Entropy Source and DRNG Manager

Stephan Müller <smueller@chronox.de>

May 10, 2022

Abstract

Almost all parts of cryptography rest on a random number generator which produces data that are indistinguishable from a perfect generator. This also implies that the random number generator is seeded with sufficient entropy and the entropy is maintained during processing. Also a reseed at proper intervals is important to maintain the security strength. Finally, different users and use cases have different requirements regarding seeding and the deterministic processing. All these tasks are non-trivial in nature. The Entropy Source and DRNG Manager (ESDM) provides a flexible and extensible framework. The ESDM also provides a set of well-known interfaces: an API and ABI compliant drop-in replacement for Linux /dev/random, /dev/urandom and getrandom(2) system call.

Contents

List of Figures

List of Tables

1 Introduction

The ESDM originated from the Linux Random Number Generator (LRNG) patch series. Considering the limitations of maintaining a proper entropy source management inside the confinements of a kernel and the fact that such actions do not need the privileges of executing inside the kernel, the LRNG patch series was converted into user space to form the ESDM. Even in user space, the ESDM can execute completely unprivileged and thus adds to the overall system security to isolate the operation.

Before discussing the design of the ESDM, the goals of the ESDM design are enumerated:

1. The ESDM manages the proper seeding and reseeding of DRNGs. In addition, it provides internal entropy sources which the ESDM fully controls as well as interfaces to obtain data from external entropy sources.

2. The ESDM provides a full API and ABI drop-in replacement for the Linux device files of /dev/random and /dev/urandom as well as the Linux getrandom(2) and getentropy(2) system calls. All code related to these interfaces is solely implemented in user space and executed with limited privileges.

3. The ESDM is modular consisting of a central server which is implemented with minimal dependencies. This allows the ESDM to execute on different operating systems. The aforementioned Linux interfaces are implemented with separate processes and libraries and therefore their absence do not affect the ESDM operation on operating systems other than Linux.

4. All user-visible behavior implemented by the legacy /dev/random – such as the per-NUMA-node DRNG instances are provided by the ESDM as well.

5. The ESDM must not use locking in hot code paths to limit the impact on massively parallel systems.

6. The ESDM must handle modern computing environments without a degradation of entropy. The ESDM therefore must work in virtualized environments, with SSDs, on systems without HIDs or block devices and so forth.

7. The ESDM must provide a design that allows quantitative testing of the entropy behavior.

8. The ESDM must use testable and widely accepted cryptography for conditioning.

9. The ESDM must allow the use of cipher implementations backed by architecture specific optimized assembler code or even hardware accelerators. This provides the potential for lowering the CPU costs when generating random numbers – less power is required for the operation and battery time is conserved.

10. The ESDM must separate the cryptographic processing from the entropy source maintenance to allow a replacement of these components.

11. The ESDM shall offer flexible configurations allowing vendors to apply the settings applicable to their environment.

1.1 Properties Offered by the ESDM

Apart from the fact that a user does not need to manage the DRNG and its seeding status, the ESDM provides the following properties making the ESDM a contemporary and future-proof entropy source and DRNG management framework:

- Pure user space implementation:
  - The entire ESDM as well as the Linux interfaces are fully implemented in user space.
  - An additional but optional entropy source is provided for resource-constrained systems: the scheduler-based entropy source. It, however, uses a kernel extension as it hooks into the scheduler.
- Sole use of crypto for data processing:
  - Exclusive use of a hash operation for conditioning entropy data with a clear mathematical description as given section 2.2 – non-cryptographic operations like LFSR are not used.
  - The ESDM uses only properly defined and implemented cryptographic algorithms unlike the use of the SHA-1 transformation in the legacy Linux /dev/random implementation that is not compliant with SHA-1 as defined in FIPS 180-4.
  - Hash operations use on-stack hash instances to benefit large parallel systems.
  - ESDM uses limited number of data post-processing steps as documented in section 2.2 compared to the large variation of different post-processing steps in the legacy /dev/random implementation that have no apparent mathematical description.
- Performance
  - High-performance implementation of scheduler entropy source requiring only a few cycles to collect entropy.
- Testing
  - Availability of run-time health tests of the raw unconditioned entropy source data of the scheduler-based entropy source to identify degradation of the available entropy. Such health tests are important today due to virtual machine monitors reducing the resolution of or disabling the high-resolution timer.
  - Heuristic entropy estimation for the scheduler-based entropy source is based on quantitative measurements and analysis following SP800-90B.
  - Power-on self tests for critical deterministic components (DRNG, hash implementation, and entropy collection logic).

  - Availability of test interfaces for all operational stages of the ESDM including boot-time raw entropy event data sampling as outlined in section 2.14.
  - The ESDM offers a test interface to validate the used software hash implementation and in particular that the ESDM invokes the hash correctly, allowing a NIST ACVP-compliant test cycle – see section 2.14.
  - Availability of stress testing covering the different code paths for data and mechanism (de)allocations and code paths covered with locks.
  - Availability of fully automated regression testing covering different ESDM functions in different configurations allow an unattended functional verification testing.

- Entropy collection of the internal interrupt and scheduler-based entropy source
  - The ESDM scheduler-based entropy source is fully compliant to SP800-90B requirements and is shipped with a full SP800-90B assessment and all required test tools.
  - Full entropy assessment and description is provided.
  - The ESDM provides a configuration to be compliant to SP800-90C following RBG2(NP) as well as RBG2(P) construction methods.

- Configurable
  - ESDM build-time configuration allows a flexible adoption of the ESDM to different use cases. Non-compiled additional code is folded into no-ops.
  - Configurable seeding strategies are provided following different concepts.

1.2 Document Structure

This paper covers the following topics in the subsequent chapters:

- The design of the ESDM is documented in chapter 2. The design discussion references to the actual implementation whose source code is publicly available.

- The statistical testing of the internal scheduler entropy source including the SP800-90B compliance assessment is provided in chapter 3.

- The discussion of various configurations offered by the ESDM is given in chapter 4.

- The various appendices cover miscellaneous topics supporting the general description.

2 ESDM Design

The ESDM can be characterized with figure 2.1 which provides a big picture of the ESDM processing and components.

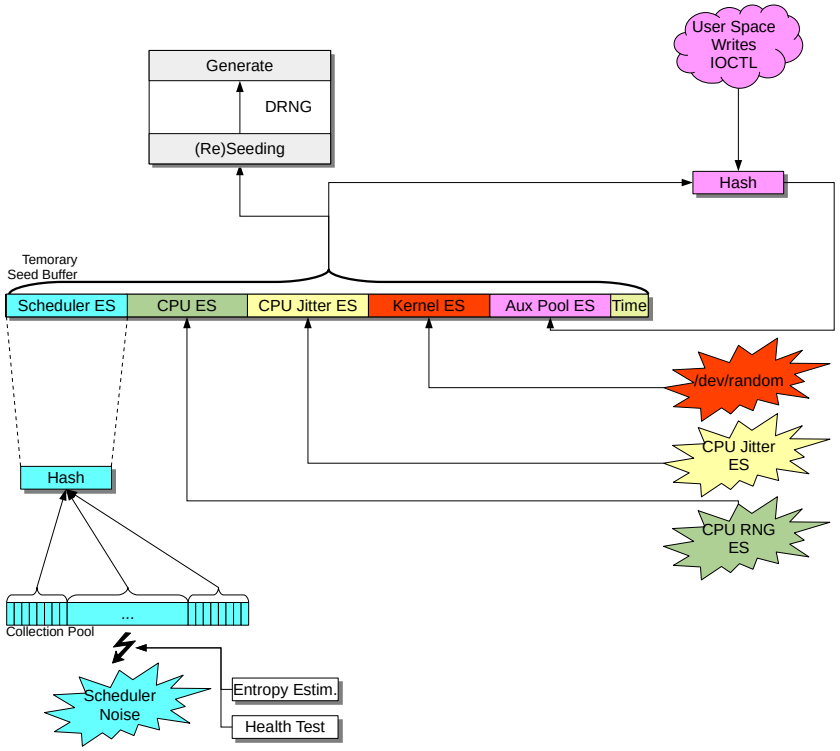

Figure 2.1: ESDM Big Picture

The colors indicate the different entropy sources managed by the ESDM.

The ESDM introduces the concept of slow and fast entropy sources. Fast entropy sources provide entropy at the time of request. A slow entropy source collects data over time into an entropy pool.

The entropy source that is under full control of the ESDM, also called the internal entropy source, comprises of the:

- The indigo marked parts refer to the scheduler entropy source that feeds per-CPU collection pools. These collection pools are hashed when seeding of the DRNG is to be performed.

The following external entropy sources are present. These entropy sources are expected to be fully self-contained. The ESDM only requests data from them and expects that the entropy estimate provided with the data is correct:

- The auxiliary pool collects data from external entropy sources which deliver data at times not controllable by the ESDM.

- The CPU entropy source obtains data from a potentially existing source in the CPU like RDSEED on Intel CPUs.

- The Jitter RNG entropy source is another external entropy source.
- The random number generator provided with the `getrandom` system call or the /dev/random device file can be enabled at compile time to serve as an entropy source.

The ESDM treats all external and internal entropy sources equally. It can handle the situations where one or more entropy sources returns little or no entropy.

The the scheduler entropy source is assessed in chapter 3 which provides its complete entropy assessment. All other entropy sources are expected to provide their own entropy assessment supporting the claim of the supplied entropy that is credited by the ESDM for these sources. The ESDM treats these additional entropy sources as black boxes and take their claimed entropy rate at face value. The ESDM, however, guarantees that all entropy sources are processed in compliance with defined standards.

The different colors used in figure 2.1 depict the different entropy sources mentioned before.

The ESDM offers various interfaces to obtain random numbers as depicted in figure 2.2.

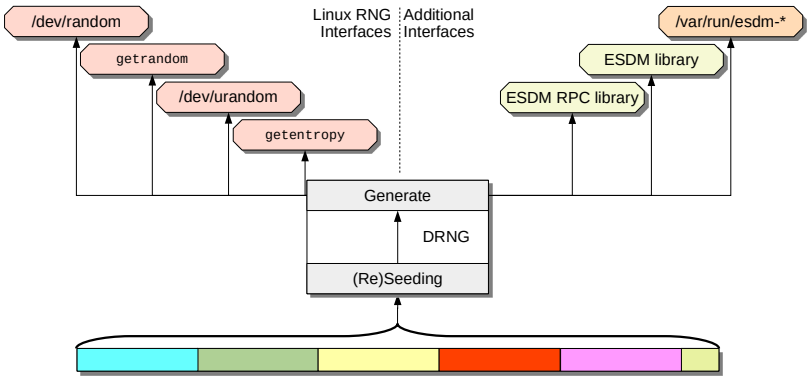

Figure 2.2: ESDM Interfaces To Obtain Random Numbers

Based on this figure, the following types of interfaces are available which are marked with the different colors in figure 2.2. Each type of interface can be selectively enabled and disabled at compile time:

- Linux RNG interfaces: The common interfaces found from today's Linux kernels include /dev/random, /dev/urandom, the `getrandom` system call or the `getentropy` C-library wrapper. These interfaces behave identically to the Linux RNG and thus allow the ESDM to act as a drop-in replacement. When not compiling those, the ESDM can be operated in parallel with the Linux RNG and all user space software divert to use the ESDM.
- The ESDM also exports a set of Unix Domain Sockets to /var/run which can be used directly, albeit it is not advised to be used as the protocol offered by those sockets is defined to be not stable, i.e. it can change from one version to another.

- To wrap the mentioned Unix Domain Sockets, the ESDM RPC library is available which offers a stable API.
- The ESDM library provides the heart of the ESDM functionality. It could be directly used by a calling application. Yet, it is advised to use the RPC interface as it is provided by a daemon that uses the ESDM library already.

The ESDM consists of the following components:

- `libesdm.so`: The ESDM provides a library with the core of the ESDM. The DRNG and entropy source managers together with all entropy sources and cryptograpphic algorithm implementations are implemented with this library. This library is wrapped by the `esdm-server` listed below. The API for using the library is exported by and documented with `esdm.h`.

  The library is provided in case a user wants to employ the ESDM in his projects instead of the `esdm-server`. Yet, the `esdm-server` provides a wrapping daemon to the ESDM library that is intended to be commonly used.

  In addition, the ESDM can be configured with the options specified in `esdm_config.h`. This would need to be performed by the consuming application. When using the tools below, this library can be ignored.

- `esdm-server`: The ESDM server provides the RPC server that encapsulates the ESDM with its random number generator and the entropy source management. When starting the server, the mentioned Unix Domain Sockets are created that allows clients to request services including random numbers.

  The ESDM server can either be started manually or with the provided (and installed) systemd unit file. When using systemd, start the server with 'systemctl start esdm-server'.

  A wrapper library to access the ESDM server RPC interface is provided with `libesdm_rpc_client.so`. Its API is specified and documented with `esdm_rpc_client.h`.

  The `esdm-server` is the backend to all of the following ESDM components.

  Note: The Unix domain sockets of the `esdm-server` are only visible in the respective mount namespace. If you have multiple mount namespaces, you need to start the daemon in each mount namespace or make the files otherwise available if its services shall be available there.

- `esdm-cuse-random`: The ESDM CUSE daemon creates a device file that behaves identically to /dev/random. It must be started as root. Reading, writing and IOCTLs are implemented in an ABI-compatible way.

  The ESDM CUSE daemon can either be started manually or with the provided (and installed) systemd unit file. When using systemd, start the server with 'systemctl start esdm-cuse-random'. Although the daemon creates a /dev/random device, the actual visible operation is atomic (a bind mount) for both creation and destruction of the new device file which implies that the daemon can be started and stopped at any time during runtime of the Linux OS.

Note: The bind mount is only visible in the respective mount namespace. If you have multiple mount namespaces, you need to start these daemons in each mount namespace or make the files otherwise available if their services shall be available there.

- `esdm-cuse-urandom`: Same as esdm-cuse-random but behaving like /dev/urandom.
- `libesdm-getrandom.so`: The library provides a wrapper to the `getrandom` and `getentropy` Linux C-library calls. To use the library for other consumers, use one of the following considerations:
  - Use `LD_PRELOAD="/path/to/libesdm-getentropy.so"` with the intended application.
  - Compile the application or library with the following options:
    - ≻ `LDFLAGS += -Wl,--wrap=getrandom,--wrap=getentropy`
    - ≻ `LDFLAGS += -lesdm-getrandom`

2.1 ESDM Components

The ESDM consists of the following components shown in figure 2.1:

1. The ESDM implements a DRNG. The DRNG always generates the requested amount of output. When using the SP800-90A terminology it operates without prediction resistance. The DRNG maintains a counter of how many bytes were generated since last re-seed and a timer of the elapsed time since last re-seed. If either the counter or the timer reaches a threshold, the DRNG is seeded from the entropy sources with the available entropy.

   In case the Linux kernel detects a NUMA system, one DRNG instance per NUMA node is maintained.

   Depending on the used interface to request data from the DRNG, the caller may be put to sleep until the ESDM is fully seeded:

   (a) All interfaces using the `esdm_get_entropy_bytes` function always generates data including when the ESDM is not properly seeded.

   (b) All interfaces using the `esdm_get_entropy_bytes_full` function generate data only when the ESDM is fully seeded and fully initialized.

2. The DRNG is seeded by concatenating the data from the following sources in case they are enabled at kernel compile time:

   (a) the output of the auxiliary pool,

   (a) the output of the per-CPU scheduler collection pools,

   (b) the Jitter RNG if available,

   (c) the CPU-based entropy source such as Intel RDSEED if available, and

   (d) the legacy RNG output.

The entropy estimate of the data of all entropy sources are added to form the entropy estimate of the data used to seed the DRNG with. The ESDM ensures, however, that the DRNG after seeding is at maximum the security strength of the used DRNG implementation of 256 bits.

The ESDM is designed such that none of these entropy sources can dominate the other entropy sources to provide seed data to the DRNG due to the following:

(a) During boot time, the amount of available entropy is the trigger point to (re)seed the DRNG following the explanation in the next section.

(b) At runtime, the the DRNG reseed is triggered by either the DRNG due to hitting the aforementioned thresholds or by a user space caller. The reseed is never triggered by the entropy sources.

3. To support backward secrecy, the following steps are applied:

(a) The temporary seed buffer holding the concatenation of data from all entropy sources to seed the DRNG is injected into the auxiliary pool like other data by hashing it together with the existing auxiliary pool data to form the new auxiliary pool content. The injection of the temporary seed buffer will not alter the entropy estimation of the auxiliary pool.

(b) The message digest created for each per-CPU entropy pool is inserted into the corresponding per-CPU entropy pool.

The ESDM allows the DRNG mechanism and the used hash to be changed at runtime. Per default, an SP800-90A Hash DRBG implementation along with a SHA-512 hash implementation is available. Both are standard C implementations which were tested against NIST's ACVP service.

The following subsections cover the different components of the ESDM from the bottom to the top.

2.2 ESDM Data Processing

The processing of entropic data from the different entropy source before injecting them into the DRNG is performed with the following mathematical operations. The operation $SHA()$ refers to the hash operation using the message digest implementation that is currently present, i.e.SHA-512.

2.2.1 Scheduler Entropy Sources

1. Truncation: The time stamps received by the IRQ as well as the scheduler entropy sources are truncated to 8 least significant bits (or 32 least significant bits during boot time) – note the GCD is a value calculated during initialization and is fixed thereafter which implies that the time stamp divided by the GCD is the raw entropy value: t_8 (or t_{32})

2. Concatenation: The time stamps received and truncated by the IRQ and scheduler entropy sources as well as auxiliary 32 bit words a_{32} are concatenated to fill the per-CPU collection pool that is capable of holding

1,024 8-bit words[1] - the order of the data a_{32} or t_8 present in the concatenation depends on the occurrence of events - the following formula depicts one possible order for illustration - the implementation is provided with functions `_ESDM_pcpu_array_add_u32` and `ESDM_pcpu_array_add_slot`:

$$CP = t_{8_{n-1019}}||a_{32_n}||t_{8_{n-1018}}|| \ ... \ ||t_{8_n} \tag{2.1}$$

Note: In case the continuous compression operation is disabled for the IRQ entropy source, the auxiliary 32 bit words a_{32} are discarded and are not injected into the collection pool. This approach is taken to prevent non-entropy data to potentially overwrite entropy data in the collection pool when the array wraps. The scheduler entropy source only records time stamps.

2.2.2 Scheduler Entropy Source

1. Hashing: For the scheduler entropy source, a message digest of all scheduler collection pools is calculated with the function `ESDM_sched_pool_hash`:

$$SP_n = SHA(CP_{SCHED_{CPU0}}||CP_{SCHED_{CPU1}}||...||CP_{SCHED_{CPUn}}) \tag{2.2}$$

2. Truncation: Just like the interrupt entropy source, the scheduler entropy source applies a truncation to the generated data as implemented by the function `ESDM_sched_pool_hash`:

$$S_n = MSB_{min(entropy,security\ strength)}(SP_n) \tag{2.3}$$

2.2.3 Auxiliary Entropy Pool

1. Hashing: When new data D_m is added to the auxiliary pool AP, the data is inserted into the auxiliary pool with a hash update operation - the implementation is provided with function `ESDM_pool_insert_aux`. The message digest generation is performed at the time entropy from the auxiliary pool is requested. To ensure backward secrecy, the temporary seed buffer T_{n-1} that holds among others the auxiliary pool digest from the previous generation round as depicted with equation 2.8 is concatenated with the received data:

$$AP_n = SHA(T_{n-1}||D_{m-(n-1)}||...||D_{m-1}||D_m) \tag{2.4}$$

2. Truncation: The MSB of the auxiliary pool of the size of the DRNG security strength are used for the seed buffer:

$$A_n = MSB_{min(digest\ size,security\ strength)}(AP_n) \tag{2.5}$$

[1]The ESDM collection size is compile-time configurable where 1,024 is a default value. When configuring a different value, the number of the concatenated data must be adjusted as needed. However, this modification has no impact to the illustration of the data processing.

2.2.4 CPU Entropy Source

1. Hashing: If the CPU entropy source provides less than full entropy, a message digest of the amount of data to be requested from it is calculated:

$$C_{cond} = SHA(C_1 || ... || C_m) \tag{2.6}$$

2. Truncation: If the CPU entropy source provides less than full entropy, the MSB defined by the requested number of bits (commonly equal to the security strength of the DRBG) or the applied message digest size, what ever is smaller, are obtained - the implementation is provided with function `ESDM_get_arch_data_compress` – otherwise C_n is the data obtained directly from the CPU entropy source:

$$C_n = MSB_{min(digest\,size, security\,strength)}(C_{cond}) \tag{2.7}$$

2.2.5 Temporary Seed Buffer Construction

1. Concatenation: The temporary seed buffer T used to seed the DRNG at the time n is a concatenation of one or more of the following entropy source data sets, depending on the compile-time configuration:

   (a) the auxiliary pool entropy source A,

   (a) the scheduler entropy source buffer S,

   (b) the Jitter RNG output J,

   (c) the CPU entropy source output C,

   (d) the legacy RNG entropy source output L, and

   (e) the current time t

   with the implementation is provided with function `ESDM_fill_seed_buffer`:

$$T_n = A_n || S_n || J_n || C_n || L_n || t \tag{2.8}$$

2.3 ESDM Architecture

Before going into the details of the ESDM processing, the concept underlying the ESDM shown in figure 2.1 is provided here.

The entropy derived from the slow entropy sources is collected and accumulated in the entropy pools which contain already compressed entropy data, supported by the collection pools which contain uncompressed, but only concatenated entropy data.

At the time the DRNG shall be seeded, the all entropy pools, any non-compressed data in the collection pools and the auxiliary pool are processed with a cryptographic hash function which can be chosen at runtime.

For the entropy pool, if the digest of the hash and the available entropy are larger than requested by the caller, the digest is truncated to the appropriate size. For the auxiliary pool, always 256 bits of data are returned irrespective of the entropy rate of this pool. This ensures that also data that is not credited with entropy but injected into the ESDM is used to stir the seed for the DRNG.

The DRNG always tries to seed itself with 256 bits of entropy, except during boot. In any case, if the entropy sources cannot deliver that amount, the available entropy is used and the DRNG keeps track on how much entropy it was seeded with. The entropy implied by the ESDM available in the entropy pool may be too conservative. To ensure that during boot time all available entropy from the entropy pool is transferred to the DRNG, the hash function always generates 256 data bits during boot to seed the DRNG. During boot, the DRNG is seeded as follows:

1. The DRNG is reseeded from the entropy sources if all entropy sources collectively have at least 32 bits of entropy available. The goal of this step is to ensure that the DRNG receive some initial entropy as early as possible.

2. The DRNG is reseeded from the entropy sources if all entropy sources collectively have at least 128 bits of entropy available.

3. The DRNG is reseeded from the entropy sources if all entropy sources collectively have at least 256 bits of entropy available.

At the time of the reseeding steps, the DRNG requests as much entropy as is available in order to skip certain steps and reach the seeding level of 256 bits. This may imply that one or more of the aforementioned steps are skipped.

In all listed steps, the DRNG is (re)seeded with a number of random bytes from the entropy pool that is at most the amount of entropy present in the entropy pool. This means that when the entropy pool contains 128 or more bits of entropy, the DRNG is seeded with that amount of entropy as well.

Before the DRNG is seeded with 256 bits of entropy in step 3, requests of random data from the blocking interfaces are not processed.

At runtime the DRNG operates as deterministic random number generator with the following properties:

- The maximum number of random bytes that can be generated with one DRNG generate operation is limited to 4096 bytes. When longer random numbers are requested, multiple DRNG generate operations are performed. The ChaCha20 DRNG as well as the SP800-90A DRBGs implement an update of their state after completing a generate request for backward secrecy.

- The DRNG is reseeded with whatever entropy is available, but at least 128 bits – in the worst case where no additional entropy can be provided by the entropy sources, the DRNG is not re-seeded and continues its operation to try to reseed again after again the expiry of one of these thresholds:

  – If the last reseeding of the DRNG is more than 600 seconds ago[2], or

  – 2^{20} DRNG generate operations are performed, whatever comes first, or

[2]Note, this value will not empty the entropy pool even on a completely quiet system. Testing of the ESDM was performed on a KVM without fast entropy sources and with a minimal user space, where only the SSH daemon was running, During the testing, no operation was performed by a user. Yet, the system collected more than 256 bits of entropy from the interrupt entropy source within that time frame, satisfying the DRNG reseed requirement.

– the DRNG is forced to reseed before the next generation of random numbers if data has been injected into the ESDM by writing data into /dev/random or /dev/urandom.

The chosen values prevent high-volume requests from user space to cause frequent reseeding operations which drag down the performance of the DRNG[3][4].

- If the DRNG was not reseeded for the last 2^{30} DRNG generate operations – i.e. the reseeding requests discussed in the previous bullets were unsuccessful – the DRNG reverts back to an unseeded state. This applies that the DRNG will not produce random numbers when accessed via the blocking interfaces. In this case, the DRNG behaves like during boot time.

With the automatic reseeding after 600 seconds, the ESDM is triggered to reseed itself before the first request after a suspend that put the hardware to sleep for longer than 600 seconds.

2.3.1 Minimally Versus Fully Seeded Level

The ESDM's DRNG is reseeded when the first 128 bits / 256 bits of entropy are received during boot as indicated above. The 128 bits level defines that that the DRNG is considered "minimally" seeded whereas reaching the 256 bits level is defined as the DRNG is "fully" seeded.

Both seed levels have the following implications:

- Upon reaching the minimally seeded level, the kernel-space callers waiting for a seeded DRNG via the API calls of either `esdm_get_random_bytes_min` is woken up.

- When reaching the fully seeded level, the user-space callers waiting for a fully seeded DRNG via the `getrandom` system call or /dev/random are woken up. Using the ESDM API of `esdm_get_random_bytes_full`, the caller is waiting synchronously until the fully seeded level is reached.

Note, the initial seeding level with 32 bits is implemented to ensure that early boot requests are served with random numbers having some entropy, i.e. the DRNG has some meaningful level of entropy for non-cryptographic use cases as soon as possible.

[3] Considering that the maximum request size is 4096 bytes defined by `ESDM_DRNG_MAX_REQSIZE` (i.e. each request is segmented into 4096 byte chunks) and at most 2^{20} requests defined by `ESDM_DRNG_RESEED_THRESH` can be made before a forced reseed takes place, at most $4096 \cdot 2^{20} = 4,294,967,296$ bytes can be obtained from the DRNG without a reseed operation.

[4] After boot, the ChaCha20 DRNG state is also used for the atomic DRNG state. Although both DRNGs are controlled by separate and isolated objects, the DRNG state is identical. As the `ESDM_DRNG_RESEED_THRESH` is enforced local to each DRNG object, the theoretical maximum number of random bytes the ChaCha20 DRNG state could generate before a forced reseed is twice the amount listed before – once for the DRNG object and once for the atomic DRNG object.

2.3.2 NUMA Systems

To prevent bottlenecks in large systems, the DRNG will be instantiated once for each NUMA node. The instantiations of the DRNGs happen all at the same time when the ESDM is initialized.

The question now arises how are the different DRNGs seeded without re-using entropy or relying on random numbers from a yet insufficiently seeded ESDM. The ESDM seeds the DRNGs sequentially starting with the one for NUMA node zero – the DRNG for NUMA node zero is seeded with the approach of 32/128/256 bits of entropy stepping discussed above. Once the DRNG for NUMA node 0 is seeded with 256 bits of entropy, the ESDM will seed the DRNG of node one when having again 256 bits of entropy available. This is followed by seeding the DRNG of node two after having again collected 256 bits of entropy, and so on. Figure 2.3 illustrates the seeding strategy showing that each DRNG instance is freshly seeded with a separate seed buffer.

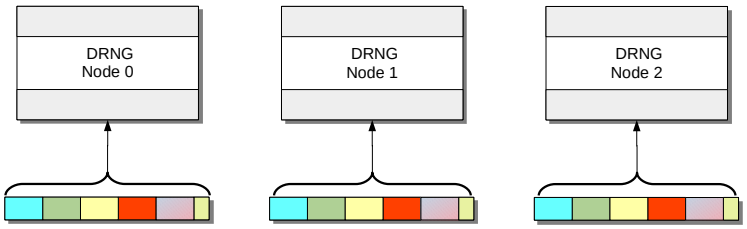

Figure 2.3: DRNG Instances on NUMA systems with seeding strategy

When producing random numbers, the ESDM tries to obtain the random numbers from the NUMA node-local DRNG. If that DRNG is not yet seeded, it falls back to using the DRNG for node zero.

Note, to prevent draining the entropy pool on quiet systems, the time-based reseed trigger, which is 600 seconds per default, will be increased by 100 seconds for each activated NUMA node beyond node zero. Still, the administrator is able to change the default value at runtime.

2.3.3 Flexible Design

Albeit the preceding sections look like the DRNG and the management logic are highly interrelated, the ESDM code allows for an easy replacement of the DRNG with another deterministic random number generator. This flexible design allowed the implementation of the ChaCha20 DRNG if the SP800-90A DRBG is not desired.

To implement another DRNG, all functions in `struct esdm_crypto_cb` in "esdm_crypto.h" must be implemented. These functions cover the allocation/deallocation of the DRNG and the entropy pool read hash as well as their usage. This function pointer data structure also holds the callbacks to the hash used to process the entropy pools.

The implementations can be changed at runtime. The default implementation is the SP800-90A Hash-DRBG using a software-implementation of the used SHA-512 message digest for accessing the entropy pools.

In addition, the ESDM allows the addition of new entropy sources. The see for other examples the ‘addon‘ directory in the source tree.

2.4 ESDM Data Structures

The ESDM uses the following main data structures:

- The scheduler entropy source defines per-CPU collection pools. The entropy pools are not maintained as compression of scheduler-based entropy sources is only performed when the DRNG shall be (re)seeded. The entire scheduler-based entropy source operates lock-less.

- The deterministic random number generator data structure for the DRNG holds the reference to the DRNG instance and the hash instance and associated meta data needed for its operation. The DRNG is managed with a separate data structure. When using the DRNG, a full read/write lock is used to guard (a) against replacement of the DRNG reference while operating on the DRNG state, and (b) to read/write the DRNG state. Contrarily when using the hash, only a read-lock is used to guard against the replacement of the hash reference. This implies that the hash state is kept on the stack of the calling application.

2.5 Scheduler Events - ESDM-internal Entropy Source

The ESDM hooks into the scheduling operation of the Linux kernel which is triggered every time a context switch is performed as initiated by the scheduler. The ESDM handling of a scheduling event is depicted with figure 2.4.

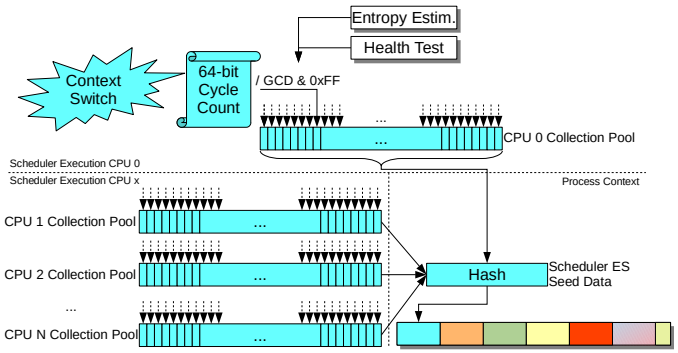

Figure 2.4: Scheduler-event Processing

When a context switch occurs, the ESDM callback is invoked which obtains a high-resolution time stamp that is concatenated into the CPU-local collection pool. When the CPU-local collection pool is full, the oldest entries are overwritten by the latest time stamp.

At the time the DRNG shall be (re)seeded, a message digest of all scheduler per-CPU collection pools is calculated. This calculation is performed in the process context of the DRNG and its caller. Therefore, the hashing operation is not performed as part of the scheduling operation. The message digest is

truncated to the entropy available in all scheduler per-CPU collection pools or the requested amount of entropy, whatever is smaller. Note, the requested amount of entropy is always smaller or equal to the size of the message digest of the used hashing algorithm.

As a side note, the scheduler entropy source potentially has some relationship with the IRQ entropy source because an IRQ ma trigger a scheduler event (e.g. with the flag `TIF_NEED_RESCHED`). Therefore, both entropy sources cannot be used at the same time and credit entropy to both. It is permissible to use both, but only one is credited with entropy.

2.5.1 Entropy Amount of Interrupts

The question now arises, how much entropy is generated with the scheduler entropy source. The current implementation implicitly assumes one bit of entropy per time stamp obtained for one scheduling event.

With the kernel compile time parameter of `ESDM_SCHED_ENTROPY_RATE` the number of interrupts that must be collected to obtain 256 bits of entropy can be specified. This value is forced by the ESDM to be at least the aforementioned limit, i.e. 256 interrupts.

The entropy of high-resolution time stamps is provided with the fast-moving least significant bits of a time stamp which is supported by the quantitative measurements shown in section 3. Although only one bit of entropy is assumed to be delivered with a given time stamp the ESDM uses the 8 least significant bits (LSB) of the time stamp to provide a cushion for ensuring that at any given time stamp there is always at least one bit of entropy collected on all types of environments.

However, the question may be raised of why not use more data of the time stamp, i.e. why not using 32 bits or the full 64 bits of the time stamp to increase that cushion? There main answer is performance and memory consumption. The collection of a time stamp is performed as part of an scheduling function. Therefore, the performance of the ESDM in this code section is highly performance-critical. To limit the impact on the scheduler, the ESDM concatenates the 8 LSB of 1,024 time stamps received by the current CPU. The second aspect is that the higher bits of the time stamp must always be considered to have zero bits of entropy when considering the worst case of a skilled attacker. As the ESDM cannot identify whether it is under attack by a skilled attacker, it always assumes it is under attack.

The Linux kernel allows unprivileged user space processes to monitor the scheduling events of interrupts by reading the process listing with a certain degree of accuracy. The ESDM uses a high-resolution time stamp that executes with nanosecond precision on 1 GHz systems. Local attackers are expected to be measure the occurrence of a scheduling event with a microsecond precision. The distance between a microsecond and a nanosecond is 2^{10}. Thus, when the attacker is assumed to predict the interrupt occurrence with a microsecond precision and the time stamp operates with nanosecond precision, 10 bits of uncertainty remains that cannot be predicted by that attacker. Hence, only these 10 bits can deliver entropy.

To ensure the ESDM interrupt handling code has the maximum performance, it processes time stamp values with a number of bits equal to a power of two. Thus, the ESDM implementation uses 8 LSB of the time stamp (after the time

stamp was divided by its GCD).

During boot time, the presence of attackers is considered to be very limited as no remote access is yet possible and no local attack applications are assumed to execute. On the other hand, the performance of the interrupt handler is not considered to be very critical during the boot process. Thus, the ESDM uses the 32 LSB of the time stamp that is injected into the per-CPU collection pool when the time stamp is collected – the ESDM still awards this time stamp one bit of entropy. Once the ESDM completed the calculation of the GCD the aforementioned runtime behavior of concatenating the 8 LSB of 1,024 time stamps before mixing them into the per-CPU entropy pool is enabled.

2.5.2 Health Tests

The ESDM implements the following health tests:

- Stuck Test
- Repetition Count Test (RCT)
- Adaptive Proportion Test (APT)

Those tests are detailed in the following sections.

Please note that these health tests are only performed for the internal entropy sources. Other entropy sources like the entropy sources feeding the auxiliary pool, the Jitter RNG, or the CPU-based entropy sources are not covered by these tests as they are fully self-contained entropy sources where the ESDM does not have access to the raw noise data and does not include a model of the entropy source to implement appropriate health tests. The ESDM considers both as external entropy source. Thus, the user must ensure that either those other entropy sources implement all health tests as needed or the kernel must be started such that these entropy sources are credited with zero bits of entropy. Not crediting any entropy to these other entropy sources can be achieved with the following kernel configuration options:

- Sources feeding the auxiliary entropy pool: The interface functions to provide entropy data have to be invoked with the value 0 for the entropy rate.
- CPU-based entropy source: `ESDM_CPU_ENTROPY_RATE=0`
- Jitter RNG: `ESDM_JENT_ENTROPY_RATE=0`

These options ensure that random data from the entropy sources are pulled, but are not credited with any entropy.

The RCT, and the APT health test are only performed when the kernel is booted with `fips=1` and the kernel detects a high-resolution time stamp generator during boot.

In addition, the health tests are only enabled if a high-resolution time stamp is found. Systems with a low-resolution time stamp will not deliver sufficient entropy for the interrupt entropy source which implies that also the health tests are not applicable.

Stuck Test The stuck test calculates the first, second and third discrete derivative of the time stamp to be processed by the per-CPU collection pool. Only if all three values are non-zero, the received time delta is considered to be non-stuck. The first derivative calculated by the stuck test verifies that two successive time stamps are not equal, i.e. are "stuck". The second derivative calculates that there is no linear repetitive signal.

The third derivative of the time stamp is considered relevant based on the following: The entropy is delivered with the variations of the occurrence of interrupt events, i.e. it is mathematically present in the time differences of successive events. The time difference, however, is already the first discrete derivative of time. Now, if the time difference delivers the actual entropy, the stuck test shall catch that the time differences are not stuck, i.e. the first derivative of the time difference (or the second derivative of the absolute time stamp) shall not be zero. In addition, the stuck test shall ensure that the time differences do not show a linear repetitive signal – i.e. the second discrete derivative of the time difference (or the third discrete derivative of the absolute time stamp) shall not be zero.

Repetition Count Test The ESDM uses an enhanced version of the Repetition Count Test (RCT) specified in SP800-90B [2] section 4.4.1. Instead of counting identical back-to-back values, the input to the RCT is the counting of the stuck values during the processing of received interrupt events. The data that is mixed into the entropy pool is the time stamp. As the stuck result includes the comparison of two back-to-back time stamps by computing the first discrete derivative of the time stamp, the RCT simply checks whether the first discrete derivative of the time stamp is zero. If it is zero, the RCT counter is increased. Otherwise, the RCT counter is reset to zero.

The RCT is applied with $\alpha = 2^{-30}$ compliant to the recommendation of FIPS 140-2 IG 9.8.

During the counting operation, the ESDM always calculates the RCT cut-off value of C. If that value exceeds the allowed cut-off value, the ESDM will trigger the health test failure discussed below. An error is logged to the kernel log that such RCT failure occurred.

This test is only applied and enforced in FIPS mode.

Adaptive Proportion Test Compliant to SP800-90B [2] section 4.4.2 the ESDM implements the Adaptive Proportion Test (APT). Considering that the entropy is present in the least significant bits of the time stamp, the APT is applied only to those least significant bits. The APT is applied to the four least significant bits.

The APT is calculated over a window size of 512 time deltas that are to be mixed into the entropy pool. By assuming that each time stamp has (at least) one bit of entropy and the APT-input data is non-binary, the cut-off value $C = 325$ as defined in SP800-90B section 4.4.2.

This test is only applied and enforced in FIPS mode.

Runtime Health Test Failures If either the RCT, or the APT health test fails irrespective whether during initialization or runtime, the following actions occur:

1. The entropy of the entire entropy pool is invalidated.

2. All DRNGs are reset which imply that they are treated as being not seeded and require a reseed during next invocation.

3. The SP800-90B startup health test are initiated with all implications discussed in section 2.5.2. That implies that from that point on, new events must be observed and its entropy must be inserted into the entropy pool before random numbers are calculated from the entropy pool.

SP800-90B Startup Tests The aforementioned health tests are applied to the first 1,024 time stamps obtained from scheduler events. In case one error is identified for either the RCT, or the APT, the collected entropy is invalidated and the SP800-90B startup health test is restarted.

As long as the SP800-90B startup health test is not completed, all ESDM random number output interfaces that may block will block and not generate any data. This implies that only those potentially blocking interfaces are defined to provide random numbers that are seeded with the interrupt entropy source being SP800-90B compliant. All other output interfaces will not be affected by the SP800-90B startup test and thus are not considered SP800-90B compliant.

To summarize, the following rules apply:

- SP800-90B compliant output interfaces
  - `/dev/random`
  - `getrandom(2)` system call when called with a flag that does not include `GRND_INSECURE`
  - `esdm_get_random_bytes_full` API call
- SP800-90B non-compliant output interfaces
  - `/dev/urandom`
  - `getrandom(2)` system call when called with `GRND_INSECURE`
  - `esdm_get_random_bytes` API call
  - `esdm_get_random_bytes_min` API call

2.6 Auxiliary Entropy Pool - ESDM-external Entropy Sources

The ESDM also supports obtaining entropy from the following data sources and entropy sources that are external to the ESDM. The data is injected into the auxiliary pool.

During the reseeding operation of the DRNG, any user-space entropy provider waiting via select(2) or kernel space entropy provider using the `add_hwgenerator_randomness` API call are triggered to provide one buffer full of data. This data is mixed into the auxiliary pool. This approach shall ensure that the ESDM-external entropy sources may provide entropy at least once each DRNG reseed operation.

2.6.1 Kernel Hardware Random Number Generator Drivers

Drivers hooking into the kernel HW-random framework can inject entropy directly into the auxiliary pool. Those drivers provide a buffer to the entropy pool and an entropy estimate in bits. The auxiliary pool uses the given size of entropy at face value. The interface function of `add_hwgenerator_randomness` is offered by the ESDM.

2.6.2 Injecting Data From User Space

User space can take the following actions to inject data into the DRNG:

- When writing data into /dev/random or /dev/urandom, the data is added to the auxiliary pool and triggers a reseed of the DRNGs at the time the next random number is about to be generated. The ESDM assumes it has zero bits of entropy.

- When using the privileged IOCTL of `RNDADDENTROPY` with /dev/random, the caller can inject entropic data into the auxiliary pool and define the amount of entropy associated with that data.

2.6.3 Auxiliary Pool

The auxiliary pool is maintained as a separate entropy source that eventually is concatenated with all other entropy sources in compliance with SP800-90C.

The auxiliary pool is processed with the available hash as follows:

1. Data is inserted the same way as data is added into the per-CPU entropy pools. The auxiliary pool technically is the message digest state where new data is inserted into the pool by performing a hash update operation.

2. When entropy is to be extracted from the auxiliary pool, a hash final operation is performed which is immediately followed by a hash init operation to initialize the hash context for new data.

3. The generated message digest is truncated to the amount of data requested by the DRNG (e.g. either 256 or 384 bits) and returned to the caller. Note, the auxiliary pool output is not truncated to the amount of entropy the data contains because the entropy provider may add data to the auxiliary pool without entropy, e.g. by simply writing to /dev/random. Thus, it may be possible that the auxiliary pool contains zero bits of entropy but yet contains data that should be used to "stir" the DRNG state.

Figure 2.5 illustrates the auxiliary pool operation for the case when user space inserts two separate buffers and a kernel driver uses the add_hwgenerator_randomness function.

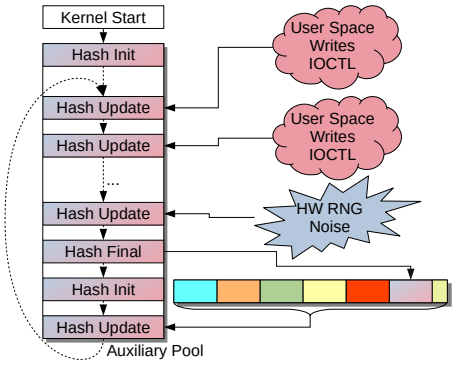

Figure 2.5: Auxiliary Pool Processing

In addition, the ESDM maintains an entropy estimator for the auxiliary pool counting the received entropy. The entropy estimator is capped to a maximum of the digest size of the used hash as this hash cannot maintain more entropy.

The auxiliary pool message digest is copied into the seed buffer when generating random numbers to seed the DRNG. The entire seed buffer is mixed back into the auxiliary pool for backward secrecy as shown in figure 2.5. The copy operation as well as the backtracking operation is atomic with respect to the auxiliary pool. This implies that both operations will always be fully completed before the next operation can commence. This ensures that the same auxiliary pool state can only be used once for a given seeding operation. Thus, both, the entropy pool and the auxiliary pool, are simultaneously used as noise data provider to seed the DRNG.

The entropy estimator is decreased by the amount of data read via the message digest.

2.7 Jitter RNG - ESDM-external Entropy Source

The Jitter RNG is treated as an external entropy source which is requested for random bits at the time the DRNG shall be seeded.

2.7.1 Entropy of CPU Jitter RNG Entropy Source

The CPU Jitter RNG entropy source is assumed provide 16th bit of entropy per generated data bit. Albeit studies have shown that significant more entropy is provided by this entropy source, a conservative estimate is applied.

The entropy value can be altered by changing the meson configuration option of `es_jent_entropy_rate`.

2.8 CPU-base Entropy Source - ESDM-external Entropy Source

The CPU-based entropy source is treated as an external entropy source which is requested for random bits at the time the DRNG shall be seeded. Depending on the underlying CPU, only one such source is available like RDSEED on Intel x86 (or RDRAND if RDSEED is not available), the POWER CPU DARN instruction, etc.

Depending whether the the CPU entropy source is documented to full entropy, the following data collection methods are applied. This approach is orthogonal to the amount of entropy the ESDM awards to the CPU entropy source.

- CPU entropy source provides full entropy: The CPU entropy source is queried for the amount of data which is stored in the temporary seed buffer.
- CPU entropy source provides less than full entropy: For this entropy source, the ESDM contains the information about how much data must be fetched from the CPU entropy source to get full entropy. The required amount of data is pulled from the CPU entropy source and conditioned with the hash currently in use by the ESDM. The calculated message digest is truncated to the requested amount of data which is stored in the temporary seed buffer.

2.8.1 Entropy of CPU Entropy Source

The entropy source of the CPU is assumed to have one 32th of the generated data size – 8 bits of entropy. The reason for that conservative estimate is that the design and implementation of those entropy sources is not commonly known and reviewable. The entropy value can be altered by changing the meson configuration option of `es_cpu_entropy_rate`.

2.9 Kernel RNG Entropy Source - ESDM-external Entropy Source

The ESDM offers the use of the kernel RNG as an entropy source. The kernel RNG derives its entropy from sampling of interrupts.

The legacy RNG is queried for random numbers using the `getrandom(2)` system call.

2.9.1 Entropy of Kernel RNG Entropy Source

When the lernel RNG is enabled, the ESDM applies the entropy rate defined at compile time with `es_kernel_entropy_rate` which is a value between 0 and 256 bits of entropy when 256 data bits are pulled from the legacy RNG.

If the ESDM is operated in FIPS mode, i.e. the kernel command line contains "fips=1", the kernel RNG entropy source's entropy rate is set to zero. The reason is that the legacy RNG is known to not comply with FIPS 140 rules like SP800-90B and thus must be assumed to provide no entropy.

2.10 DRNG Seeding Operation

The seeding operation obtains random data from all available entropy sources.

The (re)seeding logic tries to obtain 256 bits of entropy from the entropy sources. However, if less entropy can only be delivered, the DRNG reseeding is only performed if at least 128 bits of entropy collectively from all entropy sources can be obtained.

For efficiency reasons, the seeding operation uses a seed buffer depicted in figure 2.1 that is the following set of blocks of 256 bits each. If SP800-90C compliance is enabled, the initial seeding of the DRNG is seeded with a seed buffer that pulls 384 bits in the following blocks. Each block is dedicated to an entropy source. As each entropy source can be disabled at compile time, the different bullets in the following list only applies if the corresponding entropy source is enabled.

1. One block is filled with the message digest from the auxiliary pool.
2. Another block is filled with the message digest of all scheduler per-CPU collection pools.
3. One block is filled with the data from the kernel RNG entropy source.
4. The next block is filled by the Jitter RNG entropy source.
5. Finally, a block is filled by the fast entropy source of the CPU entropy source.

Finally, also a 32 bit time stamp indicating the time of the request is mixed into the DRNG. That time stamp, however, is not assumed to have entropy and is only there to further stir the state of the DRNG.

The filled seed buffer is handed to the DRNG as a seed string. In addition, the seed buffer is inserted back into the auxiliary pool for backward secrecy. The seed buffer will not alter the entropy estimation of the auxiliary pool.

2.10.1 DRNG May Become Not Fully Seeded

The ESDM maintains a counter for each DRNG instance how many generate operation are performed without performing a reseed that has full entropy. If this counter exceeds the threshold of 2^{30} generate operations, i.e. the DRNG did not receive a seed with full entropy for that many generate operations, the DRNG is set to not fully seeded. This setting implies that the DRNG instance will not be used any more for generating random numbers until the ESDM received sufficient entropy to reseed the DRNG with full entropy.

If the DRNG that becomes not fully seeded is the initial DRNG instance that was seeded during boot time as outlined in section 2.3.1, the entire ESDM is marked as not operational. This setting blocks all blocking interfaces just like during boot time when the ESDM is not yet fully seeded.

The ESDM automatically tries to recover from it when it received sufficient entropy.

2.11 Cryptographic Primitives Used By ESDM

The following subsections explain the cryptographic primitives that may be used by the ESDM.

2.11.1 DRBG

If the SP800-90A DRBG implementation is used, the default DRBG used by the ESDM is the CTR DRBG with AES-256. The reason for the choice of a CTR DRBG is its speed. The source code allows the use of other types of DRBG by

simply defining a DRBG reference using the kernel crypto API DRBG string – see the top part of the source code for examples covering all types of DRBG.

All DRNGs are always instantiated with the same DRNG type.

The implementation of the DRBG is taken from the Linux kernel crypto API. The use of the kernel crypto API to provide the cipher primitives allows using assembler or even hardware-accelerator backed cipher primitives. Such support should relieve the CPU from processing the cryptographic operation as much as possible.

The input with the seed and re-seed of the DRBG has been explained above and does not need to be re-iterated here. Mathematically speaking, the seed and re-seed data obtained from the entropy sources and the ESDM external sources are mixed into the DRBG using the DRBG "update" function as defined by SP800-90A.

The DRBG generates output with the DRBG "generate" function that is specified in SP800-90A. The DRBG used to generate two types of output that are discussed in the following subsections.

/dev/urandom and `esdm_get_random_bytes` Users that want to obtain data via the /dev/urandom user space interface or the `esdm_get_random_bytes` API are delivered data that is obtained from the DRNG "generate" function. I.e. the DRNG generates the requested random numbers on demand.

Data requests on either interface is segmented into blocks of maximum 4096 bytes. For each block, the DRNG "generate" function is invoked individually. According to SP800-90A, the maximum numbers of bytes per DRBG "generate" request is 2^{19} bits or 2^{16} bytes which is significantly more than enforced by the ESDM.

In addition to the slicing of the requests into blocks, the ESDM maintains a counter for the number of DRNG "generate" requests since the last reseed. According to SP800-90A, the number of allowed requests before a forceful reseed is 2^{48} – a number that is very high. The ESDM uses a much more conservative threshold of 2^{20}requests as a maximum. When that threshold is reached, the DRBG will be reseeded by using the operation documented in section 2.10 before the next DRNG "generate" operation commences.

The handling of the reseed threshold as well as the capping of the amount of random numbers generated with one DRNG "generate" operation ensures that the DRNG is operated compliant to all constraints in SP800-90A.

/dev/random and `esdm_get_random_bytes_full` The random numbers to be generated for /dev/random as well as `esdm_get_random_bytes_full` are defined to have a special property: it only provides data once at least 256 bits of entropy have been collected by the ESDM. In addition, the ESDM must be fully initialized before random numbers are generated, including the completion of the SP800-90B heath test if entropy from internal entropy sources is gathered.

2.11.2 ChaCha20 DRNG

If the SP800-90A DRBG is not desired, the ESDM can use a standalone C implementations for ChaCha20 to provide a DRNG.

The ChaCha20 DRNG is implemented with the components discussed in the following section. All of those components rest on a state defined by [1], section 2.3.

The operation of the ChaCha20 DRNG can be characterized with figure 2.6. This figure outlines the initialization of the DRNG, its seeding using the state update operation and the invocation of one generate operation that is requested to obtain more than 512 bits of data.

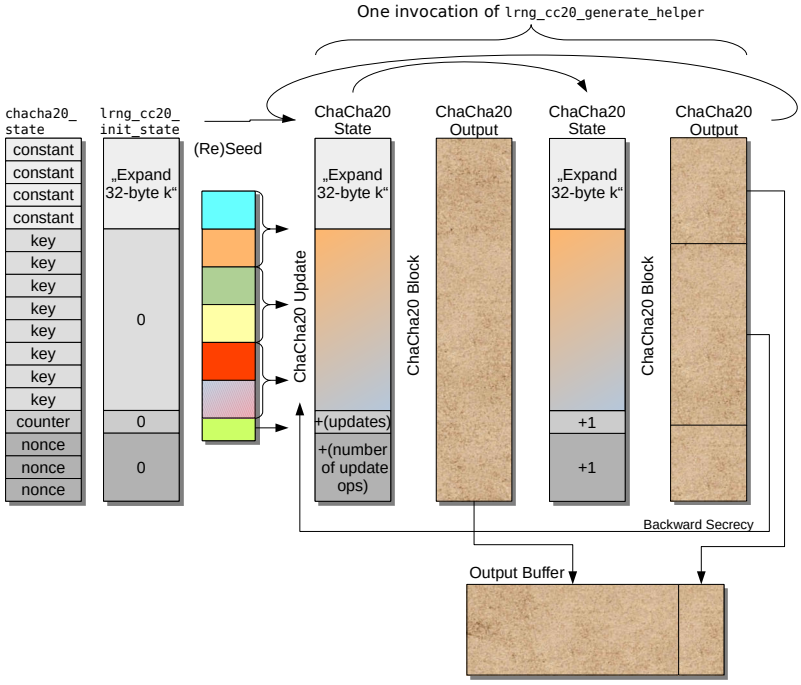

Figure 2.6: ChaCha20 DRNG Operation

State Update Function The state update function's purpose is to update the state of the ChaCha20 DRNG. That is achieved by

1. generating one output block of ChaCha20,
2. partition the generated ChaCha20 block into two key-sized chunks,
3. and XOR both chunks with the key part of the ChaCha20 state.

In addition, the nonce part of the state is incremented by one to ensure the uniqueness requirement of [1] chapter 4.

Seeding Operation The seeding operation processes a seed of arbitrary lengths. The seed is segmented into ChaCha20 key size chunks which are sequentially processed by the following steps:

1. The key-size seed chunk is XORed into the ChaCha20 key location of the state.

2. This operation is followed by invoking the state update function.

3. Repeat the previous steps for all unprocessed key-sized seed chunks.

If the last seed chunk is smaller than the ChaCha20 key size, only the available bytes of the seed are XORed into the key location. This is logically equivalent to padding the right side of the seed with zeroes until that block is equal in size to the ChaCha20 key.

The invocation of the state update function is intended to eliminate any potentially existing dependencies between the seed chunks.

Generate Operation The random numbers from the ChaCha20 DRNG are the data stream produced by ChaCha20, i.e. without the final XOR of the data stream with plaintext. Thus, the DRNG generate function simply invokes the ChaCha20 to produce the data stream as often as needed to produce the requested number of random bytes.

After the conclusion of the generate operation, the state update function is invoked to ensure enhanced backward secrecy of the ChaCha20 state that was used to generate the random numbers.

2.12 ESDM External Interfaces

The following ESDM interfaces are provided:

- ESDM library API: The API is documented in 'esdm.h'.
- ESDM server RPC API: The RPC API is documented in 'esdm_rpc_client.h'.
- /dev/random and /dev/urandom device files: See the Linux man page `random(4)`.
- getrandom and getentropy APIs: See Linux man pages `getrandom(2)` and `getentropy(3)`.

2.13 ESDM Self-Tests

All cryptographic primitives are self-tested during the startup phase of the ESDM.

2.14 ESDM Test Interfaces

During kernel compilation, the following interfaces may be enabled allowing direct access to non-deterministic aspects. It is not advisable to enable these interfaces for production systems. Yet, these interfaces are considered to be protected against misuse by allowing only the root user to access them. In addition, any data obtained through these interfaces is not used by the ESDM to feed the entropy pool. Thus, even when leaving these interfaces enabled on production systems, the impact on security is considered to be limited.

- Scheduler Entropy Source:

- The interface `/sys/kernel/debug/esdm_testing/esdm_raw_sched_hires` allows reading of the raw unconditioned noise data collected while the read operation is in progress by providing the time stamps of the events collected by the ESDM that otherwise are injected into the entropy pool. When booting the kernel with the kernel command line option `esdm_testing.boot_raw_sched_hires_test=1`, the time stamps of the first 1,024 events recorded by the ESDM are stored. The first read of the `esdm_raw_sched_hires` file after boot provides this data in this case.
- The interface `/sys/kernel/debug/esdm_testing/esdm_raw_sched_pid` allows reading of the PID value of the task that are about to be scheduled to collected while the read operation is in progress. The PID values are extracted which are collected by the ESDM that otherwise are injected into the entropy pool (if no high-resolution time stamp is detected). When booting the kernel with the kernel command line option `esdm_testing.boot_raw_sched_pid_test=1`, the time stamps of the first 1,024 events recorded by the ESDM are stored. The first read of the `esdm_raw_sched_pid` file after boot provides this data in this case.
- The interface `/sys/kernel/debug/esdm_testing/esdm_raw_starttime_pid` allows reading of the start time value of the task that are about to be scheduled to collected while the read operation is in progress. The start time values are extracted which are collected by the ESDM that otherwise are injected into the entropy pool (if no high-resolution time stamp is detected). When booting the kernel with the kernel command line option `esdm_testing.boot_raw_sched_starttime_test=1`, the time stamps of the first 1,024 events recorded by the ESDM are stored. The first read of the `esdm_raw_starttime_pid` file after boot provides this data in this case.
- The interface `/sys/kernel/debug/esdm_testing/esdm_raw_nvcsw_pid` allows reading of the numbers of context switches of the task that are about to be scheduled to collected while the read operation is in progress. The context switch values are extracted which are collected by the ESDM that otherwise are injected into the entropy pool (if no high-resolution time stamp is detected). When booting the kernel with the kernel command line option `esdm_testing.boot_raw_sched_nvcsw_test=1`, the time stamps of the first 1,024 events recorded by the ESDM are stored. The first read of the `esdm_raw_nvcsw_pid` file after boot provides this data in this case.
- The interface `/sys/kernel/debug/esdm_testing/esdm_sched_perf` allows reading of the number of cycles used to process one scheduler event. This allows measuring the performance impact of the ESDM on the scheulder. When booting the kernel with the kernel command line option `esdm_testing.boot_sched_perf=1`, the performance data of the first 1,024 events recorded by the ESDM are stored. The first read of the `esdm_sched_perf` file after boot provides this data in this case.

The helper tool getrawentropy.c is provided to read the files and format the data

for post-processing.

3 Scheduler Entropy Source Assessment

4 ESDM Specific Configurations

The ESDM offers a secure and appropriate set of features with the default configuration. Yet, use cases may arise where the ESDM should exhibit a different behavior. The flexibility of the ESDM allows a various configurations that are intended to meet different requirements.

4.1 SP800-90C Compliance

The specification of SP800-90C defines construction methods to design non-deterministic as well as deterministic RNGs. As the specification is currently in draft form, the latest available draft from January 21, 2021 is applied.

The specification defines different types of RNGs where the following mapping to the ESDM applies:

- The ESDM follows the construction of RBG2(NP) with the following entropy sources whose outputs are concatenated:
  - Auxiliary external entropy sources, such as user-space rngd, if available. The administrator is responsible to guarantee that this entropy source is compliant to SP800-90B if it alters the entropy estimator maintained by the ESDM via the `RNDADDENTROPY` IOCTL. Depending on the selected entropy source, this may be a physical or non-physical entropy source. Note, all data maintained by the auxiliary pool are processed with a vetted conditioning component. Thus, to achieve full SP800-90C compliance for such entropy sources, only one should feed data credited with entropy into the auxiliary pool.
  -
  - Scheduler entropy source is the only entropy source that is fully maintained as part of the ESDM and is subject to a full entropy analysis following 3. Furthermore, it is claimed to be fully SP800-90B compliant.
  - The Jitter RNG entropy source is used by the ESDM. This entropy source is a fully self-contained SP800-90B entropy source. Its SP800-90B compliance must be assessed separately. If SP800-90B compliance cannot be demonstrated, it must be awarded to credit zero bits of entropy with the configuration documented in section 2.7.1 or by completely disabling it by not selecting the meson configuration option of `es_jent`.
  - The CPU entropy source is used by the ESDM, if available. On Intel, this uses RDSEED. This entropy source is a fully self-contained SP800-90B entropy source. Its SP800-90B compliance must be assessed separately. If SP800-90B compliance cannot be demonstrated,

it must be awarded to credit zero bits of entropy with the configuration documented in section 2.8.1 or by completely disabling it by not selecting the meson configuration option of `es_cpu`.

By applying Method 2 of section 3.3 in SP800-90C, the entropy provided by all entropy sources can be added which is applied when the ESDM constructs the temporary seed buffer as shown with equation 2.8.

- During instantiation of the DRBG, the ESDM tries to seed the DRBG with at least (DRBG security strength) + 128 bits = 384 bits of entropy. Only when this amount of entropy was obtained to seed the DRBG is considered to be fully seeded and is allowed to produce output.

  If the SP800-90C construction is to be used as the "randomness source" following bullet 5 of section "Note to Reviewer" in the SP800-90C document to seed another DRBG with a security strength of 256 bits, either the hash or HMAC DRBG should be used as they are capable of transporting up to 512 bits of entropy and are initially seeded with 384 bits of entropy. Thus, the SP800-90C seeding requirement of providing 384 bits of entropy can be satisfied.

- Reseeding of the DRBG can be triggered by either writing data into /dev/random or by the `RNDRESEEDCRNG` IOCTL. The ESDM guarantees that the reseed operation is only performed if at least 128 bits of entropy is available. If this is not available, the reseed is attempted during the next generate operation. Yet, the generate operation is conducted in any case. Nonwithstanding, the ESDM always attempts to obtain 256 bits of entropy for reseeding. This is considered appropriate because the upper limit when a reseed is ultimately triggered is 2^{20} generate operations or after 10 minutes, whatever is reached earlier. If the DRNG cannot be reseeded it will continue to operate until the next time this threshold is reached. The DRNG will revert to an unseeded stage if it cannot be reseeded by the time it serviced 2^{30} generate requests since the last successful seeding operation.

The requirements from section 6.3 SP800-90C are met as follows:

1. The administrator must use the SP800-90A DRBG ESDM extension as mentioned above to satisfy the requirement.

2. The DRBG can be ACVTS-tested to show compliance to SP800-90A. The entropy sources are to be assessed pursuant to SP800-90B as outlined in the above listing.

3. See the above listing for the reseeding support.

4. If an entropy source is not validated, its entropy estimation must be set to zero as outlined in the above listing.

5. N/A as the ESDM is claimed to conform with RBG2(NP).

6. The entropy sources are listed above. By using concatenation of the output of all entropy sources, the Method 2 SP800-90C is implemented. The entropy of all entropy sources are added.

7. Technically it is possible that all DRBG security strengths can be chosen as the DRBG supports all security strengths. Yet, the ESDM interfaces currently only support the highest security strength of 256 bits to ensure that it can be used for all use cases.

8. The entropy source output is destroyed immediately after it was used to (re)seed the DRBG. Note, the use of the seed for backward secrecy by injecting it into the auxiliary pool via the vetted conditioning operation is considered to not violate the requirement as the seed data is unrecoverable. Furthermore, the seed data is not credited with any entropy during the backward secrecy operation. Therefore, the seed data is only used to further mix the internal state of the ESDM.

9. N/A as the ESDM does not use a CTR DRBG without derivation function.

10. The ESDM attempts to instantiate a DRBG with 3/2s bits of entropy. Only if this succeeds, the DRBG becomes available. The ESDM attempts to reseed a DRBG with s bits of entropy. See the rationale above for the discussion about the minimum entropy size of the reseeding operation of 128 bits.

11. The DRBG only provides output once it is fully seeded as mandated by SP800-90C.

12. An error occurring in the interrupt entropy source triggers a full reset of the ESDM as outlined in section 2.5.2. If the other entropy sources are subject to a health test failures, SP800-90B mandates that they do not produce entropy. Before the first initialization of the DRBG, it is subject to a power-on self test. The ESDM performs power-up self tests as outlined in section 2.13.

13. This requirement is implicitly met by the fact that the ESDM only provides DRBGs with the maximum security strength of 256 bits.

4.1.1 RBG2(P) Construction Method

It is possible to convert the ESDM into the SP800-90C type of RBG2(P). This approach requires that only physical entropy sources are credited with entropy. The following specific settings must be applied in addition to the general configurations listed in the next section:

- Configure the scheduler entropy source to not credited entropy: compile the kernel with the meson configuration option of `es_sched_entropy_rate=0`. This setting ensures that the scheduler entropy source still collects data, but it is not credited with entropy. If it shall be completely disabled, the compile time option of `es_sched` must be unset.

- Configure the Jitter RNG entropy source to not credit entropy: compile the kernel with the meson configuration option of `es_jent_entropy_rate=0`. If the value is set to 0, the entropy source still collects data, but it is not credited with entropy. If it shall be completely disabled, the compile time option of `es_jent` must be unset.

- Configure the kernel RNG entropy source to not credit entropy: compile the kernel with the meson configuration option of `es_kernel_entropy_rate=0`. If the value is set to 0, the entropy source still collects data, but it is not credited with entropy. If it shall be completely disabled, the compile time option of `es_kernel` must be unset.

- Adjust the entropy rate of the CPU entropy source as needed: compile the kernel with the meson configuration option of `es_cpu_entropy_rate=256` when full entropy is assumed to be provided by the CPU entropy source. In general, set any value between 0 and 256 if the default value is not appropriate. If the value is set to 0, still collects data, but it is not credited with entropy. If it shall be completely disabled, the compile time option of `es_cpu` must be unset.

- If other hardware entropy sources are considered deliver entropy, they may inject data into the ESDM via the IOCTL `RNDADDENTROPY`. Note, only physical entropy sources must provide data that is credited with entropy. Any other data fed into the ESDM via those interfaces must not be credited with entropy.

Important note: The entropy rate provided by the CPU entropy source plus all other physical entropy sources together must ensure they provide sufficient entropy. "Sufficient entropy" is provided when the entropy rate equals the hash type used by the ESDM. Considering the default message digest of SHA2-512, 512 bits of entropy should be provided.

Naturally, the hardware entropy sources that are credited with entropy must be compliant to SP800-90B.

In case the RBG2(P) construction method is achieved, the following additional requirement from section 6.3 SP800-90C is met:

- Requirement 5: With the required configuration mentioned before the ESDM will only count the physical entropy sources towards fulfilling the requested amount of entropy.

4.1.2 SP800-90C Compliant Configuration

SP800-90C compliance is only achieved when all of the following settings are achieved.

The following compile-time settings must be observed:

- The meson configuration option `oversample_es` is set. This option guarantees the following:

  - The final conditioning operation applied to the interrupt entropy source as well as to the auxiliary pool require 64 additional bits of entropy when obtaining data for the temporary seed buffer. This complies with the requirement specified in section 4.3.2 SP800-90C about vetted conditioning components. This ensures the conditioning components provide full entropy.
  - When the DRNG is initially seeded, it is attempted to be seeded with 384 bits of entropy at least. This complies with the requirement specified in section 6.2.1 bullet 2 of SP800-90C requiring 3/2s bits

of entropy for the initial seeding. Note, the ESDM applies a stepwise seeding of 32, 128 and 256 bits of entropy during initialization. When the final step of 256 bits is to be performed, the ESDM will guarantee that at least 384 bits of entropy are collectively pulled from all entropy sources. Only if this is achieved, the SP800-90C compliant-marked interfaces of the ESDM specified in section 2.5.2 will produce random numbers.

- Enable `es_sched` configure the scheduler entropy source compliant to SP800-90B as outlined in section 3 if the scheduler entropy source is considered to be compliant to SP800-90B by accepting the assessment in chapter 3 and by applying the testing of the entropy source as outlined in this chapter on the target system.

- The kernel must be compiled with the meson configuration option of `es_cpu_entropy_rate=0` or unset the option `es_cpu` unless the CPU-based entropy source (e.g. RDSEED on Intel) have an SP800-90B compliant entropy assessment and comply with all requirements from SP800-90B.

- The kernel must be compiled with the kernel configuration option of `es_jent_entropy_rate=0` or unset the option `es_jent` unless the Jitter RNG entropy source has an SP800-90B compliant entropy assessment and comply with all requirements from SP800-90B[5].

The following requirements apply to the runtime configuration:

- The ESDM must be operated in FIPS mode to ensure the oversampling for the conditioning and the initial seeding of the DRBG is applied. This is achieved with the kernel command line option of `fips=1`.

- Filling up the ESDM with entropy using either a user-space RNGD via the IOCTL `RNDADDENTROPY` is allowed. However, the caller is only allowed to claim entropy associated with the data and thus increase the ESDM entropy estimation if the entropy source is SP800-90B compliant with its own entropy assessment.

To verify that the SP800-90C compliance is achieved, the file /proc/ESDM_type provides an appropriate status indicator. The SP800-90C compliant is ensured only for the respectively marked ESDM interfaces specified in section 2.5.2. All other interfaces are not providing SP800-90C compliant random numbers.

4.2 AIS 20 / 31

The ESDM currently is not consistent with AIS20/31.

A Thanks

Special thanks for providing input as well as mathematical support goes to:

[5] At the time of writing, the user space Jitter RNG is SP800-90B compliant. Patches ensuring the in-kernel variant is SP800-90B compliant as well when into the kernel for version 5.8.

- DJ Johnston
- Yi Mao
- Sandy Harris
- Dr. Matthias Peter
- Quentin Gouchet

B Source Code Availability

The source code, this document as well as the test code for all aforementioned tests is available at http://www.chronox.de/ESDM.html.

C Auxiliary Testing

In addition to the entropy testing additional functional tests are applied using the meson test infrastructure. For details, see the 'tests' directory.

D Bibliographic Reference

References

[1] Y. Nir and A. Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 7539 (Informational), May 2015. URL http://www.ietf.org/rfc/rfc7539.txt.

[2] Meltem Sönmez Turan, Elaine Barker, John Kelsey, Kerry A. McKay, Mary L. Baish, and Mike Boyle. *NIST Special Publication 800-90B Recommendation for the Entropy Sources Uses for Random Bit Generation.* 2018.

E License

3. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

ALTERNATIVELY, this product may be distributed under the terms of the GNU General Public License, in which case the provisions of the GPL are required INSTEAD OF the above restrictions. (This clause is necessary due to a potential bad interaction between the GPL and the restrictions contained in a BSD-style copyright.)

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ALL OF WHICH ARE HEREBY DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF NOT ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

F Change Log

Date	ESDM Version	Change
2022-05.10	v0.2.0	Initial public release