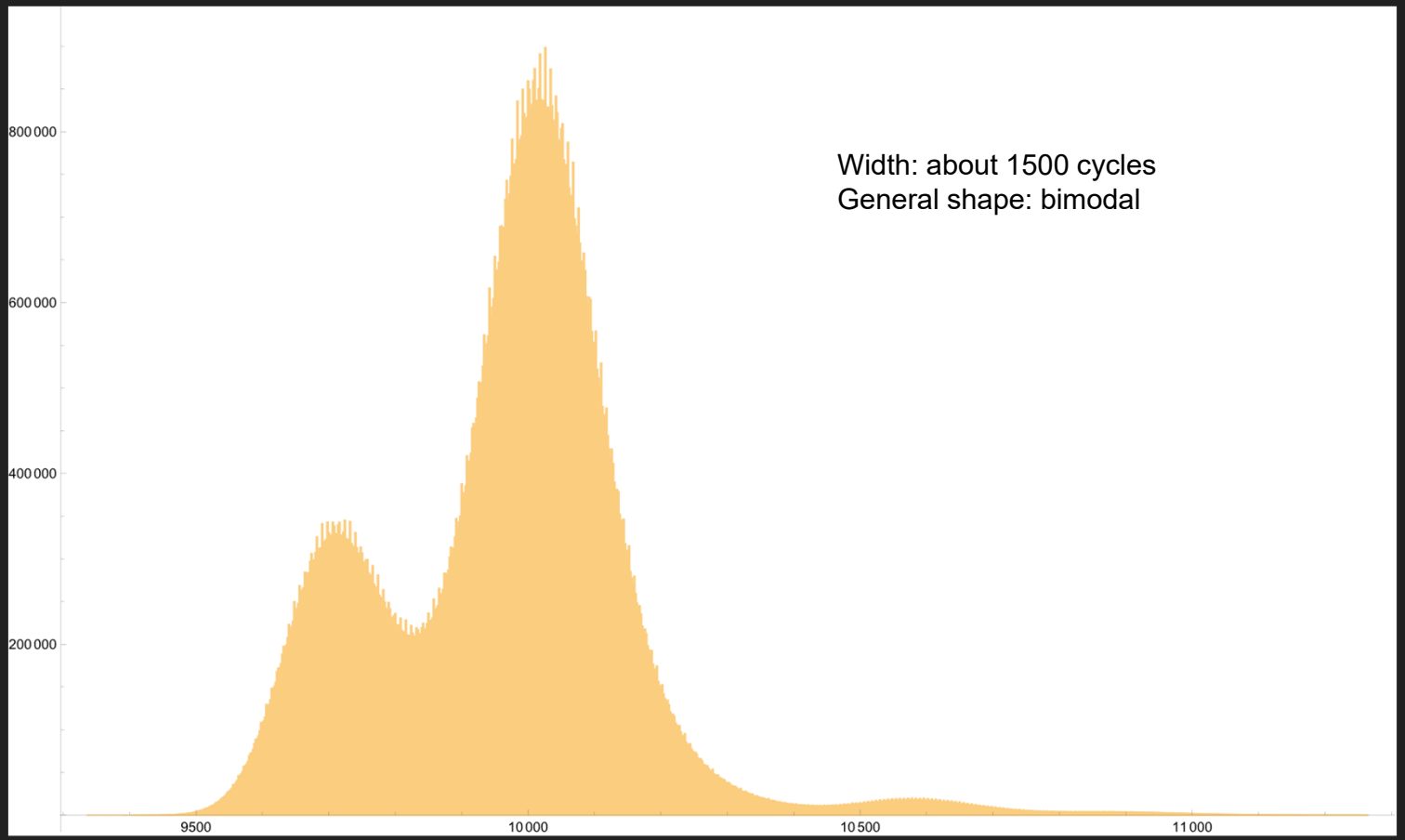


History is Complicated

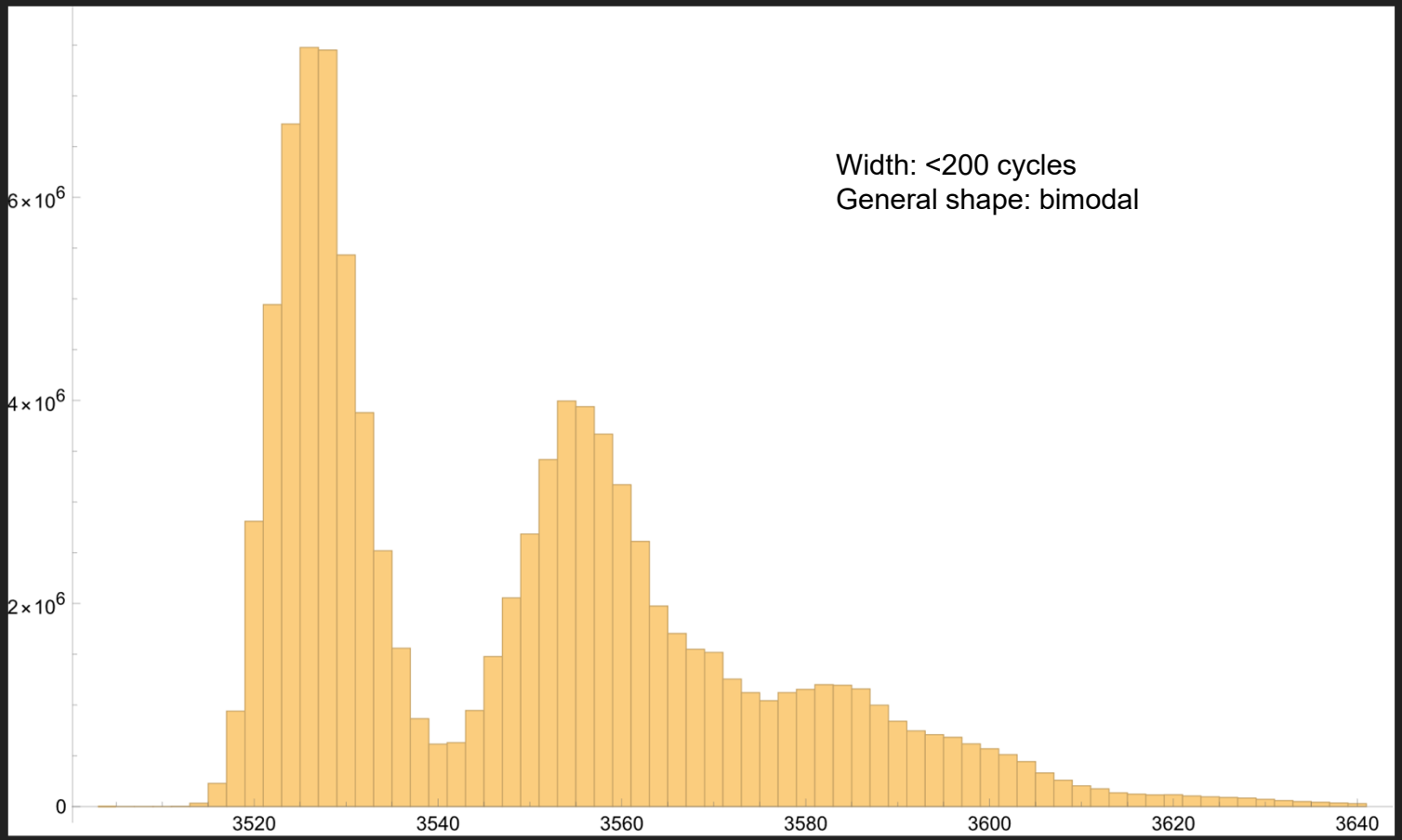
- Optimizer use may result in (catastrophically) reduced entropy: v3.0.0-v3.0.1
- No ability to force FIPS mode: v2.2.0-v3.0.1
- APT health test is non-approved: v2.2.0-v3.0.2
- APT cutoff does not vary with osr: v2.2.0-v3.0.2
- No detection of deterministic raw data: v2.2.0-v3.0.2
- Using the TSC requires manual changes: v2.2.0
- Non-vetted conditioning: v2.2.0



Example Delta Distribution: No Optimizer

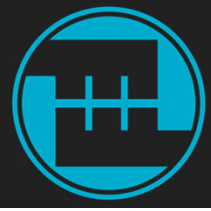


Example Delta Distribution: -O2

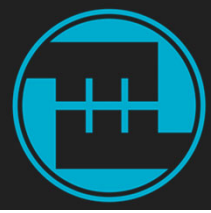


Optimizers

- These histograms tell only part of the story.
- On many platforms, optimized versions of JEnt had severe problems with varying-but-deterministic output.

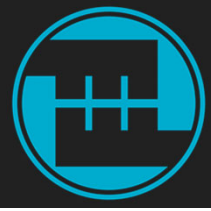


Detailed `jent_memaccess` Analysis



Why Memory Access?

- DJ mentioned that memory timings were a likely source of uncertainty in this system.
- Let's examine this source in some detail.
- Diagrams that follow are `jent_memaccess` oriented (we ignore the hash time)

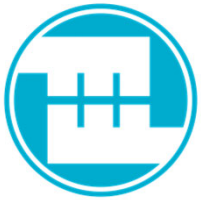


Memory Access Variation

This updates a location (hopefully) in memory.

This establishes the next location to update.

```
static void jent_memaccess(struct rand_data *ec, uint64_t loop_cnt)
{
…
        for (i = 0; i < (ec->memaccessloops + acc_loop_cnt); i++) {
                unsigned char *tmpval = ec->mem + ec->memlocation;
                /*
                 * memory access: just add 1 to one byte,
                 * wrap at 255 -- memory access implies read
                 * from and write to memory location
                 */
                *tmpval = (unsigned char)((*tmpval + 1) & 0xff);
                /*
                 * Addition of memblocksize - 1 to pointer
                 * with wrap around logic to ensure that every
                 * memory location is hit evenly
                 */
                ec->memlocation = ec->memlocation + ec->memblocksize - 1;
                ec->memlocation = ec->memlocation % wrap;
        }

        }
}
```

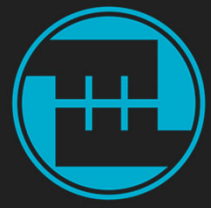


Hardware is Complicated

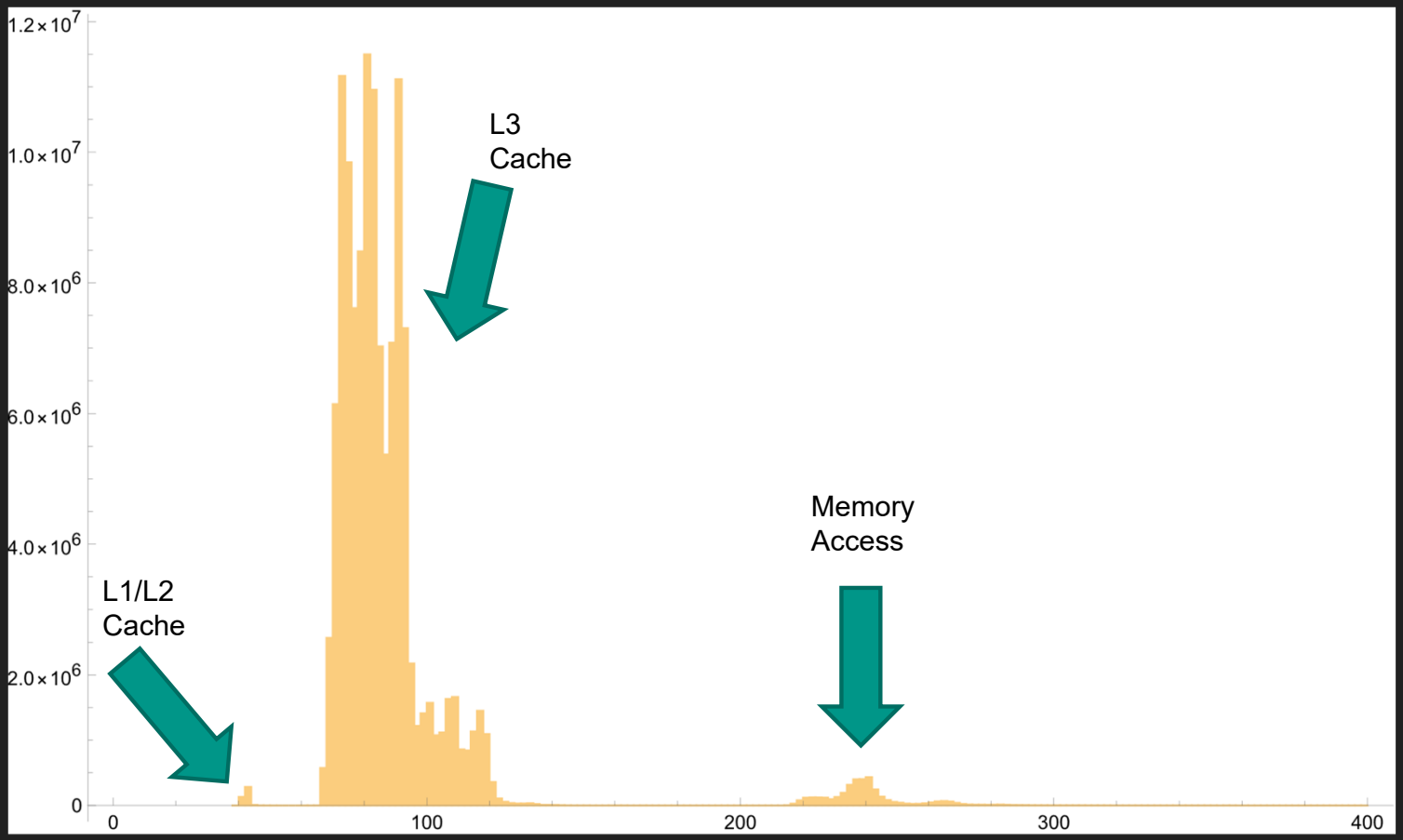
There are a few possible ways memory updates can resolve.

For this hardware (Intel Cascade Lake):

- L1 (Data cache): 4 cycles (32 KB)
- L2 Cache: 12 cycles (1 MB)
- L3 Cache: 42 cycles (36MB)
- Memory access: About 200 cycles

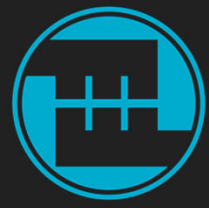


Default Memaccess Update Timing (16MB)



When Memory Access?

- L1-L3 cache is synchronous (and has deterministic timings)
- Updating memory in a very structured way will tend to produce patterns in memory access timing, resulting in correlation.
- If the memory blocks are too small, there could be multiple updates per cache line (64B here), which likely results in a cache hit.
- In this use, we are competing “against” the cache system, because when it successfully caches the updated value we get very regular timing.
  - We could develop a cache-avoidance system for each architecture, but such logic would be very hardware dependent.
- We also want some way to regularize the results, to make each update independent(ish) of the prior timings.

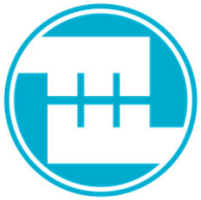


Putting the “Random Access” in RAM

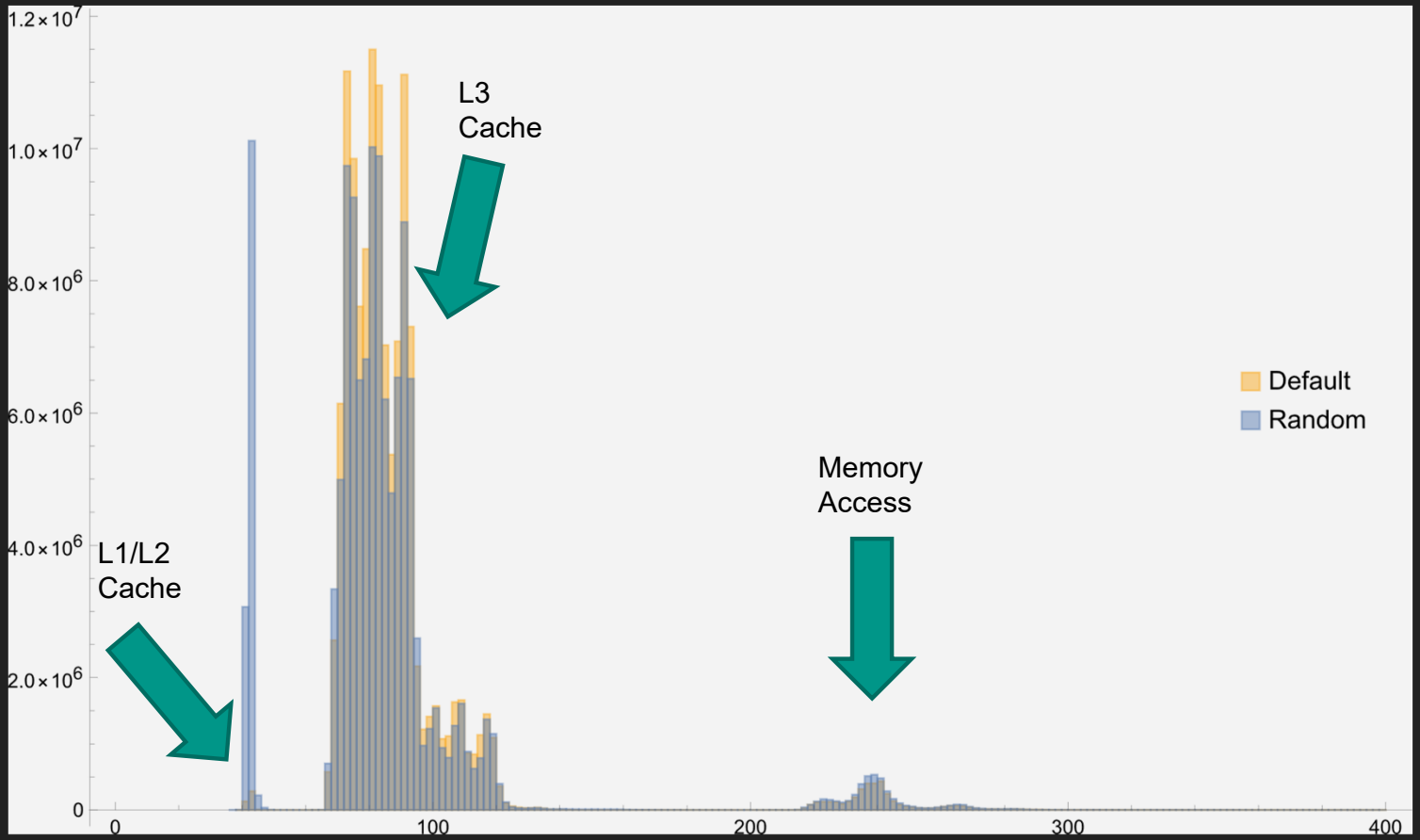
Seed a PRNG

Establish the next memory location pseudorandomly.

```
static void jent_memaccess(struct rand_data *ec, uint64_t loop_cnt)
{
...
        uint32_t prngState[4] = {0x8e93eec0, 0xce65608a, 0xa8d46b46, 0xe83cef69};
        const uint32_t addressMask = (uint32_t) ((UINT64_C(1)<<JENT_MEMORY_BITS)-1);

        /*Mix in the current data*/
        for(i=0; i<sizeof(prngState); i++) {
                uint8_t *curState = (uint8_t *)prngState;
                curState[i] ^= ec->data[i];
        }

...
        for (i = 0; i < (ec->memaccessloops + acc_loop_cnt); i++) {
                unsigned char *tmpval = ec->mem + ec->memlocation;
                /*
                 * memory access: just add 1 to one byte,
                 * wrap at 255 -- memory access implies read
                 * from and write to memory location
                 */
                *tmpval = (unsigned char)((*tmpval + 1) & 0xff);
                /*
                 * Get a new random offset.
                 */
                ec->memlocation = xoshiro128starstar(prngState) & addressMask;
        }
}
```

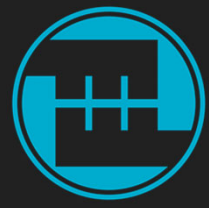



Memaccess Update Timing (16MB)

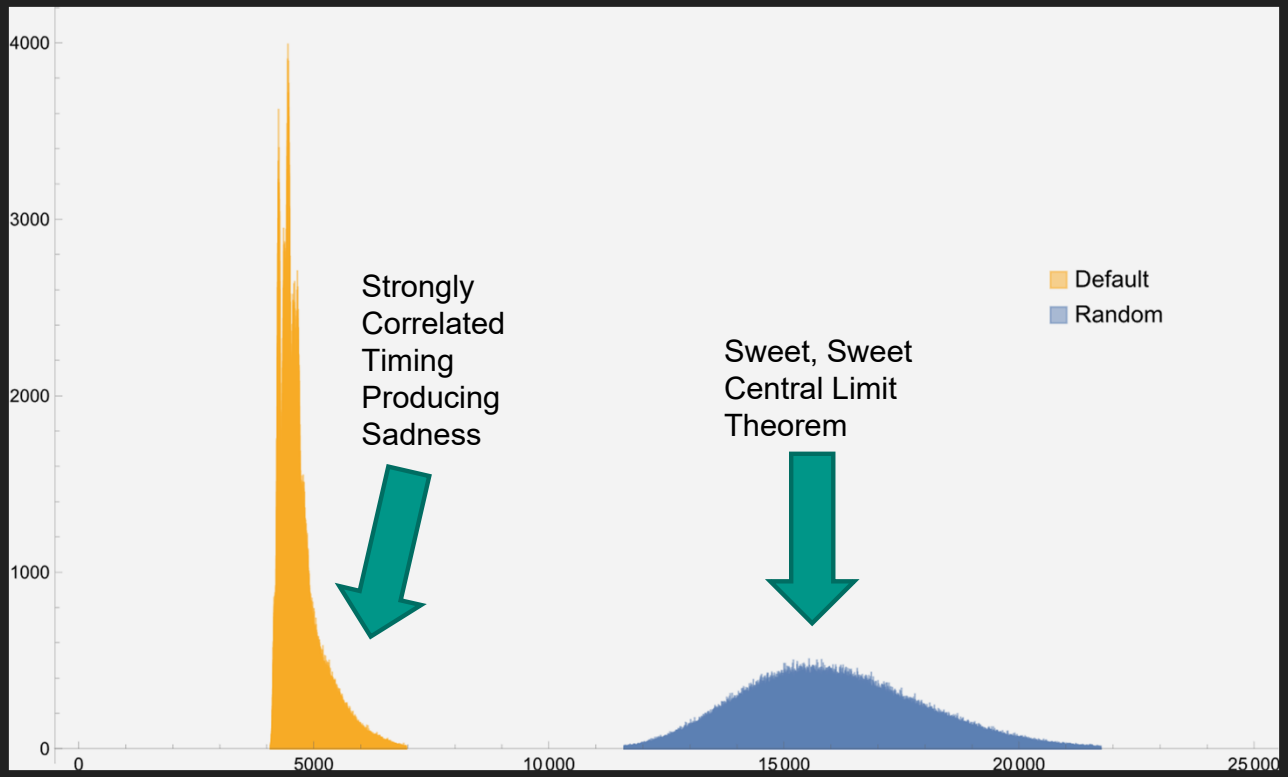


Worse? OR BETTER?!?

- The L1/L2 spike clearly indicates that some aspects of “unpredictable” also mean “undesirable”.
- This seems to make RAM access marginally more likely, but also makes the L1/L2 behavior much more likely.
- Any time you see a PRNG in a noise source, you should be concerned.
  - The PRNG doesn’t directly produce the raw noise, it just adjusts the location being updated. The timing of the update is part of the raw sample.
- The main thing this process gets you isn’t better “per-update” timing, it gets you mostly independent “per-update” timing, so we can now benefit from the Central Limit Theorem!

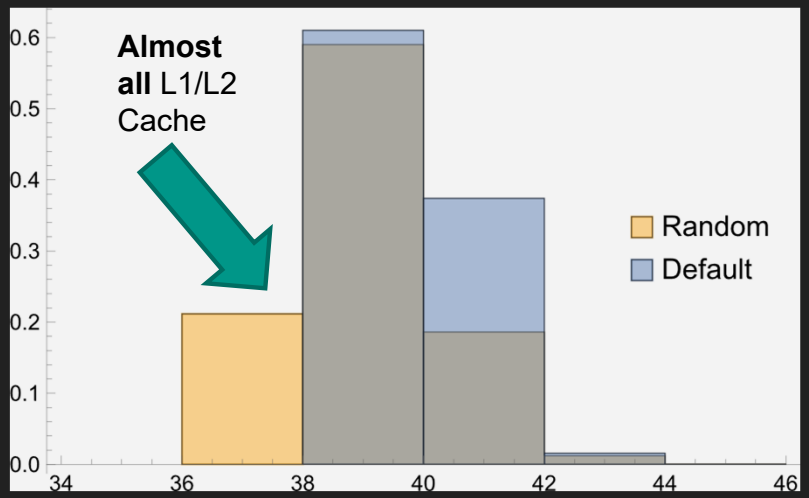


Full `jent_memaccess` Timing (16MB)

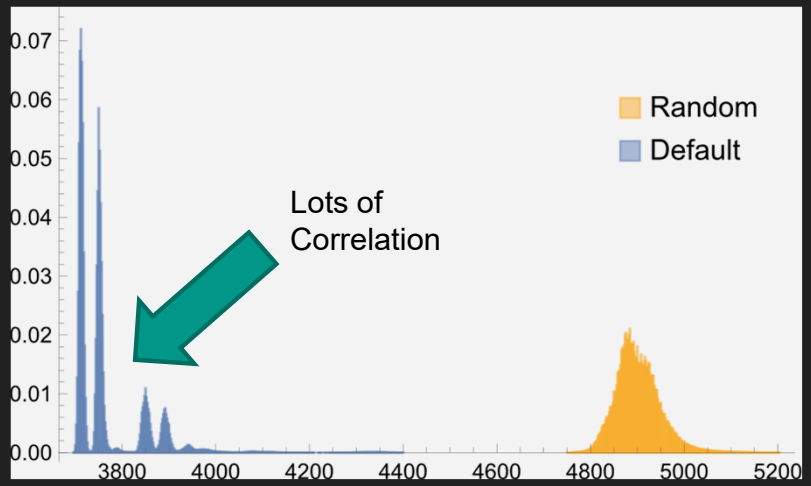


When Memory Access, Again?

- The size of the memory used here clearly should have an impact.
- If you choose too small a memory block, it may rarely actually read and write to RAM.
- The meaning of “too small” is very hardware dependent.



64 kB Block

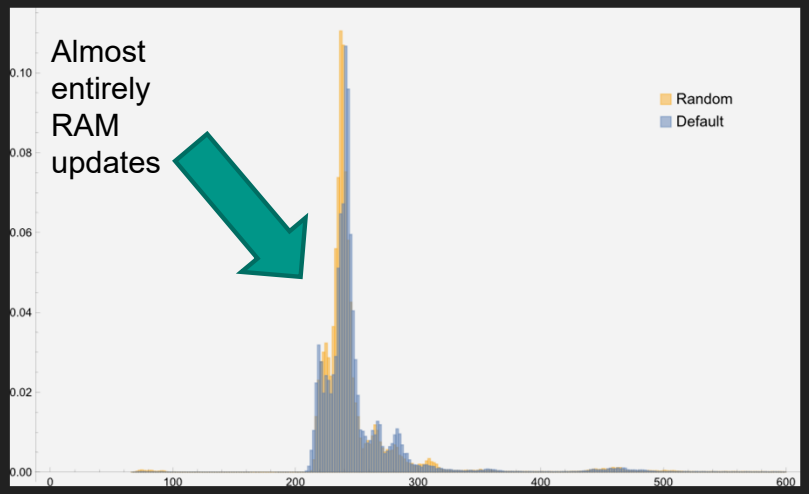


Memaccess Update Timing

Full `jent_memaccess` Timing

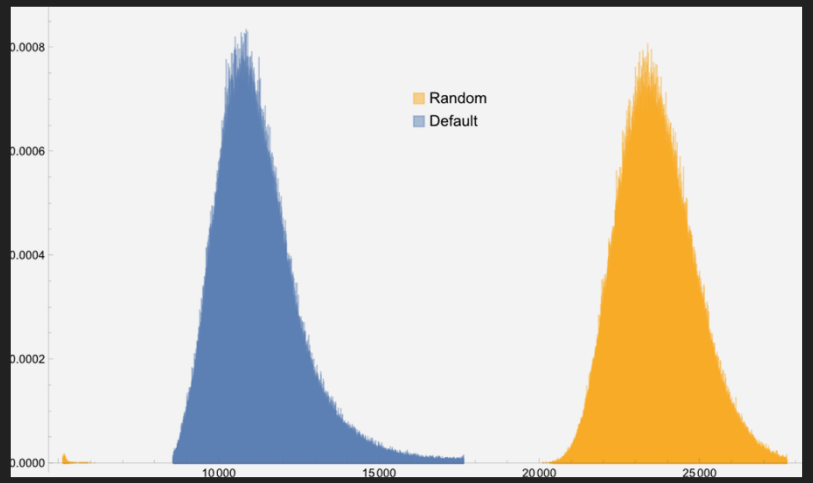
When Memory Access, Again?

- Things get better as the memory block gets larger (to a point)



Memaccess Update Timing

256 MB Block

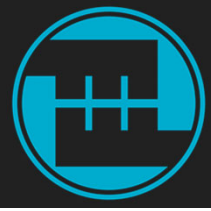


Full `jent_memaccess` Timing

Would You Say That You Had a Point?

We've done a lot of work here, and (aside from the sweet sweet Central Limit Theorem use) there hasn't been much of a payoff. Let's review:

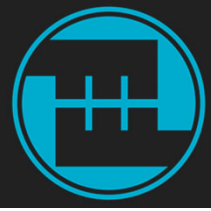
- We forced each update to follow a fixed distribution by increasing the size of the memory block.
- We forced the update timings to be independent of each other using (statistically) random selection of the update address.
- We verified that the summing 129 of these timings produces a normal-looking result (expected by the Central Limit Theorem).
- We ran the SP 800-90B IID tests on the `jent_memaccess` timing data (256MB to 2GB memory block size), and it passed.
- Claim: We have argued that (with parameters that result in profligate RAM use), this part of this noise source is IID.



Numbers, etc.

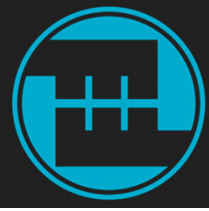
If we can credit all variation in RAM timing as uncertain, then…

Block Size	Selection Bitmask	Assessed Entropy Non-IID/IID
256 MB	0x000003F6	7.3/7.8
512 MB	0x000001EF	6.6/7.8
1 GB	0x000002F9	6.4/6.9
2 GB	0x000000FF	7.3/7.8

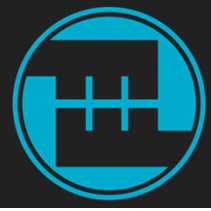


Afterward

- Updating a statistically random address has been made the default behavior as of JEnt v3.2.0 (this feature is controlled through the `JENT_RANDOM_MEMACCESS` macro).
- In this version on Linux environments, the size of the memory region is automatically set in Linux to be the smallest power of 2 greater than the size of the memory cache.



Thank You!



References

- [Müller 2021] Stephan Müller. *CPU Time Based Non-Physical True Random Number Generator*. July 25, 2021.
- [GitHub] https://github.com/smuellerDD/jitterentropy-library
- [xoroshiro] *xoshiro / xoroshiro generators and the PRNG shootout.* https://prng.di.unimi.it/