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
- 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
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 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
  – /dev/random
  – getrandom() system call
  – get_random_bytes_full in-kernel API after being triggered with
    add_random_ready_callback or after rng_is_initialized returns true
• 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
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
- Default applied
  - Either no specific seeding strategy compiled
  - Or specific seeding strategy is not enabled at boottime
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 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.
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
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 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 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**
- **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
Internal ES 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 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
- Full SP800-90B assessment documentation
- Raw entropy collection and analysis tools provided

<table>
<thead>
<tr>
<th>Test System</th>
<th>Entropy of 1,000,000 Traces</th>
<th>Sufficient Entropy</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMv7 rev 5</td>
<td>1.9344</td>
<td>Y</td>
</tr>
<tr>
<td>ARMv7 rev 5 (Freescale i.MX53)</td>
<td>7.07688</td>
<td>Y</td>
</tr>
<tr>
<td>ARMv7 rev 5 (Freescale i.MX6 UltraLite)</td>
<td>6.638399</td>
<td>Y</td>
</tr>
<tr>
<td>ARM 64 bit AppliedMicro X-Gen Mustang Board</td>
<td>5.599128</td>
<td>Y</td>
</tr>
<tr>
<td>Intel Atom Z530 – using GUI</td>
<td>3.38554</td>
<td>Y</td>
</tr>
<tr>
<td>Intel i7 7500U Skylake – 64-bit KVM environment</td>
<td>3.452064</td>
<td>Y</td>
</tr>
<tr>
<td>Intel i7 8650U Whiskey Lake – 64-bit KVM environment</td>
<td>7.400136</td>
<td>Y</td>
</tr>
<tr>
<td>Intel i7 8650U Whiskey Lake – 32-bit KVM environment</td>
<td>7.405704</td>
<td>Y</td>
</tr>
<tr>
<td>Intel i7 8650U Whiskey Lake</td>
<td>6.871</td>
<td>Y</td>
</tr>
<tr>
<td>Intel Xeon Gold 6234</td>
<td>4.434168</td>
<td>Y</td>
</tr>
<tr>
<td>IBM POWER 8 LE 8286-42A</td>
<td>6.830712</td>
<td>Y</td>
</tr>
<tr>
<td>IBM POWER 7 BE 8202-E4C</td>
<td>4.233912</td>
<td>Y</td>
</tr>
<tr>
<td>IBM System Z z13 (machine 2964)</td>
<td>4.366368</td>
<td>Y</td>
</tr>
<tr>
<td>IBM System Z z15 (machine 8561)</td>
<td>5.691832</td>
<td>Y</td>
</tr>
<tr>
<td>MIPS Atheros AR7241 rev 2(3)</td>
<td>7.159064</td>
<td>Y</td>
</tr>
<tr>
<td>MIPS Lantiq 34Kc V5.6(36)</td>
<td>7.032740</td>
<td>Y</td>
</tr>
<tr>
<td>Qualcomm IPQ8019 ARMv7(32)</td>
<td>6.638405</td>
<td>Y</td>
</tr>
<tr>
<td>SiFive HiFive Unmatched RISC-V U74</td>
<td>2.387470</td>
<td>Y</td>
</tr>
</tbody>
</table>
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 $\neq 0$
• 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
- 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, RISC-V
  – Embedded systems and Big Iron
  – Large NUMA systems with up to 160 CPUs, 8 nodes

• 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?
Backup: Legacy /dev/random
Shortcomings

- Internal noise source
  - No startup / runtime tests
  - Mix of multiple but dependent noise sources: HID / Block device sources are a “derivative” of IRQ source
  - Double accounting of entropy
- Seed data via writes/IOCTL to /dev/random or added kernel-internally does not immediately find its way to the DRNG
- Processing of multiple entropy sources
  - Intermixing of data collection from different entropy sources
  - Code intermixes entropy collection, conditioning and random number generation
- Data processing
  - No power-on self tests
  - Various non-cryptographic conditionings: 3 different LFSRs, no standards-conformant SHA-1 with folding of output
  - Missing test interfaces – different processing steps hard to test and analyze
  - Heuristic entropy estimation based on Jiffies which hardly delivers entropy – coincidental underestimation of entropy
  - IRQ entropy estimation massively underestimates available entropy
- Performance: IRQ-context and ChaCha20 performance challenges
- Significant fragmentation in user base due to inflexible structure
Backup: Shortcomings of Wiring

**DRBG Up**

- Some Linux users wire up crypto API DRBG to /dev/random user space interfaces
  - Shortcomings of legacy /dev/random still applies
  - Use of additional entropy source
  - The main entropy source of Linux is credited with zero bits of entropy
- Performance bottleneck: only one DRBG instance per NUMA node
- Seed data to via writes/IOCTL to /dev/random or added kernel-internally does not find its way to the DRBG in a decent time as DRBG reseed period is long
- DRBG is only available to user space: get_random_bytes() will go to the random.c code
- SP800-90C non-compliance: Although the Jitter RNG and the random.c data are concatenated, there is no entropy oversampling of data as required by 90C
- German NTG.1 non-compliance: The solution is not compliant to the German NTG.1 requirements
Backup: TODOs If LRNG Accepted

- Move lrng_aux.c to drivers/char/ and remove identical code from random.c
- Add documentation to kernel tree
- Make SHA256 generally available outside of crypto API – remove SHA-1 support from lrng_chacha20.c
- Simplify kernel crypto API RNG logic
  - Crypto API should only provide the crypto primitive
  - Any seeding logic / handling of entropy provided by LRNG – remove DRBG logic to seed itself from crypto/drbg.c
  - Remove the FIPS check in DRBG
- Move Jitter RNG ES from kernel crypto API to LRNG tree?