Device files /dev/random e /dev/urandom

Analisi di algoritmi e implementazione

Emanuele Paracone
emanuele.paracone@gmail.com

4 maggio 2015

Il nostro lavoro e LRNG

In questo lavoro presentiamo l'analisi degli algoritmi e dell'implementazione dei device file **/dev/random** e **/dev/urandom**. Tale elaborato è previsto a conclusione del corso di ***Linux Avanzato*** per l'*A. A. 2013-2014*

Il nostro obiettivo è fornire un'analisi del *Linux Random Number Generator* (*LRNG*), il generatore di numeri pseudocasuali del kernel Linux, attraverso una descrizione delle sue struttura e implementazione. Nello specifico, abbiamo analizzato la versione del kernel 3.14.4.

Una panoramica su LRNG...

L'implementazione del *PseudoRandomNumberGenerator* (PRNG) di Linux risiede sostanzialmente nelle $\sim$ 1700 righe di codice del file sorgente `drivers/char/random.c`, scritto da Matt Mackall e Theodore Ts'o ([MT14]).

Il LRNG è oggetto di particolare interesse per il ruolo che ricopre in applicazioni relative alla sicurezza per le quali è cruciale l'uso di un generatore di numeri pseudocasuali di buona qualità.

A dispetto di quanto ci si aspetterebbe, sebbene il LRNG sia stato inserito nel kernel nel 1994, non abbiamo trovato sue documentazioni o descrizioni in letteratura antecedenti l'articolo di Zvi Gutterman, Benny Pinkas, and Tzachy Reinman *Analysis of the Linux random number generator* [ZGR06], pubblicato nel 2006 e con oggetto la versione del kernel 2.6.10. In questo lavoro faremo spesso riferimento all'articolo di Gutterman e, soprattutto, al più recente *The linux pseudorandom number generator revisited* di Patrick Lacharme, Andrea Röck, Vincent Strubel, and Marion Videau [PLV12] pubblicato nel 2012, che effettua una nuova analisi su un kernel più recente (2.6.30.7) e suggerisce alcune modifiche dell'implementazione che sono state accolte nelle versioni attuali del kernel.

LRNG nel kernel 3.14.4

La nostra analisi si concentrerà sulle caratteristiche funzionali del LRNG e sulla sua implementazione. Nelle sezioni che seguono diamo una descrizione delle sue:

1. struttura;
2. gestione degli input di entropia esterna;

3. produzone degli output;
4. implementazione dei requisiti di sicurezza;
5. storia ed evoluzione recente.

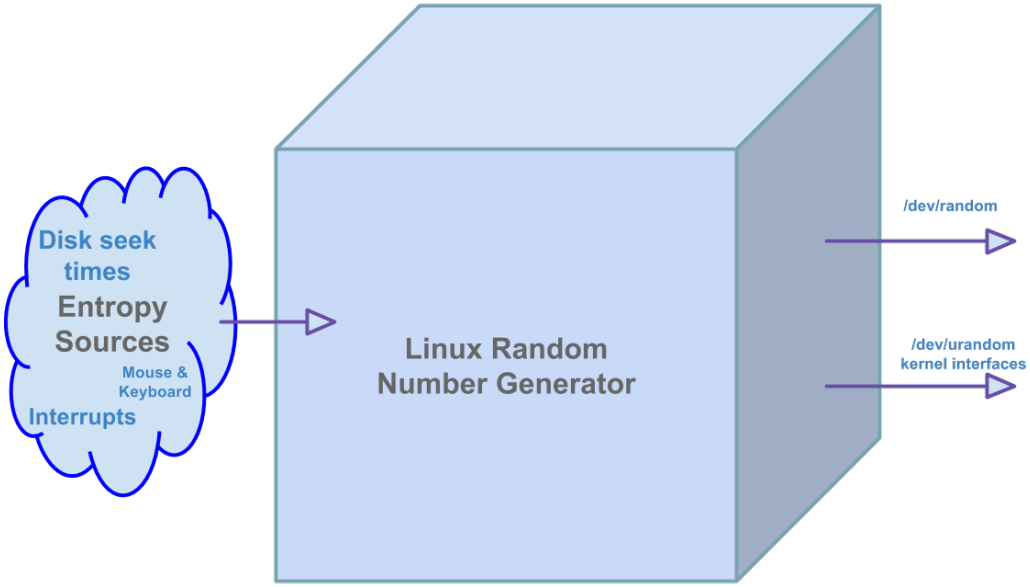
A corredo del lavoro presente esponiamo il codice sorgente di `random.c` su **github** [Par15] con i nostri commenti al codice (che si distinguono da quelli originali perché riportati in lingua italiana).

1 La Struttura del LRNG

1.1 Un po' di notazione

Gli *Entropy Inputs*

Il PRNG di Linux rientra nella classificazione di *Pseudorandom Number Generator* con *entropy inputs*, ovvero è un generatore di numeri pseudocasuali i cui bit di output sono prodotti in modo non deterministico per l'introduzione di input da sorgenti esterne di entropia (**entropy inputs**). Detti input aggiornano il generatore introducendo impredicibilità nei possibili valori che può assumerne lo **stato interno**.



L'Entropia

Poiché si richiede che i numeri generati abbiano una buona "qualità", l'impredicibilità introdotta dagli entropy input deve poter essere misurata per garantire un buon livello di randomicità dei bit di output. D'ora in avanti utilizzeremo il termine **entropia** per indicare la metrica che misura il livello di randomicità che viene "aggiunto" allo stato interno del generatore a seguito di un input di entropia.

I tipi di *Entropy Inputs*

Gli entropy inputs possono essere di tre tipi:

- **user input**: input provenienti da mouse e tastiera;
- **disk I/O**;
- **interrupts**.

L'entropia di un'entropy input è costituita dal valore dello specifico input e dal tempo intercorso tra l'input presente e l'ultimo input dello stesso tipo, misurato sia in jiffies che in cicli di CPU.

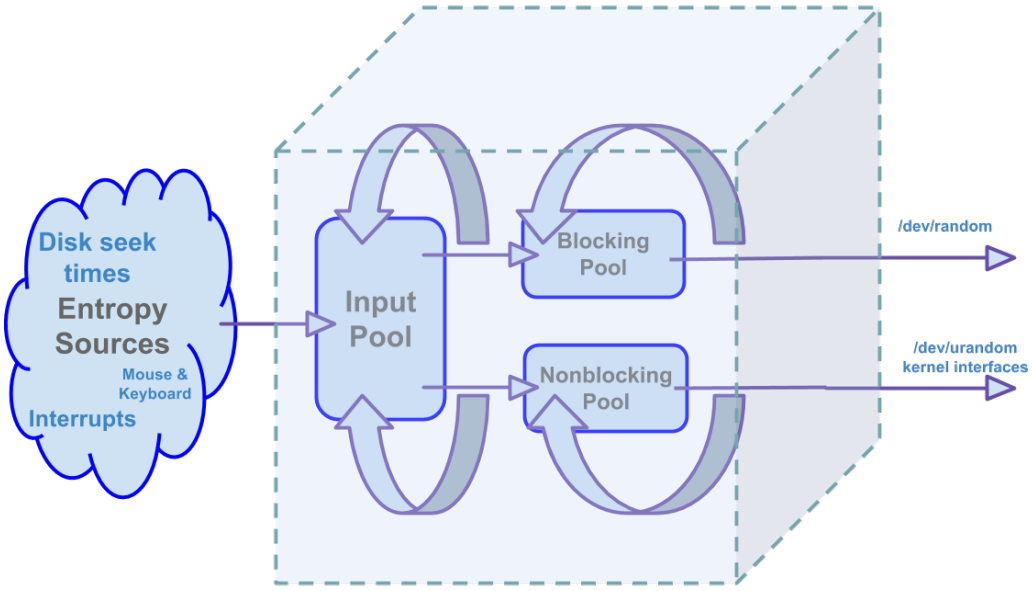
Gli *entropy inputs* hanno in comune la caratteristica di essere:

- non deterministici;
- difficili da misurare per un osservatore esterno

e aggiungono entropia al generatore in modo asincrono e indipendente rispetto alla generazione degli output.

1.2 Struttura Generale

Il cuore del LRNG è un buffer protetto, cui da ora in avanti ci riferiremo col nome di **stato interno**, inaccessibile per via diretta. Il LRNG produce i propri output a partire dai valori contenuti nel suo stato interno. Conseguentemente, lo stato interno deve essere aggiornato non solo attraverso la raccolta di quanti più *entropy input* possibile, ma anche ogni volta che viene richiesto un buffer in output al LRNG (mediante un opportuno meccanismo di feedback).



La sicurezza del generatore, ovvero l'impossibilità di prevedere gli output futuri (*backward security*) una volta che siano noti gli *entropy input* e l'impossibilità di ricostruire gli output passati (*forward security*) a

partire dalla conoscenza dello stato interno, è descritta nella sezione 4 riguardante la sicurezza del LRNG.

In quel che segue della presente sezione diamo una breve descrizione dello stato interno del generatore e delle sue interfacce.

1.2.1 Lo Stato Interno e le pool

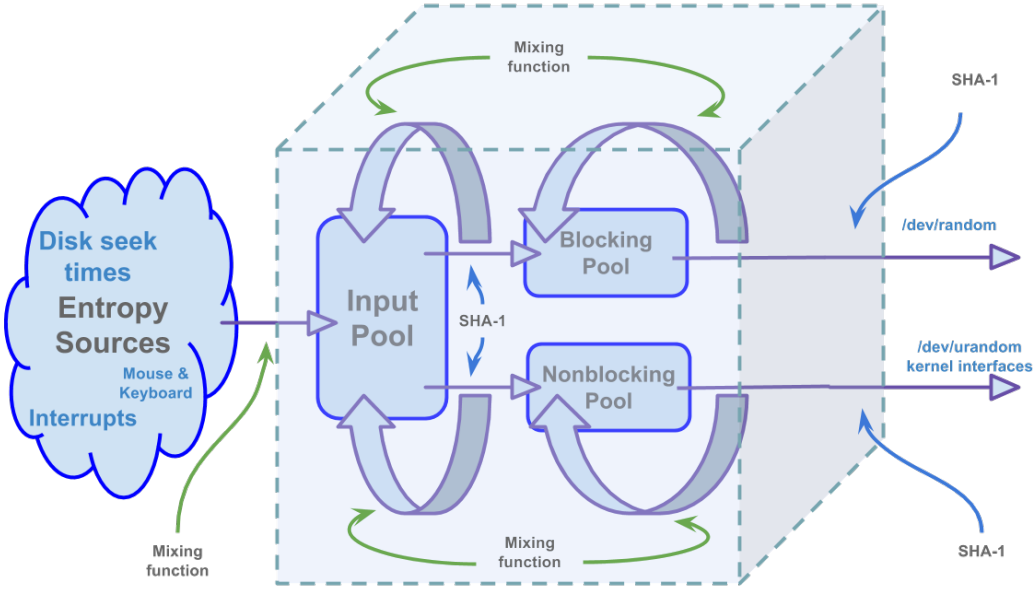
Lo stato interno del LRNG è rappresentato, all'interno del kernel, da tre strutture dette **pool**, cui nel codice viene fatto riferimento come:

- **input pool**
- **blocking pool**
- **nonblocking pool**

L'entropia aggiunta al generatore dagli *entropy inputs* viene accumulata nella *input pool* (chiamata anche *pool* primaria in [ZGR06]). L'entropia viene poi trasferita verso una delle altre due pool, *blocking* o *nonblocking*, generalmente (ma non solo!) in occasione del prelievo di bytes randomici in output dal LRNG e nella quantità strettamente necessaria. Per questo le pool *blocking* e *nonblocking* sono anche dette **output pool** o *pool* secondarie (rispettivamente in [PLV12] e [ZGR06]). Tratteremo nel dettaglio la struttura delle tre pool in 1.3.

1.2.2 La gestione dell'entropia

Il LRNG si comporta in modo diverso nel gestire il mantenimento dell'entropia interna a seconda che questa venga aggiunta o estratta rispetto alle *pool*.
Quando viene immessa nelle pool dell'entropia derivata dagli *entropy input* o proveniente dalla *input pool*, il kernel usa una mixing function lineare per efficientare l'accumulazione di entropia nell'*input pool* (viene preferita pertanto una CRC all'uso di una funzione hash sebbene la prima non sia crittograficamente robusta), mentre, quando viene trasferita entropia dalla input pool verso una output pool o vengono generati bit dalle output pool, il LRNG fa uso della primitiva crittografica SHA-1.
È facile convincersi dei benefici introdotti da questa disparità, in quanto l'aggiunta di entropia rimane un'operazione veloce nonostante gli *entropy input* abbiano un tasso di arrivo elevato (e maggiore di vari ordini di grandezza rispetto al tasso con cui mediamente sono richiesti gli output).
Anche se ci proponiamo di definire più avanti il concetto di "sicurezza" riferito al LRNG, vogliamo già dare risalto al fatto che la segretezza dello stato interno risieda fondamentalmente nell'adozione di SHA-1 per la produzione degli output.



Ogni volta che viene aggiunta dell'entropia presso una pool, viene aggiornato un contatore specifico per quella pool: l'**entropy counter**. La quantità per la quale viene incrementato il contatore è pari alla stima di entropia calcolata sul particolare *entropy input* corrente (che come abbiamo già visto si basa sul valore dell'input e sul tempo di interarrivo tra questo e l'input dello stesso tipo che lo ha preceduto).

1.2.3 Le interfacce di output

Il LRNG espone due interfacce per il prelievo di numeri pseudo-randomici dallo spazio utente:

- il device file `/dev/random`: espone verso l'user space la funzione di generazione di output dalla *blocking pool*;
- il device file `/dev/urandom`: espone verso l'user space la funzione di generazione di output dalla *nonblocking pool*;

e alcune interfacce kernel:

- il simbolo `get_random_bytes()`: l'interfaccia kernel principale, esporta verso il kernel space una funzione per la produzione di bytes generati dalla *nonblocking pool*;
- il simbolo `get_random_int()`: esporta verso il kernel space una funzione per la produzione di bytes pseudocasuali crittograficamente non sicuri indipendenti dalle *pool* (l'estrazione di valori siffatti non diminuisce l'ammontare di entropia dello stato intereno);
- il simbolo `get_random_bytes_arch()`: esporta verso il kernel space un'interfaccia per l'uso di un eventuale hardware specifico per la generazione di numeri pseudocasuali;
- il simbolo `generate_random_uuid()`: esporta verso il kernel un generatore casuale di uuid che sfrutti il LRNG.

1.3 Le Pool

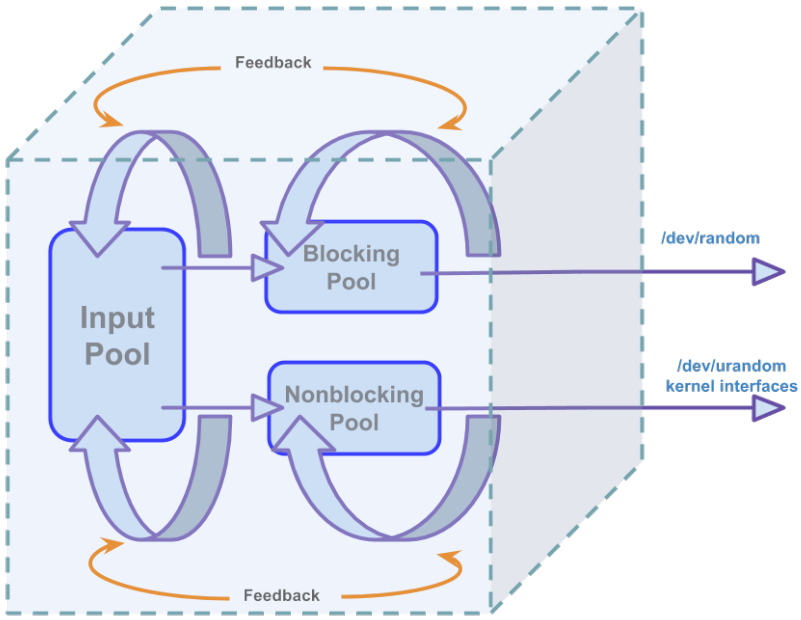
Poiché abbiamo visto come l'entropia del *LRNG* sia collezionata per mezzo delle tre pool *input*, *blocking* e *nonblocking*, non ci resta che analizzare quale sia il ruolo di queste e come concorrano nell'implementare lo stato interno.

1.3.1 A che servono le pool?

Sappiamo già che la componente più profonda dello stato interno, ovvero quella che colleziona l'entropia proveniente dagli *entropy input*, risieda nell'*input pool*: il contenuto informativo di ogni *entropy input* viene "mischiato" direttamente con il buffer in essa contenuto. Quando vengono richiesti bytes pseudocasuali al LRNG, questi vengono estratti dalla *output pool* specifica per l'interfaccia invocata (1.2.3).

I bytes prodotti a partire dalla *blocking pool* (ovvero ottenuti da `/dev/random`) hanno un'elevata qualità entropica (intesa in funzione di quanto la pool che li ha generati sia stata aggiornata con eventi non predicibili e quindi di elevato contenuto entropico), mentre tale qualità non è garantita per quelli prodotti dalla *nonblocking pool* (ottenibili tamite l'uso delle altre interfacce).
I primi, infatti, sono generati solo quando vi è sufficiente *entropia* collezionata all'iterno del LRNG: se l'entropia stimata presso lo stato interno del generatore è inferiore al numero di bytes richiesti in output, il flusso prodotto da `/dev/random` si blocca fino all'"arrivo" di nuova entropia in quantità sufficiente presso l'*input pool*.
I secondi sono ottenuti in modo non bloccante a prescindere dalla quantità di entropia stimata per lo stato interno: in mancanza di entropia sufficiente dalla *input pool*, la *nonblocking pool* produce ugualmente bytes in output via SHA-1 a partire dal proprio stato interno. Notiamo come i bytes prodotti in modo non bloccante siano comunque idonei per la grande maggioranza delle applicazioni: un attaccante in possesso di vecchi output dovrebbe aver effettuato una crittoanalisi di SHA-1 che gli consenta di prevedere futuri output a partire da vecchi output (condizione sufficiente nel solo caso di esaurimento dell'entropia e assenza di *entropy input*) mentre ad oggi si ritiene (anche se non è stato dimostrato) che una crittanalisi di questo genere non sia possibile.

Ogni volta che viene richiesto un output (o un trasferimento di entropia) e subito prima della consegna dell'output, viene aggiornato lo stato interno della pool coninvolta attraverso un meccanismo di retroazione (o *feedback*).



Notiamo anche che non esistano interfacce kernel per i bytes prodotti dalla *blocking pool*: i bytes di alta qualità sono pertanto destinati solo per applicazioni critiche dello spazio utente.
Eventuali dispositivi hardware per la generazione di numeri randomici sono esposti sia verso l'user space – tramite il device driver `hwrng`– che verso il kernel space – tramite il simbolo esportato `get_random_bytes_arch()`.
Può verificarsi inoltre che un'applicazione nello spazio utente utilizzi gli output generati da quest'ultimo tipo di driver, se la sorgente è trusted, per aumentare l'affidabilità dei numeri generati dal *LRNG*, in quanto è possibile sommare un seme al suo stato interno.
La somma di un seme allo stato interno del generatore è utile all'aumento dell'impredicibilità dello stesso, soprattutto in fase di inizializzazione del sistema (quando gli entropy input sono predicibili e ridotti in numero) oppure nel caso di sistemi senza dischi e/o input utente. D'altro canto, l'aggiunta di un nuovo seme non aumenta il valore della stima dell'entropia interna delle varie pool.

1.3.2 Com'è strutturata una Pool

Ogni pool è modellata da una struttura `entropy_store` costituita sostanzialmente da:

- un buffer interno;
- un **entropy counter**, che memorizza una stima dell'entropia aggiunta al buffer interno a partire dagli *entropy inputs*
- un *polinomio caratteristico* per la mixing function.

Il buffer interno è il cuore della pool: costituisce il set di registri a partire dal quale vengono effettuate le operazioni di mixing necessarie all'aggiunta e al perelievo di entropia. Contiene le parole (**words**) di 16 bytes (4 unsigned a 32 bit) utilizzate dal mixing ed ha una dimensione fissata, specifica rispetto alla singola pool (128 *words* per la *input pool* e 32 per le *output pool*).
Rimandiamo il lettore alla sezione dedicata alla *mixing function* (2.4) per una descrizione del polinomio caratteristico, e di come vengono prelevati e immessi bytes di entropia presso le pool.

I principali campi di una struttura `entropy_store` sono:

Campi read-only:

- `const struct poolinfo *poolinfo`: contiene informazioni sul dimensione e configurazione della pool, compreso il suo polinomio caratteristico;
- `__u32 *pool`: punta allo stato interno della pool;
- `struct work_struct push_work`: punta alla funzione `push_to_pool()` (**??**) nel caso in cui la pool sia un'*output pool*.

Campi read-write:

- `int entropy_count`: è l'*entropy counter*, conta il numero di frazionali di bit di entropia (ottavi di bit – 2^{-3} bit) presenti nella pool (secondo quanto previsto dalla funzione di stima – 2.3);
- `unsigned short add_ptr`: contatore ciclico, indica la posizione della prossima parola della pool da trattare per le funzioni di mixing. Le parole vengono lette in ordine inverso rispetto all'ordine nel buffer;
- `unsigned short input_rotate`: numero di posizioni per cui devono essere ruotati i bit della prossima parola di entropia da aggiungere alla pool. Ad ogni accesso viene incrementato di 7 posizioni, ad eccezione del primo accesso in cui viene incrementato di 14;
- `unsigned long last_pulled`: l'ultimo istante in jiffies in cui sono stati estratti bytes dalla pool;
- `unsigned int limit`: flag di un bit impostato a 1 nel caso in cui la pool sia bloccante (*input* e *nonblocking pool*).

1.4 I Device Files /dev/random e /dev/urandom

Abbiamo affermato in 1.2.3 che vi sono due interfacce per il prelevamento dell'output a partire dallo spazio utente costituite dai due device files `/dev/random` e `/dev/urandom`. In realtà è riduttivo pensare a tali device file solo come interfacce per gli output del generatore: è infatti possibile effettuare su di essi operazioni di scrittura con lo scopo di mischiare allo stato interno un hash del valore immesso.
Le azioni specifiche per i due file sono definite nelle due strutture `file_operations random_fops` e `urandom_fops`.

1.5 Dagli *Entropy Input* all'Output

Dopo aver introdotto gli ingredienti principali che costituiscono il generatore, vediamo ora come interagiscono fra di loro.

2 L'Input

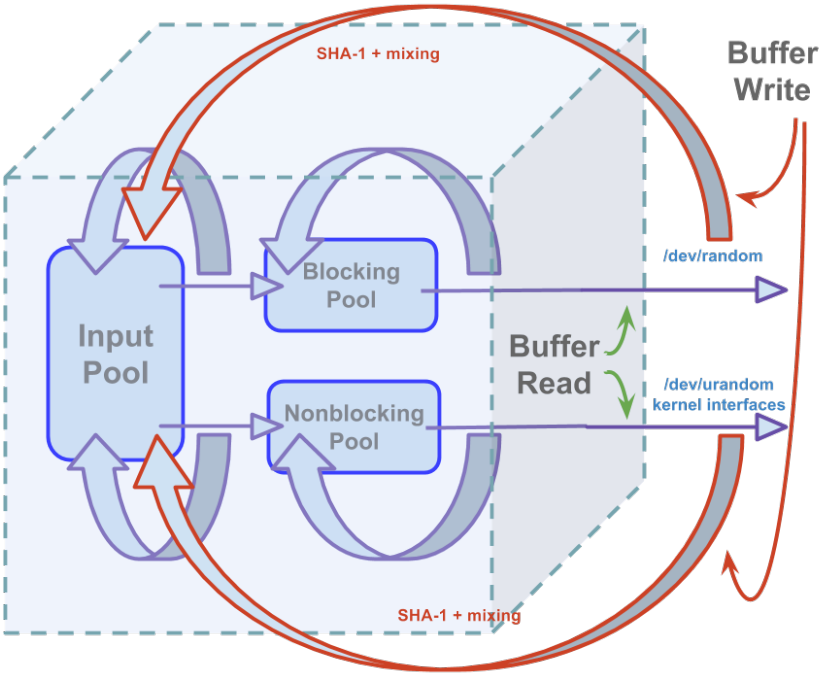
Gli input immessi nel LRNG possono essere solo di due tipi:

1. seed aggiunto al generatore direttamente dallo spazio utente (1.3.1).
2. *entropy input* (introdotti in 1.1);

Nella sezione presente analizziamo l'aggiunta di entropia presso le pool conseguente l'arrivo di nuovi *entropy input* e gli accorgimenti necessari per garantire la sicurezza del LRNG anche nel caso in cui l'entropia sia quantitativamente scarsa o nulla (per l'assenza di *input*).

2.1 Il Reseeding da Spazio Utente

Come visto in 1.4, l'operazione di scrittura di un buffer direttamente presso i device files `/dev/random` e `/dev/urandom` comporta un reseeding dello stato interno del generatore a partire da un hash del buffer stesso.



Tale operazione è incoraggiata nell'header di `random.c`, dove viene specificato che, per differenziare gli output prodotti in prossimità del momento del boot della macchina, è opportuna l'esecuzione di uno script all'avvio che scriva presso `random` o `urandom` un buffer letto dagli stessi device files prima dello shutdown della macchina. In questo modo il sistema appena avviato produrrà output differenti, rispetto a quelli prodotti a ridosso di altri startup dello stesso sistema, anche quando è stata raccolta ancora poca (o nulla) entropia (fatto inevitabile a ridosso dell'inizializzazione del sistema). Ovviamente, nel caso non siano presenti dispositivi per la memorizzazione persistente (dischi, flash, ecc) non è possibile adottare un simile accorgimento.

Vi sono altri scenari per cui può rendersi opportuno imporre un reseeding dello stato interno del generatore. Ad esempio, si può aumentare il livello di confidenza verso i valori prodotti in output da `urandom` redirigendovi gli output ottenuti a partire da un generatore di numeri randomici hardware. Nel complesso il reseeding effettuato a tempo di start-up del sistema resta comunque l'esigenza più frequente.

Quando viene scritto un buffer su `random` o `urandom`, non viene stimata dal kernel alcuna entropia, non potendo essere misurata l'impredicibilità né verificata l'affidabilità della sorgente.
`write_pool()` è la routine che si occupa di passare il buffer in input al meccanismo di mixing della *pool* specificata, processandone 16 parole da 4 bytes per volta.

2.2 Gli *Entropy Input*

Gli *entropy input* sono l'unica sorgente di casualità autentica (per quanto possibile) per il generatore: tutto il nondeterminismo introdotto nello stato interno del LRNG ha origine infatti dall'impredicibilità degli eventi esterni catturati e processati dal generatore. Eventi di questo genere devono pertanto essere, nella prospettiva di un osservatore esterno, difficili sia da rilevare che da predire, e sono causati da:

- input utente – pressione di tasti della tastiera e movimenti del mouse;

- tempi di accesso ai dischi;
- interruzioni di sistema.

Ogni *entropy input* tiene in considerazione tre valori di 4 bytes ([PLV12]):

1. un valore `num`, specifico per il particolare evento occorso (ad esempio il keycode nel caso l'evento sia un input da tastiera);
2. il numero di cicli di CPU al momento dell'evento;
3. il numero di *jiffies* al momento dell'evento.

2.2.1 Le interfacce di input

Dal 2012 (5.2) gli *entropy input* vengono immessi esclusivamente tramite la chiamata alle interfacce:

- `void add_device_randomness(const void *buf, unsigned int size);`
- `add_input_randomness(unsigned int type, unsigned int code, int value);`
- `void add_interrupt_randomness(int irq, int irq_flags);`
- `void add_disk_randomness(struct gendisk *disk)`.

Ogni interfaccia interessa specifiche sorgenti di entropia e ogni sorgente di entropia è modellata da una struct `timer_rand_state` che mantiene al suo interno:

1. l'istante, espresso in *jiffies*, in cui la sorgente ha generato l'ultimo evento;
2. i primi due ordini di delta `last_delta1` e `last_delta2`.

Il primo ordine di delta, `delta1`, è pari al tempo di interarrivo, espresso in jiffies, tra l'ultima occorrenza di un evento e quella immediatamente precedente (se esiste) di un evento dello stesso tipo.
Il secondo ordine di delta, `delta2`, è pari alla differenza tra il delta di primo ordine appena calcolato.
Da questi due ordini di delta viene poi derivato un terzo ordine `delta3` che non viene memorizzato in quanto calcolato al momento della stima dell'entropia per l'*entropy input* in analisi (2.3.2).

Vediamo nel dettaglio ciascuna delle interfacce di input (una loro descrizione è rintracciabile anche nell'header di `random.c` [MT14]).

`add_device_randomness()` serve a differenziare l'inizializzazione dello stato interno del generatore compiuta da dispositivi diversi. Sfrutta un insieme di informazioni specifiche per la macchina su cui il kernel è in esecuzione, cercando di leggere indirizzi MAC, numeri seriali e il *real-time clock* (*RTC*). È utile in particolare per differenziare gli output di dispositivi con poca entropia (caso frequente nel caso di dispositivi embedded e/o privi di dischi, schede di rete, ecc.).
Durante l'aggiornamento viene chiamata la funzione `get_cycles()`, che ritorna un valore diverso da zero solo in presenza di un *Time Stamp Counter* (*TSC*), implementato quasi in esclusiva nelle architetture `x86`. Nel caso in cui `get_cycles()` ritorni zero, l'unico contributo basato sul tempo dell'evento proviene dal valore di jiffies.

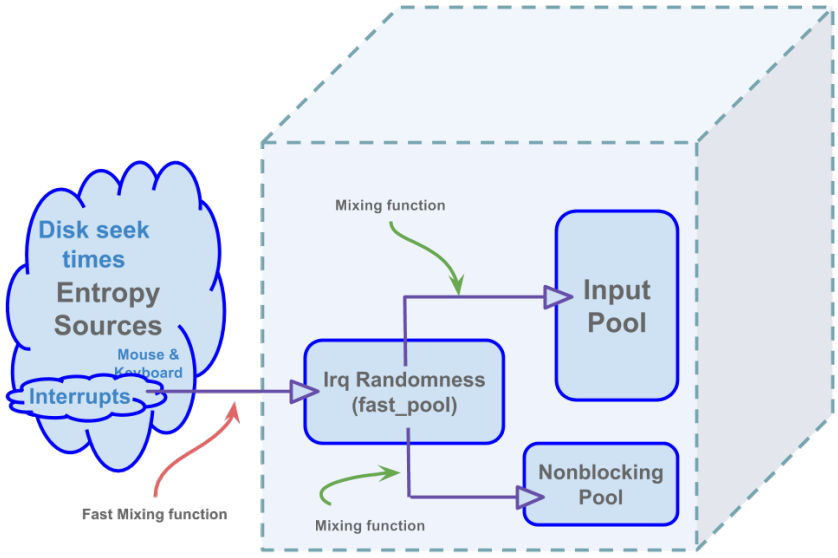
`add_input_randomness()` aggiunge entropia a partire dagli input provenienti dall'utente (mouse, tastiera, touchscreen, ecc) e agisce sulla struttura `timer_rand_state` della sorgente di entropia corrispondente, `input_timer_state`. Viene chiamata dalla routine `input_pass_values()` in `/drivers/input/input.c`, che dapprima sottopone l'input in fase di processamento ad una lista di filtri per poi, una volta superati questi, sottoporre l'input a tutti gli opportuni handler, tra cui il LRNG.
`add_input_randomness()` è parametrizzata dai tre valori

1. `type` - codifica la sorgente dell'evento di input (tastiera, mouse, touchscreen, ecc.);
2. `code` - il codice dell'input (ad esempio quale tasto della tastiera o del mouse è stato premuto o rilasciato);
3. `value` - il valore dell'input, specifica quale evento specifico è occorso per il codice code (ad esempio se il tasto è stato premuto, mantenuto o rilasciato, ecc).

L'entropia viene aggiunta allo stato interno sotto forma di un buffer ottenuto mischiando i tre parametri. Se la routine viene chiamata consecutivamente con uno stesso parametro `value`, le chiamate successive alla prima sono idempotenti, ovvero non sortiscono alcun effetto e la funzione ritorna.

`add_interrupt_randomness()` aggiunge un *entropy input* a partire da un'`irq`. L'aggiunta di entropia avviene attraverso quattro passi:

1. Viene creato un array, `input`, di 4 parole da 32 bit, ottenute attraverso un mixing elementare (xor) dei valori di *cycles* e di *jiffies*, l'identificativo dell'*irq* e il valore dell'*instruction pointer* (che punta all'istruzione successiva all'esecuzione dell'interruzione). Nel caso in cui l'architettura sia a 64 bit, *cycles* e *jiffies* vengono divisi in due valori da 32 bit ciascuno durante l'inizializzazione delle parole di *input*. *Cycles* è un valore a 64 bit (e viene pertanto diviso in due valori da 32 bit) anche nel caso in cui l'architettura non sia a 64 bit ma sia pur sempre una `x86`.
2. Viene mischiato l'array `input` alla `fast_pool` `irq_randomness`, una struttura che possiede una mini-pool di 4 valori a 32 bit inizializzata con i valori dei registri della cpu al momento della chiamata dell'interruzione. La funzione di mixing adottata è `fast_mix()`, che mischia i valori alla `fast_pool` facendo uso di una speciale tabella per il mixing `twist_table`.



Se l'ultimo interrupt visto dalla CPU è troppo recente rispetto all'*interrupt request* attuale, la routine `add_interrupt_randomness` ritorna, sortendo come unico effetto un aggiornamento di `irq_randomness`.

3. Vengono mischiati tutti i bytes della mini-pool di `irq_randomness` presso l'*input pool* (oppure presso *nonblocking pool* nel caso questa debba ancora essere inizializzata).

4. Viene accreditata una stima di entropia pari a 1 bit presso la *entropy pool* selezionata al passo 3. Tale accreditamento viene omesso nel caso in cui non vi sia un contatore di cicli valido per l'architettura specifica oppure se si osservano timer interrupt consecutivi (l'accredito è analizzato più nel dettaglio in 2.3.2).

`add_disk_randomness()` sfrutta la variabilità del *seek time* dei settori dei dischi. Dischi a stato solido sono ovviamente quasi completamente inefficaci nell'aggiungere entropia, accedendo ai blocchi di dati in tempo approssimativamente costante.
Opera sulla struttura `gendisk *disk` (definita in `include/linux/genhd.c`) specifica per il disco che ha generato l'evento. In particolare fa uso della struttura `timer_rand_state` in `disk->random` per l'aggiunta e la stima dell'entropia.

2.3 La Stima dell'Entropia

La stima dell'entropia nel LRNG ha come obiettivo di quantificare l'impredicibilità su cui si può fare affidamento al momento della generazione di bytes ed è pertanto un'attività di centrale importanza per il LRNG. Si basa su due assunzioni:

- ogni *entropy input* aggiunge una quantità finita di entropia al generatore;
- ogni output prodotto consuma una quota dell'entropia totale presente presso il generatore.

Sappiamo già (da 1.2.3 e 1.3) che sebbene il conteggio dell'entropia accumulata nelle pool (mantenuto negli *entropy counter* di ciascuna) entri in gioco per la produzione di qualsiasi output, nel caso degli output prodotti dalla *input* e *blocking pool* è la condizione in base alla quale il LRNG sceglie se generare o meno bytes verso le interfacce che ne fanno richiesta.
Gli output prodotti in assenza di entropia (provenienti dalla *nonblocking pool* quando il suo *entropy counter* ha valore zero) sono equivalenti a quelli prodotti da un generatore di numeri pseudocasuali semplice (ovvero privo di *entropy inputs*). Il grado di confidenza che si può avere con l'indipendenza (pur sempre apparente) di simili output tra loro, seppure elevata, è ovviamente inferiore rispetto agli output prodotti a partire dall'entropia esterna. L'utilizzo di sorgenti esterne infatti aumenta, per un attaccante che abbia accesso a un insieme di output, la difficoltà di inferire informazione riguardo agli altri valori (passati e futuri) prodotti.

La stima dell'entropia si basa su due meccanismi, rispettivamente:

1. per la stima dei bit di entropia per *entropy input*;
2. per l'accredito presso gli *entropy counter* dei bit di entropia derivati dagli *entropy input*.

Prendendo spunto da [PLV12] (sez. 2.3.2), ci riferiremo col termine di ***entropy estimator*** al complesso di entrambi i meccanismi.

Nota sulla sicurezza: Il grado di confidenza che un utente può accordare al LRNG deve tenere conto anche di eventuali vulnerabilità del LRNG che potrebbero annullare i benefici introdotti dall'aggiunta di entropia esterna: riportiamo un esempio interessante in questo senso nella sezione 5.3 riguardante le vulnerabilità di *RdRand*.

2.3.1 Assunzioni e vincoli per l'*entropy estimator*

Prima di analizzare com'è implementato l'*entropy estimator*, diamo ragione brevemente delle assunzioni e dei vincoli sugli *entropy input*, individuati nell'articolo di Lacharme [PLV12], a partire dai quali ha preso forma l'implementazione attuale.

Assunzioni:

- le distribuzioni di probabilità sono sconosciute e non uniformi: esse infatti cambiano, anche radicalmente, in funzione del contesto in cui si fa uso del LRNG, rendendo impossibile (o molto difficile) fare previsioni sugli input;
- la correlazione fra input è sconosciuta: sebbene sia scontato che vi siano delle correlazioni fra input consecutivi (soprattutto se provenienti dalla stessa sorgente di entropia), è difficile catturarne la natura;
- lo spazio dei campioni è ampio: le differenze tra i *jiffies* sono valori a 32 o 64 bit (a seconda dell'architettura) indefinitamente grandi, che possono pertanto variare fra 2^{32} o 2^{64} possibilità;
- non è possibile stabilire a priori quale sia la conoscenza di un attaccante.

Vincoli:

- il tempo per effettuare la stima è limitato: la stima avviene anche al termine della gestione dell'interrupt, per cui deve richiedere un tempo di esecuzione minimo;
- la stima deve essere effettuata a runtime: l'impredicibilità deve essere calcolata per ogni input, per cui non è possibile differire la stima dell'entropia successivamente all'arrivo di un set di input.

2.3.2 La stima dell'entropia per *entropy input*

A seconda delle sorgenti di entropia cambia il modo di stimare l'impredicibilità introdotta dallo specifico *entropy input*. In ciascuna delle interfacce descritte in 2.2.1 è implementato il meccanismo particolare per l'input in gestione. Specifichiamo che il tempo osservato dal meccanismo di stima dell'entropia è sempre misurato solo in *jiffies*, affiché la stima sia sempre pessimistica: la stima non tiene conto del numero di *cycles* anche quando questo sia disponibile e venga utilizzato dalla funzione di mixing.

`add_device_randomness()` non implementa alcuna stima di entropia, in quanto i valori che vengono presi in input sono quelli che servono a distinguere quanto più possibile la configurazione dell'architettura presente da tutte le altre. A fronte di avvìi distinti della macchina, il contributo di questa routine nell'aggiornamento dello stato interno del generatore è sempre lo stesso, ed è per questa ragione che non vi è alcuna "nuova" impredicibilità da considerare per la stima dell'entropia.

`add_interrupt_randomness()` prova a accreditare presso la *entropy pool* opportuna un bit di entropia per ogni interrupt ricevuto, a meno che non si siano attivati i meccanismi di filtro dell'input descritti in 2.2.1. In base alla frequenza degli arrivi degli interrupt e all'architettura sono possibili tre casi:

1. viene aggiunto il contenuto informativo dell'input alla *fast pool* e alla *pool* corretta, viene accreditata l'entropia presso la *pool*;
2. viene aggiunto il contenuto informativo dell'input alla *fast pool* e alla *pool* corretta ma non viene accreditata l'entropia presso la *pool*: questo è il caso in cui si stanno verificando interrupt dello stesso tipo consecutivamente, oppure non vi è un contatore di *cycles* valido per l'architettura corrente;
3. viene aggiunto il contenuto informativo dell'input alla *fast pool* ma non presso alcuna *entropy pool*, né viene accreditata l'entropia presso la *pool*: è quanto avviene se l'interrupt si è verificato a meno di 100 jiffies dall'ultimo aggiornamento della *fast pool* oppure il counter della fast pool è congruo a 2^6 (sebbene sia un valore `short` e possa assumere valori in 2^{16}).

`add_input_randomness()` e `add_disk_randomness()` gestiscono gli input a maggior valore entropico, cioè quelli provenienti rispettivamente dagli input utente e dagli accessi a disco. Per entrambe le routine la stima viene effettuata attraverso la chiamata ad una terza funzione, `add_timer_randomness(),` che si occupa dell'aggiunta dell'entropia presso le pool, effettua una stima dell'entropia basata sui primi tre ordini di delta (δ) e prova a accreditare la misura stimata presso l'*entropy counter* corretto.
Ogni ordine di δ è calcolato nel modo seguente:

1. δ (nel codice sorgente, `delta`) è pari al tempo di interarrivo tra gli ultimi due input, ovvero tra quello corrente e il suo predecessore;

2. δ_2 (`delta2`) è pari a $|\delta - \delta_{old}|$, ovvero al modulo della differenza tra il δ calcolato al punto 1 e il δ_{old} calcolato per l'ultimo input giunto dalla stessa sorgente di entropia;

3. δ_3 (`delta3`) è pari a $|\delta_2 - \delta_{2\ old}|$.

Una volta calcolati i tre ordini di delta, viene considerato per la stima dell'entropia quello col valore minore. Durante la chiamata a `add_timer_randomness()` viene disattivata la *preemption*.

2.3.3 L'accredito di entropia

L'accredito di entropia presso le *pool* avviene tramite le routine `credit_entropy_bits()` e `credit_entropy_bits_safe()`. Quest'ultima si differenzia dalla prima soltanto per un controllo sulla correttezza del numero di bits che si vuole accreditare presso la *pool* specificata e viene usata nel caso di reseeding provenienti dallo spazio utente (2.1).
Si possono accreditare valori positivi o negativi (in tal caso parliamo di debito o consumo di entropia) a seconda che si stia aggiungendo entropia proveniente dagli *entropy input* oppure si stia prelevando per trasferirla dall'*input pool* verso le *output pool* o per produrre un output.

Vogliamo porre l'accento su come vengano trattati in modo completamente differente i due casi di accredito e di debito di bit di entropia: mentre il debito è gestito tramite un approccio lineare, venendo sottratto dagli *entropy counter* della pool un numero di bit esattamente pari al numero di bit che si vogliono addebitare, nel caso dell'accredito l'aggiunta di bit di entropia contribuisce ad avvicinare asintoticamente l'*entropy counter* al suo valore massimo senza mai raggiungerlo. Oltre a ciò, nel caso in cui la quantità di entropia stimata per l'*input pool* sia prossima al suo valore massimo, viene fatta "traboccare" una parte di quest'entropia verso una *pool* di output.

Ricordiamo che nel conteggio dell'entropia accumulata presso le pool l'unità di misura è l'ottavo di bit (2^{-3} bit). Tale accorgimento accresce di molto la risoluzione del meccanismo di accredito asintotico di bit di entropia presso le pool, facendo sì che possano essere conteggiate le aggiunte di entropia degli *entropy input* anche quando la *pool* è quasi "satura". Ovviamente, per la produzione di bit in output vengono considerati solo valori interi.

Analizziamo di seguito i passi richiesti dall'operazione di accredito di entropia:

1. se il numero di bit di entropia da aggiungere è negativo, tale numero viene direttamente sottratto dall'*entropy counter*;
   se invece tale numero è positivo, viene calcolata quanta parte dei bit di entropia stimati vada sommata all'*entropy counter*: sia *entropy* la quantità di entropia presente nella pool, *pool_size* la massima quantità di entropia che è possibile accumularvi e *add_entropy* i frazionali di bit che si vogliono accreditare. Per dovere di completezza segnaliamo che *add_entropy* non può essere maggiore della metà della capacità della pool e, nel caso il numero di bit che si vogliono accreditare sia maggiore di tale quantità, la parte in eccesso viene scartata.
   La quantità di entropia che verrà aggiunta al conteggio in *entropy* sarà dunque pari a:

$$new\_credit \leftarrow (pool\_size - entropy) * (1 - e^{\frac{-add\_entropy}{pool\_size}})$$

Volendo evitare l'onerosità di calcolo dell'esponeziale nella formula di *new_credit* è necessario sostituirlo con una sua stima più economica.

Consideriamo che se $0 \leq x \leq \frac{1}{2}$, allora $\frac{(1-e^{-x})}{x} \geq \frac{(1-e^{-\frac{1}{2}})}{\frac{1}{2}} = 0.7869386805747332$.

Quando $add\_entropy \leq \frac{pool\_size}{2}$, possiamo quindi approssimare per difetto il termine $(1-e^{\frac{-add\_entropy}{pool\_size}})$ con $(\frac{add\_entropy}{pool\_size}) * 0.7869\ldots$, sapendo che $(1 - e^{\frac{-add\_entropy}{pool\_size}}) \geq (\frac{add\_entropy}{pool\_size}) * 0.7869\ldots$
Approssimando ancora $0.7869\ldots$ con $3/4 = 0.75$ si ottiene la stima di *new_credit* implementata dalla routine:

$$new\_credit \leftarrow (pool\_size - entropy) * \frac{add\_entropy}{pool\_size} * \frac{3}{4}.$$

2. viene controllato che l'*entropy counter* sia consistente:
   - se negativo ne viene corretto il valore a 0;
   - se maggiore della capacità della pool *pool_size* ne viene corretto il valore a *pool_size*.
3. controllo sugli accessi concorrenti: se il valore dell'*entropy count* è diverso da quello registrato all'inizio della chiamata alla funzione deve essere occorso almeno un accredito di entropia concorrente a quello presente e deve essere rieseguita l'intera routine di accredito;
4. configurazione della pool come inizializzata: nel caso in cui la *pool* abbia una stima di entropia superiore a 128 bit e non sia ancora stata inizializzata, viene configurata come inizializzata. Nel caso in cui la *pool* in esame sia la *nonblocking*, viene invocata una routine (`prandom_reseed_late()`) che ne inizializza il buffer interno con dei valori di 4 bytes senza segno (ottenuti tramite l'invocazione di `get_random_bytes()`, quindi originati dalla stessa *pool nonblocking*).
5. step finali nel caso la *pool* in esame sia la *input pool*:
   (a) se il numero di bytes di entropia della *pool* è superiore della soglia opportuna (64 bytes), cambia il flag di polling delle *file operations* consentite sul device file `random` per segnalare la possibilità di accesso non bloccante ai processi in attesa di lettura (per il polling su `random` rimandiamo alla sezione 3.3.2);
   (b) se la stima di entropia presso l'*input pool* è maggiore di 7/8 della capacità complessiva della *pool*, schedula alternatamente la *nonblocking pool* e la *blocking pool* perché vengano trasferiti, dalla *input pool* presso una di esse, fino a 8 bytes di entropia. Non viene schedulato nulla se la *output pool* selezionata è già satura per almeno il 75%. Questo meccanismo è alla base del bilanciamento dell'entropia interna, come vedremo più avanti nella sezione 3.2.2.

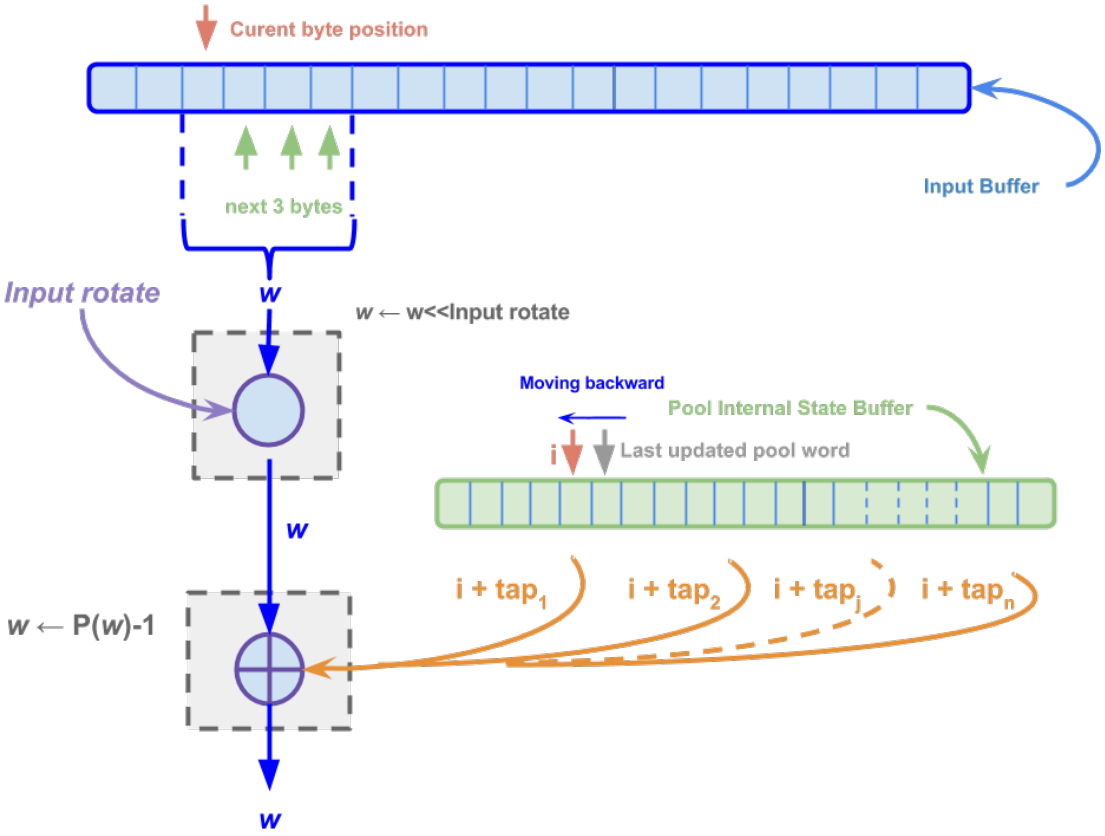
2.4 La Mixing Function

La **mixing function** si occupa di mischiare un dato numero di bytes di un buffer in input al buffer interno di una specifica pool. Nel caso del LRNG, la mixing function è una buona funzione hash non crittografica [PLV12]. In assenza di input, essa è equivalente ad un *registro a scorrimento a retroazione lineare* (*linear feedback shift register* o *LFSR*) definito su un campo finito (o campo di Galois) di 2^{32} elementi ($GF(2^{32})$), con polinomio di feedback $Q(X) = \alpha^3(P(X) - 1) + 1$ (con α primitiva di $GF(2^{32})$) e basato sui polinomi $P(X)$:

$$P(X) = X^{128} + X^{104} + X^{76} + X^{51} + X^{25} + X + 1 \text{ per la } \textit{input pool};$$
$$P(X) = X^{32} + X^{26} + X^{19} + X^{14} + X^7 + X + 1 \text{ per le } \textit{output pool}.$$

La moltiplicazione per α^3 è effettuata attraverso una tabella di lookup che nel codice è chiamata **twist table**.

Il cuore della *mixing function* è costituito dalla definizione dei polinomi nella struttura `poolinfo` e dall'implementazione del LFSR nella routine `_mix_pool_bytes()`.
`_mix_pool_bytes()` per ogni byte del buffer in input, esegue le operazioni:



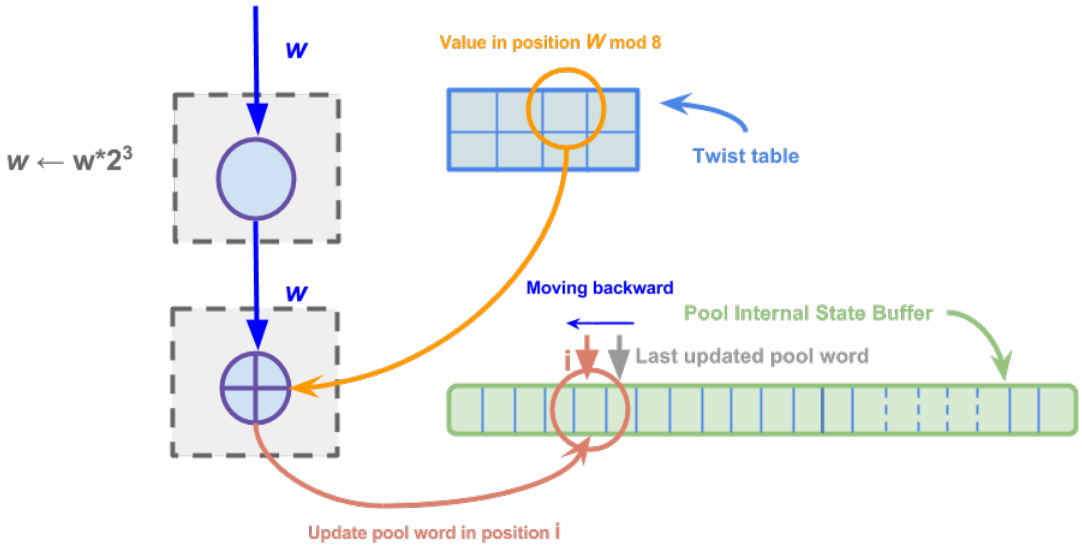
1. viene prelevato il byte corrente assieme ai 3 byte che lo seguono. Otteniamo una parola w di 4 bytes che viene ruotata del numero `input_rotate` di posizioni (in bit) specificato dalla pool;

2. viene inizializzato un cursore i con la posizione del predecessore dell'ultima parola modificata della *pool*;

3. viene effettuato il calcolo del polinomio $P(X) - 1$ per $X = w$ come:

$$w \leftarrow w \oplus pool[i + tap_j]$$

dove i è la posizione di una parola di 4 bytes della pool e tap_j è la potenza j-esima del polinomio $P(X)$ (di cui non consideriamo la potenza 0 a causa dell'addendo -1 in $P(X) - 1$); tale operazione viene applicata $\forall\ tap_j \in \{128, 104, 76, 51, 25, 1\}$ nel caso della *input pool* ($\forall tap_j \in \{32, 26, 19, 14, 7, 1\}$ nel caso delle *output pool*).



4. viene aggiornata l'i-esima parola della pool con $\alpha^3\left(P(X)-1\right)-1$:

$$pool\left[i\right] \leftarrow \left(w * 2^3\right) \oplus twist\_table\left[w \ mod(8)\right]$$

5. il numero di posizioni della rotazione per la prossima iterazione `input_rotate` viene aumentata di 7 posizioni (o 14 alla prima iterazione) e ridotta modulo 32;

6. vengono salvati nella *pool* la posizione `i` dell'ultima parola aggiornata della *pool* e `input_rotate`.

7. se tra i parametri è definito il buffer di output `out` ne vengono riempiti i primi 16 elementi con 16 valori consecutivi della pool letti al contrario (dalla posizione di scrittura nella pool corrente procedendo in senso inverso): tali valori sono quelli a maggior contenuto entropico perché sono quelli che sono stati letti meno di recente.

I polinomi caratteristici attuali sono costruiti in modo da garantire, nel caso non vi siano input di entropia, il massimo periodo per gli output di LRNG. Condizione necessaria e sufficiente perché il periodo sia massimo è la primitività del polinomio di feedback $Q(X) = \alpha^3(P(X)-1)+1$. L'attuale versione del LRNG accoglie le modifiche ai polinomi caratteristici $P(X)$ delle pool suggerite nell'articolo di Lachman et al. [PLV12] affinché sia garantita la primitività di $Q(X)$; nella sezione 5.1 rendiamo conto di dette modifiche e dei miglioramenti che hanno introdotto.

3 L'Output

La produzione degli output necessita di un meccanismo per l'estrazione dell'entropia che mantenga intatta la sicurezza del generatore. Per garantire il rispetto dei vincoli di sicurezza definiti nella sezione 4 il LRNG fa uso di SHA-1 per "mascherare" i bytes dello stato interno utilizzati per l'estrazione di entropia rispetto ai bytes in output prodotti.

In questa sezione analizziamo l'estrazione di entropia in funzione delle *pool* coinvolte:

1. dalle *output pool* vengono prodotti bytes in output ogni volta che viene invocata una tra le interfacce di output descritte in 1.2.3 ;
2. dall'*input pool*, quando è disponibile, viene estratta entropia
   - ogni volta che vengono richiesti bytes in output dalle *pool* di output e queste non hanno sufficiente entropia;
   - quando l'*input pool* è satura per più di $\frac{7}{8}$ e almeno una delle *pool* secondarie è satura meno del 75%.

In particolare, nel primo caso sono generati bytes di entropia per produrre un output verso lo spazio utente o lo spazio kernel (3.3), mentre nel secondo viene estratta entropia dalla *input pool* per trasferirla presso una delle pool di output(3.2).

3.1 Estrazione di un buffer dalla *pool*

L'estrazione di un buffer, ovviamente, non consiste nell'ottenere direttamente una porzione più o meno lunga del buffer interno della *pool*, quanto piuttosto un suo hash crittograficamente "robusto". Come abbiamo già affermato in 1.2.2, questo è il passaggio più delicato in termini di sicurezza e la sua affidabilità è delegata totalmente all'adozione della funzione crittografica SHA-1.

L'estrazione implementata dal LRNG consta sostanzialmente di tre passi: dapprima produce un hash (quello che vedremo nel dettaglio essere il buffer w) a partire dai valori interni della *pool*, prosegue estraendo dalla *pool* il buffer di 512 bit con maggiore contenuto entropico (*extract*) e termina producendo un nuovo hash (sempre via SHA-1) da copiare nel buffer di output.

Volendo osservare più dettagliatamente la funzione di estrazione dell'entropia, implementata dalla routine `extract_buf()`, troviamo la procedura seguente:

1. inizializza un buffer `w` di 5 parole a 32 bit (ricordiamo che le parole delle *pool* sono valori senza segno di 32 bit);
2. applica, per un numero di volte dipendente dalla *pool* (2 per le *output pool* e 8 per l'*input pool*) l'hash di SHA-1 ai 512 bit della pool che iniziano con la parola in posizione i (ovvero coinvolgendo anche le 15 parole che seguono l'i-esima), utilizzando `w` come digest:

   $$pool\,[i]_{512} \leftarrow f_{SHA-1}(w, pool\,[i]_{512})$$

   dove i è un cursore inizializzato a zero e incrementato con passo 16 e $pool\,[i]_{512}$ è il buffer dei 512 bit della *pool* dall'inizio della parola in posizione i. Da notare che ad ogni iterazione vengono anche modificati i valori del digest w;
3. se l'architettura è `x86` o `powerPc`, calcola un buffer `l` composto da 5 o 4 `unsigned long` (rispettivamente nel caso di architetture a 32 e 64 bit) tramite la chiamata a `arch_get_random_long()`:
   (a) in `x86`, ogni valore è ottenuto da un'invocazione dell'istruzione `RdRand` delle architetture *Intel onchip*, che ritorna un numero casuale dal generatore hardware della CPU;
   (b) in `powerPc` il buffer è riempito da valori diversi da zero solo se la linea dell'architettura è la `pSeries`: è il caso di server basati sulla virtualizzazione hardware, i valori sono ottenuti attraverso una chiamata all'interfaccia `H_RANDOM` dell'*hypervisor* (se ne espone una).

4. chiama la *mixing function* per il mixing di w nella *pool* estraendo da essa il buffer `extract` di 512 bit, composto dai valori della *pool* che sono stati aggiornati meno di recente dalla routine per il mixing `mix_pool_bytes()`;

5. applica uno SHA-1 al buffer estratto usando ancora come digest w: quest'operazione è fondamentale per crittografare `extract` e mascherare i bit prelevati direttamente dalla pool;

6. azzera il contenuto del buffer `extract` (per ragioni di sicurezza, contenendo bit copiati direttamente dalla *pool*);

7. mischia i valori di w e li salva nel buffer di output `out`;

8. azzera il buffer contenente w.

Nota: Vogliamo evidenziare qui come il punto 3a sia in contraddizione con quano affermato dal commento al codice, in cui si afferma di non fare alcun uso dei generatori hardware al momento dell'estrazione del buffer dalle *pool*. Esamineremo questo fatto nel paragrafo 5.3 su `RdRand` assieme alle critiche di cui è oggetto.

3.2 Il trasferimento di entropia

Il LRNG implementa alcuni meccanismi atti a bilanciare la quantità di entropia presente al suo interno e garantire che non venga sprecata. Perché ciò possa accedere è necessario poter trasferire entropia tra le *pool*.

Per impedire che l'*input pool* sia "prosciugata" inutilmente, le *output pool* rimangono prive di entropia fino al momento in cui viene loro richiesto di produrre un output: solo allora vengono trasferiti bytes di entropia verso la *pool* che ne ha bisogno. Oppure può succedere, al contrario, che il sistema debba accumulare tanta entropia da tendere a saturare l'*input pool*: poiché abbiamo visto come il meccanismo di accredito di entropia (descritto in 2.3.3) incrementi in modo asintotico gli *entropy counter*, è ragionevole pensare che se l'*input pool* è sufficientemente "piena" sia conveniente "stipare" entropia anche presso le *output pool*.

In entrambi i casi è necessario trasferire entropia dalla *intput pool* ad una *pool* secondaria: questo compito è assolto dalla routine `xfer_secondary_pool()`.

3.2.1 Dall'*input pool* alle *output pool*

Vediamo in questo paragrafo come opera `xfer_secondary_pool()`.

Idempotenza verso *nonblocking pool*: Quando viene invocata, `xfer_secondary_pool()` controlla per prima cosa se è stato richiesto un trasferimento di entropia verso la *nonblocking pool*: se questa è già stata aggiornata negli ultimi 60 secondi, la funzione ritorna senza operare alcun effetto. Questo meccanismo di fatto implementa l'idempotenza verso accessi frequenti e consecutivi alle interfacce della *pool* non bloccante ed è immediato convincersi della sua necessità: un qualunque processo (o più processi concorrenti), sia esso nello spazio kernel o nello spazio utente, che faccia richieste di numeri randomici di qualità inferiore ad un tasso relativamente sostenuto potrebbe esaurire velocemente l'entropia accumulata nel generatore e conseguentemente bloccare i processi dello spazio utente che invece necessitano di output altamente affidabili.

Controllo sulla disponibilità di entropia corrente: Per evitare che vengano tolti inutilmente bytes di entropia dalla *input pool*, la routine controlla, prima di trasferire effettivamente i bytes tramite `_xfer_entropy_pool()`, che nella *pool* corrente vi sia meno entropia di quella richiesta.
Continuiamo la descrizione di `xfer_entropy_pool()` attraverso quella di `_xfer_entropy_pool()`.

_xfer_entropy_pool(): Questa è la routine che si occupa effettivamente del trasferimento di bytes di entropia dalla *input pool*. Oltre che essere estesa da `xfer_entropy_pool()`, viene invocata in modo diretto dalla routine per il riequilibrio dell'entropia `push_to_pool` (che vedremo in 3.2.2).
La routine prende le mosse dall'inizializzazione delle variabili che definiscono l'upper e il lower bound dell'entropia trasferibile, stabilendo che debbano essere trasferiti non meno di 8 bytes di entropia (corrispondenti alla soglia di attivazione per i processi sospesi nella lettura di `/dev/random`) e non più di 128 bytes (pari alla dimensione massima di una *pool* di output).
Le operazioni successive sono naturalmente conseguenti:

1. viene estratto un buffer dalla *input pool* con le caratteristiche sopra citate attraverso la chiamata `extract_entropy()` (descritta più avanti in 3.3.1);

2. viene effettuato il mixing del buffer così estratto attraverso `mix_pool_bytes()` (già descritta nella sezione 2.4 sulla *mixing function*);

3. viene aggiornato l'*entropy counter* via `credit_entropy_bits()` (vista in 2.3.3).

3.2.2 Il bilanciamento dell'entropia

Alla fine della sezione 2.3.3 sull'accredito di entropia abbiamo descritto il meccanismo per cui, se l'*input pool* ha un entropia superiore ai 7/8 della sua capacità, viene trasferita parte di quest'entropia (8 bytes per volta) alternativamente in una delle due *pool* secondarie se questa non è già impegnata per oltre il 75% della propria capacità.

La routine che viene schedulata da `credit_entropy_bits()` per bilanciare l'entropia in eccesso è `push_to_pool()`, puntata dal campo `push_work` della struttura `entorpy_store` che modella le *pool* di output. `push_to_pool()` non fa altro che invocare `_xfer_secondary_pool` per trasferire gli 8 bytes di entropia dalla *input pool* alla *outpul pool*.

3.2.3 L'accounting di entropia

L'accounting nel LRNG è l'operazione che valuta quanti bytes possono essere effettivamente prelevati da una pool a fronte di una richiesta di una quantità pari a `nbytes`. Il modo in cui viene compiuta questa misura cambia a seconda che la *pool* sia bloccante o non bloccante: nel primo caso infatti viene restituito il minimo tra i bytes di entropia disponibili per l'estrazione e `nbytes`, nel secondo il valore di ritorno è lo stesso `nbytes`. In entrambi i casi comunque l'*entropy counter* della pool viene aggiornato con il valore corretto. La routine addetta all'operazione di accounting è `account()`.

account(): Alla routine viene passato come parametro, oltre all'intero `nbytes`, anche i valori `min`, che definisce il minimo valore di entropia che deve essere estratto, e `reserved`, che indica la quantità minima di bytes di entropia che devono essere lasciati nella *pool* dopo l'estrazione.
Diamo un breve descrizione degli step principali compiuti dalla funzione:

1. nel caso in cui la pool sia bloccante, la quantità di bytes estraibile viene calcolata come la differenza tra i bytes della *pool* e i bytes `reserved` che devono essere lasciati "per terra";

2. viene calcolato il nuovo valore dell'*entropy counter* della *pool* garantendo che venga rispettato il vincolo su `reserved`;

3. se l'*entopy counter* letto all'inizio della routine è stato modificato da un altro thread prima che il thread corrente vi abbia eseguito il proprio aggiornamento, occorre fare un nuovo tentativo per operare l'accounting e viene eseguito un *goto* al punto 1;

4. se il numero (intero) di bit dell'*entropy counter* cade sotto la soglia `random_write_wakeup_thresh` (pari a 28 word) viene impostato a 1 lo switch `wakeup_write`: questo ha lo scopo di abilitare la scrittura per i processi che invocano la chiamata di sistema `poll` sui device files `/dev/random` e `/dev/urandom`;

5. se `wakeup_write` è stato impostato manda un segnale di wake up alla *wait queue* `random_write_wait`: la conseguenza è l'esecuzione della routine `random_poll()` che aggiorna la *poll mask* per i device file `random` e `urandom`;

6. ritorna con il numero di bytes che è possibile prelevare.

Da notare che durante tutta la sua esecuzione, la funzione di accounting opera avendo disabilitato le *interrupt request* per poi ripristinarne lo stato al termine della sua esecuzione, attraverso le chiamate a `spin_lock_irqsave()` e `spin_unlock_irqrestore()`.

3.3 La produzione dell'output

Nella sezione 1.2.3 abbiamo distinto le interfacce di output tra quelle esportate verso lo spazio kernel e verso lo spazio utente (i device file `/dev/random` e `/dev/urandom`). Analizziamo l'output del LRNG mantenendo questa distinzione. Le pool interessate a questo livello sono le *output pool*.

3.3.1 Output nello spazio kernel

Tutte le routine esportate verso lo spazio kernel (`get_random_bytes()`, `get_random_bytes_arch()`, `generate_random_uuid()` e `get_random_int()`) producono output in modo non bloccante: tutte infatti generano valori a partire dal buffer interno della *nonblocking pool*, con la sola eccezione di `get_random_int()` che genera valori meno crittograficamente robusti senza consumare l'ammontare dell'entropia complessiva del generatore.

L'output basato sul buffer interno della *nonblocking pool* è un hash ottenuto tramite la chiamata alla routine `extract_entropy()`, che cerca di estrarre dalla *pool* un numero di bytes pari a quello specificato per parametro nel modo seguente:

1. trasferisce dalla *input pool* (possibilmente) il numero di bytes richisto e li mischia alla *nonblocking pool*;

2. sottrae (addebita) all'*entropy counter* il numero di bytes passato per parametro (routine `account()`);

3. estrae bytes in output dalla *nonblocking pool* attraverso la chiamata a `extract_buf()` finché i bytes estratti sono pari al minimo tra il numero di bytes di entropia disponibile e quello richiesto dalla chiamata a funzione;

4. ritorna il buffer complessivo dei bytes estratti.

`get_random_bytes()` è senza dubbio l'interfaccia più utilizzata dal kernel: essa si occupa di produrre come output un hash dei bytes del buffer interno della *nonblocking pool* attraverso la chiamata alla routine `extract_entropy()`. Essa consente l'estrazione di bytes randomici da parte delle componenti del kernel - tipicamente quelle che implementano alcuni tra driver, protocolli di rete, routines di crittografia, protocolli di sicurezza - in modo non bloccante.

`get_random_bytes_arch()` utilizza il generatore hardware di numeri pseudocasuali specifico per l'architettura in uso se ve n'è uno. Il vantaggio nell'uso di un dispositivo hardware risiede nella maggiore velocità di questo rispetto ad una implemetazione in software, ma l'utilizzatore deve ritenere trusted il *manifacturer* del device e avere la ragionevole confidenza nel fatto che questi non abbia inserito in esso back door. Il funzionamento della routine è abbastanza semplice: dapprima cerca di riempire il buffer passato per riferimento con tanti `long` quanti sono richiesti dal parametro `nbytes`; nel caso non sia stato possibile per un qualche

motivo estrarre tutti i bytes richiesti, si esegue l'estrazione nel modo canonico dalla *nonblocking pool*, ovvero mediante `extract_entropy()` (proprio come avverrebbe in `get_random_bytes()`).

`generate_random_uuid()` viene utilizzato dai protocolli e dalle procedure che necessitano di un *Universally Unique IDentifier*. L'uso di base che ne fa il kenrel è quello relativo all'assegnazione di *uuid* alle partizioni del filesystem e per i driver per i filesystem *Lustre*, *Btrfs*, *ubifs* e *reiser fs* ma, come spiegato nel commento al codice, l'interfaccia è esportata per consentirne l'uso anche agli eventuali driver esterni al kernel che ne abbiano necessità. Un *uuid* è un identificativo composto da 16 bytes, ottenuto estraendo 16 bytes randomici dalla nonblocking pool via `extract_entropy()`.

`get_random_int()` si discosta dalle routine precedenti in quanto non consuma bytes di entropia da alcuna *pool*. Il suo uso è infatti previsto per i casi in cui servono bytes generati casualmente ma su di essi non vi è alcun requisito di sicurezza né di qualità, ma unicamente di velocità di calcolo: un consumo dell'entropia del generatore per la loro produzione corrisponderebbe pertanto ad uno spreco. La routine produce un intero a 32 bit cercando dapprima di prelevarli da un eventuale generatore hardware di numeri casuali; in caso non sia possibile, ritorna un hash composto a partire dal pid del processo corrente, dal valore attuale del contatore dei *jiffies* e del contatore di cicli se implementato.

3.3.2 Output nello spazio utente

Sappiamo che le interfacce per gli output (ma anche gli input) del LRNG nello spazio utente sono i due device files `/dev/random` e `/dev/urandom`: questi vengono creati assieme agli altri device files dalla componente in `/drivers/char/mem.c` e hanno i permessi in lettura e scrittura (0666 in notazione ottale), mentre le operazioni consentite su di essi sono definite nelle struct `file_operations` rispettivamente `random_fops` e `urandom_fops`. Le *file operations* definite su di essi differiscono, com'era da aspettarsi, unicamente per le funzioni di lettura, ovvero nell'atto di estrazione degli output del LRNG.

extract_entropy_user() Prima di descrivere le funzioni di lettura è necessario introdurre la routine per l'estrazione dell'entropia dalle *pool* `extract_entropy_user()`, l'equivalente di `extract_entropy()` per le richieste provenienti dallo spazio utente.
Sostanzialmente il suo comportamento è del tutto analogo a quello di `extract_entropy()` (che abbiamo descritto in 3.3.1):

1. trasferisce dalla *pool* primaria (possibilmente) nbytes e li mischia alla *pool* secondaria;
2. preleva e ritorna un hash estratto dalla *pool* passata per riferimento.

ma si distingue da essa per alcuni dettagli funzionali e implementativi:

- non viene mai chiamata sulla *input pool* ma esclusivamente sulle *pool* di output;
- finché ci sono bytes da estrarre, prima dell'estrazione controlla che il processo corrente non sia bloccato e, nel caso, chiede al kernel di rischedularlo;
- copia il buffer dei bytes letti nel buffer in output dello spazio utente.

Lettura da /dev/random: La funzione addetta all'operazione di lettura su `/dev/random` è `random_read()`, estrae dalla *blocking pool* 512 byte di entropia per volta e li consegna al buffer nello spazio utente tramite `extract_entropy_user()` fino a raggiungere il numero di bytes `nbytes` richiesto. Nel caso l'estrazione fallisca, l'intera funzione ritorna il codice di errore corrispondente. Se non è stato possibile estrarre alcun byte e il file passato per riferimento alla routine ha impostato il flag `O_NONBLOCK`, la funzione ritorna con vil valore `EAGAIN`, ovvero è consentito il polling del processo in lettura. Se invece non è stato possibile prelevare byte entropici e la lettura del file è bloccante, la lettura del device file è bloccata in attesa che venga segnalato il

nuovo superamento della soglia di wakeup da parte della *input pool*. Non appena il processo è stato risvegliato, ovvero non appena è stata aggiunta sufficiente entropia alla *input pool* da superare la soglia di wakeup, invia il codice di errore `ERESTARTSYS` che serve al processo chiamante per re-invocare la chiamata di sistema per la lettura del device file. Al termine dell'esecuzione della routine, la funzione ritorna con il numero di bytes estratti oppure con l'eventuale codice di errore nel caso non sia stato possibile estrarne alcuno.

Lettura da /dev/urandom: La lettura degli output dal device file `/dev/urandom` è definita dalla routine `urandom_read()`, che si limita a eseguire `extract_entropy_user()` sulla *nonblocking pool* per estrarre i bytes richiesti in output e consegnarli direttamente allo spazio utente.

Blocco dell'output o output continuo: Pur avendo visto le funzioni di lettura non abbiamo ancora chiarito come facciano le due interfacce `random` e `urandom` rispettivamente a bloccare la produzione dei bytes in output oppure a produrli anche in assenza di entropia. La differenza funzionale dei due device file tra bloccante e non bloccante risiede non solo nelle diverse implementazioni delle funzioni di lettura `random_read()` e `urandom_read()`, ma anche da come cambia il comportamento delle routine `extract_entropy_user()` e `account()` in funzione delle *pool* su cui vengono chiamate.

In `random_read()` `extract_entropy_user()` viene chiamata sulla *blocking pool*, prova a trasferire nel buffer dello spazio utente il numero di bytes richiesto e produce un valore di ritorno n che può essere:

1. il numero di bytes effettivamente trasferiti se > 0: `random_read()` prova a trasferire un altro chunk di massimo 512 bytes, iterando fino al completamento della richiesta;
2. 0 se non c'è entropia nelle *pool*: in questo caso il comportamento di `/dev/random` dipende da come è stato impostato il flag `O_NONBLOCK` che, nel caso di `random`, è *device specific*([Ker15]):
   (a) se attivo, una `random_read()` ritorna immediatamente con un errore `EAGAIN`: l'invocazione di una nuova `read()` sarà quindi gestita interamente nello spazio utente;
   (b) se disattivato, il processo chiamante è sospeso per essere poi "risvegliato" al sopraggiungere di nuova entropia in quantità sufficiente: una volta sbloccato il processo questo riceve come valore di ritorno dalla `read` il codice di errore ERESTARTSYS, che segnala di rieseguire la chiamata di sistema su cui si era bloccato precedentemente.
3. un codice di errore (< 0) nel caso il flusso di esecuzione dell'estrazione si sia interrotto per altri motivi.

Quando `extract_entropy_user()` viene chiamata sulla *nonblocking pool* in `urandom_read()`, si comporta in modo diverso: in questo caso infatti essa copia nel buffer dello spazio utente esattamente il numero di bytes randomici richiesti a prescindere da quanti bytes sono conteggiati negli *entropy count* dell'*input pool* e della *nonblocking pool*. Questo è possibile perché la routine `account()` (3.2.3), su cui si basa `extract_entropy_user()`, oltre ad aggiornare l'*entropy counter* distingue a sua volta tra *pool* bloccanti e non bloccante ritornando, nel primo caso, con il numero di bytes di entropia estraibili e, nel secondo, con il numero stesso di bytes richiesti. In questo modo `extract_entropy_user()`, chiamata sulla *nonblocking pool*, preleva da essa un hash esattamente della dimensione desiderata ignorando la quantità di entropia che contiene.

4 Sicurezza nel LRNG

In questa sezione diamo una rapida panoramica di come sono implementati i requisiti di sicurezza nel LRNG, mentre per il lettore alla ricerca di una descrizione approfondita rimandiamo agli articoli [PLV12] (sezione 4) e [ZGR06] (sezioni 3.1, 3.4).

4.1 I requisiti di sicurezza

In *"The Linux Pseudorandom Number Generator Revisited"* [PLV12] (estendendo quanto analizzato in *"Analysis of the Linux Random Number Generator"* [ZGR06]) vengono elencati i seguenti requisiti di sicurezza:

1. **buona stima di entropia:** l'entropia introdotta deve essere stimata in modo da rendere difficile per un attaccante prevedere gli output senza conoscere gli input;
2. **pseudorandomicità:** non deve essere possibile, per un attaccante con una conoscenza parziale o totale delle sorgenti di entropia, predire i futuri output o conoscere lo stato interno a partire dalla conoscenza di output correnti;
3. **forward security:** un attaccante che conosca lo stato interno del generatore non deve poter ricostruire i vecchi output prodotti (*backtracking resistence*), ovvero non deve essere in grado di ricostruire i vecchi stati assunti dallo stato interno anche in assenza di nuova entropia;
4. **backward security:** (detto anche *break in recovery*) un attaccante che conosca lo stato interno del generatore ad un dato istante non deve poter prevedere gli output futuri del generatore se è stata introdotta una quantità sufficiente di entropia dall'esterno.

Come viene fatto notare in [PLV12] i requisiti 3 e 4 assumono che un attaccante conosca lo stato interno del generatore ad un dato istante, e prevedono che il sistema sia resiliente ad attacchi di tipo crottoanalitico; allo stesso tempo, gli autori osservano come sia difficile conoscere in pratica lo stato interno del generatore in quanto questo vive ed opera interamente all'interno del kernel e non vi sono interfacce che ne consentano la lettura in maniera diretta.

Nel seguito di questa sezione tratteremo i requisiti 2, 3 e 4 con riferimento alle caratteristiche del LRNG esposte fino a questo momento, mentre tralasciamo il requisito 1 in quanto necessiterebbe di un'attenzione più teorica rispetto allo scopo del nostro lavoro.

4.2 Pseudorandomicità

Il requisito sulla pseudorandomicità considera il caso in cui l'attaccante conosca completamente o in parte le *entropy sources*; di seguito esaminiamo il suo soddisfacimento considerando il caso ancora peggiore della totale assenza di afflusso di entropia esterna.
Il LRNG applica un feedback al proprio stato interno ogni volta che viene estratto un output attraverso la *mixing function* descritta in 2.4 . Lacharme *et al.* affermano che non vi sia conoscenza di attacchi realizzabili, su un generatore di numeri pseudorandomici (PRNG) siffatto, che possa fare a meno della conoscenza dello stato interno: questo vale in ragione del fatto che SHA-1 è una funzione crittografica a senso unico (o, come troviamo in letteratura, *one-way*) e che dunque è teoricamente impossibile derivare informazione sullo stato interno a partire dagli output prodotti. L'operazione di *folding* dei bytes estratti dalla *pool* rispetto a quelli restituiti come output impedisce inoltre di risalire allo stesso hash prodotto da SHA-1.

4.3 Resilienza alla crittoanalisi

Come abbiamo anticipato, i requisiti di sicurezza 3 e 4 (visti in 4.1) assumono che l'attaccante abbia una conoscenza parziale o totale dello stato interno del generatore. Prima di vedere come il LRNG implementi il *forward* e *backward recovery*, vogliamo analizzare velocemente in cosa consista tale conoscenza e quali siano gli scenari in cui si possa considerare una possibilità concreta.

Come già sappiamo (1.3.1), lo stato interno è implementato dalle tre *pool input*, *blocking* e *nonblocking*: ne consegue che quando parliamo di conoscenza dello stato interno ci riferiamo alla capacità di un osservatore esterno di conoscere i valori nei buffer di almeno una di esse. Abbiamo anche osservato in 1.3 come SHA-1 sia una protezione robusta affinché gli output non "tradiscano" nulla sul contenuto informativo delle *pool*. Per poter avere accesso ai buffer interni alle *pool* è necessario pertanto che l'osservatore sia in una condizione (e posizione) particolarmente privilegiata (pensiamo ad esempio al caso di *ring deprivileging* e *ring compression* che si può avere nella virtualizzazione di un sistema operativo).

Per poter fare previsioni sugli output è necessario conoscere lo stato delle *pool* di output *blocking* e *nonblocking*. Sappiamo però che queste sono tendenzialmente prive di entropia fino al momento in cui viene richiesto loro di produrre un buffer in output, ed anche in questo caso i bytes di entropia sono consumati all'istante: l'attaccante dovrebbe pertanto catturare lo stato del buffer interno della *pool* tra il momento in cui avviene il reseeding con i bytes provenienti dall'*input pool* (tramite `xfer_secondary_pool()` 3.2.1) e quello in cui viene estratto l'hash e mischiato al buffer in output; conoscere lo stato interno prima di questa brevissima finestra sarebbe intutile a meno di possedere (e quindi aver catturato) anche l'hash estratto dalla *input pool* da `xfer_secondary_pool()`. Continuando questa ricerca a ritroso, troviamo che l'unico modo per poter fare delle previsioni, in modo quanto più possibile deterministico, sugli output di almeno una fra le interfacce disponibili, occorre conoscere sia lo stato della rispettiva *output pool* che quello della *input pool*. Anche in questo caso un attaccante a conoscenza dello stato prima della generazione dell'output potrebbe inferire lo stato successivo solo a meno dei 160 bit mischiati nuovamente alle *pool* per il meccanismo di feedback: in assenza di ulteriori informazioni un attacco generico avrebbe un overhead di $O(2^{160})$ e produrrebbe 2^{80} soluzioni ([PLV12]).

Possiamo quindi considerare che l'assunzione sulla conoscenza da parte di un attaccante dello stato interno, su cui si basano i due requisiti in esame, sia fortemente difficile da realizzare.

Forward security: Secondo Elaine Barker e John Kelsey (in "*Recommendation for random number generation using deterministic random bit generators*" [BK07]), se un generatore di numeri pseudocasuali, ogni volta che produce i propri output, lo fa sia attraverso una funzione *one-way* sia adottando un meccanismo di feedback per aggiornare il proprio stato interno, allora si può considerare resistente al *backtracking* come richiesto dal requisito sul *forward recovery*. Il LRNG implementa tale requisito utilizzando la primitiva SHA-1 per produrre l'hash su cui si basano il buffer restituito per output e i bytes per il feedback verso la *pool* (sezione 3.3).

Backward security: La *backward security* è garantita soltanto se si è effettuato un reseeding opportuno dello stato interno tra due richieste di output distinte ([BK07]). Il meccanismo che implementa il requisito è dato dal mancato trasferimento di bytes di entropia dalla *input pool* alle *pool* secondarie se la prima non ha accumulato almeno 64 bit di entropia: per questo motivo la resistenza a questo genere di attacchi dichiarata per il LRNG è pari a 64 bit ([PLV12]).

5 Cos'è cambiato?

Nella realizzazione del presente lavoro ci siamo imbattuti spesso nelle "tracce" lasciate dalle vecchie implementazioni del codice – dalle transizioni tra soluzioni differenti alle ottimizzazioni di varia natura –, come ad esempio le incongruenze tra quanto affermato nei commenti del codice, le differenza tra il codice attuale e quanto affermato negli articoli e la presenza nel codice alcuni rami morti del flusso di esecuzione (provenienti da meccanismi funzionali rimossi da poco tempo).
Questa sezione ha lo scopo di "raccontare" alcuni fra i cambiamenti che abbiamo rintracciato, o perché funzionalmente di rilievo (come la modifica al polinomio caratteristico delle *mixing function*) e le criticità (e le critiche) che sono emerse recentemente sull'uso del generatore hardware di numeri randomici delle ultime architetture `x86` della *Intel*.

5.1 Mixing Function e Polinomio Caratteristico

L'ottimizzazione più evidente avvenuta negli ultimi due anni all'implementazione del LRNG risiede nel polinomio caratteristico della *mixing function*: come rimarcato nei commenti stessi del codice, la *mixing function* utilizzato fin dall'origine non generava, in assenza di entropia esterna, una sequenza di stati interni di periodo massimo. Questo fatto è stato notato per la prima volta proprio nell'articolo di Lacharme *et al.* *"The linux pseudorandom number generator revisited"* ([PLV12] a pagina 7), dove si suggerivano anche le modifiche che si sarebbero dovute applicare ai polinomi caratteristici relativi alle *pool* primaria e secondarie per rendere massimo tale periodo in $GF(32)$.

I cambiamenti suggeriti sono stati accolti dalle nuove versioni del kernel, così che oggi il polinomio relativo alla *input pool*:

$$P(X) = X^{128} + X^{103} + X^{76} + X^{51} + X^{25} + X + 1$$

è diventato:

$$P(X) = X^{128} + X^{104} + X^{76} + X^{51} + X^{25} + X + 1$$

mentre il polinomio caratteristico per le *pool* di output è mutato da:

$$P(X) = X^{32} + X^{26} + X^{19} + X^{14} + X^{7} + X + 1$$

a:

$$P(X) = X^{32} + X^{26} + X^{20} + X^{14} + X^{7} + X + 1$$

I periodi sono così aumentati da meno di $2^{92*32} - 1$ a $2^{128*32} - 1$ per la *input pool* e da meno di $2^{26*32} - 1$ a $2^{32*32} - 1$ per le *output pool* (sempre da [PLV12]).

5.2 Rimozione di IRQF_SAMPLE_RANDOM

Fino alla versione 3.6 del kernel, era necessario dichiarare direttamente nei driver se gli interrupt specifici per ciascuno di essi dovessero essere considerati sorgenti di entropia o meno. In [PLV12] si descrive come, per fare ciò, si doveva impostare per ciascuno di essi il flag `IRQF_SAMPLE_RANDOM`. Dal 2009 si è iniziato a soppiantare `IRQF_SAMPLE_RANDOM`, nella gestione degli interrupt, con un approccio completamente automatizzato, al punto da sostituirlo completamente nel 2012 come affermato dallo stesso Theodore Ts'o in una mail della *Linux Kernel Mailing List* (rintracciabile nell'archivio in [Ts'12]):

> *The IRQF_SAMPLE_RANDOM flag is finally gone*
> *from the kernel tree, only three years late.* ☺

Il motivo alla base di questa sostituzione furono gli usi sbagliati che venivano fatti del flag nell'implementazione dei driver. Durante questi tre anni di transitorio, gli interrupts vennero di fatto ignorati come sorgenti di entropia perché si è ritenuto non vi si potesse più fare affidamento con un sufficiente grado di confidenza e le uniche sorgenti di entropia rimaste erano soltanto gli input utente e i *disk timings* (pagina 4 di [PLV12]). Notiamo (come visto in 2.2) che anche nell'implementazione attuale ogni interrupt concorre al più per solo un bit all'aumento dell'entropia del generatore, mentre gli input utente e i tempi di accesso a disco sono stimati avere un contenuto di entropia pari a 64 bit.

5.3 Dibattito su RdRAND

Nel settembre 2013, sul *New York Times*, esce un articolo ([NPS13]) che titola "*N.S.A. Able to Foil Basic Safeguards of Privacy on Web*" e in cui si afferma, tra le varie cose, che il governo statunitense, nelle vesti dell'agenzia *N.S.A.*, avrebbe fatto pressione verso diversi manifacturer di hardware per inserire *back doors* nei propri prodotti dedicati alla sicurezza.

Con riferimento all'articolo del *NYT*, il "padre" del LRNG Theodore Ts'o pubblica sul suo profilo *Google Plus* [Ts'13]:

> *≪I am so glad I resisted pressure from Intel engineers to let /dev/random rely only on the RDRAND instruction. To quote from the article below:*
>
> *"By this year, the Sigint Enabling Project had found ways inside some of the encryption chips that scramble information for businesses and governments, either by working with chipmakers to insert back doors..."*
>
> *Relying solely on the hardware random number generator which is using an implementation sealed inside a chip which is impossible to audit is a BAD idea.≫*

Taylor Hornby, un ricercatore e programmatore canadese, due mesi più tardi afferma in un tweet ([Hor13a]) di essere riuscito a modificare l'istruzione `RDRAND` per "*cancellare l'altra entropia*", ovvero l'entropia raccolta dal LRNG senza l'ausilio del generatore hardware dei processori `x86`, pubblicando contestualmente l'immagine del codice dell'hacking e del suo effetto su `/dev/urandom`. Oltre a ciò, sempre Hornby pubblica un suo commento al codice ([Hor13b]) in cui mette in rilievo alcune contraddizioni dei commenti originali; fra questi, vogliamo rilevare in particolare il commento alla routine `get_random_bytes()` dove si evidenzia la non veridicità dell'affermazione, presente nel codice, sul non uso di generatori hardware di numeri randomici:

> *≪This function is the exported kernel interface.*
> *It returns some number of good random numbers,*
> *suitable for key generation, seeding TCP sequence numbers, etc.*
> ***It does not use the hw random number generator, if available;***
> *use get_random_bytes_arch() for that.≫*

Come abbiamo avuto modo di verificare noi stessi nel presente lavoro (nelle sezioni 3.1 e 3.3.1), nelle architetture `x86` *Intel onchips* viene in effetti fatto uso del generatore hardware presente nei chipset durante l'estrazione di un buffer di bytes randomici dalle *pool* (la chiamata `RDRAND` è implementata nelle architetture *Intel* in `get_random()`). L'incoerenza del commento al codice rispetto all'implementazione effettiva deve aver tratto in inganno anche gli autori di [PLV12]: a pagina 4, infatti, affermano che nessun generatore hardware di numeri pseudocasuali è coinvolto nella produzione di output del LRNG ("*[...]hardware RNGs ... are not mixed into the Linux PRNG*").
Le nostre osservazioni pertanto confortano quanto affermato da Hornby.

L'hacking compiuto da Hornby però non sembra essere applicabile per le versioni del kernel successive alla 3.12.3 ma, come lui stesso afferma, è ragionevole non escludere l'esistenza di altri punti deboli negli output del LRNG:

> *≪It's fixed in 3.13-rc2, in that you can't control /dev/(u)random output, but you can still weaken some other properties.≫*

5.4 E in futuro?

Sia Guttermann *et al.* in [ZGR06] che Lacharme *et al.* in [PLV12] sono concordi nella necessità di una documentazione del codice più approfondita sull'implementazione del LRNG. Soprattutto, abbiamo notato nel nostro lavoro come addirittura il commento di corredo al codice sorgente, oltre che carente di dettagli, sia

in alcuni punti obsoleto e non coerente rispetto al codice stesso che si propongono di descrivere. Il bisogno di una documentazione aggiornata è stata alla base della pubblicazione degli articoli sopra citati e, con necessaria e molta modestia, di questa nostra analisi, e si giustifica considerando la sensibilità degli scenari in cui il LRNG deve impiegato e l'affidabilità che conseguentemente gli viene richiesta. Le problematiche emerse di recente sulla possibile presenza di *back door* nelgli stessi chipset utilizzati dal generatore impone un'analisi più approfondita e, a nostro avviso, la necessità di rimuovere l'uso di generatori hardware dalle routine per l'estrazione dei buffer dalle *pool* del LRNG, ancora ad oggi presente.

Conclusioni

Nel lavoro che abbiamo presentato sono state esposte le funzionalità del Linux Random Number Generator, con particolare attenzione alle interfacce `/dev/random` e `/dev/urandom` esposte verso lo spazio utente. Oltre alla descrizione funzionale, è stata analizzata l'implementazione della gestione dell'entropia raccolta dal sistema, sia al momento della sua raccolta a partire da *entropy inputs* sia relativamente alla produzione di bytes di entropia come output.

Riferimenti bibliografici

[BK07] Elaine Barker and John Kelsey. Recommendation for random number generation using deterministic random bit generators (revised). *Technical Report SP800-90, NIST*, 2007.

[Hor13a] Taylor Hornby. `https://twitter.com/defusesec/status/408975222163795969` Twitter 'My RDRAND backdoor proof of concept is working!, 2013.

[Hor13b] Taylor Hornby. `http://pastebin.com/A07q3nL3` hacking and comment to random.c - untitled, 2013.

[Ker15] Michael Kerrisk. `http://man7.org/linux/man-pages/man4/random.4.html` Random(4) - Linux Programmer's Manual, The Linux man-pages project, 2015.

[MT14] Matt Mackall and Theodore Ts'o. random.c – a strong random number generator, june 2014, 2014.

[NPS13] Jeff Larson Nicole Perloth and Scott Shane. `http://www.nytimes.com/2013/09/06/us/nsa-foils-much-internet-encryption.html?pagewanted=all&_r=0` New York Times 'N.S.A. Able to Foil Basic Safeguards of Privacy on Web', 2013.

[Par15] Emanuele Paracone. `https://github.com/knizontes/random/blob/master/random.c` comment to random.c - untitled, 2015.

[PLV12] Vincent Strubel Patrick Lacharme, Andrea Röck and Marion Videau. the linux pseudorandom number generator revisited, `http://eprint.iacr.org/2012/251.pdf`. *IACR Cryptology ePrint Archive*, 2012.

[Ts'12] Theodore Ts'o. `https://lkml.org/lkml/2012/7/17/650` Linux Kernel Mailing List archive, 2012.

[Ts'13] Theodore Ts'o. `https://plus.google.com/+TheodoreTso/posts/SDcoemc9V3J` Google Plus 'I resisted pressure from Intel engineers', 2013.

[ZGR06] Benny Pinkas Zvi Gutterman and Tzachy Reinman. Analysis of the linux random number generator. *IACR Cryptology ePrint Archive*, 2006.