What to Expect When You're Expecting

(Part II: … to Assess the CPU Jitter Random Number Generator v2.2.0 Against SP 800-90B)

Joshua E. Hill, PhD

KeyPair CONSULTING

CMUF Entropy WG
20230711-2

(Presentation Version *20230711*)

The CPU Jitter Random Number Generator (JEnt)

- In Part 1 of this presentation, we discussed the general JEnt design, focusing on more modern JEnt versions.
- I had imagined that most JEnt use would be with these more modern versions, because they behave better, have more meaningful health testing, and are more clearly compliant with SP 800-90B.
- It has become clear that there is still some residual demand for v2.2.0 of this library, in particular the version integrated into the Linux Kernel.
- With that in mind, here is a more detailed review of the design of this earlier version.

JEnt v2.2.0 Notes Summary

- Getting access to “raw” data is hard.
  - Models for the entropy source need to be supported.
    - Start-up health tests extract data from a model of the noise source.
    - Relevancy of this model must be verified.
  - Use of debug interfaces run the risk of altering the timing.
    - Some investigation should be included in the validation.
- “Shuffling” obscures:
  - The entropy rate.
  - Failure from health tests.
  - (It ought[*] to be disabled.)

[*] This feature is not compatible with the implemented health tests WRT the health testing requirements of SP 800-90B, but some of these requirements are presently optional within the ESV program.

JEnt v2.2.0 Notes Summary

- (LFSR-based) Non-vetted conditioning cannot make a full-entropy claim.
- Health testing doesn’t use the approved tests.
    - The RCT test is clearly a refinement of the approved RCT.
    - The APT test has a few bugs.
- Health testing misses some common failure modes.
    - The lag health test was added for a reason…
- Versioning is ambiguous.

Raw Data

Raw Data

- Raw data produced in `jent_measure_jitter()`.
  - Not programmatically externally available.
  - Raw data is not returned by this interface.
- Raw data conditioned in `jent_lfsr_time()`.

The Central Logic

Do the memory update(s).

Calculate raw data.

Health Testing

Conditioning

```
static int jent_measure_jitter(struct rand_data *ec)
{
        …
        jent_memaccess(ec, 0);

        jent_get_nstime(&time);
        current_delta = jent_delta(ec->prev_time, time);
        ec->prev_time = time;

        stuck = jent_stuck(ec, current_delta);

        jent_lfsr_time(ec, current_delta, 0, stuck);

        return stuck;
}
```

Getting “Raw Data” Externally

- Data for testing must come from somewhere…
  - Extracting from a system debug interface.
    - E.g. SystemTap.
    - Allowed, as Shall ID #56 is optional.
  - Using the publicly available test tools (e.g. `jitterentropy-foldtime`).
    - This makes a model for the noise source and extracts the data from the model.
    - Prohibited by Shall ID #58.
  - Making a new interface in a debug build.
    - Prohibited by Shall ID #58.

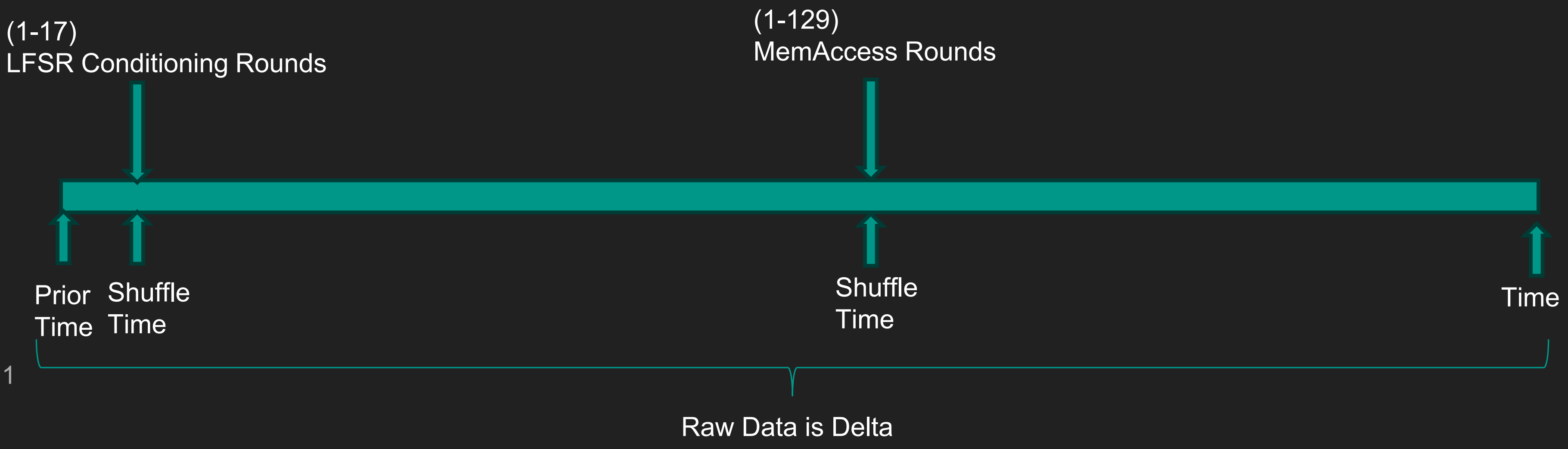
Getting “Raw Data” Internally

- Start-up health tests don’t have access to any special interface.
- The start-up code implements a model for the noise source.
- The submitter would need to argue for the meaningfulness of such start-up health tests.

The Hazards of “Shuffling”

When Is Time Used?

- The credited variation is the amount of time to
    - Integrate (the prior) data into the pool using (some number of) LFSR round(s).
    - (wave hands) Perform other system activity.
    - Perform (some number of) `jent_memaccess` rounds.
- Intermediate values are used…



Loop Shuffle Code

Use the current time (already credited!) to perturb the calculation.

Integrate the previously output value.

XOR blocks together

```
… jent_loop_shuffle(struct rand_data *ec,
                    unsigned int bits, unsigned int min){
        …
        jent_get_nstime(&time);
        …
        time ^= ec->data;
        …
        for (i = 0; ((DATA_SIZE_BITS + bits - 1) / bits) > i; i++) {
                shuffle ^= time & mask;
                time = time >> bits;
        }
        …
return (shuffle + (1<<min));
}
```

Impact of the “Shuffle”

- The timestamps are related (but different than) the raw data delta.
  - If credited, this would necessarily reduce the raw data entropy.
- This transitory data is only locally available.
- Use of previously output data can’t contribute entropy but does make the result pseudorandom.
- The default parameters result in 4 bits of variation for rounds of conditioning and 7 bits of variation for rounds of `jent_memaccess`.
  - Depending on the relationship between the expected timing value of the conditioning and `jent_memaccess`, this contributes between 7 and 11 bits of pseudorandomness to the output
  - We have produced a JEnt variant that allows for independent testing of this pseudorandom variation.
    - Results from this variant are not interesting; the data looks pseudorandom.

Impact of the "Shuffle"

- Any empirical (i.e., data-based) heuristic entropy estimation strategy used with this design has to account for this pseudorandom variation.
- Health testing needs to be "shuffle aware" in order to be meaningful, but this isn't the case.
    - If the runtime became fixed (and thus the system produced zero entropy), the output could continue to look random due to the (at least) 7 bits of pseudorandomness present in the timing (sampled via the timer).
    - This shuffling behavior also masks cases where the architecture prevents entropy production.
    - Using the approved health tests, this feature is not compatible with the health testing requirements of SP 800-90B, but some of these requirements are presently optional within the ESV program.

How to Avoid the “Shuffle”

- There are two compile time macros that control this functionality. To disable this functionality:
  - MAX_FOLD_LOOP_BIT and MAX_ACC_LOOP_BIT must be set to 0 *during compilation.*

Non-Vetted Conditioning

Non-Vetted Conditioning Consequences

- In the abstract, one can use (some types of) non-vetted conditioning to produce outputs that have substantial min entropy.
- SP 800-90B disallows a “full entropy” claim when using only non-vetted conditioning components.
  - When using non-vetted conditioning, Section 3.1.5.2 necessarily prevents any claim higher than a 99.9% entropy rate.
  - A practical limit is actually closer to about 95% (due to the $h' \times n_{\text{out}}$ term).
  - It was the stated intent of the SP 800-90B authors to only allow a “full entropy” claim when vetted conditioning is used.
- IG D.K Resolution 19 is (as of March 2023) the only way to make a “full entropy” claim. This resolution allows a “full entropy” claim when:
  - Using a vetted conditioning component, AND
  - $h_{\text{in}} \geq n_{\text{out}} + 64$, AND
  - The security strength of the conditioning component is greater than or equal to n_{out}.

And Yet…

- JEnt v2.2.0 uses (LFSR-based) non-vetted conditioning.
- Various JEnt v2.2.0 ESV certs (#E8, #E37, #E47, #E48, #E50, and #E54) make full entropy claims, without listing any vetted conditioning.

Health Testing Oddities

RCT Variations

- The implemented RCT isn't the approved RCT
- A "stuck value" occurs if the raw value (1st delta), 2nd delta, or 3rd delta repeat.
- The failure mode is a strict superset of the RCT.
- If the RCT cutoff is selected using the SP 800-90B procedure, then the calculated cutoff isn't associated with the targeted false positive rate.
  - The argument for the targeted false positive rate for the approved RCT should be modified for this test (as this RCT test fails at roughly three times the rate of the approved RCT).
  - The observed false positive rate also has the standard SP 800-90B non-IID disconnect between the targeted false-positive rate and observed false-positive rate.

APT Variations

- The base v2.2.0 library’s implemented APT isn’t the approved APT.
    - Only the lowest nibble is used for the test.
- The Linux Kernel version tries to change the base JEnt v2.2.0 APT to use the full raw symbol.
    - The start-up health tests still use the lowest nibble.
- There are data conversion problems within the test (due to a stored uint32_t comparison with the current uint64_t value)
- The APT logic isn’t quite right.
    - The last symbol from the prior window is used as the new reference symbol.
    - The window size is thus 513 rather than 512.

Health Testing Impacts

- From a technical perspective, not much.
- From a programmatic perspective, this is a bit of a pain.

Anticipated Failure Modes

- Later versions of JEnt integrated an additional health test (the Lag Health Test) due to observed failure modes.
  - The Lag Health Test detects short cycles of highly regular outputs.
- This health test is not backported in the base JEnt v2.2.0 library or its public Linux integrations.

Versioning

Public-Facing Versioning and Code

- “v2.2.0” doesn’t uniquely describe a single version.
  - There is the public JEnt v2.2.0 library release.
  - There are a few closely-related variations of this library integrated into the Linux Kernel.
    - This integration fixed some problems and added some different problems.
    - These versions are also marked as “v2.2.0”, despite being clearly different than the base library release.
  - The label may have similarly been applied to other v2.2.0 variants (e.g., the Linux version of v2.2.0 + some additional changes).
- Using these (evidently) opensource versions has an impact.
- Using the same version on public certificates may mislead other vendors / labs regarding the current SP 800-90B / ESV requirement set.

References

- [Müller 2021] Stephan Müller. *CPU Time Based Non-Physical True Random Number Generator*. July 25, 2021.
- [GitHub] https://github.com/smuellerDD/jitterentropy-library