Des nombres aléatoires dans le noyau Linux

Damien Goutte-Gattat <dgouttegattat@incenp.org>

2020/09/02

Résumé

Cet article présente les mécanismes de génération de nombres (pseudo-)aléatoires dans le noyau Linux, et fait le point sur les principales évolutions survenues entre les versions 3.17 (octobre 2014) et 5.6 (mars 2020).

Table des matières

1. Typologie des générateurs de nombres aléatoires

Brièvement, on distingue plusieurs types de RNG (*Random Number Generator*). Dans tous les cas, les propriétés minimales que l'on attend d'un RNG quel que soit son type sont que :

- chaque nombre généré doit être statistiquement indépendant des nombres précédents (un nombre donné n'a pas plus de chance qu'un autre d'apparaître après une certaine séquence) ;

- les nombres générés sont uniformément distribués dans la plage de valeurs souhaitées (aucun nombre ne doit apparaître plus fréquemment qu'un autre).

Un *True Random Number Generator* ou TRNG produit des nombres *réellement* aléatoires en se basant sur des phénomènes imprévisibles — par exemple, le délai entre deux émissions de particules par radioactivité, le bruit thermique d'une résistance, le bruit de fond d'un microphone ou d'une caméra, etc. Ce type de générateur a typiquement un débit assez faible (puisque limité par les propriétés du phénomène imprévisible sous-jacent) et nécessite une étape de *conditionnement* pour assurer la distribution uniforme requise.

Un *Pseudo-Random Number Generator* ou PRNG est un algorithme pour générer des nombres *semblant* aléatoires de façon déterministe à partir d'une graine (*seed*) donnée.

Un *Cryptographically Secure Pseudo-Random Number Generator* ou CSPRNG est un type particulier de PRNG dont la production est imprévisible pour un observateur ne connaissant pas l'état interne du générateur, et est de fait convenable pour des applications cryptographiques (par exemple la création de clefs). Formellement, un PRNG est un CSPRNG s'il n'existe pas d'algorithme en temps polynomial capable, à partir de n bits produits par le générateur, de prédire le bit n + 1 en se trompant moins d'une fois sur deux.[1]

En général, on attend aussi d'un CSPRNG qu'il fournisse une certaine résistance même dans l'hypothèse où son état interne est partiellement ou totalement compromis. En particulier, il doit être infaisable, connaissant l'état interne du générateur à un instant t, d'en déduire les nombres précédemment produits (*backtracking resistance*).

Un *PRNG with entropy inputs*[2] est un PRNG dont l'état interne est régulièrement ré-initialisé (totalement ou partiellement) avec des nombres réellement aléatoires. Schématiquement, c'est la combinaison d'un TRNG et d'un CSPRNG, le premier étant utilisé pour (ré-)initialiser le second. Par rapport à un CSPRNG seul, on attend d'une telle combinaison qu'elle fournisse en plus une *forward resistance* : il doit être infaisable, connaissant l'état interne du générateur à un instant t, de déduire les prochains nombres une fois que le CSPRNG a été ré-initialisé.

Enfin, un *HardWare Random Number Generator* ou HWRNG est un RNG matériel, par opposition à un RNG logiciel. Un tel RNG peut prendre la forme d'une puce spécialisée sur une carte-mère (comme une puce TPM — *Trusted Platform Module*), d'une carte d'extension, ou encore d'un périphérique externe (comme par exemple le périphérique libre NeuG [https://www.gniibe.org/memo/development/gnuk/rng/neug.html]).

On verra plus loin qu'au sein du noyau Linux, une distinction supplémentaire est faite entre les HWRNG *périphériques*, qui se présentent sous la forme d'un périphérique interne ou externe, et le HWRNG éventuellement présent directement dans le processeur et accessible via une simple instruction (comme l'instruction `RDRAND` des processeurs X86 récents) — ce dernier est qualifié de RNG *architectural*.

La notion de RNG matériel ou logiciel est orthogonale à celle de RNG produisant des nombres réellement aléatoires (TRNG) ou pseudo-aléatoires (PRNG) : ce n'est pas parce qu'un RNG est matériel qu'il s'agit nécessairement d'un TRNG ! En fait c'est même rarement le cas, la plupart des HWRNG sont des implémentations matérielles d'un CSPRNG.

Éventuellement le CSPRNG peut être associé à un composant TRNG pour son initialisation, formant ainsi un *PRNG with entropy inputs* comme décrit plus haut (c'est ainsi par exemple qu'est implémentée l'instruction `RDRAND` des processeurs Intel [https://software.intel.com/content/www/us/en/develop/articles/intel-digital-random-number-generator-drng-software-implementation-guide.html#inpage-nav-3]) ; dans d'autres cas le CSPRNG est initialisé en usine avec une graine unique pour chaque exemplaire du matériel (c'est le cas par exemple pour de nombreuses cartes à puce, notamment en entrée de gamme).

2. Gestion des HWRNG dans le noyau Linux

Le sous-système *hw_random* du noyau Linux (implémenté dans le dossier `drivers/char/hw_random`) fournit le support des différents HWRNG éventuellement dis-

[1] A. Menezes, P. van Oorschot, et S. Vanstone (1996). *Handbook of Applied Cryptography*, CRC Press, p. 171. http://cacr.uwaterloo.ca/hac/.

[2] Ce type de PRNG ne semble pas avoir d'acronyme consacré dans la littérature — je proposerais bien CRPRNG pour *Continuously Reseeded PRNG* mais ça n'engage que moi.

ponibles sur une machine. Il fournit une interface unique pour accéder à ces générateurs, constituée du fichier de périphérique `/dev/hwrng` et du dossier `/sys/class/misc/hw_random`.

Le fichier `/dev/hwrng` donne directement accès à un RNG matériel si au moins un tel RNG est présent sur la machine. S'il y en a plusieurs, un seul d'entre eux peut être connecté au périphérique `/dev/hwrng` à la fois. Le fichier `/sys/class/misc/hw_random/rng_available` donne la liste des HWRNG disponibles, et le fichier `/sys/class/misc/hw_random/rng_current` indique celui qui est présentement accessible via le périphérique `/dev/hwrng`. On peut passer d'un HWRNG à l'autre en écrivant le nom du HWRNG souhaité (tel qu'il apparaît dans `rng_available`) dans `rng_current`.

Le périphérique `/dev/hwrng` fournit les octets (pseudo-)aléatoires « bruts », tels que produits par le HWRNG sous-jacent ; le noyau n'applique aucun traitement intermédiaire, et ne procède à aucune vérification de la qualité des nombres aléatoires générés (tests statistiques pour vérifier l'absence de biais par exemple). Il appartient entièrement aux applications utilisatrices de décider de la confiance à accorder au HWRNG.

Le noyau lui-même n'utilise (quasiment) pas le sous-système *hw_random*, dont le seul but est d'exposer les HWRNG de la machine à l'espace utilisateur. Le seul usage que le noyau fait de *hw_random* est d'utiliser les éventuels HWRNG disponibles pour contribuer un peu au *pool d'entropie* au cœur du pilote *random*, comme on le verra plus loin.

Le sous-système *hw_random* ne concerne que les HWRNG *périphériques*. Le HWRNG *architectural* (intégré au processeur), s'il existe, n'est pas géré par *hw_random*. Il est en effet directement accessible aux applications via une instruction du processeur (`RDRAND` ou `RDSEED` sur les processeurs X86), de sorte qu'aucun support par le noyau n'est nécessaire.

3. Le pilote *random*

Le pilote *random* (entièrement contenu dans `drivers/char/random.c`) est le RNG du noyau. Il est en charge de fournir tous les nombres aléatoires dont le système a besoin, que ce soit à l'intérieur du noyau ou dans l'espace utilisateur. À l'intérieur du noyau, il est appelable via la fonction `get_random_bytes()` ; en espace utilisateur, il est accessible via les fichiers de périphériques `/dev/random` et `/dev/urandom`, et via l'appel système `getrandom()`.

Cette section décrit l'implémentation « historique » du pilote, jusqu'à la version 3.17 du noyau.[3] Les évolutions majeures introduites à partir de cette version jusqu'à l'implémentation actuelle seront décrites plus loin.

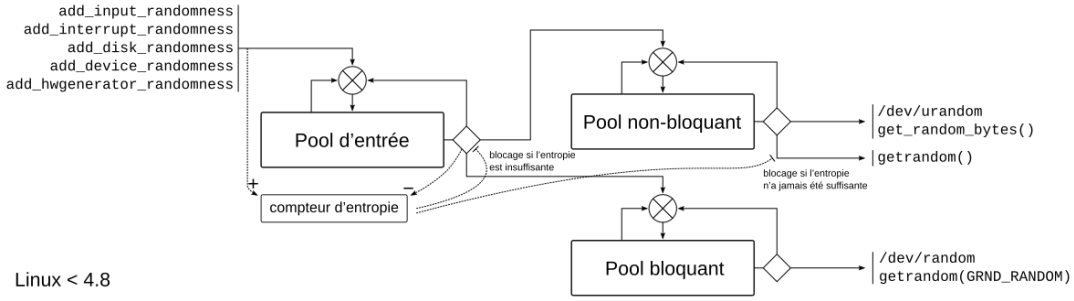
3.1. Architecture générale

Le pilote *random* implémente un RNG de type *PRNG with entropy inputs*. Il est bâti autour de trois *pools d'entropie* (Figure 1) :

- un *pool d'entrée*, qui collecte l'entropie à partir de plusieurs sources dans le système (décrites dans une section suivante) ;
- un *pool de sortie non-bloquant*, qui alimente le périphérique `/dev/urandom` ainsi que la fonction interne `get_random_bytes()` ;
- et un *pool de sortie bloquant*, qui alimente le périphérique `/dev/random`.

[3] Une analyse détaillée de l'implémentation historique, couvrant les versions 2.6.30 à 3.1.10, est disponible dans Lacharme *et al*. (2012) [https://eprint.iacr.org/2012/251].

Figure 1. Architecture historique du RNG du noyau Linux



Chaque pool est un tampon de mémoire (de 512 octets pour le pool d'entrée, 128 octets pour les pools de sortie) associé à un *compteur d'entropie* fournissant une estimation, en nombre de bits, du caractère imprévisible du contenu du pool ; ce compteur est incrémenté lorsque des octets aléatoires sont mixés dans le pool (ajout d'entropie) et décrémenté lorsque des octets aléatoires en sont extraits.

Le compteur d'entropie du pool d'entrée est le seul qui soit réellement important. C'est d'ailleurs le seul dont la valeur est lisible depuis l'espace utilisateur (via le fichier `/proc/sys/kernel/random/entropy_avail`) et sauf précision contraire, c'est toujours de ce compteur dont il est question quand on parle de « l'entropie disponible ».

Les octets contenus dans un pool sont brassés par une *fonction de mixage* (`mix_pool_bytes()`, représentée par le symbole ⊗ en Figure 1). Cette fonction est aussi utilisée pour ajouter des octets dans le pool, et assure que l'ajout de mêmes quelques octets seulement a des répercussions sur l'ensemble du pool.

La production d'octets pseudo-aléatoires depuis un pool est réalisée par une *fonction d'extraction* (`extract_entropy()`, représentée par le symbole ◇ en Figure 1). Brièvement, cette fonction calcule un condensat SHA-1 sur le contenu du pool, ré-injecte les 20 octets du condensats dans le pool via la fonction de mixage, et produit dix octets finaux en sortie, obtenus en « repliant » le condensat (les dix premiers octets sont XORés avec les dix derniers).

3.2. Principe de fonctionnement

Chaque pool avec sa fonction de mixage et sa fonction d'extraction est assimilable à un CSPRNG indépendant.

Le pool d'entrée est continuellement alimenté par les différentes sources d'entropie du système, qui sont collectivement assimilables à un TRNG.

En l'absence de sollicitations, l'entropie collectée reste dans le pool d'entrée, et les pools de sortie sont maintenus « vides » (leur compteur d'entropie est proche de zéro). Lorsqu'un des périphériques `/dev/random` ou `/dev/urandom` est sollicité en lecture, de l'entropie est transférée depuis le pool d'entrée vers le pool de sortie correspondant (le pool bloquant pour `/dev/random`, le pool non-bloquant pour `/dev/urandom`) : des octets sont extraits du pool d'entrée via sa fonction d'extraction (décrémentant son compteur d'entropie au passage) et sont ajoutés au pool de sortie via sa fonction de mixage. Le pool de sortie désormais plein se vide alors immédiatement dans le périphérique associé. L'opération est répétée jusqu'à ce que le pool de sortie ait produit autant d'octets pseudo-aléatoires qu'il lui en a été demandé, tant que le compteur d'entropie du pool d'entrée reste au-dessus d'un certain seuil.

Lorsque le compteur d'entropie du pool d'entrée tombe en-deça de ce seuil, les transferts d'entropie entre le pool d'entrée et le pool de sortie sont interrompus. C'est là que se

joue la différence entre le pool de sortie bloquant et le pool de sortie non-bloquant (et donc entre `/dev/random` et `/dev/urandom`) :

- Le pool bloquant, comme son nom l'indique… se bloque : il ne produira plus rien tant que les transferts d'entropie depuis le pool d'entrée n'auront pas repris (ce qui ne pourra arriver que lorsque le pool d'entrée aura lui-même reçu de l'entropie en provenance des différentes sources du système).

- Dans le cas du pool non-bloquant en revanche, même en l'absence d'octets arrivant du pool d'entrée, la fonction de mixage continue de brasser le contenu existant du pool, autant de fois que nécessaire jusqu'à ce que le pool ait produit le nombre demandé d'octets pseudo-aléatoires.

En quelque sorte, la différence clef entre le pool bloquant et le pool non-bloquant est que la fonction de mixage du pool non-bloquant peut tourner « à vide » (mixant seulement les octets déjà présents dans le pool), alors que la fonction de mixage du pool bloquant ne fonctionne qu'en recevant des octets en provenance du pool d'entrée.

3.3. Sources d'entropie

Les sources d'entropie qui alimentent le pool d'entrée sont représentées par une série de fonctions `add_`*`XXXX`*`_randomness()`, que le pilote *random* met à disposition des autres composants du noyau.

Ces fonctions sont :

- `add_input_randomness()`, appelée par le sous-système gérant les entrées des utilisateurs à chaque évènement en provenance des périphériques d'entrée (typiquement, un appui sur une touche du clavier ou un mouvement de la souris) :

- `add_interrupt_randomness()`, appelée par le gestionnaire d'interruptions à chaque interruption matérielle ou logicielle ;

- `add_disk_randomness()`, appelée par le sous-système gérant les périphériques de stockage à la fin de chaque opération de lecture ou d'écriture sur un disque.

Chacune de ces fonctions ajoute au pool d'entrée de l'entropie basée sur le moment auquel la fonction est appelée (exprimé à la fois en nanosecondes, en nombres de cycles, et en nombres d'interruptions, depuis le démarrage de la machine) et sur certaines propriétés de l'évènement sous-jacent (par exemple le code de la touche clavier ou le numéro de l'exception).

Une autre fonction, `add_device_randomness()`, permet à n'importe quel pilote de périphérique de contribuer au pool en utilisant des données propres au périphérique dont ce pilote a la charge. Un pilote appelle typiquement cette fonction une seule fois, lors de son initialisation. Contrairement aux fonctions précédentes, `add_device_randomness()` ne change pas le compteur d'entropie associé au pool d'entrée — les octets collectés par cette fonction sont mixés au pool mais « ne comptent pas » pour l'estimation de l'entropie contenue dans le pool.

Enfin, en-dehors du noyau, un programme en espace utilisateur peut contribuer au pool d'entrée via un appel `ioctl(RNDADDENTROPY)` sur un descripteur de fichier ouvert sur `/dev/random` ou `/dev/urandom`. Il est aussi possible d'écrire simplement dans ces fichiers, mais dans ce cas : 1) les données écrites sont mixées directement dans les pools de sortie, sans contribuer au pool d'entrée (contrairement à l'appel `ioctl()`), et 2) conséquemment, le compteur d'entropie du pool d'entrée n'est pas affecté.

4. Évolution du pilote *random*

Cette section retrace brièvement les principaux changements dans l'interface ou l'implémentation du pilote *random* à partir du noyau 3.17 jusqu'à la forme actuelle (versions 5.6 et ultérieures).[4]

4.1. Linux 3.17 : getrandom() et add_hwgenerator_randomness()

La version 3.17 du noyau (publiée en octobre 2014), si elle ne change pas fondamentalement l'architecture décrite ci-dessus, apporte deux changements significatifs : le nouvel appel système `getrandom()`, et l'utilisation directe par le noyau des HWRNG éventuellement présents sur le système.

4.1.1. L'appel système `getrandom()`

Avant cette version, une application en espace utilisateur souhaitant utiliser le RNG du système avait le choix entre lire depuis `/dev/random` ou depuis `/dev/urandom`. L'utilisation de `/dev/urandom` était la méthode recommandée, mais comportait un défaut : comme le pool non-bloquant ne bloque jamais quelque soit l'état du pool d'entrée, il y avait un risque, dans les premiers instants après le démarrage du système, que le générateur n'ait pas été initialisé avec suffisamment d'entropie. Utiliser `/dev/random` éliminait ce risque, mais au prix de potentiellement bloquer l'application même si le générateur avait déjà été initialisé avec bien assez d'entropie.

Il manquait donc une sorte de compromis entre `/dev/urandom`, qui renvoie sans rougir des nombres pseudo-aléatoires provenant d'un générateur potentiellement non-initialisé, et `/dev/random`, qui même avec un générateur pleinement initialisé refuse obstinément de renvoyer quoi que ce soit si l'entropie disponible à un instant donnée est jugée trop basse.

Le compromis a pris la forme d'un nouvel appel système appelé `getrandom()`, qui est désormais la méthode recommandée pour accéder au RNG.

Par défaut, `getrandom()` renvoie des octets pseudo-aléatoires en provenance du pool non-bloquant et est donc équivalent à une lecture depuis `/dev/urandom`, à cette différence près que l'appel bloquera si, depuis le démarrage du système, l'entropie disponible dans le pool d'entrée n'a *jamais* atteint un certain seuil. De cette manière, `getrandom()` ne renvoie jamais des nombres pseudo-aléatoires venant d'un générateur insufisamment initialisé, tout en ne bloquant concrètement presque jamais.

Appelé avec le drapeau GRND_RANDOM, `getrandom()` puise dans le pool bloquant et est de fait exactement équivalent à une lecture depuis `/dev/random`, le seul intérêt étant alors de simplifier le code (un seul appel à `getrandom()` remplaçant trois appels successifs à `open()`, `read()`, et `close()`) et de faire l'économie d'un descripteur de fichier.

4.1.2. Utilisation d'un HWRNG comme source d'entropie

Il a été mentionné plus haut que le noyau n'utilise pas directement les HWRNG périphériques de la machine, et que le seul propos du sous-système *hw_random* est d'exposer ces HWRNG à l'espace utilisateur.

Si la machine dispose d'un HWRNG périphérique, il peut être parfaitement raisonnable de l'utiliser pour contribuer à remplir le pool d'entropie. À cet effet, le projet rng-tools [https://github.com/nhorman/rng-tools] fournissait le démon en espace utilisateur rngd. Ce démon obtenait régulièrement des nombres aléatoires depuis le fichier `/dev/hwrng`

[4] Pour aller (beaucoup) plus loin, on pourra se référer aux différents articles de LWN [https://lwn.net/Kernel/Index/#Random_numbers] sur le sujet, qui décrivent l'évolution du RNG du noyau de manière beaucoup plus détaillée.

(donc, en provenance directe du HWRNG) et les envoyait vers le pool d'entrée, via l'appel `ioctl(RNDADDENTROPY)` mentionné plus haut.

La version 3.17 du noyau supprime ce passage obligé par l'espace utilisateur, en ajoutant au pilote *random* une source d'entropie supplémentaire sous la forme d'une fonction `add_hwgenerator_randomness()`. Lors de l'initialisation du sous-système *hw_random*, un *thread* noyau est créé, qui se charge d'appeler régulièrement cette fonction en lui passant des nombres aléatoires en provenance du HWRNG. Ce *thread* remplace ainsi le démon rngd (le *thread* est d'ailleurs démarré par une fonction appelée `start_khwrngd()`, traduisant le fait qu'il s'agit d'une version *in-kernel* de rngd).

Le démon rngd peut obtenir des octets aléatoires depuis un fichier arbitraire eu lieu et place de `/dev/hwrng` et n'est donc pas complètement obsolète — il peut toujours être utilisé pour alimenter le pool d'entropie à partir d'une source non gérée par *hw_random*, comme par exemple un périphérique NeuG. Mais l'utiliser pour lire depuis `/dev/hwrng`, ce qui est son comportement par défaut, est bien désormais inutile puisque redondant avec ce que le noyau fait déjà automatiquement.

Il s'agit de la *seule* utilisation que le noyau fait des HWRNG périphériques, dont l'usage est à part ça réservé à l'espace utilisateur.

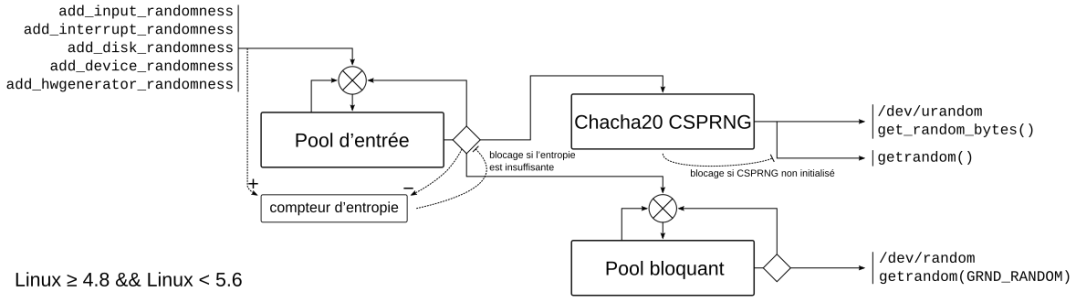
4.2. Linux 4.8 : remplacement du pool non-bloquant

La version 4.8 du noyau (publiée en octobre 2016) introduit un changement majeur dans le pilote *random* : le pool non-bloquant est supprimé et remplacé par un CSPRNG basé sur l'algorithme de chiffrement par flux ChaCha20 [http://cr.yp.to/chacha/chacha-20080128.pdf].

L'architecture générale du pilote reste inchangée, le nouveau CSPRNG prenant simplement la place du pool non-bloquant (Figure 2). Il est périodiquement ré-initialisé à partir du contenu du pool d'entrée et produit autant d'octets pseudo-aléatoires que nécessaire sans jamais bloquer.

L'interface en espace utilisateur est inchangée également, y compris dans le fait que `/dev/urandom` peut sans aucune honte renvoyer des octets pseudo-aléatoires alors que le CSPRNG n'a pas été initialisé avec suffisamment d'entropie. L'appel système `getrandom()`, lui, bloquera si le CSPRNG n'a pas reçu au moins 128 bits d'entropie en provenance du pool d'entrée, après que le CSPRNG est quoi qu'il arrive considéré comme suffisamment initialisé pour tous les besoins futurs (ce qui n'empêche pas qu'il sera périodiquement ré-initialisé quand même).

Figure 2. Architecture du RNG à partir de Linux 4.8



4.3. Linux 5.4 : initialisation rapide du pool par *jitter entropy*

La version 5.4 du noyau (publiée en novembre 2019) introduit un changement dans la manière de remplir le pool d'entrée au démarrage du système. Le but est de collecter *ra-*

pidement les 128 bits d'entropie nécessaire pour initialiser le CSPRNG ChaCha20 — plus rapidement que les sources d'entropie listées en Section 3.3 ne le permettent —, de manière à éviter à `getrandom()` de bloquer trop longtemps.

Des améliorations récentes du système de fichiers *ext4* avaient permis de réduire drastiquement le nombres d'opérations d'entrées/sorties au démarrage du système — réduisant malheureusement au passage la quantité d'entropie collectée par la fonction `add_disk_randomness()`, au point sur certains systèmes de bloquer le lancement d'une session graphique.

L'approche retenue, déjà suggérée depuis 2013 [http://www.chronox.de/jent/doc/CPU-Jitter-NPTRNG.html] et aussi implémentée en espace utilisateur par le démon haveged [http://www.issihosts.com/haveged/index.html], repose sur le fait que la durée d'exécution d'une séquence d'instructions sur un processeur moderne est imprévisible et non-reproductible, et peut donc faire office de source d'entropie (appelée *jitter entropy*).

Concrètement, le noyau appelle en boucle l'instruction `RDTSC`, qui obtient du processeur le *Time Stamp Counter*. Ce compteur donne le nombre de cycles écoulés depuis le démarrage de la machine. Il est donc nécessairement différent (il contient une valeur plus élevée) à chaque appel, et l'ampleur de l'incrément entre chaque appel est imprévisible.

Avec cette méthode, les 128 bits d'entropie nécessaire pour initialiser le CSPRNG peuvent être obtenus en une seconde au maximum dans le pire des cas (en absence de toute autre source d'entropie utilisable).

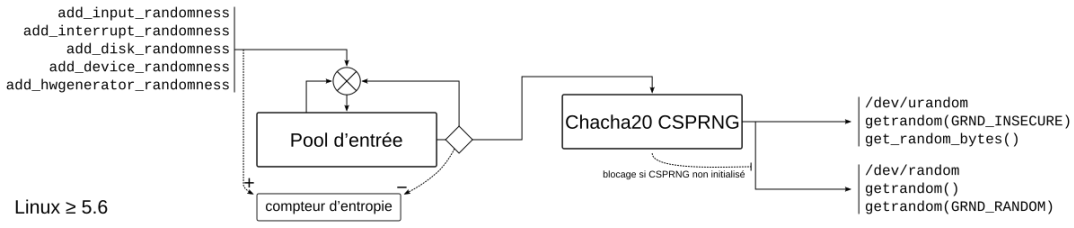
Le noyau compte néamoins toujours sur les sources d'entropie « classiques » pour remplir le pool d'entrée : cette méthode n'est utilisée que si elle est nécessaire, c'est-à-dire si `getrandom()` est appelée alors que le CSPRNG n'a pas encore été initialisé — elle assure alors que l'appel ne bloquera pas pour plus d'une seconde.

4.4. Linux 5.6 : suppression du pool bloquant

Le dernier changement en date dans le pilote *random*, introduit avec la version 5.6 du noyau (publiée en mars 2020), est particulièrement significatif puisqu'il s'agit de la suppression pure et simple du pool bloquant, qui depuis le début était un aspect caractéristique du RNG de Linux.

En conséquence, le RNG a désormais une architecture beaucoup plus simple (Figure 3), seulement constituée du pool d'entropie (toujours appelé le « pool d'entrée » dans le code, mais cette dénomination ne veut plus dire grand'chose maintenant qu'il n'y a plus de pools dits « de sortie »), qui continue à collecter l'entropie en provenance des différentes sources, et du CSPRNG basé sur ChaCha20 introduit dans le noyau 4.8, qui est toujours périodiquement ré-initialisé depuis le pool d'entropie.

Figure 3. Architecture du RNG à partir de Linux 5.6



La raison derrière la suppression du pool bloquant est principalement que celui-ci n'avait plus vraiment de raison d'être depuis l'introduction du CSPRNG ChaCha20, dont la qualité est suffisante pour répondre à tous les besoins en matière de nombres aléatoires, y compris pour la génération de clefs cryptographiques de long terme. Il devenait difficile de justifier son maintien et le maintien de tout le code associé.

L'interface en espace utilisateur est préservée. Le périphérique `dev/random`, qui puisait dans le pool bloquant, est maintenant connecté au CSPRNG au même titre que `/dev/urandom`. La différence entre les deux périphériques devient la même que celle qui existait déjà entre `/dev/urandom` et `getrandom()` : `/dev/urandom` ne bloque jamais même si le CSPRNG n'a pas été initialisé, alors que `/dev/random` est susceptible de bloquer une fois, le temps d'initialiser correctement le CSPRNG (en faisant si nécessaire intervenir le mécanisme d'initialisation rapide décrit en section précédente) — après quoi `/dev/random` ne bloquera plus jamais.

Le drapeau GRND_RANDOM de `getrandom()`, qui instruisait la fonction de puiser dans le pool bloquant, n'a plus lieu d'être. Il est maintenu pour ne pas casser le code existant, mais n'a plus d'effet. Un nouveau drapeau GRND_INSECURE est introduit à la place, qui instruit `getrandom()` de ne *pas* bloquer même si le CSPRNG n'a pas été initialisé ; l'utilisation de ce drapeau rend un appel à `getrandom()` strictement équivalent à une lecture depuis `/dev/urandom`.

On voit mal l'intérêt de ce drapeau, qui est contraire à la raison d'être même de `getrandom()`. Tout l'objet de cet appel système est de donner au code appelant l'assurance de n'obtenir que des nombres aléatoires provenant d'un générateur proprement initialisé…

5. Utilisation du HWRNG architectural

On a mentionné plus haut que le noyau Linux distinguait les HWRNG *périphériques*, exposés à l'espace utilisateur via le sous-système *hw_random* (`/dev/hwrng`), et l'éventuel HWRNG dit *architectural*, directement implémenté dans le processeur. Cette section décrit comme le noyau utilise ce dernier, s'il existe.

Il ne sera question ici que de l'architecture X86, autrement dit des instructions `RDRAND` et `RDSEED` introduites par Intel (sous le nom *Secure Key*) à partir de 2012 et par la suite reprises par AMD. Je n'ai pas particulièrement regardé ce que faisait le noyau sur les autres architectures offrant un HWRNG directement dans le processeur, ni mêmes quelles autres architectures offraient cette fonctionnalité. Selon la formule consacrée, « cela est laissé en exercice aux lectrices. »

Préalablement, un mot sur la différence entre `RDRAND` et `RDSEED`. Le HWRNG des processeurs Intel [https://software.intel.com/content/www/us/en/develop/articles/intel-digital-random-number-generator-drng-software-implementation-guide.html#inpage-nav-3] est composé d'un TRNG (basé sur le bruit thermique à l'intérieur de la puce) qui alimente un CSPRNG basé sur AES-256 (*CTR_DRBG*, tel que défini dans le standard NIST SP800-90A [http://dx.doi.org/10.6028/NIST.SP.800-90Ar1]). L'instruction `RDRAND` renvoie la sortie du CSPRNG, tandis que l'instruction `RDSEED` renvoie la sortie du TRNG (l'implémentation des processeurs AMD est similaire [https://www.amd.com/system/files/TechDocs/amd-random-number-generator.pdf]). `RDSEED` est supposément plus adaptée pour initialiser un CSPRNG en aval, là où `RDRAND` est plutôt supposée fournir des nombres aléatoires directement utilisables par le code appelant.

Le pilote *random* fait une utilisation assez intensive de `RDSEED/RDRAND`, saisissant la moindre opportunité de mixer la sortie de ces instructions au flux normal de l'entropie. Précisément, `RDSEED` ou `RDRAND` sont appelées aux occasions suivantes :

- lors de l'initialisation du pilote, le pool d'entrée est rempli avec des valeurs extraites de `RDSEED` (si disponible) ou `RDRAND` ;
- chaque fois que le CSPRNG ChaCha20 est ré-initialisé, la graine extraite du pool d'entropie est XORée avec une valeur extraite de `RDSEED` ou `RDRAND` ;
- chaque fois qu'une valeur aléatoire est extraite du CSPRNG ChaCha20, l'état interne du générateur est partiellement XORé avec une valeur extraite de `RDRAND` ;

- chaque fois que la fonction `add_interrupt_randomness()` est appelée par le gestionnaire d'interruption pour contribuer au pool d'entropie, une valeur extraite de `RDSEED` est mixée au pool (en plus des octets tirés de l'interruption elle-même) ;

- chaque fois que de l'entropie est extraite d'un pool (le pool d'entrée, et les pools de sortie dans les versions du noyau où ils existent encore), une valeur extraite de `RDRAND` est utilisée comme vecteur d'initialisation pour calculer le condensat SHA-1 sur le contenu du pool ;

- chaque fois que le pool est alimenté depuis l'espace utilisateur (via l'appel `ioctl(RNDADDENTROPY)` ou une écriture sur `/dev/random` ou `/dev/urandom`), les octets mixés au pool sont XORés avec une valeur extraite de `RDRAND`.

Dans toutes les utilisations ci-dessus, les valeurs extraites de `RDRAND` ou `RDSEED` ne sont pas considérées comme ajoutant de l'entropie. En particulier, même si le pool d'entrée est complètement rempli avec des valeurs provenant de ces instructions lors de l'initialisation du pilote, cela n'a aucun effet sur le compteur d'entropie associé au pool qui à ce moment-là est considéré comme « vide » d'entropie. La seule exception à cette règle est l'appel au sein de la fonction `add_interrupt_randomness()`, dont le noyau considère qu'il ajoute *un* bit d'entropie au pool.

Une autre utilisation de ces instructions est conditionnée à l'option de configuration `RANDOM_TRUST_CPU`. Si le noyau est compilé avec cette option, `RDSEED` (si disponible, sinon `RDRAND`) est utilisée pour initialiser immédiatement le CSPRNG ChaCha20, indépendamment du pool d'entropie. Cela permet au CSPRNG d'être utilisable au plus tôt, même sans recourir au mécanisme d'initialisation rapide (*jitter entropy*) décrit en Section 4.3. Ce comportement peut être désactivé à l'exécution, sans avoir à recompiler le noyau, en passant en paramètre au noyau l'option `random.trust_cpu=off`.

Toutes les utilisations de `RDRAND` et `RDSEED` sont conditionnées à l'option `nordrand` : si cette option est passée en paramètre au noyau au démarrage, `RDRAND` et `RDSEED` ne sont *jamais* utilisées ; cette option prévaut sur `RANDOM_TRUST_CPU` (du point de vue du noyau c'est comme si le processeur ne supportait pas ces instructions).

6. Vu depuis le code en espace utilisateur

En conclusion, quelles sont les options pour du code en espace utilisateur ayant besoin de nombres aléatoires ?

L'option la plus portable (vis-à-vis du matériel) est évidemment d'utiliser le RNG du système. À cet effet, `getrandom()` est l'interface recommandée, vu qu'elle évite complètement d'avoir à se soucier de l'état (initialisé ou non) du générateur. Comme mentionné plus haut elle a été introduite en 2014 (noyau 3.17), et la glibc fournit un *wrapper* depuis février 2017 (glibc 2.25). Si pour une raison ou une autre `getrandom()` n'est pas souhaitable (glibc trop ancienne ?), l'ancienne recommandation d'utiliser `/dev/urandom` est toujours valable, *sauf* si votre code est destiné à être appelé très tôt au cours de la vie du système (potentiellement avant que le CSPRNG ne soit initialisé) — exactement le cas de figure pour lequel `getrandom()` a été inventée…

Pour une portabilité vis-à-vis du *système*, on pourra éventuellement préférer la fonction `getentropy()`, disponible sur les systèmes *BSD mais pour laquelle la glibc fournit un *wrapper* compatible basé sur l'appel système `getrandom()`.

Pour une portabilité allant au-delà de Linux et des *BSD, on pourra utiliser une bibliothèque cryptographique tierce, comme par exemple libgcrypt [https://gnupg.org/software/libgcrypt/index.html], qui fournit une fonction `gcry_randomize()` ; par défaut, cette fonction utilise son propre CSPRNG, initialisé à partir du RNG du système (via `getrandom()`, sous Linux et si disponible).

À l'inverse, si la portabilité n'est pas requise (même pas vis-à-vis du matériel), on peut choisir de se passer complètement du RNG du système et soit lire depuis `/dev/hwrng`, soit utiliser `RDRAND`.[5] Pour ce dernier cas, GCC fournit depuis sa version 4.6 l'option `-mrdrnd`, qui rend disponible la fonction intrinsèque `__builtin_ia32_rdrand32_step()`. Cette fonction est aussi disponible sous le nom `_rdrand32_step()` (via une macro définie dans `immintrin.h`), qui a l'avantage d'être compatible avec les compilateurs d'Intel et de Microsoft.

A. À propos de ce document

[5] Dans les deux cas cela implique évidemment de faire totalement confiance au RNG matériel sous-jacent, là où le RNG du système réduit ce besoin de confiance puisqu'il se repose toujours sur *plusieurs* sources d'entropie (hors le cas de `RANDOM_TRUST_CPU`).