Linux /dev/random
A New Approach

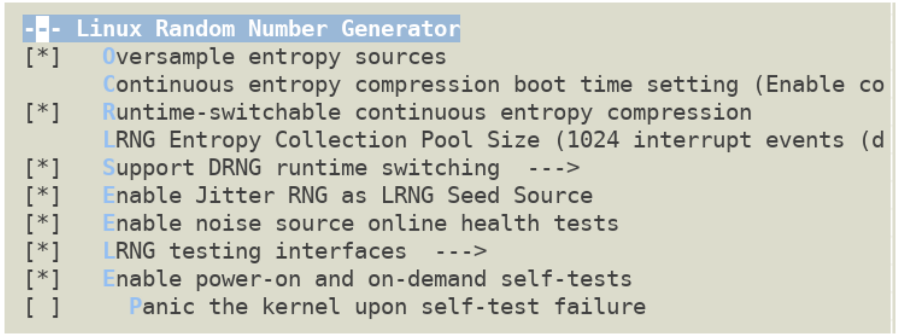
Stephan Müller
<smueller@chronox.de>

Agenda

- LRNG Goals
- LRNG Design
- Entropy Sources

LRNG Goals

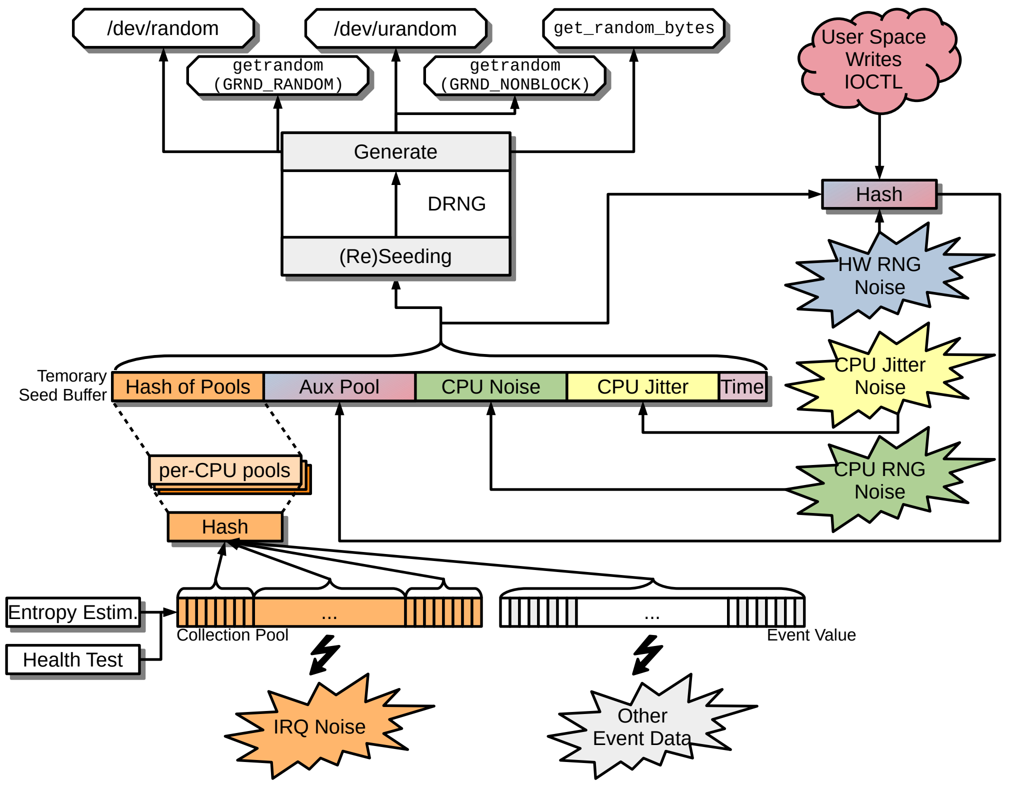
- Sole use of cryptography for data processing
- High-Performance lockless IRQ handler
- Test interfaces for all LRNG processing steps
- Power-up and runtime tests
- API and ABI compliant drop-in replacement of existing /dev/random
- Flexible configuration supporting wide range of use cases
- Runtime selection of cryptographic implementations
- Clean architecture
- Standards compliance: SP800-90A/B/C, AIS 20/31



```
--- Linux Random Number Generator
[*]   Oversample entropy sources
      Continuous entropy compression boot time setting (Enable co
[*]   Runtime-switchable continuous entropy compression
      LRNG Entropy Collection Pool Size (1024 interrupt events (d
[*]   Support DRNG runtime switching  --->
[*]   Enable Jitter RNG as LRNG Seed Source
[*]   Enable noise source online health tests
[*]   LRNG testing interfaces  --->
[*]   Enable power-on and on-demand self-tests
[ ]     Panic the kernel upon self-test failure
```

LRNG Design

- 4 Entropy Sources
  - 3 external
  - 1 internal
  - All ES treated equally
  - No domination by any ES – seeding triggered by boot process or DRNG
- All ES can be selectively disabled at runtime
- ES data fed into DRNG
- DRNG accessible with APIs

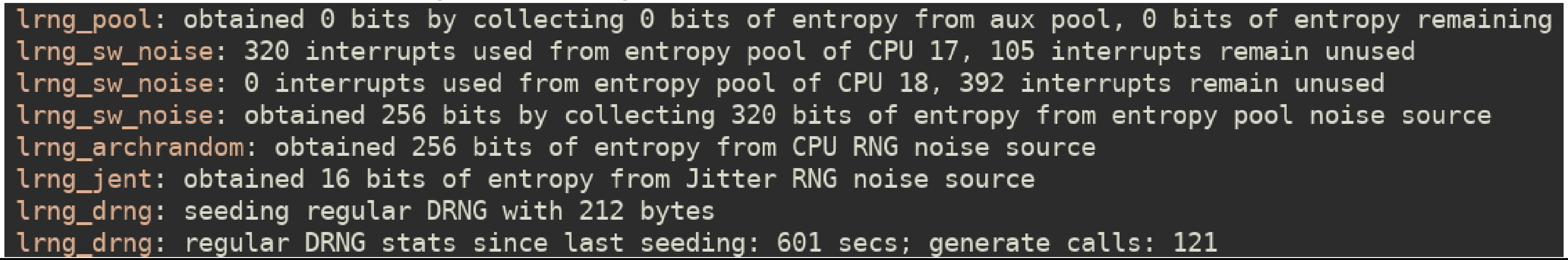


DRNG Output APIs

- Blocking APIs – deliver data only after fully initialized and fully seeded:
  - /dev/random
  - getrandom() system call
  - get_random_bytes_full in-kernel API
- All other APIs deliver data without blocking
  - No guarantee of LRNG being fully initialized / seeded

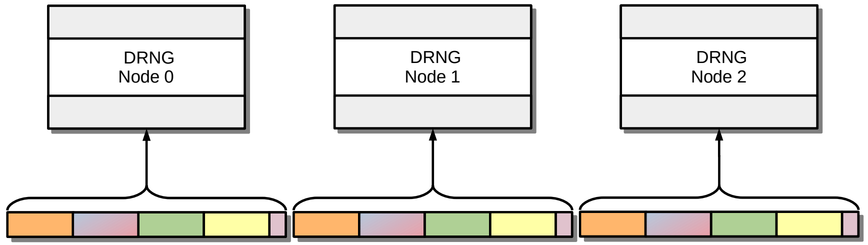
DRNG Seeding

- Temporary seed buffer: concatenation of output from all ES
- Seeding during boot: when 32/128/256 bits of entropy are available
- Seeding at runtime:
  - After 2^20 generate requests or 10 minutes
  - After forced reseed by user space
  - At least 128 bits of total entropy must be available
  - 256 bits of entropy requested from each ES – ES may deliver less
  - Seed operation occurs when DRNG is requested to produce random bits



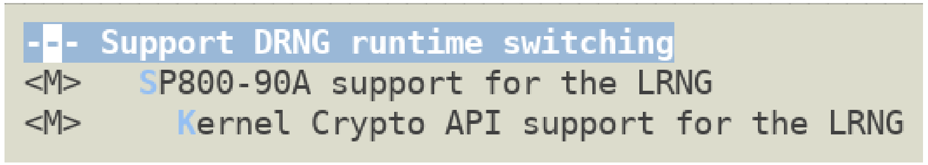
```
lrng_pool: obtained 0 bits by collecting 0 bits of entropy from aux pool, 0 bits of entropy remaining
lrng_sw_noise: 320 interrupts used from entropy pool of CPU 17, 105 interrupts remain unused
lrng_sw_noise: 0 interrupts used from entropy pool of CPU 18, 392 interrupts remain unused
lrng_sw_noise: obtained 256 bits by collecting 320 bits of entropy from entropy pool noise source
lrng_archrandom: obtained 256 bits of entropy from CPU RNG noise source
lrng_jent: obtained 16 bits of entropy from Jitter RNG noise source
lrng_drng: seeding regular DRNG with 212 bytes
lrng_drng: regular DRNG stats since last seeding: 601 secs; generate calls: 121
```

DRNG Management



- One DRNG per NUMA node
- Hash contexts NUMA-node local
- Each DRNG initializes from entropy sources
- Sequential initialization of DRNG – first is Node 0
- If DRNG on one NUMA node is not yet fully seeded → use of DRNG(Node 0)
- Each DRNG instance managed independently
- To prevent reseed storm – reseed threshold different for each node
  - Node 0: 600 seconds
  - Node 1: 700 seconds
  - …
- NUMA support code only compiled if CONFIG_NUMA → only one DRNG present

Data Processing Primitives



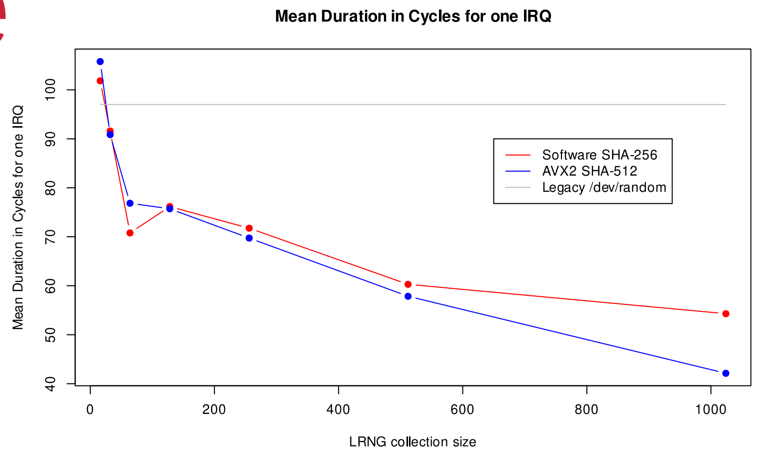
```
-*- Support DRNG runtime switching
<M>   SP800-90A support for the LRNG
<M>     Kernel Crypto API support for the LRNG
```

- Sole use of cryptographic mechanisms for data compression
- Cryptographic primitives Boot-Time / Runtime switchable
  - Switching support is compile-time option
  - DRNG, Conditioning hash
  - Built-in: ChaCha20 DRNG / SHA-256
  - Available:
    - SP800-90A DRBG (CTR/Hash/HMAC) using accelerated AES / SHA primitive, accelerated SHA-512 conditioning hash
    - Hardware DRNG may be used (e.g. CPACF)
    - Well-defined API to allow other cryptographic primitive implementations
- Complete cryptographic primitive testing available:
  - Full ACVP test harness available: https://github.com/smuellerDD/acvpparser
  - ChaCha20 DRNG userspace implementation: https://github.com/smuellerDD/chacha20_drng
- Other data processing primitives:
  - Concatenation of data
  - Truncation of message digest to heuristic entropy value
- Entropy behavior of all data processing primitives based on fully understood and uncontended operations

External Entropy Sources

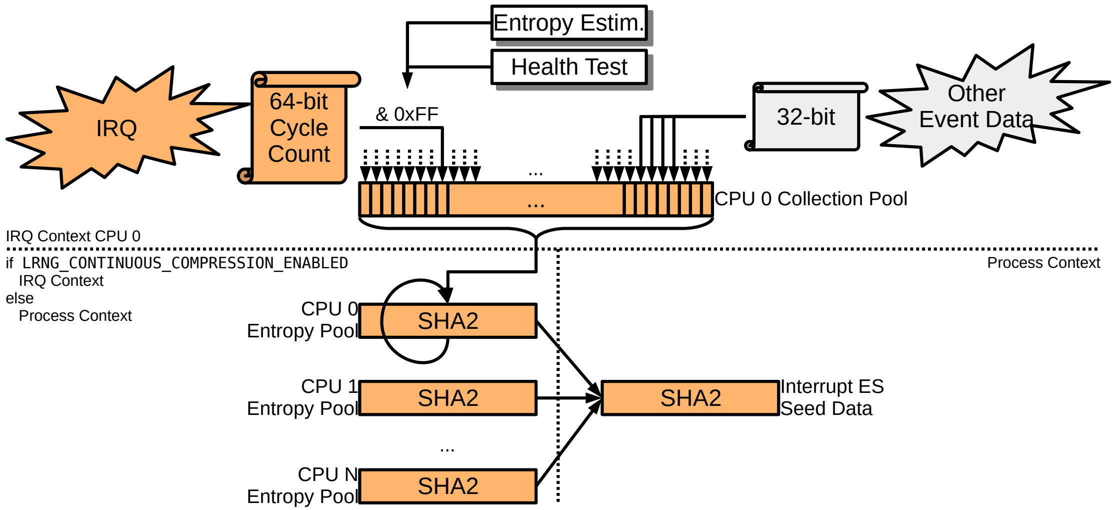
- Use without additional conditioning – fast source
  - Jitter RNG
  - CPU (e.g. Intel RDSEED, POWER DARN, ARM SMC Calling Convention or RNDR register)
  - Data immediately available when LRNG requests it
- Additional conditioning – slow source
  - RNGDs
  - In-kernel hardware RNG drivers
  - All received data conditioned into “auxiliary pool“
  - Data “trickles in” over time
- Every entropy source has individual entropy estimate
  - Taken at face value – each ES requires its own entropy assessment

Internal Entropy Source



- Interrupt timing
  - All interrupts are treated as one entropy source
- Data collection executed in IRQ context
- Data compression executed partially in IRQ and process context
- High performance: up to twice as fast as legacy /dev/random in IRQ context with LRNG_CONTINUOUS_COMPRESSION enabled
  - Even faster without continuous compression

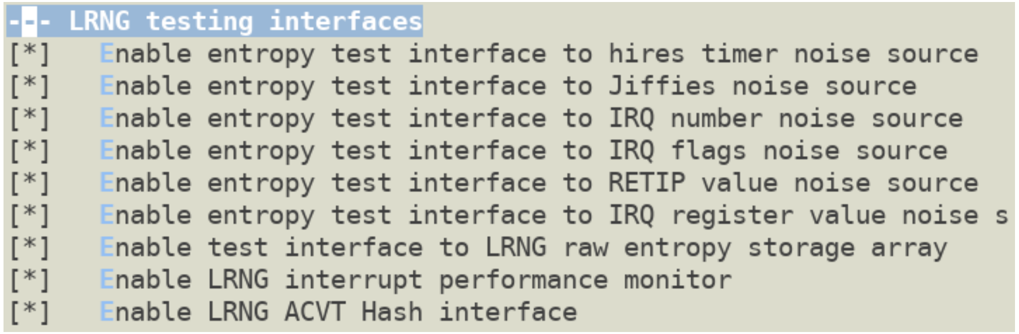
Internal ES Data Processing



- 8 LSB of time stamp concatenated into per-CPU collection pool
  - Entropy estimate
  - Health test
- 32 bits of other event data concatenated into per-CPU collection pool
- When array full → conditioned into per-CPU entropy pool
  - When entropy is required → conditioning of all entropy pools into one message digest
  - Addition of all per-CPU entropy estimates

Internal ES Testing Interfaces

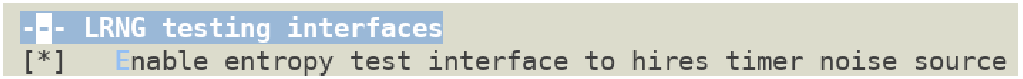
- Testing code is compile time option
- Access via DebugFS
- Testing supports data collection at boot time and runtime:
  - Raw unprocessed entropy time stamps for IRQ
  - Raw Jiffies for IRQ
  - IRQ value
  - IRQ flags value
  - _RET_IP_ per IRQ
  - Performance data for LRNG's IRQ handler
- Hash testing interface for built-in SHA-256



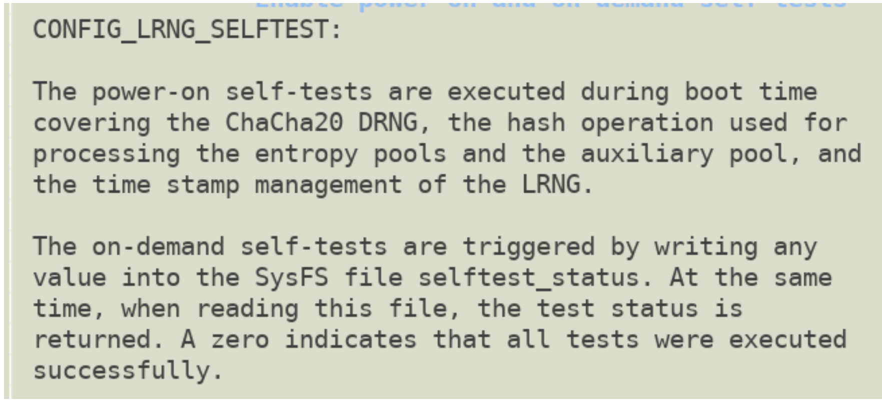
```
--- LRNG testing interfaces
[*]   Enable entropy test interface to hires timer noise source
[*]   Enable entropy test interface to Jiffies noise source
[*]   Enable entropy test interface to IRQ number noise source
[*]   Enable entropy test interface to IRQ flags noise source
[*]   Enable entropy test interface to RETIP value noise source
[*]   Enable entropy test interface to IRQ register value noise s
[*]   Enable test interface to LRNG raw entropy storage array
[*]   Enable LRNG interrupt performance monitor
[*]   Enable LRNG ACVT Hash interface
```

Internal ES Health Test

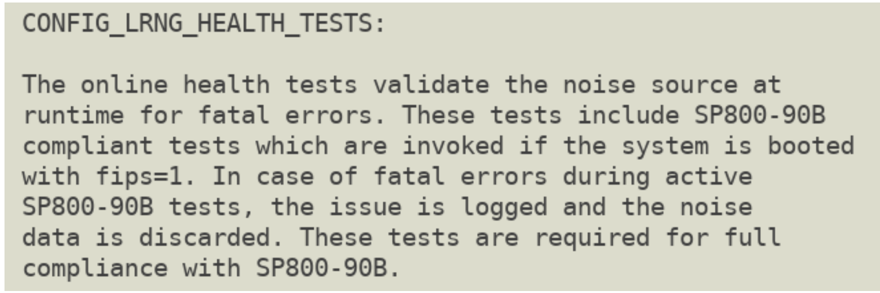
- Health test compile-time configurable
- Power-Up self tests
  - All cryptographic mechanisms
  - Time stamp management
- APT / RCT
- Time-Stamp Pattern detection: 1st/2nd/3rd discrete derivative of time ≠ 0
- Blocking interface: Wait until APT power-up testing complete
- Full SP800-90B assessment documentation
- Raw entropy collection and analysis tools provided



```
-*- LRNG testing interfaces
[*]   Enable entropy test interface to hires timer noise source
```



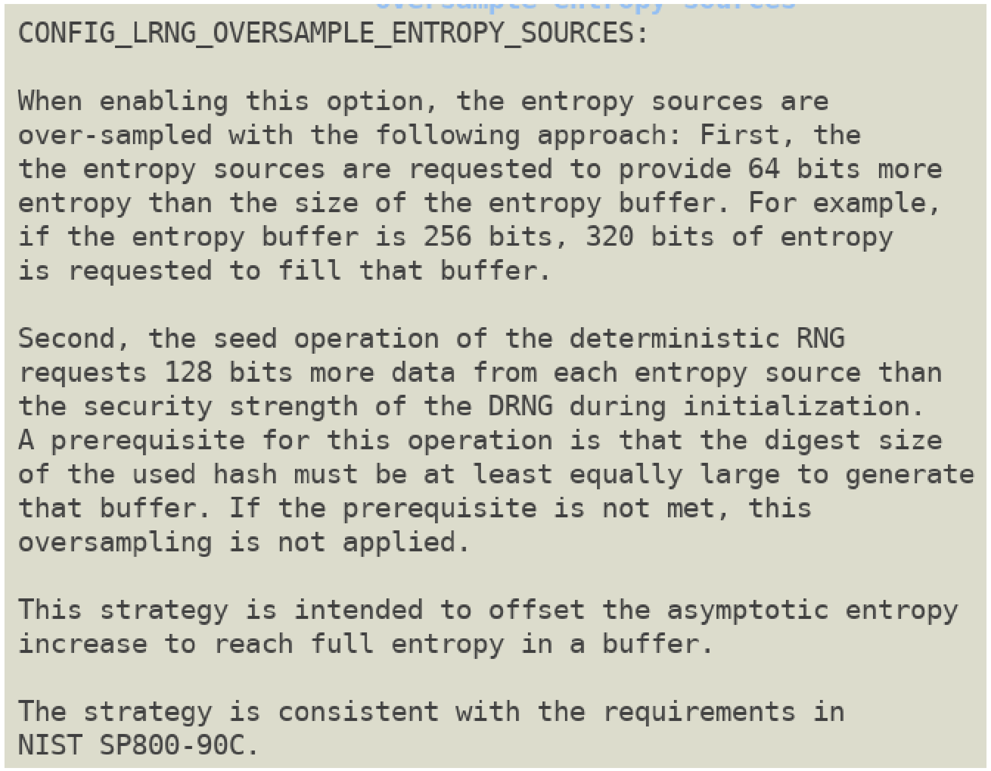
```
CONFIG_LRNG_SELFTEST:

The power-on self-tests are executed during boot time
covering the ChaCha20 DRNG, the hash operation used for
processing the entropy pools and the auxiliary pool, and
the time stamp management of the LRNG.

The on-demand self-tests are triggered by writing any
value into the SysFS file selftest_status. At the same
time, when reading this file, the test status is
returned. A zero indicates that all tests were executed
successfully.
```



```
CONFIG_LRNG_HEALTH_TESTS:

The online health tests validate the noise source at
runtime for fatal errors. These tests include SP800-90B
compliant tests which are invoked if the system is booted
with fips=1. In case of fatal errors during active
SP800-90B tests, the issue is logged and the noise
data is discarded. These tests are required for full
compliance with SP800-90B.
```

Entropy Source Oversampling

- Compile time option
  - Function only enabled in FIPS mode
  - Function only enabled if message digest of conditioner >= 384 bits
- Final conditioning: s + 64 bit
- Initial DRNG seeding: every entropy source requested for s + 128 bits
  - Every ES alone could provide all required entropy
- All ES data concatenated into seed buffer
- Runtime debug mode: display of all processing steps
- SP800-90C compliance:
  - SP800-90A DRBG with 256 bit strength / SHA-512 vetted conditioning component
  - Complies with RBG2(NP) per default
  - Can be configured to provide RBG2(P)



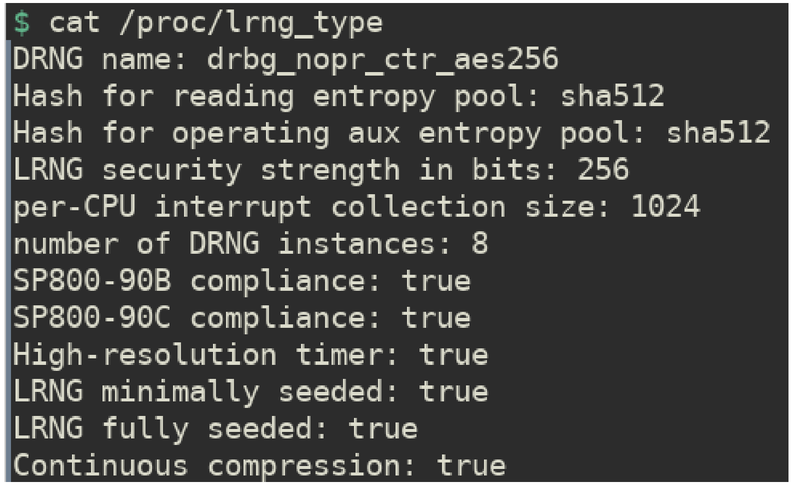
```
CONFIG_LRNG_OVERSAMPLE_ENTROPY_SOURCES:

When enabling this option, the entropy sources are
over-sampled with the following approach: First, the
the entropy sources are requested to provide 64 bits more
entropy than the size of the entropy buffer. For example,
if the entropy buffer is 256 bits, 320 bits of entropy
is requested to fill that buffer.

Second, the seed operation of the deterministic RNG
requests 128 bits more data from each entropy source than
the security strength of the DRNG during initialization.
A prerequisite for this operation is that the digest size
of the used hash must be at least equally large to generate
that buffer. If the prerequisite is not met, this
oversampling is not applied.

This strategy is intended to offset the asymptotic entropy
increase to reach full entropy in a buffer.

The strategy is consistent with the requirements in
NIST SP800-90C.
```

General Testing

- Locking torture test of loading/unloading DRNG extensions under full load
- Applied kernel framework tests:
  - KASAN
  - UBSAN
  - Lockdep
  - Memory leak detector
- Use of LRNG without kernel crypto API
- Performance tests of DRNG
- Syscall validation testing
- Test of LRNG behavior in atomic contexts

LRNG - Resources

- Code / Tests / Documentation: https://github.com/smuellerDD/lrng
- Testing conducted on
  - Intel x86, AMD, ARM, MIPS, POWER LE / BE, IBM Z
  - Embedded systems and Big Iron
- Backport patches available
  - LTS: 5.10, 5.4, 4.19, 4.14, 4.4
  - 5.8, 4.12, 4.10
- Why is it not upstream?



```
$ cat /proc/lrng_type
DRNG name: drbg_nopr_ctr_aes256
Hash for reading entropy pool: sha512
Hash for operating aux entropy pool: sha512
LRNG security strength in bits: 256
per-CPU interrupt collection size: 1024
number of DRNG instances: 8
SP800-90B compliance: true
SP800-90C compliance: true
High-resolution timer: true
LRNG minimally seeded: true
LRNG fully seeded: true
Continuous compression: true
```