Linux /dev/random
A New Approach

Stephan Müller
<smueller@chronox.de>

Agenda

- LRNG Goals
- LRNG Design
- Initial Seeding Strategies
- Entropy Sources

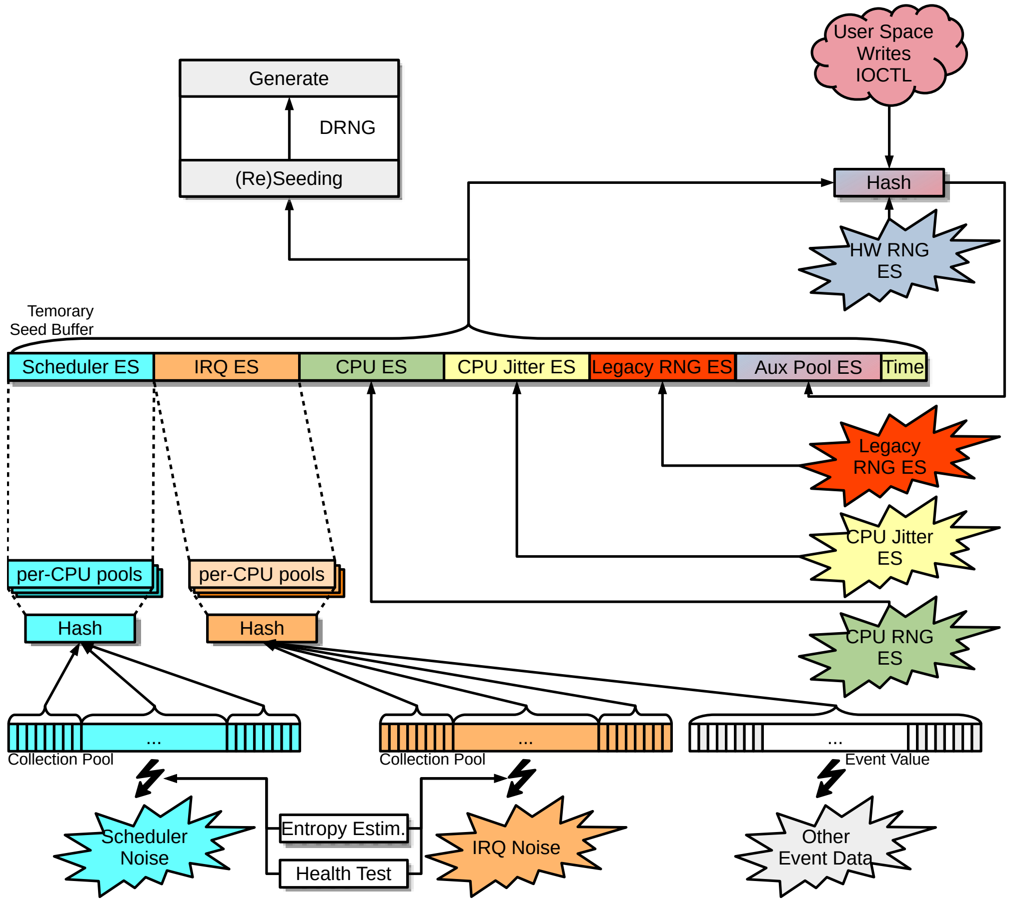
LRNG Goals

- Sole use of cryptography for data processing
- High-Performance lockless IRQ handler
- Test interfaces for all LRNG processing steps
- Power-up and runtime tests
- Compile-time enabling of 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 – all permutations of options of the LRNG always lead to a secure random bit generation
- Standards compliance: SP800-90A/B/C, AIS 20/31, FIPS IG 7.19 / D.K (use of DRBG as conditioner)

```
--- Linux Random Number Generator
      Specific DRNG seeding strategies  --->
      Entropy Source Configuration  --->
[*]   Support DRNG runtime switching  --->
[*]   LRNG testing interfaces  --->
[*]   Enable power-on and on-demand self-tests
[ ]     Panic the kernel upon self-test failure
```

LRNG Design

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



DRNG Output APIs

- Blocking APIs – deliver data only after fully initialized and fully seeded
  - lrng_get_random_bytes_full in-kernel API
  - When /dev/random compliant API enabled:
    - /dev/random
    - getrandom() system call
    - get_random_bytes in-kernel API after being triggered with add_random_ready_callback or after rng_is_initialized returns true
- Prediction Resistance API – deliver data only after fully initialized and successful reseed returning at most data equal to the amount of entropy
  - Separate DRNG instance operating with prediction resistance generating at most as much data as seed entropy was inserted
  - Using /dev/random with O_SYNC
  - Using getrandom(2) with flag GRND_RANDOM
  - Compliant with:
    - FIPS IG 7.19 / D.K to use DRBG as conditioning component for seeding other DRBGs
    - German AIS 20/31 (2011) NTG.1 requirements
- Get seed: getrandom(2) with flag GRND_SEED to obtain data from entropy sources directly
- All other APIs deliver data without blocking until complete initialization
  - 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
  - After new DRNG is loaded
  - At least 128 bits (SP800-90C mode: LRNG security strength) 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
  - DRNG returns to not fully seeded when last seed with full entropy was > 2^{30} generate operations ago
  - Pictures shows regular and SP800-90C initial seeding behavior

```
[   58.360166] lrng_es_irq: 256 interrupts used from entropy pool of CPU 17, 0 interrupts remain unused
[   58.360171] lrng_es_irq: 0 interrupts used from entropy pool of CPU 18, 256 interrupts remain unused
[   58.360175] lrng_es_irq: 0 interrupts used from entropy pool of CPU 19, 256 interrupts remain unused
[   58.360177] lrng_es_irq: obtained 256 bits by collecting 256 bits of entropy from entropy pool noise source
[   58.360183] lrng_es_archrandom: obtained 256 bits of entropy from CPU RNG noise source
[   58.364772] lrng_es_jent: obtained 16 bits of entropy from Jitter RNG noise source
[   58.365128] lrng_es_aux: obtained 256 bits by collecting 256 bits of entropy from aux pool, 0 bits of entropy remaining
```

```
[93745.008780] lrng_es_irq: 256 interrupts used from entropy pool of CPU 17, 0 interrupts remain unused
[93745.008785] lrng_es_irq: 192 interrupts used from entropy pool of CPU 18, 64 interrupts remain unused
[93745.008789] lrng_es_irq: 0 interrupts used from entropy pool of CPU 19, 256 interrupts remain unused
[93745.008791] lrng_es_irq: obtained 384 bits by collecting 448 bits of entropy from entropy pool noise source
[93745.008800] lrng_es_archrandom: obtained 384 bits of entropy from CPU RNG noise source
[93745.015528] lrng_es_jent: obtained 24 bits of entropy from Jitter RNG noise source
[93745.015887] lrng_es_aux: obtained 192 bits by collecting 256 bits of entropy from aux pool, 0 bits of entropy remaining
```

Initial Seeding Strategy I
Default Operation

- DRNG is initially seeded with at least 32 bits of entropy
- DRNG is minimally seeded with at least 128 bits of entropy
- DRNG is fully seeded with 256 bits of entropy
- Blocking interfaces released after DRNG is fully seeded
  - After 5 requests received, forced seeding with available entropy to achieve fully seeded level
- Default applied
  - Either no specific seeding strategy compiled
  - Or specific seeding strategy is not enabled at boottime

```
[ ] Oversample entropy sources
[ ] AIS 20/31 NTG.1 seeding strategy
```

Initial Seeding Strategy II
Entropy Source Oversampling

- Initial / minimal seeding steps apply – fully seeded step changed
- 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)
- Can be used in parallel with seeding strategy III

```
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.
```

Initial Seeding Strategy III
Two Entropy Sources

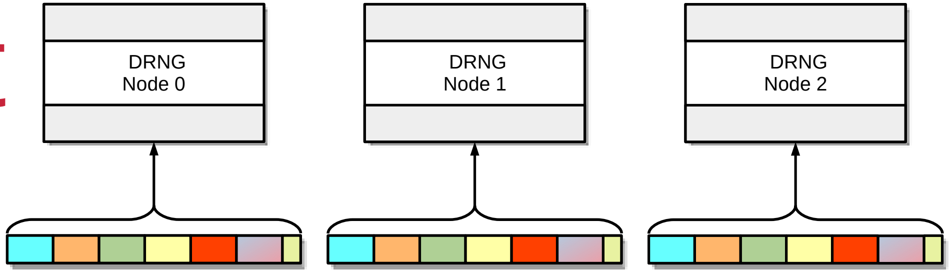
- Initial / minimal seeding steps apply unaltered – fully seeded step changed
- Compile time option
  - Function only enabled with lrng_es_mgr.ntg1=1
- Initial DRNG seeding: two entropy sources must deliver 220 bits of entropy each
- All ES data concatenated into seed buffer
- Runtime debug mode: display of all processing steps
- German AIS 20/31 compliance
  - Caveat: Applies to draft version of AIS20/31 as of September 2022
  - NTG.1: LRNG configuration ensures two entropy sources can reach at least 220 bits each
  - PTG.3 / DRG.4: LRNG can be configured to provide a PTG.3 or DRG.4
- Can be used in parallel with seeding strategy II
- German AIS 20/31 compliance (2011): access /dev/random with O_SYNC or getrandom(2) with GRND_RANDOM

```
CONFIG_LRNG_AIS2031_NTG1_SEEDING_STRATEGY:

When enabling this option, two entropy sources must
deliver 220 bits of entropy each to consider a DRNG
as fully seeded. Any two entropy sources can be used
to fulfill this requirement. If specific entropy sources
shall not be capable of contributing to this seeding
strategy, the respective entropy source must be configured
to provide less than 220 bits of entropy.

The strategy is consistent with the requirements for
NTG.1 compliance in German AIS 20/31 and is only enforced
with lrng_pool.ntg1=1.
```

DRNG Management

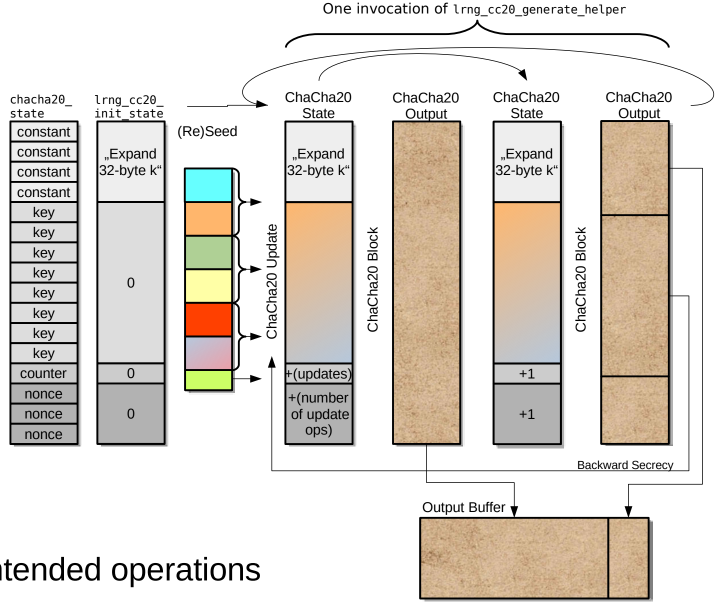


- One DRNG per NUMA node
- Hash contexts NUMA-node local
- Each DRNG initializes from entropy sources
- Sequential initialization of DRNG – first is Node 0
- If DRNG on one NUMA node is not yet fully seeded → use of DRNG(Node 0)
- Each DRNG instance managed independently
- To prevent reseed storm – reseed 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

- 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

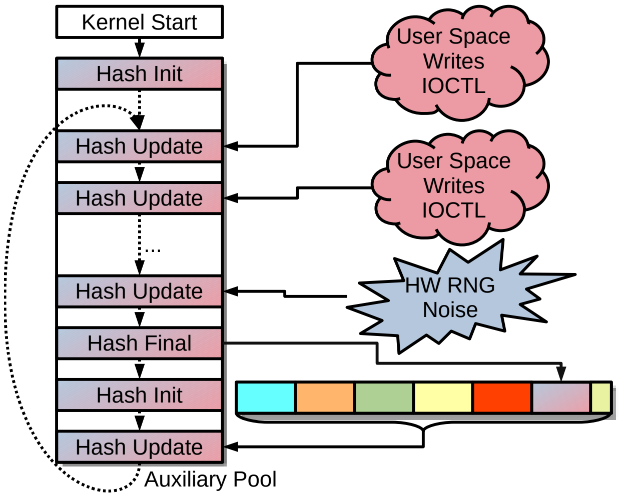
```
-*- Support DRNG runtime switching
<M>   SP800-90A support for the LRNG
<M>     Kernel Crypto API support for the LRNG
```



External Entropy Sources

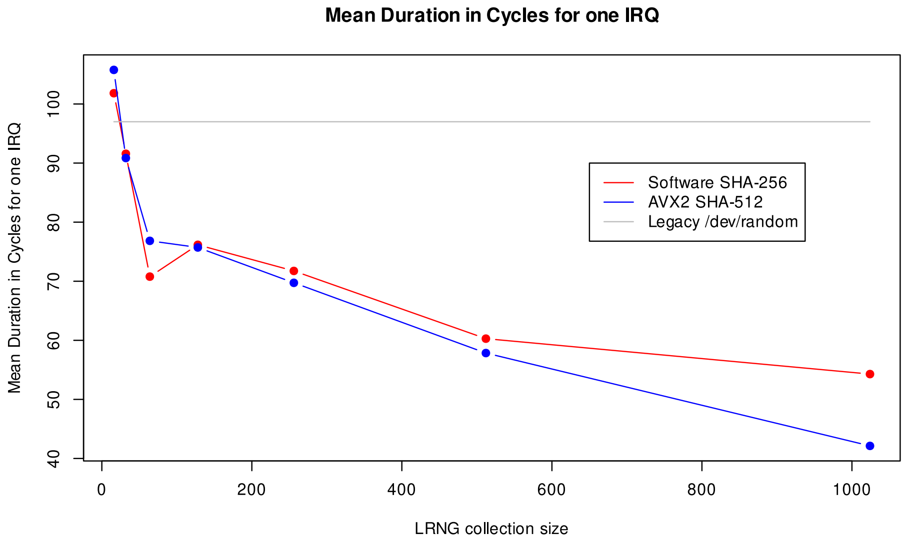
```
    *** Jitter RNG Entropy Source ***
[*] Enable Jitter RNG as LRNG Seed Source
(16)  Jitter RNG Entropy Source Entropy Rate
    *** CPU Entropy Source ***
[*] Enable CPU Entropy Source as LRNG Seed Source
(8)   CPU Entropy Source Entropy Rate
```

- Use without additional conditioning – fast source
  - Jitter RNG
  - Kernel RNG (mutually exclusive with internal IRQ ES)
  - 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 added to “auxiliary pool“ with hash update operation
  - Data “trickles in” over time
- Every entropy source has individual entropy estimate
  - Taken at face value – each ES requires its own entropy assessment



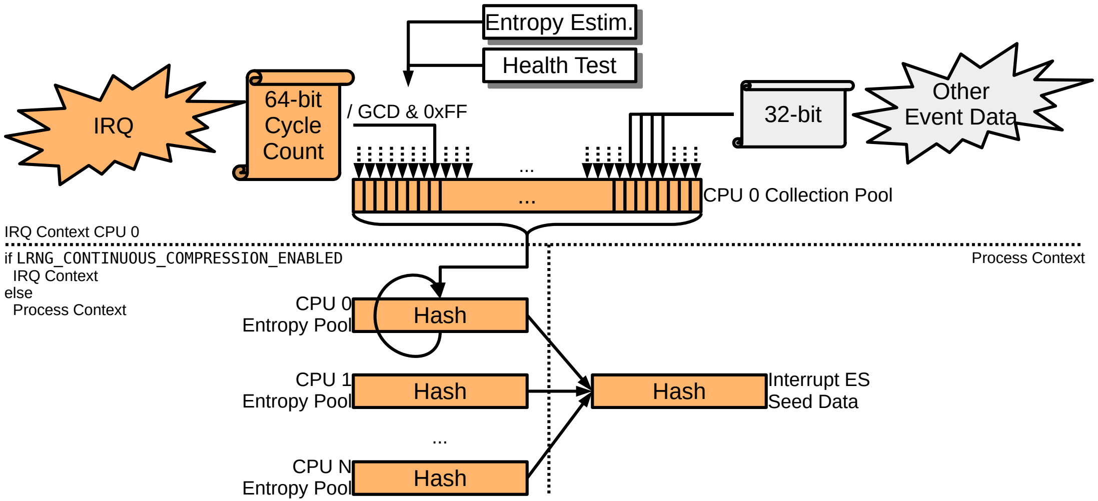
Internal ES: Interrupts

- Interrupt timing
  - All interrupts are treated as one entropy source
- Mutually exclusive with Kernel RNG ES
- Data collection executed in IRQ context
- Data compression executed partially in IRQ and process context
- Data compression is a hash update operation
- High performance: up to twice as fast as legacy /dev/random in IRQ context with LRNG_CONTINUOUS_COMPRESSION enabled
  - Even faster without continuous compression

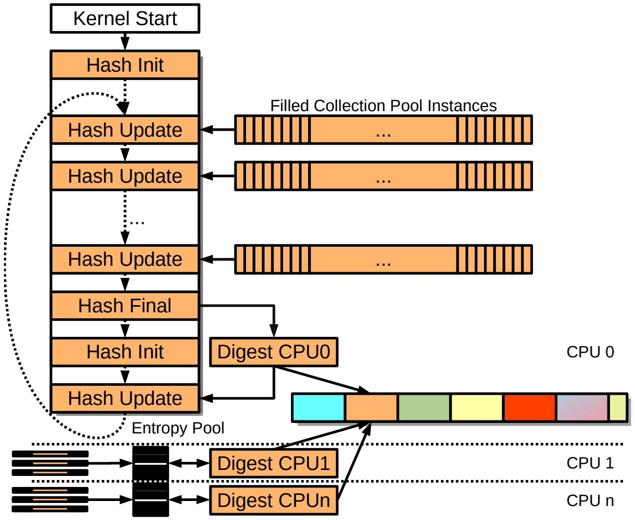



```
    *** Interrupt Entropy Source ***
[*] Enable Interrupt Entropy Source as LRNG Seed Source
      Continuous entropy compression boot time setting (Enable co
[*] Runtime-switchable continuous entropy compression
    LRNG Entropy Collection Pool Size (1024 interrupt events (def
[*] Enable interrupt entropy source online health tests
(256) Interrupt Entropy Source Entropy Rate
```

Internal ES: IRQ Data Processing



- 8 LSB of time stamp divided by GCD 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



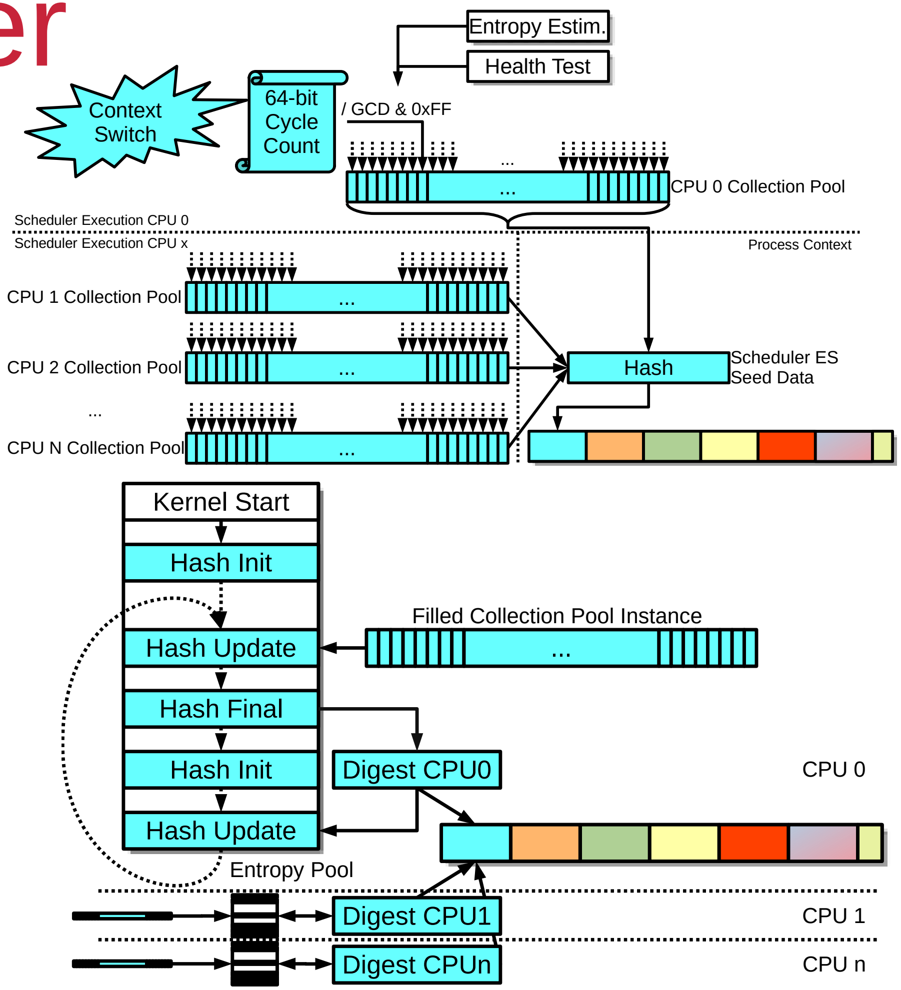
Internal ES: Scheduler Events

- Scheduler-based context switch timing
  - All context switches are treated as one entropy source
- Data collection executed in scheduler context
  - Collection: adding data into collection array → high-performance (couple of cycles)
- Data compression executed in process context during reseeding of DRNG
- Data compression is a hash operation

```
    *** Scheduler Entropy Source ***
[*] Enable Scheduer Entropy Source as LRNG Seed Source
(256) Scheduler Entropy Source Entropy Rate
```

Internal ES: Scheduler Data Processing

- 8 LSB of time stamp divided by GCD concatenated into per-CPU collection pool
  - Entropy estimate
  - Health test
- When array full → overwriting of oldest value
- 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 ES
  - Raw auxiliary IRQ data
  - Raw unprocessed entropy time stamps for Scheduler ES
  - Performance data for LRNG's IRQ handler
  - Performance data for LRNG's Scheduler handler
- Hash testing interface for built-in SHA-256
- Full SP800-90B assessment documentation
- Raw entropy collection and analysis tools provided

Test System	Entropy of 1,000,000 Traces	Sufficient Entropy
ARMv7 rev 5	1.9344	Y
ARMv7 rev 5 (Freescale i.MX53)[22]	7.07088	Y
ARMv7 rev 5 (Freescale i.MX6 Ultralite)[23]	6.638399	Y
ARM 64 bit AppliedMicro X-Gene Mustang Board	5.599128	Y
Intel Atom Z530 – using GUI	3.38584	Y
Intel i7 7500U Skylake - 64-bit KVM environment	3.452064	Y
Intel i7 8565U Whiskey Lake – 64-bit KVM environment	7.400136	Y
Intel i7 8565U Whiskey Lake – 32-bit KVM environment	7.405704	Y
Intel i7 8565U Whiskey Lake	6.871	Y
Intel Xeon Gold 6234	4.434168	Y
IBM POWER 8 LE 8286-42A	6.830712	Y
IBM POWER 7 BE 8202-E4C	4.233912	Y
IBM System Z z13 (machine 2964)	4.366368	Y
IBM System Z z15 (machine 8561)	5.691832	Y
MIPS Atheros AR7241 rev 1[24]	7.157064	Y
MIPS Lantiq 34Kc V5.6[25]	7.032740	Y
Qualcomm IPQ4019 ARMv7[26]	6.638405	Y
SiFive HiFive Unmatched RISC-V U74	2.387470	Y

```
      *** Interrupt Entropy Source Test Interfaces ***
[*]   Interface to obtain raw unprocessed IRQ noise source data
[*]   Entropy test interface to Jiffies of IRQ noise source
[*]   Entropy test interface to IRQ number noise source
[*]   Entropy test interface to IRQ flags noise source
[*]   Entropy test interface to RETIP value of IRQ noise source
[*]   Entropy test interface to IRQ register value noise source
[*]   Test interface to LRNG raw entropy IRQ storage array
[*]   LRNG interrupt entropy source performance monitor
      *** Scheduler Entropy Source Test Interfaces ***
[*]   Interface to obtain raw unprocessed scheduler noise source
[*]   Entropy test interface to PID value
[*]   Entropy test interface to task start time value
[*]   Entropy test interface to task context switch numbers
[*]   LRNG scheduler entropy source performance monitor
      *** Auxiliary Test Interfaces ***
[*]   Enable LRNG ACVT Hash interface
[*]   Enable runtime configuration of entropy sources
[*]   Enable runtime configuration of max reseed threshold
[ ]   Force CPU ES compression operation
```

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
- Provides SP800-90B compliance of internal ES

```
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.
```

General Testing

- Automated regression test suite covering the different options of LRNG
- Locking torture test of loading/unloading DRNG extensions under full load
- Applied kernel framework tests
  - KASAN
  - UBSAN
  - Lockdep
  - Memory leak detector
  - Sparse
- Performance tests of DRNG
- Syscall validation testing
- Test of LRNG behavior in atomic contexts

```
Executing test with kernel command line fips=1 lrng_jent.jitterrng=256 lrng_arch
random.archrandom=256
[PASSED] Jitter RNG: Jitter RNG working on system
[PASSED] Jitter RNG: used for seeding
Executing test with kernel command line lrng_jent.jitterrng=256 lrng_archrandom.
archrandom=256
[PASSED] Jitter RNG: Jitter RNG working on system
[PASSED] Jitter RNG: used for seeding
Executing test with kernel command line lrng_pool.ntg1=1 lrng_jent.jitterrng=256
 lrng_archrandom.archrandom=256
[PASSED] Jitter RNG: Jitter RNG working on system
[PASSED] Jitter RNG: used for seeding
Executing test with kernel command line fips=1 lrng_pool.ntg1=1 lrng_jent.jitter
rng=256 lrng_archrandom.archrandom=256
[PASSED] Jitter RNG: Jitter RNG working on system
[PASSED] Jitter RNG: used for seeding
[PASSED] no failures
========= Testing ended Do 10. Jun 11:30:27 CEST 2021 ================
========= Testing started Do 10. Jun 11:30:27 CEST 2021 ================
Executing test with kernel command line fips=1 lrng_jent.jitterrng=256 lrng_arch
random.archrandom=256
[PASSED] Atomic: LRNG executing in atomic contexts
Executing test with kernel command line lrng_jent.jitterrng=256 lrng_archrandom.
archrandom=256
[PASSED] Atomic: LRNG executing in atomic contexts
Executing test with kernel command line lrng_pool.ntg1=1 lrng_jent.jitterrng=256
 lrng_archrandom.archrandom=256
[PASSED] Atomic: LRNG executing in atomic contexts
Executing test with kernel command line fips=1 lrng_pool.ntg1=1 lrng_jent.jitter
rng=256 lrng_archrandom.archrandom=256
[PASSED] Atomic: LRNG executing in atomic contexts
[PASSED] no failures
========= Testing ended Do 10. Jun 11:30:44 CEST 2021 ================
[PASSED] ALL TESTS PASSED
[PASSED] no failures
========= Testing ended Do 10. Jun 11:30:44 CEST 2021 ================
```

DRNG Type	Cipher	Cipher Impl.	Read Size	Performance
HMAC DRBG	SHA-512	C	64 bytes	13.8 MB/s
HMAC DRBG	SHA-512	AVX2	16 bytes	4.7 MB/s
HMAC DRBG	SHA-512	AVX2	32 bytes	11.6 MB/s
HMAC DRBG	SHA-512	AVX2	64 bytes	23.3 MB/s
HMAC DRBG	SHA-512	AVX2	128 bytes	38.3 MB/s
HMAC DRBG	SHA-512	AVX2	4096 bytes	92.1 MB/s
Hash DRBG	SHA-512	C	64 bytes	27.9 MB/s
Hash DRBG	SHA-512	AVX2	16 bytes	13.1 MB/s
Hash DRBG	SHA-512	AVX2	32 bytes	25.9 MB/s
Hash DRBG	SHA-512	AVX2	64 bytes	51.1 MB/s
Hash DRBG	SHA-512	AVX2	128 bytes	83.3 MB/s
Hash DRBG	SHA-512	AVX2	4096 bytes	217.8 MB/s
CTR DRBG	AES-256	C	16 bytes	15.4 MB/s
CTR DRBG	AES-256	AES-NI	16 bytes	24.4 MB/s
CTR DRBG	AES-256	AES-NI	32 bytes	49.3 MB/s
CTR DRBG	AES-256	AES-NI	64 bytes	96.2 MB/s
CTR DRBG	AES-256	AES-NI	128 bytes	177.1 MB/s
CTR DRBG	AES-256	AES-NI	4096 bytes	1.247 GB/s
ChaCha20	ChaCha20	C	16 bytes	42.0 MB/s
ChaCha20	ChaCha20	C	32 bytes	84.5 MB/s
ChaCha20	ChaCha20	C	64 bytes	131.0 MB/s
ChaCha20	ChaCha20	C	128 bytes	194.7 MB/s
ChaCha20	ChaCha20	C	4096 bytes	550.3 MB/s
Legacy /dev/random	SHA-1	C	10 bytes	12.9 MB/s
Legacy /dev/random	ChaCha20	C	16 bytes	29.2 MB/s
Legacy /dev/random	ChaCha20	C	32 bytes	58.6 MB/s
Legacy /dev/random	ChaCha20	C	64 bytes	80.0 MB/s
Legacy /dev/random	ChaCha20	C	128 bytes	118.7 MB/s
Legacy /dev/random	ChaCha20	C	4096 bytes	220.2 MB/s

LRNG - Resources

- Code / Tests / Documentation: https://github.com/smuellerDD/lrng
- Testing conducted on
  - Intel x86, AMD, ARM, MIPS, POWER LE / BE, IBM Z, RISC-V
  - Embedded systems and Big Iron
  - Large NUMA systems with up to 160 CPUs,
  - 8 nodes
- Backport patches available
  - LTS: 5.15, 5.10, 5.4, 4.19, 4.14, 4.9

```
$ cat /proc/lrng_type
DRNG name: drbg_nopr_ctr_aes256
LRNG security strength in bits: 256
number of DRNG instances: 8
Standards compliance: SP800-90C
Entropy Sources: IRQ Scheduler JitterRNG CPU Auxiliary
LRNG minimally seeded: true
LRNG fully seeded: true
Auxiliary ES properties:
 Hash for operating entropy pool: sha512
IRQ ES properties (internal ES 0):
 Hash for operating entropy pool: sha512
 per-CPU interrupt collection size: 8192
 Standards compliance: SP800-90B
 High-resolution timer: true
 Continuous compression: true
Scheduler ES properties (internal ES 1):
 Hash for operating entropy pool: sha512
 per-CPU scheduler event collection size: 8192
 Standards compliance: SP800-90B
 High-resolution timer: true
JitterRNG ES properties:
 Enabled: true
CPU ES properties:
 Hash for compressing data: N/A
 Data multiplier: 1
```