Entropy and NIST 800-90b

A personal journey

John Manferdelli
JohnManferdelli@hotmail.com

What are cryptographic random numbers?

- A cryptographic random number consisting of n bits has 2^n possible values.
- All these values should be “equally likely.”
- If you are told k of the bits, you should have no better chance of guessing the remaining bits than $\frac{1}{2^{n-k}}$.
- If you are handed a set of n bits cryptographic random numbers of size N knowing $N-1$ of them should give you no advantage in guessing the remaining one.

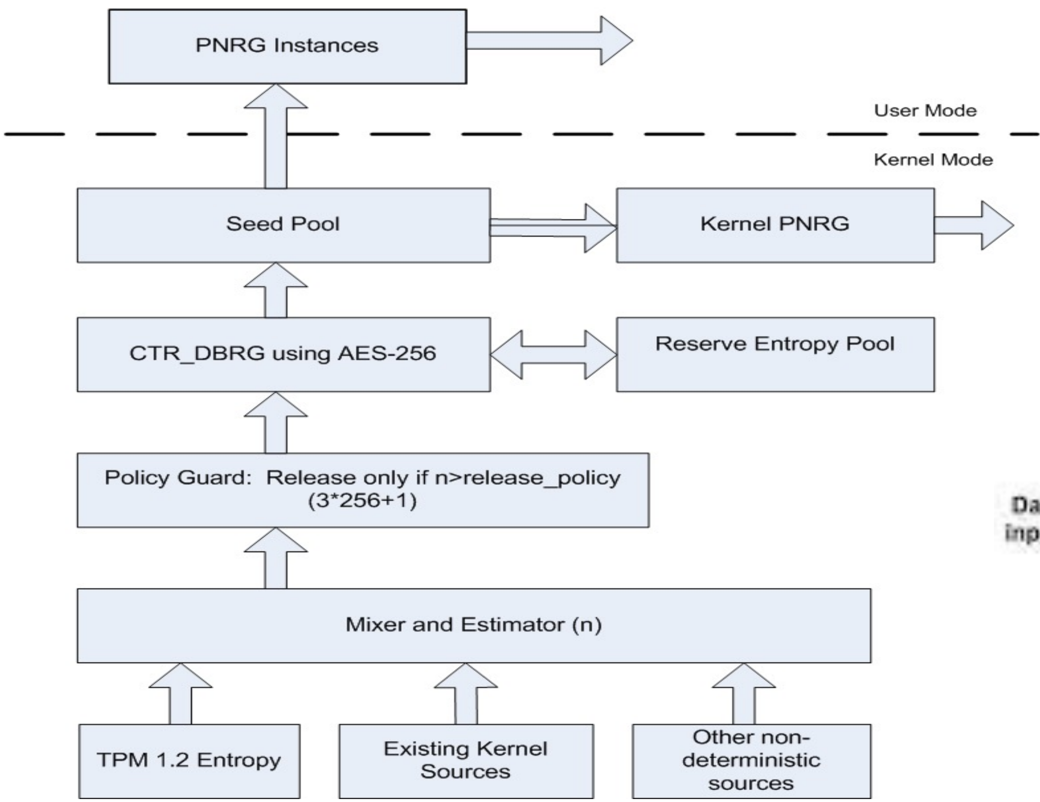
Cryptographic random numbers

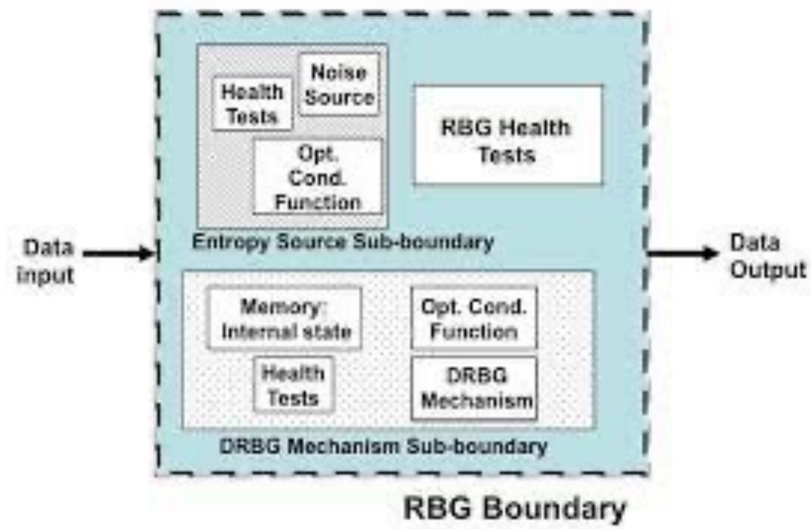
- Critical in cryptographic algorithms and protocols
  - Past weaknesses have been catastrophic.
  - Random number weakness and bad key management are greatest points of attack on cryptographic systems.
- Bad entropy is principal basis for practical attacks
  - Netscape browser attack is famous example.
  - More recent Debian entropy attack (Mind your p’s and q’s) is another.
  - Hall of fame, epic fails for bad entropy
    - “But the entropy looked random”
    - “No one could guess the sample values, it’s too complex”
- Other attacks
  - Intrusion (read privileged entropy pool)
  - Incremental guessing attacks

How can you produce cryptographic random numbers in the real world?

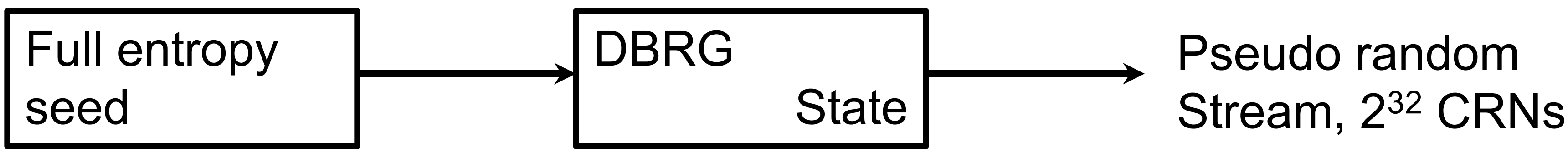
- NIST 800-90C specifies overall design of a cryptographic random number system.
- Components are:
  - Entropy Subsystem including characterized noise source, health tests, entropy conditioning. This is the critical component which prevents adversaries from guessing keys. The output of this system is a seed containing enough “entropy” (more later) to generate keys. The entropy subsystem is specified in NIST 800-90B. This is the hard part.
  - A deterministic random number generator (DRNG). This takes a seed and safely produces a long sequence of cryptographically secure random numbers. This is specified in NIST 800-90A. This is the easy part.

Sample 800-90 RNG System





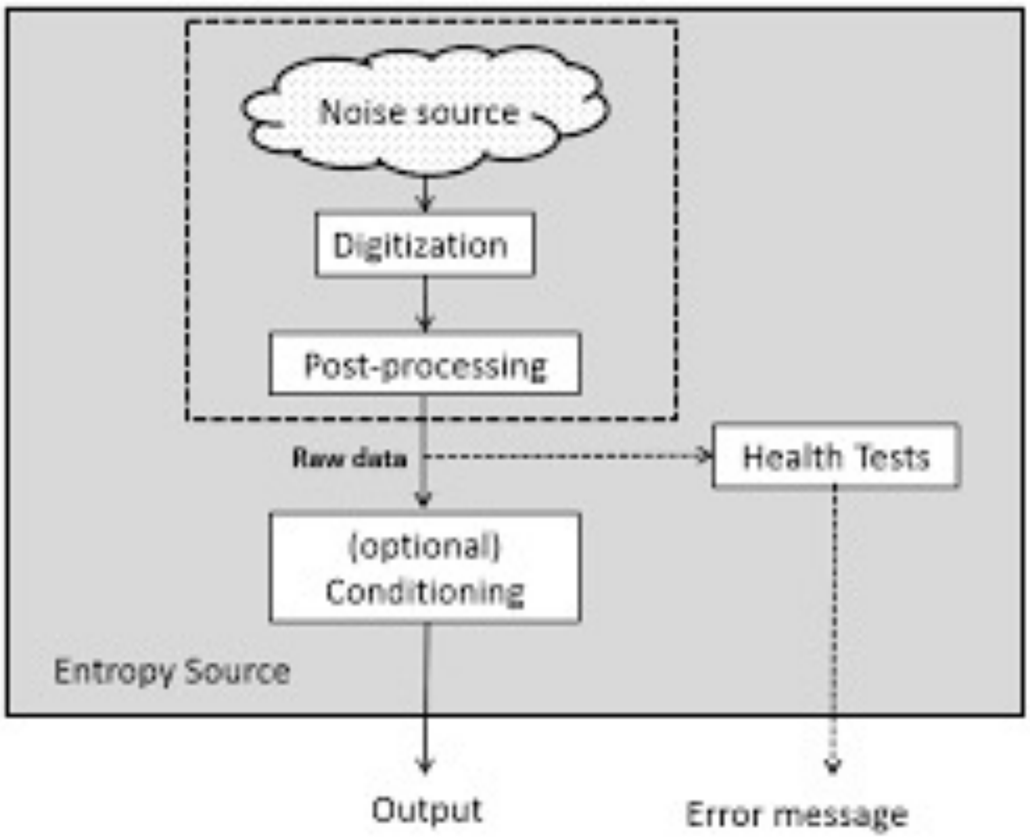
The easy part of cryptographic random number generation



- "Anyone discussing deterministic generation of random number is, strictly speaking, already in a state of sin" – von Neuman.
  - So, a "seed" with full entropy is critical
- Smooths and stretches entropy
- DBRG's can be built using
  - Block ciphers
  - Hash functions
  - Stream ciphers
  - Even public key systems
- Building good, certifiable DRBG's is a "solved problem"
  - Only a few "gotcha's" to be careful about

NIST 800-90B entropy subsystem

- This is the hard part we'll talk about.
  - No finite number of statistical tests can "prove" entropy.



What is entropy?

- Entropy is a measure of uncertainty or equivocation. It comes from thermal physics.
  - Entropy is related to how easy it is to "guess" the outcome of an experiment.
  - It is measured in bits (as we'll see). If you have n bits of entropy, you should be able to determine the outcome after 2^n "guesses."
  - In symmetric crypto, for example, if a key has n bits of entropy and you have a solid encryption algorithm, given ciphertext, an adversary should need to try 2^n keys to get the plaintext.
- Caution
  - Entropy is defined with respect to probability distributions. It cannot be calculated using statistical tests.
  - Example probability distribution: A fair coin toss has the distribution $P(X = heads) = \frac{1}{2}, P(X = tails) = \frac{1}{2}$.
  - *If you have data from an experiment whose trial outcomes are about half heads and half tails, it does not mean it has the foregoing distribution or the foregoing distribution's entropy.*
  - *Conditioned output can masquerade as entropy rich.*

Shannon's mathematical definition

- Suppose we have an experiment, with a finite set of outcomes, $\Omega = \{x_1, x_2, \ldots, x_n\}$ where the outcomes occur with probability $p_1, p_2, \ldots, p_n$ respectively. The probability distribution is $P(X = x_i) = p_i$. Note:
  - $\sum_{i=1}^{n} p_i = 1$
  - In general, $p_i \neq p_j$ for $i \neq j$
  - For a probability distribution to be useful, it should be *stationary*, that is, every time you perform an experiment, the probability distribution should be the same. This does *not* mean the outcome of two successive experiments should be the same!
  - These are *very* strong conditions.
- Shannon entropy: $H(X) = -\sum_{i=1}^{n} p_i \lg(p_i)$, $\lg(x) = log_2(x)$

Some entropy source calculations

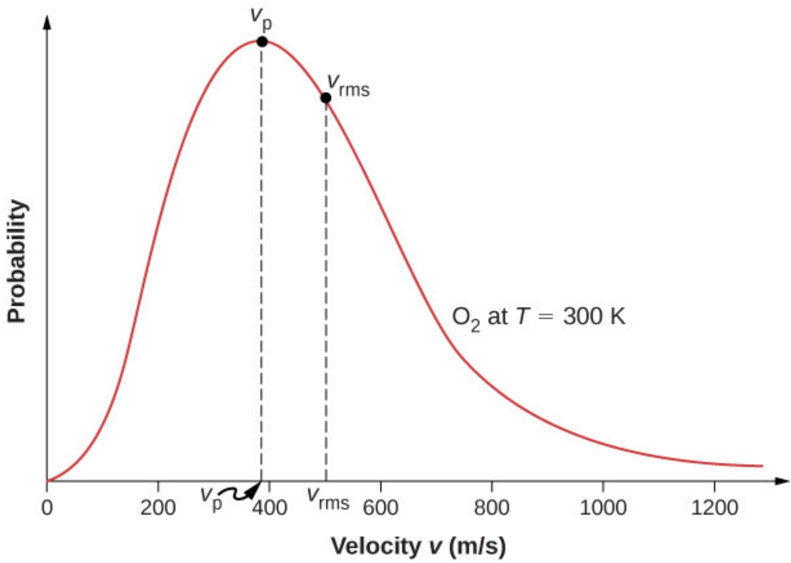
- Fair coin toss
  - $P(X = 1) = \frac{1}{2}\ (heads), P(X = 0) = \frac{1}{2}\ (tails)$
  - $H(X) = -\left(\frac{1}{2}\lg\left(\frac{1}{2}\right) + \frac{1}{2}\lg\left(\frac{1}{2}\right)\right) = 1.$
  - Note: A fair coin is unbiased: $p_{heads} = p_{tails}$
- Biased (but independent) coin tosses
  - $\Pr(x = 1) = \frac{1}{4}, \Pr(x = 0) = \frac{3}{4}$
  - $H(X) = -\left(\frac{1}{4}\lg\left(\frac{1}{4}\right) + \frac{3}{4}\lg\left(\frac{3}{4}\right)\right) =$ .85 $bits$
  - A “conditioner,” like a hash function, can take biased noise samples and “even them.”

Other measures of entropy

- Renyi entropy: $H_\alpha(X) = \frac{1}{1-\alpha} \sum_{i=1}^{n} {p_i}^{\alpha}$
  - Useful when calculating collision properties, usually $\alpha = 2$
- "Min" entropy: $H_{min}(X) = -\lg(max_{x_i}\, p_i)$
- $H_{shannon}(X) \geq H_{renyi}(X) \geq H_{min}(X)$, they are equal for a flat distribution.
- NIST focuses on H_{min}. Here's why:
  - Suppose we have the distribution, $P(X = x_1) = \frac{1}{2}$, $P\left(X = x_j\right) = \frac{1}{2(n-1)},\ j = 2,3,\dots,n$
  - An optimal adversarial strategy is to guess x_1 all the time, thus succeeding half the time.
  - $H_{shannon}(X) = -\left(\frac{1}{2}\lg\left(\frac{1}{2}\right) + \frac{n-1}{2(n-1)}\lg\left(\frac{1}{2(n-1)}\right)\right) \approx \frac{1}{2} + \frac{\lg(2n)}{2}$, for large n. This gives a distortedly pessimistic measure of an attacker's chance of succeeding.

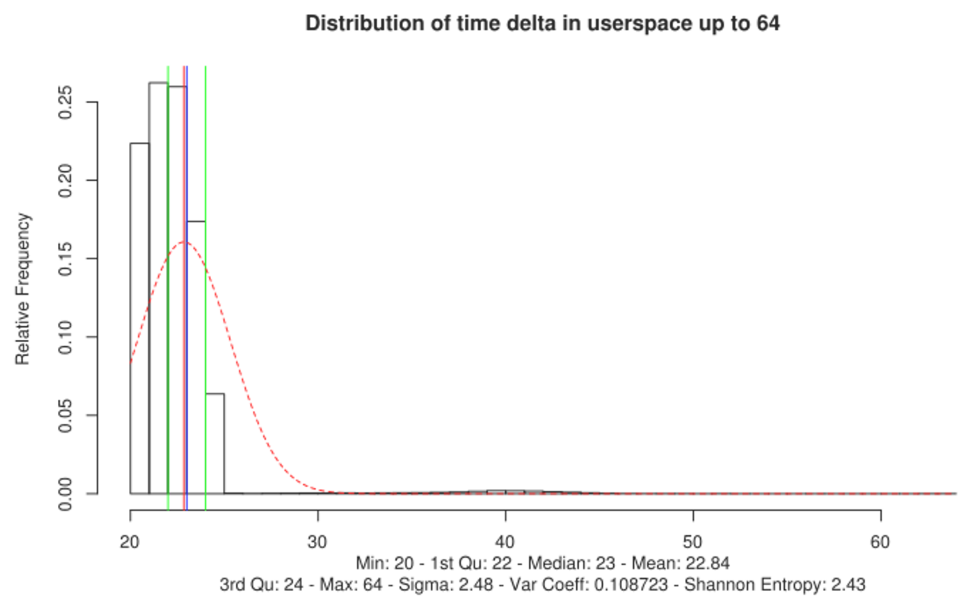
HW sources of entropy (God)

- Hardware
  - Thermodynamics (Johnson noise, …)
  - Oscillator jitter
  - Unsynchronized ring oscillators (Intel's HW RNG is based on this)
  - Noisy diodes
  - Radioactive decay
  - "Open pins" on Raspberry Pi's
  - Coin tosses (with a fair coin)
- Finding the probability distribution is easy: ask a physicist
- Easy to implement in microelectronics



SW sources of entropy (the devil)

- Software sources have been pseudo-science based
- Here is a list (Red is bad. Why? Don’t know distribution, also entropy starvation, non-stationarity. Vulnerable to side channels. Green is new and evidently does not have these drawbacks.)
  - Disk arm speed variation
  - Process id, thread id (predictable)
  - Interrupt arrival time
  - Ticks since boot
  - Cursor, mouse
  - New: execution jitter
- Finding the probability distribution is hard or impossible except for jitter then you can ask a cryptographer



Jitter entropy from Mueller

NIST 800-90B evolution

- 2008: Basic structure: We know it's hard. Document it.
- 2012: We're worried about entropy, here are a bunch of tests to run
  - Justification for software entropy is ad hoc or non-existent: "interrupt arrival times are impossible to guess." (wrong).
  - Use HW if you can: Intel's Ivy bridge RNG (launched 2012)
- 2016: People who don't have a good probability model for their noise sources, don't have entropy.
  - Let's use hardware as a model, hardware sources have distributions
  - Health tests are important because there can be failures
  - Should software entropy have more lax standards? [No!]
- 2018: No, seriously, you have to justify entropy estimators even for a software noise source, so you need source probability models.
  - Here are more tests (restart) so it's harder to cheat especially at boot
  - New software techniques arise (jitter)
  - Linux and some BSD entropy is justified
  - By the way, future standard will be stricter [2021]

A new hope

- Jitter execution entropy
  - High quality and relatively easy (i.e.- possible) to analyze.
  - Adopted by Linux, some BSD’s and Apple plus others.
  - Prediction: Eventually everyone will adopt it.

- History
  - B. Sunar, W. J. Martin, D. R. Stinson, *A Provably Secure True Random Number Generator with Built-in Tolerance to Active Attacks* IEEE. Mostly HW focused.
  - Stinson, part 2: What about software based on predicting execution time on modern processor?
  - Works on small processors too: Keaton Mowery, Michael Wei, David Kohlbrenner, Hovav Shacham, and Steven Swanson, *Welcome to the Entropics: Boot-Time Entropy in Embedded Devices.*
  - Mueller, *CPU Time Jitter Based Non-Physical True Random Number Generator.*
  - There’s lots more

How does Jitter execution work?

- Collect Entropy

    for (i= 0 to n-1)

        Get real time clock (t_{start})

        Execute standard code block

        Get real time clock (t_{end})

        $\Delta_i = t_{end} - t_{start}$
- The Δ_i's (usually one byte per sample as specified by NIST 800-90B) are the noise source for constructing a seed.
- Why is there uncertainty in the Δ_i?
    - Answer: Thank you ARM, Intel, RISC-V and IBM

Why is there uncertainty in the Δ_i?

- CPU instruction pipelines fill level affects execution time of an instruction. These pipelines add to execution jitter.
- The CPU clock cycle is different than the memory bus clock speed. Wait states for the synchronization of memory access adds to time variances (this also reflects hardware variability effects).
- The CPU frequency scaling alters the processing speed of instructions.
- The CPU power management may disable CPU features.
- Instruction and data caches
  - Tests showed that before the caches are filled, the time deltas are bigger by a factor of two to three.
- CPU topology and caches used jointly by multiple CPUs affect execution time.

But wait, there’s more

- CPU frequency scaling depending on the work-load.
- Branch prediction units
- TLB hits and misses
- Kernel locks
- Moving processes from one CPU to another
- Hardware interrupts can occur regardless what the operating system was doing in the meanwhile. [*This is not the same as interrupt arrival time.*]
- Large memory segments whose access times vary due to the physical distance from the CPU.

- Aren’t these variations predictable?
  - Amazingly, no
  - An adversary outside the kernel basically can’t affect them.

Why is Jitter execution entropy "good"

- Can be modelled
  - Jitter execution depends only on "core" hardware: CPU's, memory system, interconnect.
  - Component probability models are relatively simple (like HW): normal distributions around an average performance (memory, interconnect, speculation and prediction).
  - Some sources of variation derived from physical phase jitter (HW).
  - Easy to pick good, short blocks to measure on any CPU even as architectures change.
  - You can validate stationarity with chi squared tests.
    - Important for "boot entropy," where critical machine keys are derived.
  - Naturally "unobservable" by adversary. Protected from side channels.
- Gives very high entropy rates
- Doesn't need to be in "critical sections." Works well in kernel and user mode. Works well on all CPU's, under all workloads.

Why are other sources (say interrupt arrival time) “bad”

- Interrupt arrival modelling difficult (actually impossible, I think)
  - Presumptive distribution is Poisson arrival which is complicated and depends critically on stable average arrival time.
  - Definitely not stationary.
  - Depends on analysis of potentially huge number of devices.
  - Terrible during boot.
  - Past attacks exploited “observable” interrupt artifacts by adversary. Also subject to side channel attacks (see Dodis et. al., 2014).
  - Dependent on workloads and their imposed interrupt activity so accurate estimate would be painfully complex even with a few devices.
  - Gives low entropy rates (even “guessed” rates) so more intermediate processing. It takes much longer to “reseed.”
  - Requires ongoing care to measure interrupt times correctly (If you measure arrival time in the wrong place the entropy is greatly reduced).
- Kernel mode only
- I don’t know of any credible analysis of interrupt noise and I’ve looked hard.

Three weekends and a NIST reading

- I built a (basically) fully NIST compliant RNG in my existing open-source crypto project (which I use for teaching) with justified HW and SW entropy in about three weekends.
  - Used standard NIST 800-90A certified SHA-256 hash-df based DBRG.
  - Used Intel analyzed HW noise source (Noise justification: Rachael J Parker, *Justification for Metastability Based Nondeterministic Random Bit Generator*, Intel and previous papers by *Kocher, Cox, Walker, Gueron, Brickell, et al*).
  - Developed SW Jitter based noise source (Noise justification: You'll see.)
  - Implemented full health and restart tests.
  - Both HW and SW entropy qualified.
  - All in user mode (kernel version is almost identical).

RNG

```
jlm@New-MacBook-Pro cryptobin % ./test_full_rng.exe --print_all=true

Hardware test
HW noise : 2efccc36185532aecc5714c76a328d2e55bb5b6868cf6ee28ce7c0dd9178c43e
dffda69eb78b66ec
Entropy from pool: 300 bits.
seed: 2efccc36185532aecc5714c76a328d2e55bb5b6868cf6ee28ce7c0dd9178c43e
dffda69eb78b66ec

Hardware derived random numbers:
  random number 0: 38c4606144d00c417be407466237b3e08b8f08f81fd98b8a94dd54d74f914fed
  random number 1: 74283705ff3ad67c00d029ba8344cc9d014ef98a58edc9c93259cc8cf042cf37
  random number 2: 272f7b36a8694254368ca0f534ee45fd56756ef7e90bd49a3d35e87019f9ee68
  random number 3: d81302e1a5d30dc7735e62cc2c790ab3595c5cce34665c590556293f1f61e826
  random number 4: c804ada32bbe35d75a16cde90ec044d1f09350a82b8355ff50a7ef00f3a90ec3
  random number 5: c3b090ddf0c9d6dcd54a3d25ac9ba24813df5dd9119f683cff52c0a4487db23a
  random number 6: d25fa7625b34646d2d5402e3b8f65dff2fb806c4a9cfa978303b27a133fd9dbe
  random number 7: 5baa492e703d63b3074e40cbb041723203659f42951ce70db1d083dcb25777a6
  random number 8: 88210cb9e669ecf79dd13091c4da7de0beb60f553dd6bf6ad39360c587d2370a
  random number 9: 7943f321315f440803a377e9ddce3696b4f30515b05510e8f368c15837c4b63d
```

RNG

Software test
SW noise: 8604aa5c9683f61c5765fb4bf610ff4026afcde5
SW noise: d24e2528dc24a560b5ecdde9f941ff555ba18758
SW noise: 3758ba50858922f3bd919b14b0da4375e3281129
…10 more
Entropy from pool: 259 bits.
seed: 56580a6f33dffb67271838514b72609d292960989636c17b593018764b2c3caf36ebf42e

Software derived random numbers:
  random number 0: 5704f41ef758d66ef82483217c9291fd79f5aa18b5bd1dc114049df5e81c1d34
  random number 1: c1ae285d23a91a399f5e2c095a769c4195f4c3753a4aca0d78b7d8197378c672
  random number 2: b860d249bbddcaf87666815de2def42d8e73c6de798b667d68f55068a5b79d7c
  random number 3: 4370488c2a28ed85161cd216b3caf2caae2705e9b893c9e082666a4c93622337
  random number 4: 21f3a1438634aa05d4071617ce420786aa574ab4c4258fbbc8d975b2f6c205bb
  random number 5: 10e1fc08d6df99515727b7d68c973deee25b52a28304d29db85aec71735939dc
  random number 6: d9cd2731623a3ee43ba2652bb07e41d00c153d72ab814a24df29883f5c234fb4
  random number 7: 18b9e00e397c7ef8c05f61078ce1884c30f3b74e61286574abac5d01991a8c6b
  random number 8: f2d71f29ee1aeec8109da2a3e83a1aa9f1ded83bb573441c7202e6892381106e
  random number 9: cb467402c79b90e02959d35e0ffe39cf0c2b4da75583f30b2921c0b4073fcc5b

Software jitter sources in my code

- Used five different blocks for jitter including two from Mueller
- For each 8-bit (as required by NIST) noise sample, the estimated entropy varied from 1.8 bit/sample to over 6 bits/sample (!) over the jitter blocks
  - Jitter gives amazingly high rate independent of other activity Can’t be tampered with by adversary
- Most important: Can be analyzed and justified.
  - Isn’t this just an academic concern?
  - No!

Simple Jitter Block 0

- Based on a simple loop adding to an accumulator (Optimization turned off!)

Simple jitter test 0, cpc: 1999533010

largest: 63, smallest: 31, non-zero: 1000, mean: 37.204, adjusted mean: 37.141

44 bins, lower 30, upper: 42, bins from 30 to 42 selected:

0000 0005 0015 0059 0078 0181 0055 0146 0080 0217 0034 0094 0033

Samples: 1000, num loops: 5, expected bin: 37.097, deviation: 32.243

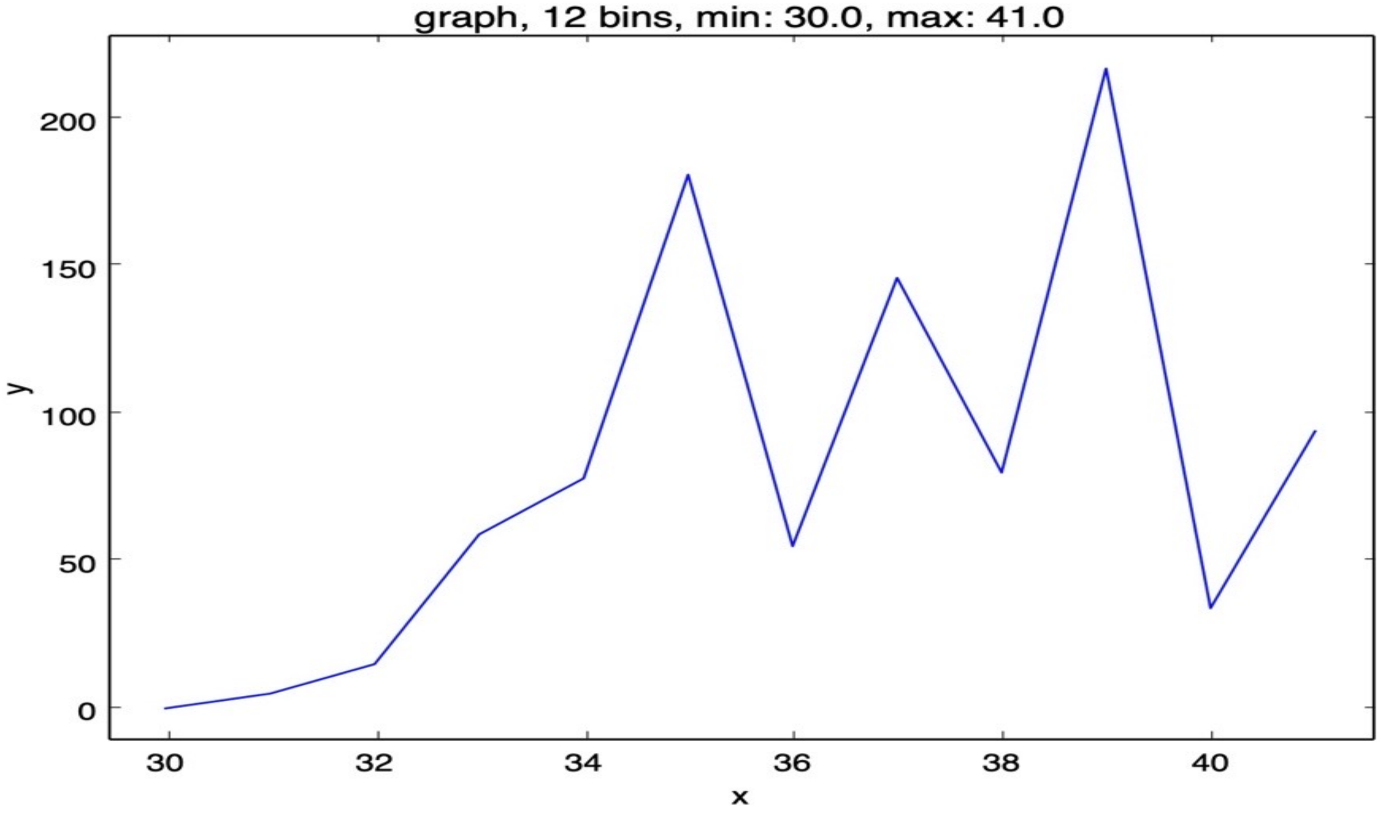
probabilities:

31,0.005 32,0.015 33,0.059 34,0.078 35,0.181 36,0.055 37,0.146

38,0.080 39,0.217 40,0.034 41,0.094

Shannon entropy: 3.168, renyi entropy: 2.927, min entropy: 2.204

Simple Jitter Block 0



Simple Jitter Block 1

Simple jitter test 1, cpc: 2003954612

largest: 125, smallest: 32, non-zero: 1000, mean: 39.565, adjusted mean: 39.440

47 bins, lower 31, upper: 45, bins from 31 to 45 selected:

0000 0001 0010 0024 0045 0073 0090 0038 0286 0094 0110 0073 0104 0036 0013

Samples: 1000, num loops: 5, Expected bin: 39.286, deviation: 33.994

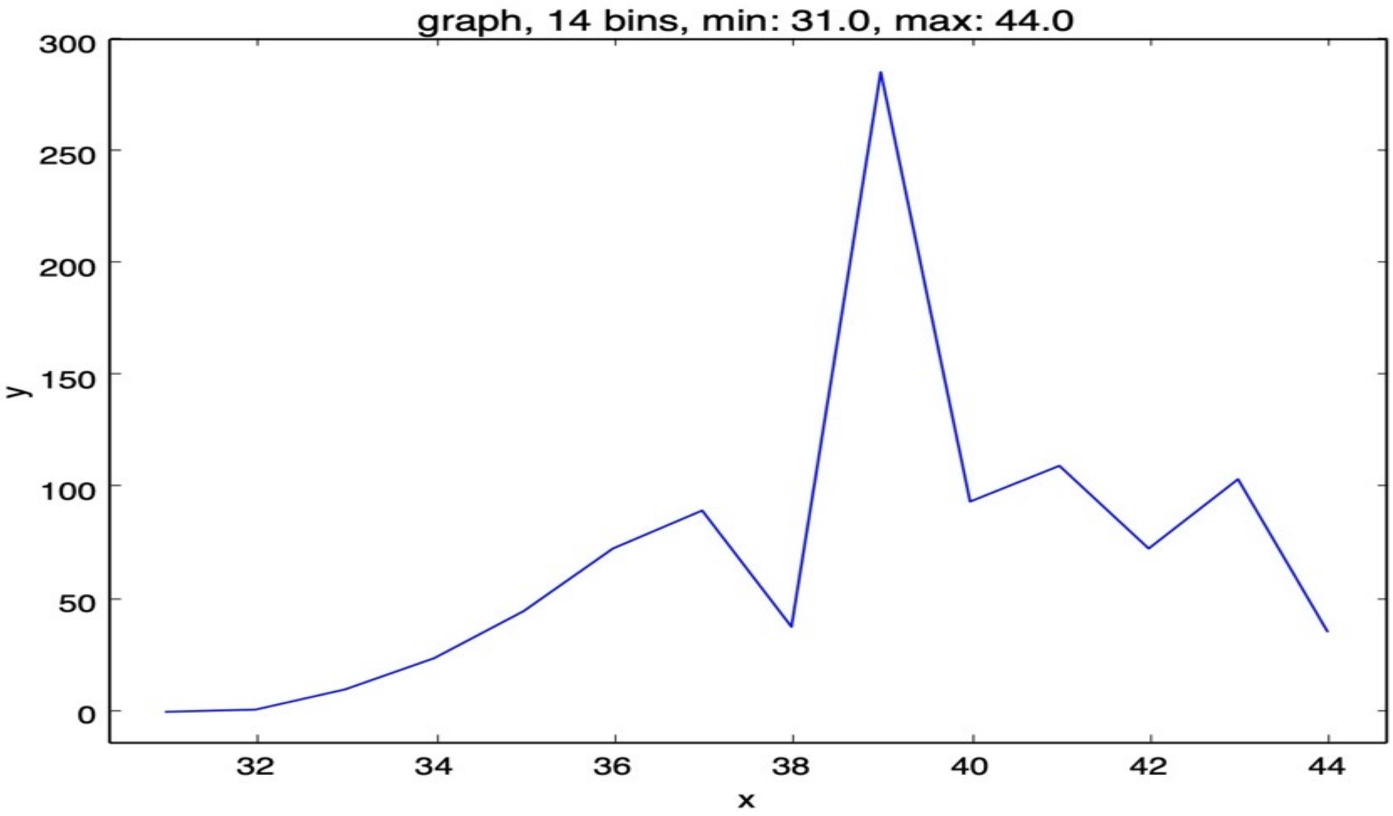
probabilities:

32,0.001 33,0.010 34,0.024 35,0.045 36,0.073 37,0.090 38,0.038

39,0.286 40,0.094 41,0.110 42,0.073 43,0.104 44,0.036

Shannon entropy: 3.231, renyi entropy: 2.858, min entropy: 1.806

Simple Jitter Block 1



Simple Jitter Block 2

Simple jitter test 2, cpc: 2003743142

largest: 165, smallest: 49, non-zero: 1000, mean: 64.158, adjusted mean: 63.993

79 bins, lower 50, upper: 73, bins from 50 to 73 selected:

0002 0000 0012 0010 0032 0009 0046 0082 0050 0079 0088 0103 0031 0067 0079 0100 0031 0043 0022 0028 0025 0014 0003 0007

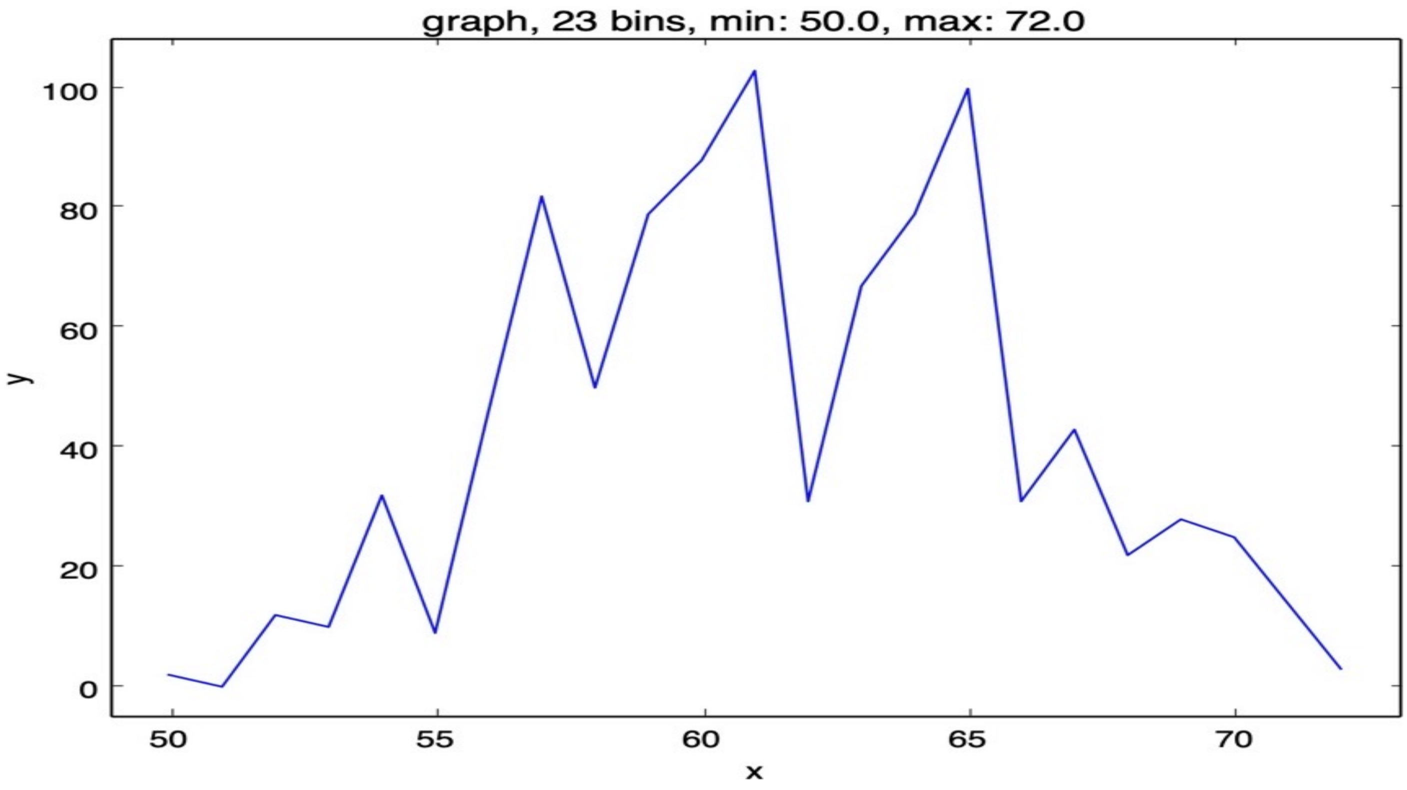
Samples: 1000, num loops: 5, Expected bin: 59.797, deviation: 54.901

probabilities:

50,0.002 52,0.012 53,0.010 54,0.032 55,0.009 56,0.046 57,0.082 …

Shannon entropy: 4.034, renyi entropy: 3.967, min entropy: 3.279

Simple Jitter Block 2



Memory Jitter

- Memory jitter comes from CPU/DRAM clock differences and cache fill wait states.

Memory jitter test, cpc: 1998883639

largest: 255, smallest: 2, non-zero: 1000, mean: 71.828, adjusted mean: 71.573

141 bins, lower 30, upper: 71, bins from 30 to 71 selected:

0004 0003 0018 0010 0015 0006 0017 0011 0020 0028 0014 0022 …

Samples: 1000, num loops: 5, Expected bin: 50.529, deviation: 46.755

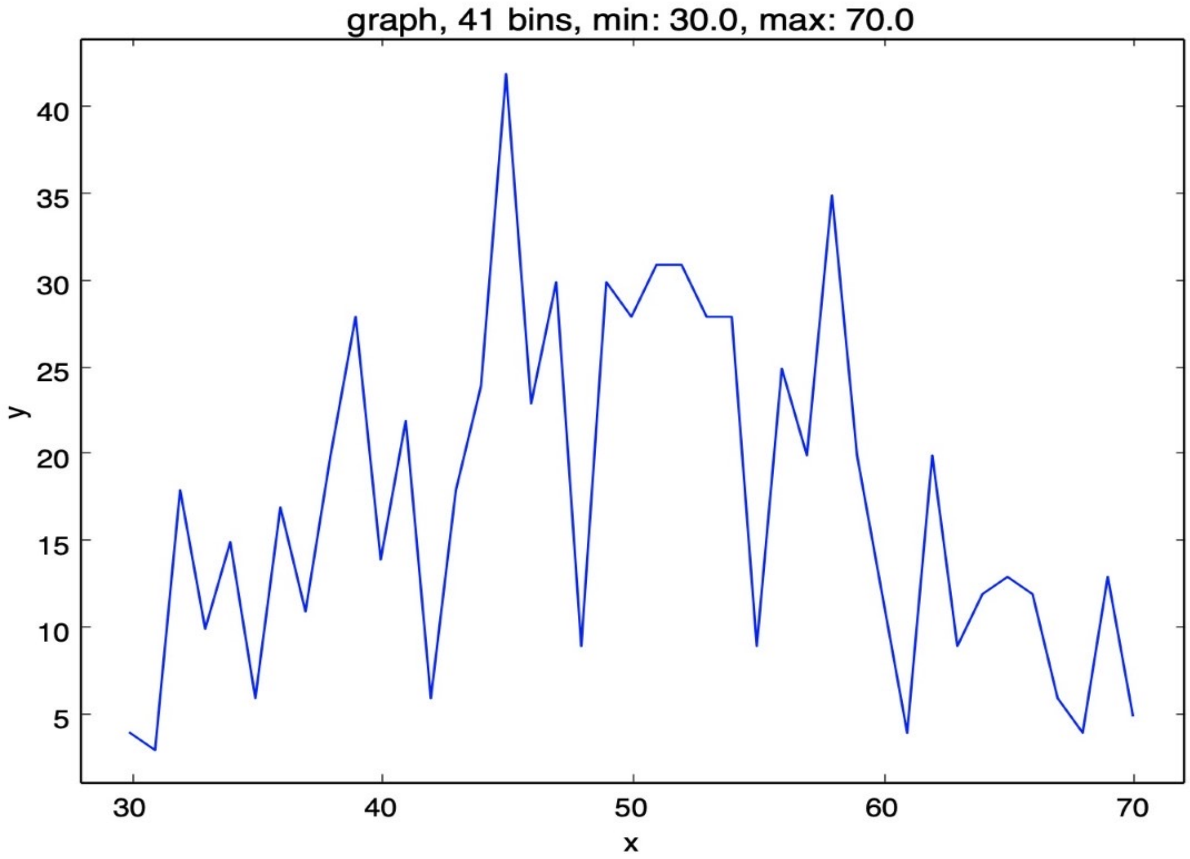
probabilities:

30,0.004; 31,0.003; 32,0.018; 33,0.010; 34,0.015; 35,0.006; …

…

Shannon entropy: 5.476, renyi entropy: 5.873, min entropy: 4.573

Memory Jitter



Hash based execution jitter

- Hash jitter is based on the combined effect of all the sources of variation exhibited in the course of multiple SHA-3 digest computations.

Hash jitter test, cpc: 2003971500

largest: 255, smallest: 0, non-zero: 1000, mean: 132.188, adjusted mean: 131.933

264 bins, lower 66, upper: 237, bins from 66 to 237 selected:

…

Samples: 1000, num loops: 5, Expected bin: 132.188, deviation: 131.502

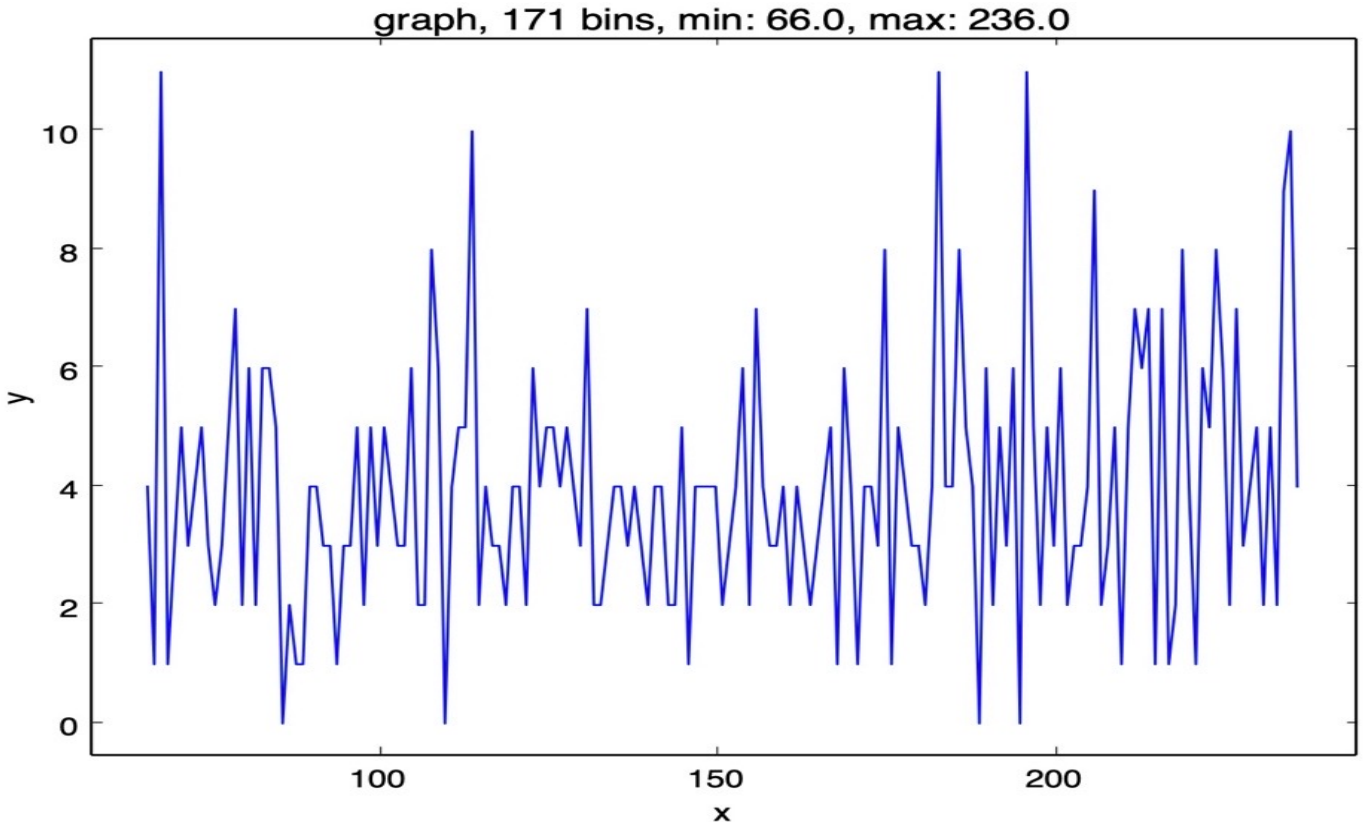
probabilities:

66,.004 67,.001 68,.011 69,.001 70,.003 71,.005 72,.003 73,.004

…

Shannon entropy: 7.791, renyi entropy: 7.637, min entropy: 6.506

Hash based execution jitter



Future work

- Work out hardware model in detail for individual sources of entropy (memory jitter, …).
- Combine complicated jitter sources to obtain different distributions (the “mixture problem”). Done accurately, this is interesting research!
- The level of proof in this presentation is informal or “heuristic,” which is all NIST demands for now.
- NIST will increase level of rigor required in future standards. These arguments can be developed to justify a full stochastic model NSA would be proud of.

Conclusion

- **NIST was completely right!**
  - I apologize for earlier skepticism
- New standard greatly improves security (if you follow it)
  - In cryptography, don’t trust anything you can’t quantifiably analyze --- you’re only fooling yourself.
  - In cryptography, sometimes there are good surprises (jitter).
- Benefits from Jitter
  - No entropy starvation at boot.
  - Defense in depth (qualified HW and SW entropy).
  - Simpler than interrupts and less performance impact
  - Works on all devices (even embedded devices).
  - Widely accepted despite initial skepticism: Linux, (some) BSD versions and Apple (during boot).
  - You can stop drinking the kool-aid: Jettison previous software entropy “pseudo-science.”

Security Model

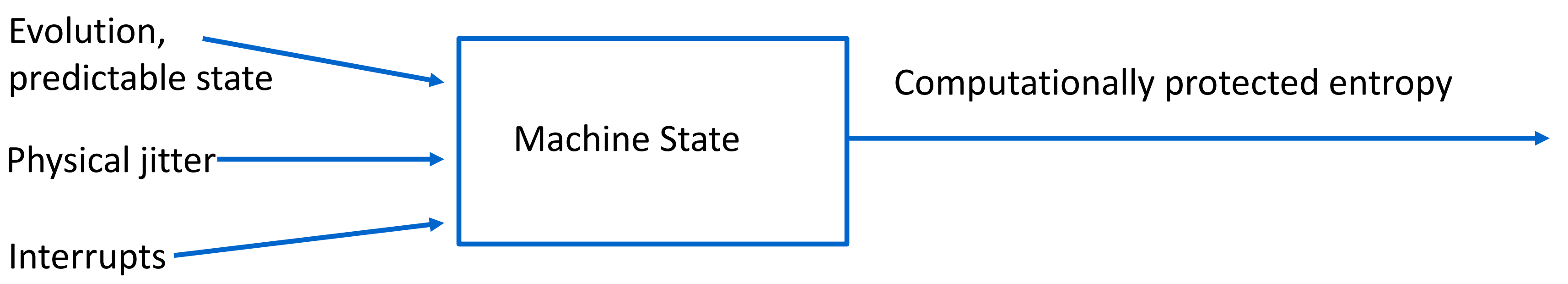
- There are two different security foundations for “execution jitter” as an entropy source.
  - Unpredictable timing artifacts that are caused by pure physical processes that affect jitter. For example, cross domain clocking environments which affects timing jitter. This is physical.
  - The computational barrier to recreate complex CPU state: caches, branch prediction, frequency scaling, intervening interrupts, locks, cross CPU performance differences, TLB misses, speculative execution.
  - Both are present.

Security Model --- symmetric crypto

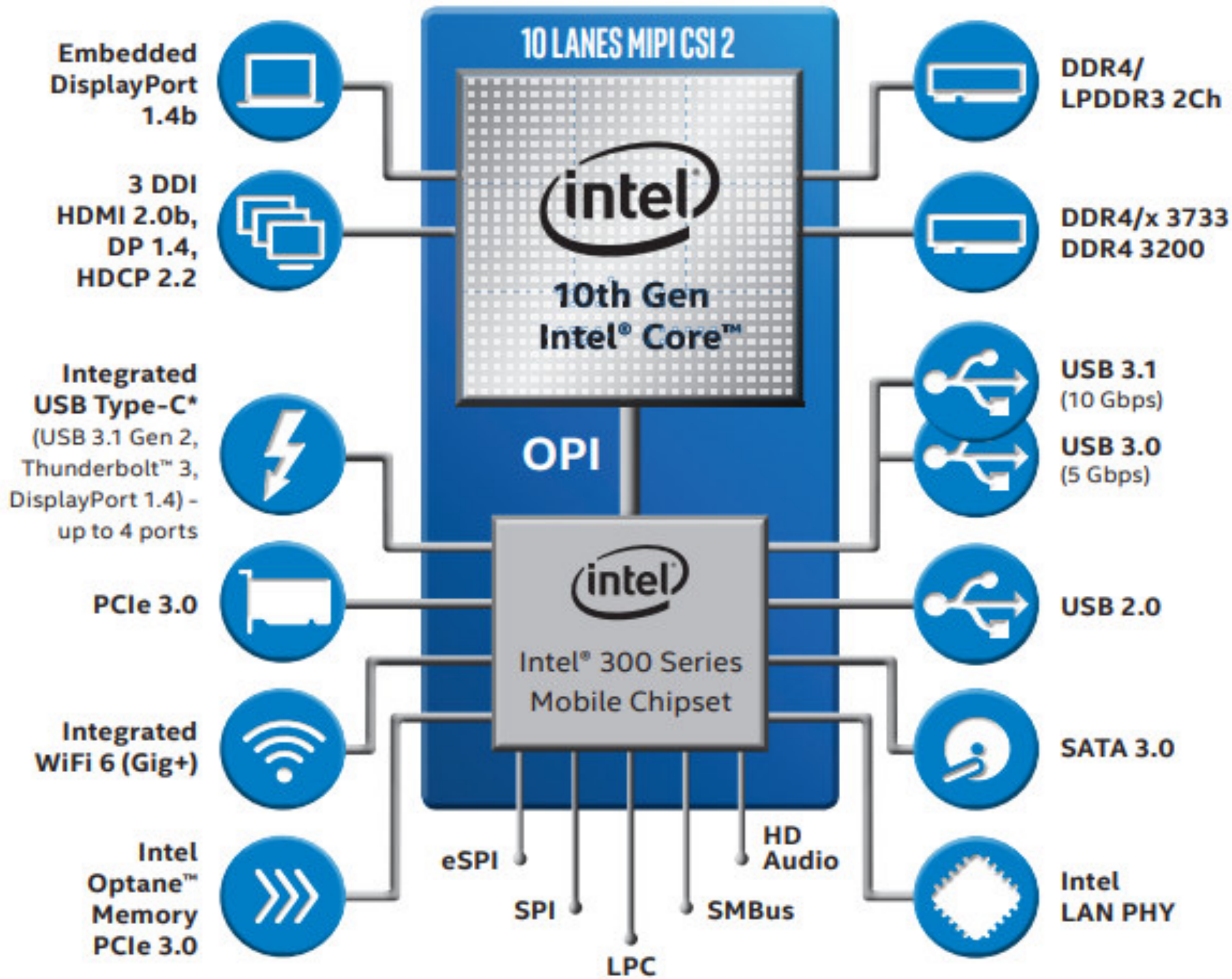
- Two security models
  - One time pad corresponds to “physical entropy.” Safe regardless of computational assumptions.
  - Real security based on computation
    - Model: Given a known cipher (say AES) with a prescribed key size (say 128 bits) and one block of corresponding known plain and ciphertext. What is “gold standard” to evaluate its security?
    - Answer: If best adversary’s most efficient attack is brute force, cipher is good. Guaranteed success in 2^{128} steps, expected cost is 2^{127} steps.
- Computational entropy
  - If I produce n bits of entropy and any successful adversary must carry out 2^n operations (say, to recreate a deterministic process used to produce the entropy), the entropy generation has “equivalent” security to the underlying cipher that uses it.
  - So, we have to “prove” an adversary must perform 2^n operations to condition machine state to produce the environment that generated the entropy scheme.

Translating this to execution jitter

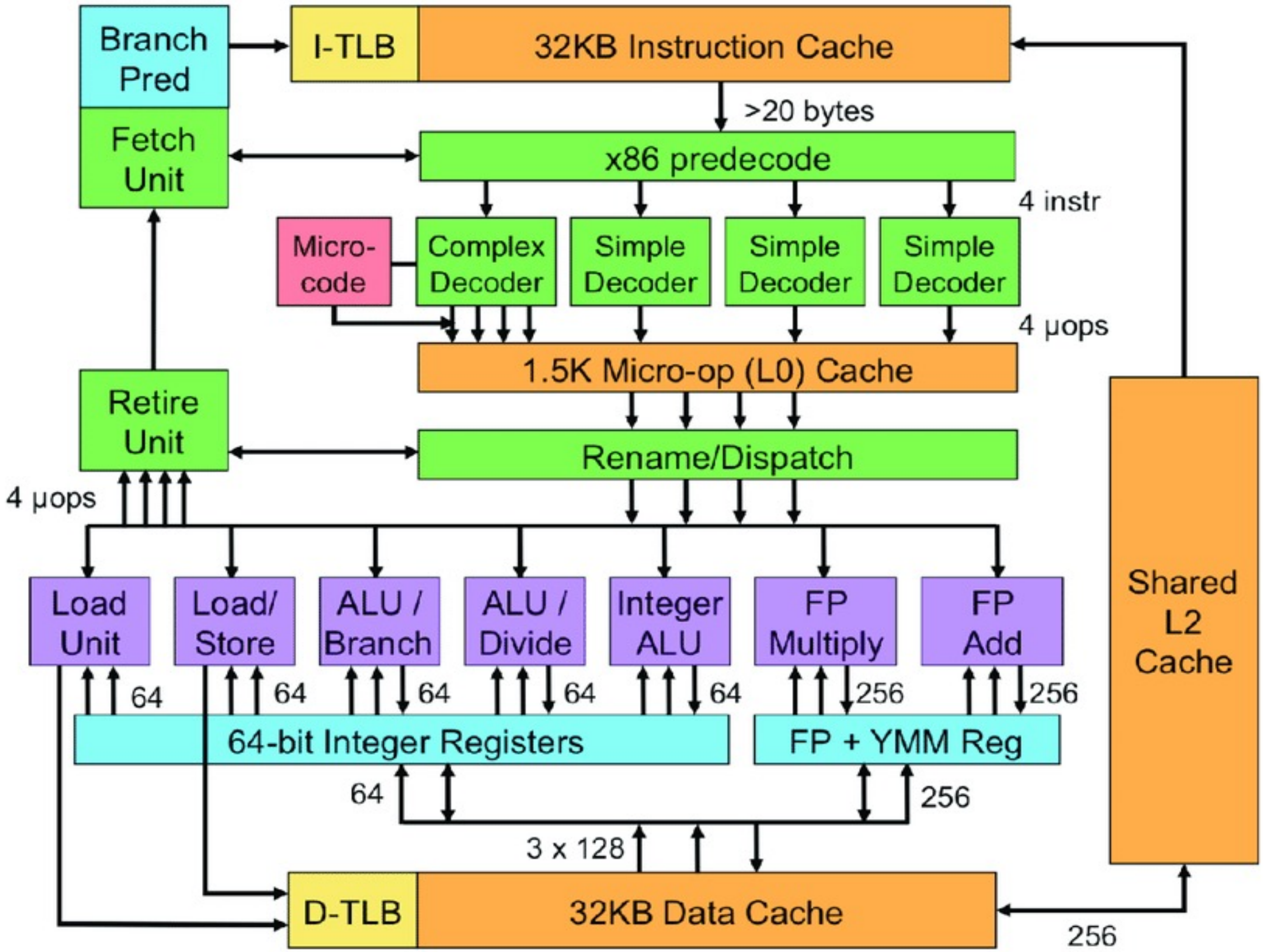
- We want to show that the expected effort to “recreate” the machine state to successfully reproduce execution jitter purporting to provide n bits of entropy requires 2^n operations (this includes guesses or conditioning)
  - Reduction to scheduling problem (given actual physical jitter)
  - Thousands of bits of machine state contribute to producing the precise environment:
    - Cache state, TLB, branches, precise timing of board level interrupts and their affect on state, races that affect physical maps, microcode.
    - Not all internal state can be “set.” Some require reproducing execution traces and interrupts as well as full knowledge of all associated software affecting the platform and even data associated with the computation.



Machine state (Intel x64)



Machine state (Intel x64)



Machine state (Intel x64)

Feature	Core i7
Number of cores	4 (each core hyperthreaded)
Pipeline	OoO
Number of stages	19
Width of fetch	6 instructions
Width of decode	4-7 fused u-ops
Size of decode queue	56 entries
Width of reissue/rename	4 fused u-ops
Width of dispatch	8 u ops
Width of commit	4 fused u-ops
Reservation station	60 u-ops
Reorder buffer	192 u-ops
L1 (Data cache)	8 way, 32KB (8KB/core) 128 64B cache blocks, 16 sets
L1 (Instruction cache)	8 way 32 KB (8KB/core)
L2	8 way, 256 KB
L3	16 way, 16MB

Machine state (Intel x64)

Feature	Core i7
L1 latency	4 cycles
L2 latency	12 cycles
L3 latency	36 cycles
Size of cache block	64Bytes
BTB associativity	4096, 4 way
RAS	16 entries
Branch mispredict penalty	14 cycles
Integer/float registers	168/168
Instruction TLB	128 entries
Data TLB, associativity	64, 4 way
L2 TLB size, associativity	1024 entries, 8 way

Machine state (Intel x64)

- Time of day clock
- CPU cycle rate
- DRAM cycle and refresh rates
- CPU instruction pipelines fill levels and stalls
- The CPU frequency scaling
- The CPU power management may disable CPU features.
- Instruction and data caches
- TLB (Miss penalty: 9 cycles)
- CPU topology and caches used jointly by multiple CPUs affect execution time.
- Branch prediction (Mis-predict penalty between 10 and 40 cycles)
- Kernel locks/barriers
- CPU configuration on multi-core machine
- Hardware interrupts

Machine Data

- Client Intel i7-8650 @ 1.9GHz
- CPU MHz: 947-1710MHz
- 8KB/core instruction and data cache
- Quiet system
- 253 soft irqs/sec, .25 interrupts/irq
- 509 interrupts/sec, .51 interrupts/ms
- For, say, hash jitter, what is the average running time?
  - With 10 repeats: 19200 cycles (10 μsec)
- What is the probability of an interrupt during a run?
- Need to know what CPU services interrupt
  - Probably balanced

Sources of Equivocation in Execution Jitter

- CPU/DRAM jitter in cache misses
- Unpredictability of initial state of caches, branch predictors, pipelines on entry to jitter execution sampling caused by evolution of state prior to entry.
  - Evolution caused by planned intermediate execution
  - Core assignment
  - “Unpredictable” interrupts affecting state evolution prior to sampling.
- Interrupts occurring *during* an execution jitter sample
  - Delays processing (hence sample time)
  - Changes processor state

State evolution on x64

- 20% of code is branches
- Branches correctly predicted about 80% of the time
- L1 cache misses about 2% of the time
- L2 and L3 caches also effect execution. Should we model this?
- Locks and barriers in kernel
- Pipeline improves performance by 20 – 100% (10 cycle deviation)
  - When does pipeline flush?
- Effect of virtual to physical memory assignment and paging and TLB misses
- Serial run of 50000 samples collected serially (jitter_block_0), establishes lower bound of 2 bits of “variability” in execution time caused by a routine of about 30 instructions .

Effect of memory jitter (More needed)

- Variability attributable or cache misses:
  - -(.98 lg(.98)+.02lg(.02))= .14 bit per access
  - Each miss causes 8 cycle difference
  - 1 bit on entropy/sample (Rambus data)
  - How much physical jitter happens per cache miss

Effect of state randomization on jitter execution

- Serial run of 50000 samples collected serially, establish lower bound of 2.5 bits of variability due to state randomization discounting interrupts during execution and memory jitter. This includes TLB, branch and speculation. Almost no paging.
- Branch prediction succeeds 80% of the time and each failure results in a 40-80 cycle difference.

Interrupts and state evolution

- Things that are monotonic: TOD clock, counters
- The data collected on my NUC shows that there are approximately 600 interrupts per sec on my 4 core i7.
- Each interrupt results in more than 40 instructions executed.
  - Give an example
  - Markov model of state affecting jitter samples

Effect of interrupts on state evolution and execution time - I

- As “jitter_block_0” shows, every series of 30 instructions contributes about 2 bits of apparent entropy to jitter execution, even in a single execution context, indicating that 15 instructions are sufficient to cause a state change which causes a difference in jitter execution time between two jitter execution measurements.
- Consider a single core with 150 expected interrupts where each interrupt introduces about 40 instructions.
- An adversary must model $\frac{150 \times 40}{15} = 400$ possible state changes every second to accurate model the jitter execution time.
- Thus, an adversary’s jitter prediction model requires a work factor of 2^{400} after 1 second of “random” interrupts to guess the initial state affecting jitter execution performance.

Effect of interrupts on state evolution and execution time -- II

- If we delay collecting execution jitter for 1 second on a device experiencing 150 interrupts/sec interrupts, an adversary cannot compute the deterministic state to reproduce the first jitter execution sequence with a probability of more than 2^{-400}. [This needs more justification.]
- Thereafter, interrupts contribute no less than $\frac{150 \times 40}{15}$ bits of entropy per second to jitter measurements.
- It remains to schedule execution jitter sampling to ensure a minimum entropy (due to state) of 1 bit which we can achieve by sampling no more frequently than every 2.5 ms. Each such sample has “at least” 1 bit of entropy.

Bottom line

- Memory jitter and execution jitter caused by state evolution supports estimated entropy provided we sample less frequently than every 2.5 ms and these estimates are very conservative.
- We have not included effects adding to entropy based on interrupts that occur during a jitter sample execution.
- We do not assume that interrupts are randomly distributed. We only require that the number of interrupts occurring every 2.5 ms can be predicted with a probability of no better than $\frac{1}{2}$.

Notes

- mpstat –I ALL –P ALL