ROMA TRE
UNIVERSITÀ DEGLI STUDI

CORSO DI LAUREA MAGISTRALE

INGEGNERIA DELLE TECNOLOGIE DELLA COMUNICAZIONE E DELL'INFORMAZIONE

Generatori di Numeri Pseudocasuali

Diego Belvedere, Alessandro Brugnola, Alessia Vennarini

Prof. Francesca Merola

Roma, Maggio, 2009

Anno Accademico 2008/2009

Indice

Capitolo 1. Introduzione

Per iniziare a comprendere l'importanza della casualità bisogna capire bene la differenza tra numero casuale e numero pseudocasuale. Il numero **casuale** è un numero estratto da un insieme di valori equiprobabili, mentre il numero **pseudocasuale** è un numero casuale generato da calcolatore.

1.1 Sequenze "veramente casuali" e "pseudocasuali"

Nelle lotterie con un premio in denaro, i numeri vincenti sono di solito *veri* numeri casuali ed ognuno è determinato da una pallina numerata messa con delle altre in un contenitore. Un generatore di numeri veramente casuali ha bisogno di una sorgente randomica esistente in natura ed è un compito difficile progettare un dispositivo hardware o un programma software che sfrutti questa casualità e produca una sequenza di numeri libera da errori e correlazioni. Questo ovviamente non è possibile praticamente con le simulazioni al calcolatore, specialmente quando sono richiesti milioni di numeri casuali. Infatti, le tecniche che facevano uso dei numeri veramente casuali e altri metodi Monte Carlo (dal legame che esiste tra gioco e simulazioni probabilistiche, in onore del famoso casinò di Monaco) sono stati abbandonati per varie ragioni.
Sui moderni computer, i numeri pseudocasuali sono generati da algoritmi interamente deterministici e l'obiettivo più soddisfacente è quello per cui nessuno possa distinguere la sequenza d'uscita del generatore da una sequenza veramente random in un tempo ragionevole (qualche anno di tempo di CPU su potenti computer), osservando solamente la sequenza d'uscita e non conoscendo la struttura del generatore.
Secondo quanto affermava John Von Neumann, non è possibile produrre numeri casuali tramite metodi matematici, in quanto un vero generatore di numeri casuali è uno strumento capace di fornire una sequenza di numeri non deterministici. Questi numeri sono idealmente infiniti e non sono influenzabili da alcun fattore esterno. Il computer, o qualsiasi macchina, non ha la possibilità di generare questo tipo di sequenza.
L'unico modo è utilizzare opportuni algoritmi che generino numeri apparentemente casuali. Vengono quindi chiamati numeri *pseudocasuali* poiché venendo a conoscenza dell'algoritmo e del primo elemento (seme) utilizzato, è possibile determinare la sequenza che verrà generata.
Inoltre la sequenza non è infinita, ovvero, la sequenza di numeri generati si ripete ciclicamente con un intervallo fisso (periodo del generatore). L'unico elemento puramente aleatorio è il seme, poiché scelto dai dati casuali presenti nella macchina: ad esempio, il numero di file presenti, il numero di battute della tastiera, l'orario, ecc.

1.2 Requisiti delle sequenze

Una sequenza di numeri, per essere definita casuale, deve possedere due caratteristiche fondamentali: *distribuzione uniforme* e *indipendenza*.
La prima fa riferimento alla distribuzione uniforme (equidistribuzione) che devono possedere i numeri generati all'interno di un intervallo determinato e solitamente l'intervallo ha ampiezza (0,1).
La seconda significa che numeri successivi devono essere indipendenti tra di loro. L'output al tempo *t* non deve quindi influire sull'output generato al tempo *t+1*.
Procediamo con un esempio chiarificatore. L'ipotetica sequenza di numeri pseudocasuali:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

è certamente equidistribuita nell'intervallo (1, 10) e quindi il principio della distribuzione, in questo caso, è stato rispettato. Lo stesso non si può dire nei confronti dell'indipendenza. Tutte le coppie di numeri sono infatti nella forma (n, n+1).
Rientrando nel campo della definizione intuitiva dei PRNG, se osservassimo i primi 8 numeri della sequenza, potremmo facilmente ipotizzare il valore del nono output, e questo contrasta con la definizione di generatore casuale.

1.3 Generatori pericolosi

La maggior parte dei generatori disponibili sui computer non ha le caratteristiche ideali elencate nel paragrafo precedente. Possono essere inadatti o addirittura pericolosi per l'applicazione in uso. Per la crittografia, molti generatori sono pericolosi perché ci sono modi efficienti per predire il valore successivo, data una sequenza di valori già prodotta dal generatore. Molti dei generatori di default correntemente offerti nei software in commercio, sono vecchi e non competitivi con quelli basati sulla recente teoria.

Costruire un buon generatore non è per niente facile e, per usare una frase di Knuth: "I numeri casuali non dovrebbero essere generati con un metodo scelto a caso!".

1.4 Applicazioni

L'utilizzo maggiore dei numeri casuali si trova nella simulazione e nella crittografia.
Nella simulazione è necessario creare campioni "virtuali" capaci di rappresentare una popolazione reale o un suo aspetto. Si effettuano simulazioni di natura probabilistica di fenomeni fisici (reattori nucleari, traffico stradale, aerodinamica), di problemi decisionali e finanziari (es.: econometria, previsione Dow Jones), informatica (progettazione VLSI, rendering) o come semplice fonte di divertimento (videogiochi).
Nella crittografia, invece, c'è la necessità di generare numeri che non possano essere determinati da terzi. E' difficile immaginare un'applicazione crittografica ben progettata che non faccia uso dei numeri casuali. Le chiavi di sessione, i vettori di inizializzazione, l'hash, i parametri unici nelle operazioni sulle firme digitali ed i nonce (number used once) nei protocolli sono assunti casuali dai progettisti dei sistemi. Sfortunatamente, come si accennava all'inizio, molte applicazioni crittografiche non hanno una fonte affidabile di numeri casuali reali, come il rumore termico nei circuiti elettronici, il decadimento atomico o altri fenomeni fisici con alta entropia. Quindi usano un meccanismo crittografico chiamato Generatore di Numeri Pseudocasuali (PNRG – Pseudo-Random Number Generator), per generare questi valori. I PRNG raccolgono la casualità da vari flussi d'ingresso a bassa entropia e provano a generare uscite che sono in pratica indistinguibili dai veri flussi casuali.

1.5 Una definizione di generatore

I generatori di numeri casuali odierni sono software che producono una sequenza di numeri periodica e deterministica. L'Ecuyer ha dato la seguente definizione:

Un *generatore di numeri pseudocasuali* è una struttura $g = (S, s_0, T, U, G)$ dove:

- S è un insieme finito di stati (spazio degli stati),
- $s_0 \in S$ è lo stato iniziale,
- μ è la distribuzione di probabilità utilizzata per selezionare il seme dallo spazio degli stati S,
- $T : S \rightarrow S$ è la funzione di transizione utilizzata per determinare lo stato al tempo $t+1$ dato lo stato al tempo t, più formalmente: $s_{i+1} = T(s_i)$,
- U è un insieme finito di simboli di uscita (spazio degli output),
- $G : S \rightarrow U$ è la funzione d'uscita. Dato un qualsiasi stato s_i, $u_i = G(s_i) \in U$.
  Gli output $u_0, u_1, \ldots$ sono i numeri casuali prodotti dal generatore.

Un generatore opera nel modo seguente: parte dallo stato iniziale s_0 (chiamato seme) e definisce $u_0 := G(s_0)$. Poi, per $i := 1, 2, \ldots$ imposta $s_i := T(s_{i-1})$ e $u_i := G(s_i)$. Si assume che sono disponibili procedure efficienti per calcolare T e G. La sequenza $\{u_i\}$ è l'output del generatore e gli elementi u_i sono chiamati *osservazioni*. Per i generatori di numeri pseudocasuali ci si aspetta che l'uscita abbia il comportamento di una variabile casuale uniformemente distribuita sullo spazio U. L'insieme U spesso è un insieme di interi $\{0, \ldots, m-1\}$ o un insieme finito di valori compresi tra 0 e 1 per approssimare la distribuzione $U(0,1)$.

Dal momento che lo spazio degli stati S è finito, selezionando un qualsiasi seme s_i, esisterà sempre un valore l tale per cui:

$$s_{i+l} = s_i$$

In pratica, qualsiasi sia lo stato iniziale, dopo un numero l di iterazioni, il PRNG torna inevitabilmente allo stato iniziale. Dal momento che sia la funzione di transizione che la funzione di output sono deterministiche, allora anche gli output generati torneranno inevitabilmente allo stato iniziale.

Il valore di l più piccolo per cui si realizza il ritorno allo stato iniziale è chiamato *periodo* del PRNG ed è individuato dal simbolo ρ .
Il valore di ρ è minore o uguale alla cardinalità dello spazio degli stati S. Nella pratica, essendo S memorizzato in un calcolatore sotto forma di stringa binaria di lunghezza k, allora ρ avrà lunghezza:

$$\rho \leq 2^k$$

I generatori di buona qualità si distinguono per valori di ρ prossimi a $|S|$. Il valore di ρ dipende anche dal seme ed i buoni generatori possiedono valori del periodo uniformi per tutti i possibili stati iniziali.
Un PRNG con $l = 0$ è detto puramente periodico.

Da questa definizione, lo stato iniziale s_0 è assunto dato (deterministico). Per introdurre un po' di casualità "reale", questo stato iniziale può essere scelto casualmente (come estrarre dei numeri da un contenitore). In altre parole, possiamo generalizzare questa definizione dicendo che lo stato iniziale s_0 è generato casualmente secondo qualche distribuzione di probabilità μ su S. Generare un seme veramente casuale richiede molto meno lavoro ed è molto più ragionevole che generare una lunga sequenza di veri numeri casuali. Un generatore con un seme casuale può essere visto come un

estensore di casualità, il cui scopo è quello di simulare il "lancio della moneta" ed estende un seme di breve lunghezza, veramente casuale, in una lunga sequenza di valori che si suppone comportarsi come una sequenza veramente casuale.

1.6 Bontà di un generatore

Per valutare la bontà di un generatore vengono considerati il periodo del generatore e la casualità della sequenza. Per il periodo è semplice: più è lungo, migliore sarà il generatore; mentre per la casualità, è difficile attribuire l'aggettivo casuale ad una sequenza di numeri. Prendiamo come esempio due serie di 1 e 0:

0, 1, 0, 1, 0, 1, 0, 1, 0, 1, ...
1, 1, 0, 1, 0, 1, 1, 0, 0, 1, ...

Apparentemente si è portati e definire la prima sequenza deterministica, poiché si riconosce una certa periodicità o un algoritmo capace di generarla; la seconda sembra essere "più casuale" e non si trova alcuna regola capace di formarla.
In realtà entrambe potrebbero essere generate dal lancio di una moneta come da un algoritmo. Il "buon senso" quindi non basta per dare una valutazione ed esistono altri criteri più obiettivi fra i quali:

- *criterio di Turing*. Se una sequenza numerica generata da estrazioni meccaniche è apparentemente indistinguibile da quella generata da un algoritmo, allora anche quest'ultima può considerarsi casuale.
- *criterio di Von Neumann*. Esistono dei procedimenti matematici per determinare la "bontà" di un generatore di numeri casuali. Questi metodi si fondano sull'assunto che ogni numero casuale sia equiprobabile e dunque l'estrazione di N numeri diversi dia luogo ad una distribuzione uniforme.

Da questo si nota che il primo criterio resta legato al "buon senso", mentre il secondo è più obiettivo ed introduce all'adattamento.

1.7 Caratteristiche dei PRNG

Analizziamo ora le caratteristiche che determinano la qualità di un generatore. Oltre al periodo, altri parametri di valutazione dei PRNG sono: Efficienza, Ripetibilità e Portabilità.

Analizziamoli nel dettaglio:

- **Periodo**: I migliori PRNG sono quelli caratterizzati da periodi lunghi. Valori di ρ prossimi a S assicurano che il sistema non entri in cicli brevi e prevedibili.

- **Efficienza**: Un buon PRNG è un software che utilizza un quantitativo ridotto di risorse computazionali, misurate in termini di memoria allocata.

- **Ripetibilità**: Gli PRNG devono essere in grado di riprodurre esattamente la stessa sequenza di numeri pseudo casuali partendo dallo stesso stato iniziale.

- **Portabilità**: Gli algoritmi PRNG devono essere realizzati in maniera da essere il più possibile indipendenti dal contesto hardware e software in cui sono implementati.

Oltre a queste 4 caratteristiche, per valutare la qualità di un PRNG è fondamentale valutare l'uniformità della distribuzione dei numeri generati.

1.8 PRNG crittograficamente sicuri (CSPRNG)

I CSPRNG sono generatori di numeri pseudocasuali crittograficamente sicuri, il che significa che hanno proprietà tali da renderli adatti per la crittografia. La qualità della casualità richiesta varia in base all'applicazione considerata. Per esempio, creare una chiave di sessione in alcuni protocolli richiede solo l'unicità, mentre la generazione delle master key richiede una qualità più alta, o meglio, un'entropia più alta. Nel caso poi del one-time pad, la teoria impone che per avere una segretezza perfetta la chiave dev'essere ottenuta da una vera sequenza casuale con alta entropia.
Idealmente, la generazione dei numeri casuali nei CSPRNG usa l'entropia ottenuta da una sorgente di alta qualità, la quale può essere un generatore hardware di numeri casuali. Da un punto di vista della teoria dell'informazione, l'entropia che può essere generata è uguale all'entropia fornita dal sistema. Ma qualche volta, nelle situazioni pratiche, sono richiesti più numeri casuali rispetto all'entropia disponibile ed inoltre, i processi per estrarre la casualità da un sistema sono molto lenti. In tali circostanze può essere usato un CSPRNG, dato che può estendere l'entropia disponibile su più bit.
I requisiti di un PRNG ordinario sono soddisfatti anche da un PRNG crittograficamente sicuro, ma non è vero il contrario.

I requisiti di un CSPRNG cadono in due gruppi, infatti questi devono:

- passare i test di casualità statistica (*next-bit test*);
- resistere bene ad attacchi seri anche quando parte del loro stato iniziale (o corrente) sia disponibile ad un attaccante (*attacco di compromissione dello stato*);

Molti PRNG non sono adatti per essere dei CSPRNG e non soddisfano nessuno dei due requisiti; altri invece passano i test statistici ma non resistono ad attacchi di ingegneria inversa.
I CSPRNG sono esplicitamente progettati proprio per resistere alla crittoanalisi.

1.8.1 CSPRNG: Generatore Fortuna

Fortuna è un generatore di numeri casuali crittograficamente sicuro ed è stato chiamato così in onore della dea romana della fortuna dai suoi due inventori Bruce Schneier e Niels Ferguson. E' composto dalle seguenti parti:

- *Generatore*. Una volta che il suo seme è inizializzato produce una quantità indefinita di numeri pseudocasuali;
- *Accumulatore di entropia*. Seleziona i numeri casuali da varie sorgenti e li usa per inizializzare di volta in volta il seme del generatore (*reseeding*) quando è disponibile abbastanza entropia;
- *Seme*. Memorizza un numero sufficiente di stati per abilitare il computer ad iniziare la generazione di numeri casuali non appena è avviato.

Capitolo 2. Generatori di bit pseudo casuali

2.1 Introduzione

Un generatore di bit pseudo casuali, *pseudorandom bit generator* (PRBG), è un algoritmo deterministico che, data una sequenza binaria realmente casuale di lunghezza *k*, restituisce una sequenza binaria di lunghezza $l \gg k$, che sembra essere casuale. L'ingresso del PRBG è chiamato seme, mentre l'uscita del PRBG è chiamata sequenza di bit pseudo casuali. Qui di seguito viene data una definizione più formale:

Definizione: Siano *k*, *l* due interi positivi tali che $l \geq k+1$. Un generatore di bit (k,l) è una funzione $f:(\mathbb{Z}_2)^k \to (\mathbb{Z}_2)^l$ che può essere calcolata in tempo polinomiale (come una funzione di k). L'input $s_0 \in (\mathbb{Z}_2)^k$ è detto *seed* (seme), l'output $f(s_0) \in (\mathbb{Z}_2)^l$ è la *stringa di bit generata.* E' necessario che *l* sia una funzione polinomiale di *k*.

L'uscita di un PRBG non è casuale. Lo scopo è di considerare una piccola sequenza realmente casuale e di espanderla in una sequenza molto più grande, in modo tale che un attaccante non posso distinguere efficientemente la sequenza di uscita del PRBG dalla sequenza realmente casuale.

Si dice che un generatore di bit pseudo casuali passa il test next-bit se non si conoscono algoritmi eseguibili in tempo polinomiale che, ricevendo in ingresso i primi *l* bit di una sequenza di uscita *s*, riescano a prevedere il bit $(l+1)$ di *s* con probabilità statisticamente più grande di $1/2$. Se il PRBG passa il test next-bit è chiamato generatore di bit pseudo casuali crittograficamente sicuro (CSPRBG).

2.2 Generatore ANSI X9.17

Uno dei più robusti PRBG (crittograficamente parlando) è specificato nell'ANSI X9.17. Molte applicazioni utilizzano questa tecnica, incluse le applicazioni di sicurezza finanziaria e il PGP.

La figura 1 mostra l'algoritmo, che utilizza il Triplo DES (EDE) per la cifratura. I parametri sono i seguenti:

- **Input:** il generatore è guidato da due ingressi pseudo casuali. Uno è una rappresentazione a 64 bit del giorno e dell'ora corrente (timestamp), che è aggiornata a ogni generazione del numero. L'altro è un seme a 64 bit; quest'ultimo è inizializzato con un valore arbitrario ed è aggiornato durante il processo di generazione.
- **Chiavi:** il generatore utilizza 3 moduli di cifratura TripleDES. Tutti e tre fanno uso della stessa coppia di chiavi a 56 bit, che devono essere mantenute segrete ed essere utilizzate solo per la generazione di numeri pseudo casuali.
- **Output:** l'uscita è costituita da un numero pseudo casuale a 64 bit e da un seme a 64 bit.

Sono definite le seguenti quantità:

DT_i Valore Giorno/Ora all'inizio dell'i-esima iterazione

V_i Valore del seme all'inizio dell'i-esima iterazione

R_i Numero pseudo casuale generato all'i-esima iterazione

K_1, K_2 Chiavi DES utilizzate in ogni fase

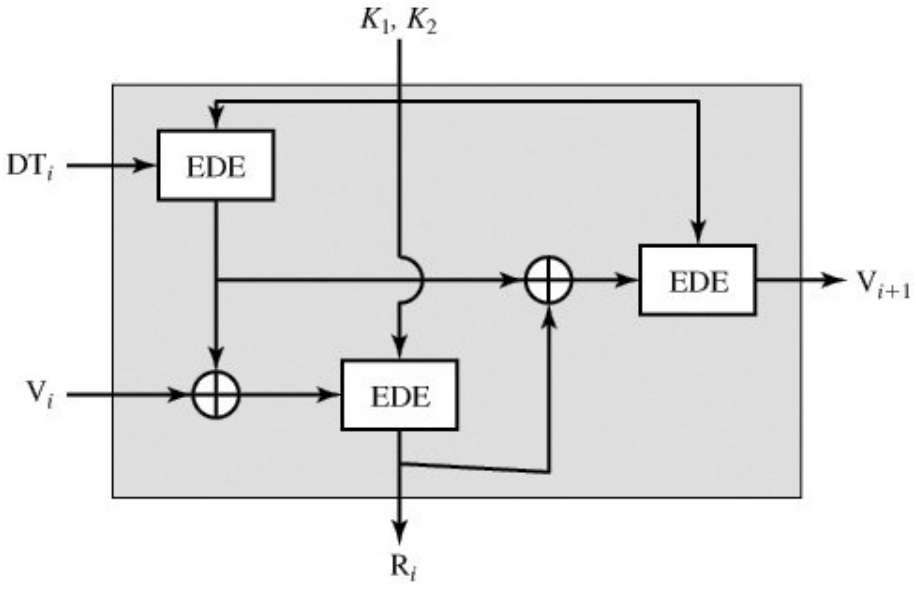


Fig.1

Si ha:

$$R_i = EDE\left(\left[K_1, K_2\right], \left[V_i \oplus EDE\left(\left[K_1, K_2\right], DT_i\right)\right]\right)$$

$$V_{i+1} = EDE\left(\left[K_1, K_2\right], \left[R_i \oplus EDE\left(\left[K_1, K_2\right], DT_i\right)\right]\right)$$

dove $EDE\left(\left[K_1, K_2\right], X\right)$ indica la sequenza di operazioni cifratura - decifratura - cifratura che utilizza due chiavi TripleDES per cifrare *X*.

Diversi fattori contribuiscono alla forza crittografica di questo metodo. La tecnica richiede una chiave a 112 bit e tre cifrature EDE per un totale di nove cifrature DES. Lo schema è guidato da due ingressi pseudo casuali, la data e l'ora e un seme prodotto dal generatore che è distinto dal numero pseudo casuale prodotto dal generatore. Perciò, l'ammontare del materiale che deve essere compromesso da un attaccante è scoraggiante. Anche se un numero pseudo casuale R_i è compromesso, sarebbe impossibile dedurre V_{i+1} da R_i perché viene utilizzata un'operazione EDE aggiuntiva per produrre V_{i+1}. Ogni stringa di bit in uscita R_i può essere utilizzata come vettore d'inizializzazione (IV) per il funzionamento del DES; per ottenere, invece, una chiave per il DES da R_i, si considerano 56 bit di R_i, i restanti 8 bit sono utilizzati per il controllo di parità.

2.3 Linear Congruential Generator (LCG)

Un generatore di bit molto noto (ma insicuro) è il *Linear Congruential Generator* (Generatore Congruenziale Lineare), presentato nell'Algoritmo 1.

L'idea alla base è di generare una sequenza di residui modulo *M*, in cui ciascun elemento della sequenza è una certa funzione lineare modulo *M* del precedente elemento della sequenza. Il seme sarà un residuo modulo *M*, e i bit meno significativi degli elementi della sequenza formano la stringa di bit generati.

Algoritmo 1: *Linear Congruential Generator*

Supponiamo che $M \geq 2$ sia un intero e che $1 \leq a, b \leq M - 1$.

Definiamo $k = 1 + \lfloor \log_2 M \rfloor$ e sia $k + 1 \leq l \leq M - 1$.

Il seme è un intero s_0 , con $0 \le s_0 \le M-1$. (Si noti cha la rappresentazione binaria del seme è una stringa di lunghezza non superiore a k; comunque non tutte le stringhe di lunghezza di k sono semi ammissibili). Per $1 \le i \le l$ si eseguono i seguenti passi:

1. $s_i = (as_{i-1} + b) \bmod M$.
2. $z_i = s_i \bmod 2$.
3. La sequenza d'uscita è $f(s_0) = (z_1, z_2, ..., z_l)$.

f è un (k,l)-*Linear Congruential Generator* (generatore lineare congruenziale). Qui di seguito c'è un piccolo esempio per illustrare il funzionamento dell'*LCG*. Questo esempio mostra anche la natura periodica dei PRBG, vale a dire, che i PRBG eventualmente si ripetono se sono generati un numero sufficiente di bit.

Esempio 1: Supponiamo di costruire un generatore di bit lineare congruenziale (5, 10), prendendo $M = 31$, $a = 3$ e $b = 5$ per l'*LCG*. Consideriamo la mappatura $s \to 3s + 5 \bmod 31$. Se il seme è 13 si ha che 13 → 13, gli altri 30 residui sono permutati in un ciclo di lunghezza 30,

0, 5, 20, 3, 14, 16, 22, 9, 1, 8,

29, 30, 2, 11, 7, 26, 21, 6, 23, 12,

10, 4, 17, 25, 18, 28, 27, 24, 15, 19.

Se il seme è qualcosa di diverso da 13, allora questo rappresenta un punto di partenza nel ciclo, e i prossimi 10 elementi, residui modulo 2, formano la sequenza pseudo casuale.

Le possibili 31 stringhe di bit prodotte da questo generatore sono mostrate in Tabella 1.

Per esempio, la sequenza costruita dal seme 0 si ottiene prendendo i 10 interi seguenti lo 0 della sequenza mostrata prima, cioè, 5, 20, 3, 14, 16, 22, 9, 1, 8, 29, e la riduzione in modulo 2.

Tabella 1:

Stringa di bit prodotta da un *LCG*

Seme	Sequenza	Seme	Sequenza
0	1010001101	16	0110100110
1	0100110101	17	1001011010
2	1101010001	18	0101101010
3	0001101001	19	0101000110
4	1100101101	20	1000110100
5	0100011010	21	0100011001
6	1000110010	22	1101001101
7	0101000110	23	0001100101
8	1001101010	24	1101010001
9	1010011010	25	0010110101
10	0110010110	26	1010001100
11	1010100011	27	0110101000
12	0011001011	28	1011010100
13	1111111111	29	0011010100
14	0011010011	30	0110101000
15	1010100011		

Questi generatori sono comunemente utilizzati per le simulazioni e per algoritmi probabilistici, sono predicibili e quindi completamente insicuri per scopi crittografici: data una sequenza parziale di uscita, si può ricostruire il resto della sequenza, anche se non si conoscono i parametri *a, b* e *M.* Infatti se Eve scopre quattro valori prodotti s_0, s_1, s_2, s_3 risolvendo il sistema di equazioni

$$s_1 = as_0 + b \bmod M$$
$$s_2 = as_1 + b \bmod M$$
$$s_3 = as_2 + b \bmod M$$

riesce a trovare *a, b* e *M.*

2.4 Lagged Fibonacci Generator (LFG)

L'algoritmo lagged_Fibonacci per generare numeri pseudo casuali nasce dal tentativo di generalizzare il metodo delle congruenze lineari che, come noto, è dato dalla ricorrenza lineare

$$X_{n+1} = (aX_n + c) \bmod m .$$

Uno dei motivi che spingono alla ricerca di generatori nuovi è la necessità - utile per molte applicazioni, specie nel calcolo parallelo - di allungare il periodo del generatore. Il periodo di un generatore lineare quando *m* è approssimativamente pari alla grandezza della parola del computer, è all'incirca dell'ordine di 10^9, sufficiente per molte applicazioni ma non per tutte.

Una delle tecniche indagate è quella di far dipendere X_{n+1} da entrambi X_n e X_{n-1} invece che solo da X_n; allora il periodo può arrivare vicino al valore m^2 perché la sequenza non si ripeterà finché non si avrà

$$(X_{n+\lambda}, X_{n+\lambda+1}) = (X_n, X_{n+1})$$

Il più semplice generatore di questo tipo è la successione di Fibonacci

$$X_{n+1} = (X_n + X_{n-1}) \bmod m$$

Questo generatore è stato analizzato negli anni '50 e fornisce un periodo *m*, ma la sequenza non supera i più semplici test statistici.

Anche i generatori del tipo

$$X_{n+1} = (X_n + X_{n-k}) \bmod m$$

pur migliori della successione di Fibonacci, non danno risultati soddisfacenti. Bisogna attendere il 1958 quando Mitchell e Moore propongono la sequenza

$$X_n = (X_{n-24} + X_{n-55}) \mod m,\ n \geq 55$$

dove *m* è pari e $X_0, \ldots X_{54}$ sono interi arbitrari non tutti pari. Le costanti 24 e 55 non sono scelte a caso, sono numeri che definiscono una sequenza i cui bit meno significativi $(X_n \bmod 2)$ hanno un

periodo di lunghezza $2^{55}-1$; perciò la successione (X_n) deve avere un periodo di lunghezza almeno $2^{55}-1$. In realtà la successione ha periodo $2^{M-1}\left(2^{55}-1\right)$ dove $m=2^M$.

I numeri 24 e 55 sono comunemente chiamati *lags* e la (X_n) si dice essere una successione lagged-Fibonacci (LFG).

La successione LFG può essere generalizzata in

$$X_n=\left(X_{n-l}+X_{n-k}\right) \mod 2^M$$

$$l>k>0$$

ma solo alcune coppie (k,l) danno periodi sufficientemente lunghi; in questi casi si dimostra che il periodo è $2^{M-1}\left(2^l-1\right)$. Le coppie (k,l) devono essere scelte in modo opportuno. L'unica condizione sui primi *l* valori è che almeno uno di essi deve essere dispari (altrimenti la successione sarà composta di numeri pari).

2.5 Mersenne Twister

Mersenne Twister è un algoritmo per la generazione di numeri pseudocasuali di tipo lineare congruenziale sviluppato nel 1997 da Makoto Matsumoto e Takuji Nishimura.

Analizziamo nel dettaglio le caratteristiche che hanno reso popolare questo PRNG:

1. È stato progettato per avere un periodo a dir poco colossale: $2^{19937}-1$. Questo periodo spiega l'origine del nome: è un Numero primo di Mersenne e alcune delle costanti dell'algoritmo sono anch'esse numeri primi di Mersenne.

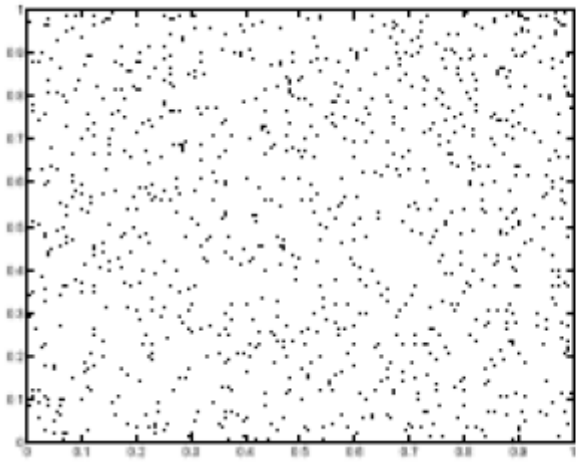
2. Permette di generare punti equidistribuiti in spazi fino a 623 dimensioni (molti altri generatori che sono "buoni" per selezionare singoli numeri mostrano le loro mancanze utilizzando N valori consecutivi per selezionare un punto in uno spazio N-dimensionale).

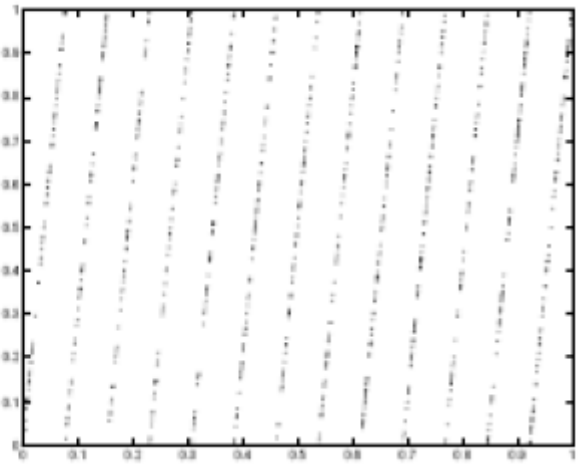
3. È più veloce della maggior parte degli altri algoritmi, compresi quelli notevolmente inferiori in quanto a qualità.

4. Ha passato numerosi test statistici di casualità.

Il Mersenne Twister e i generatori lineari congruenziali hanno la capacità di generare sequenze più o meno correlate.

Immaginiamo di avere a disposizione un PRNG e di generare una serie di sequenze di numeri pseudo casuali di lunghezza uniforme k. Utilizziamo queste sequenze per identificare una serie di punti all'interno di uno spazio k-dimensionale. Se i numeri generati fossero realmente casuali, i punti individuati dalle sequenze si distribuirebbero in maniera uniforme nello spazio. Nella realtà i punti si distribuiscono in piani k-1 dimensionali. Prendiamo per esempio una serie di sequenze di numeri pseudo casuali di lunghezza pari a 2 (cioè k = 2). Se le sequenze generate fossero statisticamente simili a sequenze realmente casuali e quindi non-correlate, riportando i numeri su un piano cartesiano dovremmo osservare una distribuzione uniforme dei punti (immagine di sinistra). Nel caso contrario, i numeri finirebbero per disporsi all'interno di aree circoscritte del piano, come illustrato nell'immagine di destra:





2.6 Generatori di bit pseudocasuali crittograficamente sicuri (CSPRBG)

In questo paragrafo sono presentati dei generatori di bit pseudocasuali crittograficamente sicuri.

2.6.1 Generatore RSA

Il *Generatore RSA* sceglie un elemento iniziale in $\mathbb{Z}_n$ (interi modulo *n*) come seme. Si forma una sequenza di elementi di $\mathbb{Z}_n$, in cui ciascun elemento della sequenza è la cifratura RSA dell'elemento precedente. I bit meno significativi degli elementi della sequenza formano la stringa $z_1, z_2, ..., z_l$ di lunghezza l. Il generatore RSA di bit pseudo casuali è generatore di bit pseudo casuali crittograficamente sicuro se si assume che sia intrattabile l'RSA-problem (dati *b* e *n* calcolare le radici *b*-esime modulo *n*, non si sa se l'RSA-problem è equivalente alla fattorizzazione).

Algoritmo. *RSA Generator*

1. Si considerano due numeri primi *p* e *q*, e si calcola $n = pq$ e $\phi(n) = (p-1)(q-1)$.
2. Si seleziona un intero casuale *b*, $1 \le b \le \phi(n)$ tale che $MCD(b, \phi(n)) = 1$. Come sempre, *n* e *b* sono pubblici mentre *p* e *q* sono privati.
3. Si seleziona un intero casuale s_0 (il seme) in $\mathbb{Z}_n^*$ (gruppo moltiplicativo di $\mathbb{Z}_n$).
4. Per $1 \le i \le l$
   4.1 $s_i = s_{i-1}^b \bmod n$.
   4.2 $z_i = s_i \bmod 2$.
5. La sequenza di uscita è $f(s_0) = (z_1, z_2, ..., z_l)$.

f è un (*k, l*)-*RSA Generator*.

Esempio: Supponiamo *n* = 91261 = 263 × 347, *b* = 1547, e s_0 = 75634.

I primi 20 bit prodotti dall'*RSA Generator* sono calcolati come mostrato nella Tabella 1. La stringa di bit risultante per questo seme è

10000111011110011000.

Tabella 1:

Bit prodotti dall'*RSA Generator*

i	s_i	z_i
0	75634	
1	31483	1
2	31238	0
3	51968	0
4	39796	0
5	28716	0
6	14089	1
7	5923	1
8	44891	1
9	62284	0
10	11889	1
11	43467	1
12	71215	1
13	10401	1
14	77444	0
15	56794	0
16	78147	1
17	72137	1
18	89592	0
19	29022	0
20	13356	0

2.6.2 Blum-Blum-Shub Generator (BBS)

Uno dei più famosi PRBG è il Blum-Blum-Shub Generator. Il generatore di bit pseudo casuali Blum-Blum-Shub (anche conosciuto come generatore $x^2 \bmod n$ o generatore BBS) è un CSPRBG se si assume che la fattorizzazione degli interi sia intrattabile.

Per ogni numero intero dispari n, indichiamo con $\mathrm{QR}(n)$ i residui quadratici modulo n, dove $\mathrm{QR}(n) = \left\{x^2 \bmod n : x \in \mathbb{Z}_n^*\right\}$. Il Blum-Blu-Shub Generator è presentato nell'Algoritmo1.

Algoritmo1: *Blum-Blum-Shub Generator*

Siano *p, q* due numeri primi ciascuno congruo a 3 modulo 4, $p \equiv q \equiv 3 \bmod 4$, e sia $n = pq$. Sia $\mathrm{QR}(n)$ l'insieme dei residui quadratici modulo *n*.

Un seme s_0 è un qualsiasi elemento di $\mathrm{QR}(n)$.

6. Per $1 \le i \le l$
   6.1 $s_i = s_{i-1}^2 \bmod n$.
   6.2 $z_i = s_i \bmod 2$.
7. La sequenza d'uscita è $f(s_0) = (z_1, z_2, ..., z_l)$

f è una (*k,l*)-PRBG, chiamato *Blum-Blum-Shub Generator* (*BBS Generator)*.

Un modo per scegliere un seme appropriato è quello di selezionare un elemento $s_{-1} \in \mathbb{Z}_n^*$ e calcolare $s_0 = s_{-1}^2 \bmod n$. Questo garantisce che $s_0 \in \mathrm{QR}(n)$.

Il generatore lavora molto semplicemente. Dato un seme $s_0 \in \mathrm{QR}(n)$, si calcola la sequenza $s_1, s_2, ..., s_l$ da quadrature successive modulo *n*, e poi si riducono gli s_i modulo 2 per ottenere z_i. Ne segue che

$$z_i = \left(s_0^{2^i} \bmod n\right) \bmod 2, \qquad 1 \le i \le l.$$

Di seguito ci sono due semplici esempi di *BBS Generator.*

Esempio1: Supponiamo che *n* = 192649 = 383 × 503 e $s_0 = 101355^2 \bmod n = 20749$. I primi 20 bit prodotti dal *BBS Generator* sono calcolati come mostrato nella Tabella. Quindi la stringa risultante da questo seme è

11001110000100111010.

Tabella: Bit prodotti dal *BBS Generator*

i	s_i	z_i
0	20749	
1	143135	1
2	177671	1
3	97048	0
4	89992	0
5	174051	1
6	80649	1
7	45663	1
8	69442	0
9	186894	0
10	177046	0
11	137922	0
12	123175	1
13	8630	0
14	114386	1
15	14863	1
16	133015	1
17	106065	1
18	45870	0
19	137171	1
20	48060	0

Esempio2: Sia $n = pq = 7 \cdot 19 = 133$ e sia $s = 100$. Si ottiene $s_0 = 100^2 \bmod 133 = 25$, $s_1 = 25^2 \bmod 133 = 93$, $s_2 = 93^2 \bmod 133 = 4$, $s_3 = 4^2 \bmod 133 = 16$, $s_4 = 16^2 \bmod 133 = 123$. La sequenza d'uscita è 1 0 0 1.

Definizione. Supponiamo che *n* sia un intero positivo dispari e che la fattorizzazione in numeri primi di n sia la seguente:

$$n = \prod_{i=1}^{k} p_i^{e_i} .$$

Sia *a* un intero positivo. Il *Jacobi symbol* $\left(\frac{a}{n}\right)$ è definito come segue:

$$\left(\frac{a}{n}\right) = \prod_{i=1}^{k} \left(\frac{a}{p}\right)^{e_i} .$$

Supponiamo, ora, che *p* e *q* siano due primi dispari distinti, e sia $n = pq$.

Dalla definizione del *Jacobi symbol*, è facile vedere che

$$\left(\frac{x}{n}\right) = \begin{cases} 0 \quad \text{se } MCD(x,n) > 1 \\ 1 \quad \text{se } \left(\frac{x}{p}\right) = \left(\frac{x}{q}\right) = 1 \text{ o se } \left(\frac{x}{p}\right) = \left(\frac{x}{q}\right) = -1 \\ -1 \quad \text{se uno tra } \left(\frac{x}{p}\right) \text{ e } \left(\frac{x}{q}\right) = 1 \text{ e gli altri sono uguali a -1} \end{cases}$$

Ricordiamo che *x* è un residuo quadratico modulo *n* se e solo se

$$\left(\frac{x}{p}\right) = \left(\frac{x}{q}\right) = 1 .$$

Definiamo

$$\widetilde{\mathrm{QR}}(n) = \left\{ x \in \mathbb{Z}_n^* \setminus \mathrm{QR}(n) : \left(\frac{x}{n}\right) = 1 \right\}.$$

Quindi

$$\widetilde{\mathrm{QR}}(n) = \left\{ x \in \mathbb{Z}_n^* : \left(\frac{x}{p}\right) = \left(\frac{x}{q}\right) = -1 \right\}.$$

Un elemento $x \in \widetilde{\mathrm{QR}}(n)$ è chiamato *pseudo-quadrato* modulo *n*. Non è difficile vedere che $|\mathrm{QR}(n)| = |\widetilde{\mathrm{QR}}(n)| = (p-1)(q-1)/4$.

La sicurezza del *BBS Generator* è basata sull'intrattabilità del **Composite Quadratic Residues** problem, che è definito nel Problema1.

Problema1: Composite Quadratic Residues

istanza: Un intero positivo *n* che è il prodotto tra due primi dispari distinti incogniti *p* e *q*, e un intero $x \in \mathbb{Z}_n^*$ tale che $\left(\frac{x}{n}\right) = 1$.

domanda: $x \in \mathrm{QR}(n)$?

Fondamentalmente, il **Composite Quadratic Residues** problem ci richiede di distinguere residui quadratici modulo *n* da pseudo-quadrati modulo *n*. Questo può non essere più difficile che fattorizzare *n*. Se si può calcolare la fattorizzazione $n = pq$, allora sarà semplice calcolare $\left(\frac{x}{p}\right)$.

Dato che $\left(\frac{x}{n}\right) = 1$, ne segue che *x* è un residuo quadratico modulo *n* se e solo se $\left(\frac{x}{p}\right) = 1$.

Questo non sembra essere un modo efficiente per risolvere il **Composite Quadratic Residues** problem se non è nota la fattorizzazione di *n*. Pertanto, questo problema è intrattabile se non è possibile fattorizzare *n*.

Questa è una proprietà del *BBS Generator* che è importante quando si considera la sua sicurezza. Poiché $n = pq$ con $p \equiv q \equiv 3 \bmod 4$, ne segue che, per ogni residuo quadratico *x*, vi è un'unica radice quadrata di *x* che è anche un residuo quadratico. Questa particolare radice quadrata è chiamata *radice quadrata principale* di *x*. Di conseguenza, la mappatura $x \to x^2 \bmod n$, che è utilizzata per definire il BBS Generator, è una permutazione su $\mathrm{QR}(n)$, l'insieme dei residui quadratici modulo n.

Esempio: Supponiamo che $n = 253 = 11 \text{ x } 23$. Si ha allora

$$\left|\mathrm{QR}(n)\right| = \frac{10\text{x}22}{4} = 55.$$

Si può dimostrare che il BBS Generator in $\mathbb{Z}_{55}$ permuta gli elementi di $\left|\mathrm{QR}(55)\right|$ in un ciclo di lunghezza 1, un ciclo di lunghezza 4, un ciclo di lunghezza 10 e due cicli di lunghezza 20.

Capitolo 3. Test statistici

In questo paragrafo sono presentati alcuni test progettati per misurare la qualità di un generatore che si fa passare per un generatore di bit pseudo casuali. Sebbene non sia possibile fornire una prova matematica che il generatore sia veramente un generatore di bit pseudo casuali, i test qui proposti scoprono alcune debolezze che potrebbero avere i generatori. Questo viene fatto prendendo una sequenza di uscita campione di un generatore e sottoponendola a vari test statistici. Ognuno di questi test determina se la sequenza possiede alcuni attributi che dovrebbe mostrare una sequenza realmente casuale; la conclusione di questi test non è definitiva, ma piuttosto probabilistica. Se si ritiene che la sequenza abbia fallito qualsiasi test, il generatore deve essere scartato poiché non è casuale; altrimenti il generatore deve essere sottoposto ad altri test. Se, invece, la sequenza passa tutti i test statistici, il generatore può essere *accettato* casuale. Più precisamente, il termine "accettato" dovrebbe essere sostituito da "non scartato", poiché passare i test prova solamente che il generatore produce sequenze che hanno alcune caratteristiche delle sequenze casuali.

3.1 La distribuzione Normale e χ^2

La distribuzione Normale

La distribuzione normale si presenta nella pratica quando si sommano un gran numero di variabili casuali indipendenti aventi la stessa media e varianza. Una variabile casuale (continua) X ha distribuzione normale con media μ e varianza σ^2 se la sua funzione di densità di probabilità è definita da

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{\frac{-(x-\mu)^2}{2\sigma^2}\right\}, \quad -\infty < x < \infty \ .$$

X è detta $N(\mu,\sigma^2)$. Se X è $N(0,1)$, si dice che X ha distribuzione normale standard. In Fig.1 è rappresentato un grafico della distribuzione $N(0,1)$.

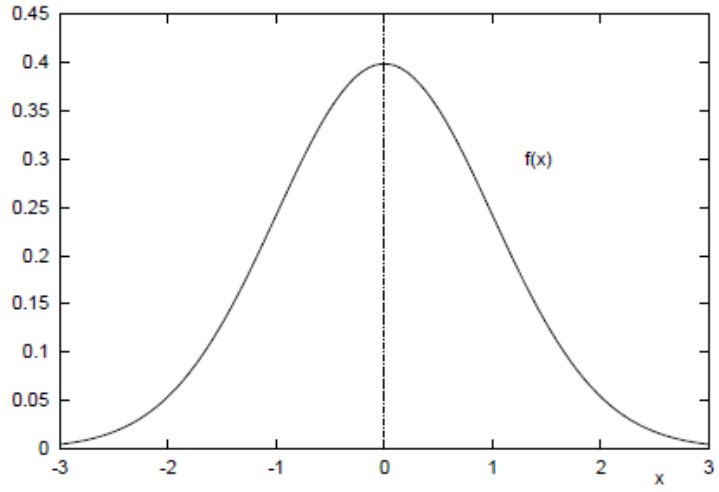


Fig. 1

Il grafico è simmetrico rispetto all'asse verticale, e quindi $P(X > x) = P(X < -x)$ per ogni x. La Tabella 1.1 da dei percentili per la distribuzione normale standard.

α	0.1	0.05	0.025	0.01	0.005	0.0025	0.001	0.0005
x	1.2816	1.6449	1.9600	2.3263	2.5758	2.8070	3.0902	3.2905

Tabella 1.1: Se X è una variabile random avente distribuzione normale standard sia ha che $P(X>x)=\alpha$

La distribuzione χ^2

La distribuzione χ^2 può essere utilizzata per confrontare la bontà di adattamento delle frequenze di eventi osservati alla loro frequenza prevista rispetto a distribuzioni ipotizzate. La distribuzione χ^2 con v gradi di libertà si presenta nella pratica quando si sommano i quadrati di v variabili casuali indipendenti che hanno delle distribuzioni normali standard. Sia $v \geq 1$ un intero. Una variabile casuale (continua) X ha una distribuzione χ^2 con v gradi di libertà se la sua funzione di densità di probabilità è definita da

$$f(x) = \begin{cases} \dfrac{1}{\Gamma(v/2)2^{v/2}} x^{(v/2)-1} e^{-x/2}, & 0 \le x < \infty, \\ 0, & x < 0, \end{cases}$$

dove Γ è la funzione gamma. La media e la varianza della distribuzione sono $\mu = v$, e $\sigma^2 = 2v$. In Fig. 2 è rappresentato un grafico della distribuzione χ^2 con $v = 7$ gradi di libertà.

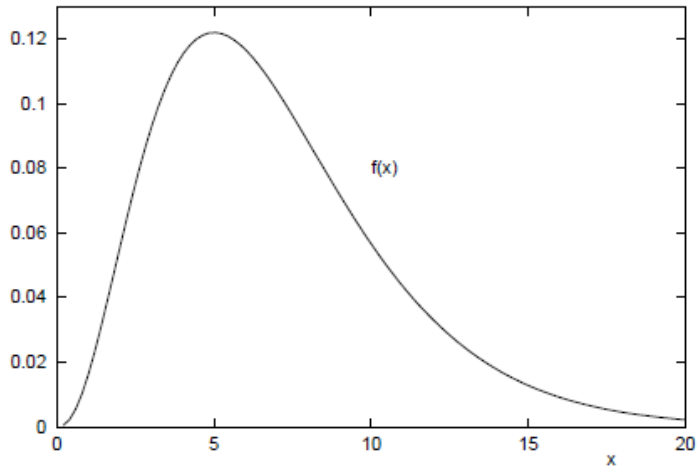


Fig. 2

La Tabella 2.2 da dei percentili per la distribuzione χ^2 per vari gradi di libertà. Per esempio se si pone $v = 5$ e $\alpha = 0.05$ si ha $x = 11.0705$.

ν	α					
	0.100	0.050	0.025	0.010	0.005	0.001
1	2.7055	3.8415	5.0239	6.6349	7.8794	10.8276
2	4.6052	5.9915	7.3778	9.2103	10.5966	13.8155
3	6.2514	7.8147	9.3484	11.3449	12.8382	16.2662
4	7.7794	9.4877	11.1433	13.2767	14.8603	18.4668
5	9.2364	11.0705	12.8325	15.0863	16.7496	20.5150
6	10.6446	12.5916	14.4494	16.8119	18.5476	22.4577
7	12.0170	14.0671	16.0128	18.4753	20.2777	24.3219
8	13.3616	15.5073	17.5345	20.0902	21.9550	26.1245
9	14.6837	16.9190	19.0228	21.6660	23.5894	27.8772
10	15.9872	18.3070	20.4832	23.2093	25.1882	29.5883
11	17.2750	19.6751	21.9200	24.7250	26.7568	31.2641
12	18.5493	21.0261	23.3367	26.2170	28.2995	32.9095
13	19.8119	22.3620	24.7356	27.6882	29.8195	34.5282
14	21.0641	23.6848	26.1189	29.1412	31.3193	36.1233
15	22.3071	24.9958	27.4884	30.5779	32.8013	37.6973
16	23.5418	26.2962	28.8454	31.9999	34.2672	39.2524
17	24.7690	27.5871	30.1910	33.4087	35.7185	40.7902
18	25.9894	28.8693	31.5264	34.8053	37.1565	42.3124
19	27.2036	30.1435	32.8523	36.1909	38.5823	43.8202
20	28.4120	31.4104	34.1696	37.5662	39.9968	45.3147
21	29.6151	32.6706	35.4789	38.9322	41.4011	46.7970
22	30.8133	33.9244	36.7807	40.2894	42.7957	48.2679
23	32.0069	35.1725	38.0756	41.6384	44.1813	49.7282
24	33.1962	36.4150	39.3641	42.9798	45.5585	51.1786
25	34.3816	37.6525	40.6465	44.3141	46.9279	52.6197
26	35.5632	38.8851	41.9232	45.6417	48.2899	54.0520
27	36.7412	40.1133	43.1945	46.9629	49.6449	55.4760
28	37.9159	41.3371	44.4608	48.2782	50.9934	56.8923
29	39.0875	42.5570	45.7223	49.5879	52.3356	58.3012
30	40.2560	43.7730	46.9792	50.8922	53.6720	59.7031
31	41.4217	44.9853	48.2319	52.1914	55.0027	61.0983
63	77.7454	82.5287	86.8296	92.0100	95.6493	103.4424
127	147.8048	154.3015	160.0858	166.9874	171.7961	181.9930
255	284.3359	293.2478	301.1250	310.4574	316.9194	330.5197
511	552.3739	564.6961	575.5298	588.2978	597.0978	615.5149
1023	1081.3794	1098.5208	1113.5334	1131.1587	1143.2653	1168.4972

Tabella 2.2: Se X è una variabile random avente distribuzione χ^2 con ν gradi di libertà, sia ha che $P(X>x)=\alpha$

3.2 Controllo delle ipotesi

Un'ipotesi statistica, indicata con H_0, è un'asserzione sulla distribuzione di una o più variabili casuali. Un test di un'ipotesi statistica è una procedura, basata sui valori osservati delle variabili casuali, che porta all'accettazione o al rifiuto dell'ipotesi H_0. Il test ci restituisce solo una misura della potenza dell'evidenza fornita dai dati contro l'ipotesi; quindi, la conclusione del test non è definitiva, ma piuttosto probabilistica. (Ipotesi $H_0 \rightarrow$ "*La sequenza è un campione di variabili casuali indipendenti e identicamente distribuite U(0,1)*")

Definizione: *Il livello di significatività* α *di un test dell'ipotesi statistica* H_0 *è la probabilità di scartare* H_0 *quando è vera*.

In questa situazione H_0 sarà l'ipotesi che una data sequenza binaria è stata prodotta da un generatore di bit casuali. Se il livello di significatività α di un test di H_0 è troppo alto, il test potrebbe scartare una sequenza che in realtà è stata prodotta da un generatore di bit casuali (questo errore è chiamato *Type I error*). Invece, se il livello di significatività α di un test di H_0 è troppo basso, c'è il rischio che il test possa accettare delle sequenze, anche se non sono state prodotte da un generatore di bit casuali (questo errore è chiamato *Type II error*). E' quindi importante che il test sia progettato attentamente per avere livelli di significatività α che siano adeguati ai diversi scopi; un livello di significatività α compreso tra 0.001 e 0.05 può essere utilizzato nella pratica. Un test statistico è implementato specificando una statistica su un campione casuale. Le statistiche sono generalmente scelte in modo che possano essere calcolate efficientemente e che abbiano (approssimativamente) una distribuzione $N(0,1)$ o χ^2. Il valore della statistica per una sequenza di uscita campione viene calcolato e confrontato con il valore supposto per una sequenza casuale come descritto di seguito.

1. Si suppone che una statistica X per una sequenza casuale abbia una distribuzione χ^2 con v gradi di libertà e supponiamo che X possa essere presa su grandi valori di sequenze non casuali. Per ottenere un livello di significatività α, si sceglie (utilizzando la tabella 2.2) un valore di soglia x_α tale che $P(X > x_\alpha) = \alpha$. Se il valore della statistica X_s per la sequenza

di uscita campione è tale che $X_s > x_\alpha$, la sequenza fallisce il test; altrimenti, passa il test. Questo test è chiamato *One-sided* test.

2. Si suppone che una statistica X per una sequenza casuale abbia una distribuzione $N(0,1)$ e supponiamo che X possa essere presa su grandi e piccoli valori di sequenze non casuali. Per ottenere un livello di significatività α, si sceglie (utilizzando la tabella 1.1) un valore di soglia x_α tale che $P(X > x_\alpha) = P(X < -x_\alpha) = \alpha/2$. Se il valore della statistica X_s per la sequenza di uscita campione è tale che $X_s > x_\alpha$ o $X_s < -x_\alpha$, la sequenza fallisce il test; altrimenti, passa il test. Questo test è chiamato *Two-sided* test.

3.3 **Cinque test di base**

Sia $s = s_0, s_1, s_2, ..., s_{n-1}$ una sequenza binaria di lunghezza *n*. Si presentano ora cinque test statistici che sono comunemente utilizzati per determinare se la sequenza binaria s possiede alcune specifiche caratteristiche che dovrebbe probabilmente mostrare una sequenza realmente casuale. Si sottolinea ancora che il risultato del test non è definitivo, ma piuttosto probabilistico. Se la sequenza passa tutti e cinque i test, non c'è nessuna garanzia che è stata veramente prodotta da un generatore di bit casuali.

Frequency test (monobit test)

Lo scopo di questo test è di determinare se il numero degli 0 e degli 1 in *s* sia approssimativamente lo stesso, come ci si aspetta da una sequenza casuale. Si indicano con n_0 e n_1, rispettivamente, il numero di 0 e di 1 in *s*. la statistica utilizzata è

$$X_1 = \frac{(n_0 - n_1)^2}{n}$$

che approssimativamente ha una distribuzione χ^2 con 1 grado di libertà se $n \geq 10$.

Serial test (two-bit test)

Lo scopo di questo test è di determinare se il numero di occorrenze di 00, 01, 10, 11, come sottosequenze di s è approssimativamente lo stesso, come ci si aspetta da una sequenza casuale. Si

indicano con n_0 e n_1 il numero di o e di 1 in *s*, e con $n_{00}, n_{01}, n_{10}, n_{11}$, rispettivamente, il numero di occorrenze di 00, 01, 10, 11 in *s*. Si noti che $n_{00} + n_{01} + n_{10} + n_{11} = (n-1)$, poiché è permesso alle sottosequenze di sovrapporsi. La statistica utilizzata è

$$X_2 = \frac{4}{n-1}(n_{00}^2 + n_{01}^2 + n_{10}^2 + n_{11}^2) - \frac{2}{n}(n_0^2 + n_1^2) + 1$$

che approssimativamente ha una distribuzione χ^2 con 2 gradi di libertà se $n \geq 21$.

Poker test

Sia *m* un intero positivo tale che $\left\lfloor \frac{n}{m} \right\rfloor \geq 5 \cdot (2^m)$, e sia $k = \left\lfloor \frac{n}{m} \right\rfloor$. Si divide la sequenza *s* in *k* parti che non si sovrappongono ognuna di lunghezza *m*, e sia n_i il numero di occorrenze di tipo *i* della sequenza di lunghezza m, $1 \leq i \leq 2^m$. Il poker test determina se le sequenze di lunghezza *m* appaiono approssimativamente lo stesso numero di volte in *s*, come ci si aspetta da una sequenza casuale. La statistica utilizzata è

$$X_3 = \frac{2^m}{k}\left(\sum_{i=1}^{2^m} n_i^2\right) - k$$

che approssimativamente ha una distribuzione χ^2 con $2^m - 1$ gradi di libertà. Si noti che il poker test è una generalizzazione del frequency test: ponendo $m = 1$ nel poker test si produce il frequency test.

Runs test

Lo scopo del runs test è di determinare se il numero di sequenze (di 0 o di 1) di varie lunghezze nella sequenza *s* è come ci si aspetta per una sequenza casuale. Il numero previsto di intervalli (o di blocchi) in una sequenza casuale di lunghezza *n* è $e_i = (n - i + 3)/2^{i+2}$. Sia *k* l'intero più grande per il quale $e_i \geq 5$. Siano B_i, G_i, rispettivamente, il numero di blocchi e di intervalli di lunghezza *i* in *s* per ogni i, $1 \leq i \leq k$. La statistica utilizzata è

$$X_4 = \sum_{i=1}^{k} \frac{(B_i - e_i)^2}{e_i} + \sum_{i=1}^{k} \frac{(G_i - e_i)^2}{e_i}$$

che ha approssimativamente una distribuzione χ^2 con $2k-2$ gradi di libertà.

Autocorrelation test

Lo scopo di questo test è di controllare le correlazioni tra la sequenza *s* e una versione (non ciclica) di questa shiftata. Sia *d* un intero fissato, $1 \le d \le \lfloor n/2 \rfloor$. Il n° di bit diversi tra *s* e la sua versione *d*-shiftata è $A(d) = \sum_{i=0}^{n-d-1} s_i \oplus s_{i+d}$, dove $\oplus$ indica l'operatore XOR. La statistica utilizzata è

$$X_5 = 2\left(A(d) - \frac{n-d}{2} \right) / \sqrt{n-d}$$

che ha approssimativamente una distribuzione $N(0,1)$ se $n-d \ge 10$.

Esempio. Consideriamo una sequenza s (non casuale) di lunghezza $n = 160$ ottenuta replicando la seguente sequenza quattro volte:

11100 01100 01000 10100 11101 11100 10010 01001.

(*frequency test*) $n_0 = 84$, $n_1 = 76$ e il valore della statistica X_1 è 0.4.

(*serial test*) $n_{00} = 44$, $n_{01} = 40$, $n_{10} = 40$, $n_{11} = 35$ e il valore della statistica X_2 è 0.6252.

(*poker test*) Qui $m = 3$ e $k = 53$. I blocchi 000, 001, 010, 011, 100, 110, 111 appaiono, rispettivamente, 5, 10, 6, 4, 12, 3, 6, e 7 volte e il valore della statistica X_3 è 9.6415.

(*runs test*) Qui $e_1 = 20.25$, $e_2 = 10.0625$, $e_3 = 5$, e $k = 3$. Ci sono 25, 4, 5, blocchi di lunghezza, rispettivamente, 1, 2, 3 e 8, 20, 12 intervalli di lunghezza , rispettivamente, 1, 2, 3. Il valore della statistica X_4 è 31.7913.

(*autocorrelation test*) Se $d = 8$ si ha $A(8) = 100$. Il valore della statistica X_5 è 3.8933.

Per un livello di significatività $\alpha = 0.05$, le soglie di valori di X_1, X_2, X_3, X_4 e X_5 sono, rispettivamente, 3.8415, 5.9915, 14.0671, 9.4877, e 1.96 (vedi le tabelle 1.1 e 2.2). Quindi la sequenza data s passa il frequency, serial, e poker test, ma fallisce il runs e l'autocorrelation test.

3.4 Maurer's universal statistical test

L'idea di base dietro il test statistico universale di Maurer è che non dovrebbe essere possibile comprimere significativamente (senza perdita d'informazione) una sequenza di output di un generatore di bit pseudo casuale.

Se una sequenza di uscita campione *s* di un generatore di bit può essere compressa significativamente, il generatore dovrebbe essere scartato in quanto difettoso. In realtà, invece di comprimere la sequenza *s*, l'universal statistical test calcola una quantità che è collegata alla lunghezza della sequenza compressa.

L'"universalità" di questo test sta nel fatto che è in grado di rilevare qualsiasi possibile difetto che un generatore di bit potrebbe avere.

Uno svantaggio del test universale nei confronti dei cinque test di base è che richiede sequenze di uscita campioni molto più lunghe per essere efficace.

Maurer's universal statistical test

Fissato un intero *L*, divido la sequenza *s* in blocchi (non sovrapponibili) di *L*-bit scartando i restanti bit. Per ogni *i*, con $1 \le i \le Q + K$, dove $Q + K$ rappresenta il numero totale di blocchi, e sia b_i l'intero che in binario rappresenta l'*i*-esimo blocco. Viene creata una tabella *T* in modo tale che ad ogni passo *T*[*j*] rappresenti la posizione dell'ultima occorrenza del blocco corrispondente all'intero *j*, con $0 \le j \le 2^L - 1$. Q rappresenta il numero dei primi blocchi utilizzati per inizializzare *T* e deve essere almeno pari a $10 \cdot 2^L$. Di conseguenza, *K* rappresenta il numero dei blocchi rimanenti e deve essere almeno pari a $1000 \cdot 2^L$. Per ogni *i*, con $Q + 1 \le i \le Q + K$, sia $A_i = i - T[j]$, dove A_i indica il numero di posizioni dall'ultima occorrenza del blocco b_i.

La statistica utilizzata da Maurer è: $X_u = \frac{1}{k} \sum_{i=Q+1}^{Q+K} \lg A_i$.

Per testare la sequenza *s* viene utilizzato un livello di significatività α compreso tra 0,001 e 0,01.

3.5 Next bit test

Le specifiche che devono soddisfare i PRNG per essere CSPRNG si dividono in due gruppi:

1. devono avere buone proprietà statistiche (devono superare i test di casualità);
2. devono resister bene agli attacchi, anche nel caso in cui vengano scoperte parte delle variabili che dovrebbero rimanere segrete.

Più precisamente:

1. Un CSPRNG deve soddisfare il Next bit test.

2. Un CSPRNG può dirsi tale se nel caso in cui parte o tutto lo stato interno del generatore è stato rilevato (o indovinato) sia impossibile ricostruire i bit casuali generati prima della scoperta. Inoltre se c'è una sorgente di entropia deve essere computazionalmente insostenibile usare la conoscenza dello stato per predirne i cambiamenti futuri.

Next bit test

Dati i primi k bit di una sequenza casuale non esiste alcun algoritmo eseguibile in tempo polinomiale che possa predire il bit k+1 con una probabilità maggiore di ½.

Andrei Yao ha dimostrato nel 1982 che un generatore casuale che passa il Next bit test passerà anche tutti gli altri test statistici di casualità eseguibili in tempo polinomiale.

Definizione: Una funzione $f:\{0,1\}^* \rightarrow \{0,1\}^*$ è next-bit unpredictable se:

1. è calcolabile in tempo polinomiale;

2. l'output è un'estensione dell'input;

3. se l'input (il seme) è casuale, allora l'output passa il Next bit test.

Teorema: Una funzione $f:\{0,1\}^* \rightarrow \{0,1\}^*$ è next-bit unpredictable se e solo se è un generatore pseudo casuale.

Capitolo 4. Attacchi di Crittoanalisi sui PRNG

In questa trattazione discuteremo i possibili attacchi contro i PRNG , daremo delle linee guida per il progetto e per l'utilizzo dei PRNG e concluderemo e con alcuni sviluppi possibili.

4.1 Classi di attacchi

1. *Direct Cryptanalytic Attack*: Quando un attaccante è direttamente in grado di distinguere tra le uscite di un PRNG e output casuali. Questo tipo di attacco non è applicabile a tutti i PRNG. Per esempio, un PRNG usato per generare chiavi triple-DES non può mai essere vulnerabile a questo tipo di attacco, dato che le uscite del PRNG non sono mai viste direttamente.

2. *Input-Based Attack*: Quando un attaccante è in grado di usare la conoscenza o il controllo degli input del PRNG per analizzare crittograficamente il PRNG stesso, cioè per distinguere tra le uscite del PRNG e valori casuali. Questa classe di attacchi può essere suddivisa in *Known-Input Attacks, Replayed_Input Attacks* e *Chosen-Input Attacks*.

3. *State Compromise Extension Attack*: Un attacco di compromissione dello stato S tenta di estendere i vantaggi di un tentativo precedente con esito positivo che ha recuperato S il più possibile. Un attacco del genere riesce quando un attaccante o è capace di recuperare le uscite sconosciute del PRNG prima che S sia compromesso o quando recupera gli output dopo che il PRNG ha raccolto una sequenza di input che l'attaccante non può indovinare. Questa classe di attacchi si suddivide in *Backtracking Attacks , Permanent Compromise Attacks, Iterative Guessing Attacks* e *Meet-in-the-Middle Attacks*.

4.2 Linee guida per l'utilizzo di PRNG vulnerabili

Qui è proposta una lista di modi per proteggere un PRNG contro ognuna delle classi di attacchi che abbiamo discusso.

1. Utilizzare funzioni hash per proteggere le uscite di un PRNG vulnerabile (vulnerabilità a *Direct Cryptanalytic Attacks*).

2. Fare l'hash degli input del PRNG con un contatore o un timestamp (timbro orario) prima di usarlo (vulnerabilità a *Chosen-Input Attacks*).

3. Generare occasionalmente nuovi stati d'inizializzazione del PRNG (per i PRNG che lasciano gran parte dei loro stati immutati una volta inizializzati).

4. Fare particolare attenzione ai punti di partenza del PRNG e ai file sei semi (vulnerabilità a *State Compromise Extension Attacks*).

4.3 Linee guida per il progetto di un PRNG

Proponiamo delle linee guida per sviluppare nuovi PRNG che dovranno resistere alle classi di attacchi descritte precedentemente.

1. **Basare il PRNG su qualcosa di robusto**. Il PRNG dovrebbe essere progettato in modo tale che un *Direct Cryptanalytic Attack* riuscito implichi un attacco riuscito a qualcosa di originariamente crittografico che si credeva essere resistente. Idealmente, questo dovrebbe essere provato.
2. **Assicurarsi che l'intero stato del PRNG cambi più volte nel tempo.** L'intero stato segreto interno dovrebbe cambiare più volte nel tempo. Questo previene che un singolo stato compromesso sia irrecuperabile.
3. **Fare "reseeding catastrofici" del PRNG**. La parte dello stato interno che viene utilizzata per generare le uscite dovrebbe essere separata dal gruppo dell'entropia. Lo stato di generazione dovrebbe essere cambiato solo quando è stata raccolta l'entropia sufficiente per resistere agli attacchi *Iterative Guessing Attacks*, in accordo con valutazioni conservative.
4. **Resistere al backtracking**. I PRNG dovrebbero essere progettati per resistere al *Backtracking Attack*. Idealmente, questo vorrebbe dire che un'uscita t dovrebbe essere non predicibile nella pratica da un attaccante che ha compromesso lo stato del PRNG al tempo $t+1$. Si dovrebbe accertare di passare semplicemente gli stati di un PRNG ad una

funzione unidirezionale (*one-way function*) per ogni uscita, limitando la possibilità di un *Backtracking Attack*.

5. **Resistere a Chosen-Input Attacks**. Gli ingressi del PRNG dovrebbero essere associati nello stato del PRNG in modo tale che, data una sequenza di ingressi non predicibili, un attaccante che inizia a capire lo stato del PRNG ma non la sequenza d'ingresso, e un altro attaccante che inizia a capire la sequenza d'ingresso ma non lo stato, sono entrambi incapaci di indovinare lo stato finale. Questo fornisce delle protezioni contro *Chosen-Input Attacks* e *State Compromise Extension Attacks*.
6. **Riprendersi velocemente dalle compromissioni**. Il PRNG dovrebbe avvantaggiarsi di ogni bit di entropia che riceve in ingresso. Un attaccante che cerca di capire gli effetti di una sequenza di ingressi su un PRNG dovrebbe indovinare l'intera sequenza d'ingresso.

4.4 Problemi aperti

Vi mostriamo importanti aree d'interesse che non abbiamo trattato in questa sezione:

1. **Progetti di PRNG dedicati**. I PRNG esistenti sono tutti costruiti su primitive crittografiche esistenti. Questo solleva la questione se ha senso costruire algoritmi PRNG dedicati. Tipicamente la motivazione di costruire un algoritmo dedicato è di migliorare le performance.
2. **Prove di sicurezza**. Sarebbe simpatico vedere alcune prove di sicurezza, che dimostrano che facendo alcune classi di attacchi al PRNG è equivalente a rompere un blocco cifrato, un flusso cifrato o una funzione hash.
3. **Punti di inizializzazione**. Un modo semplice per un attaccante di compromettere lo stato di un PRNG è di inizializzare il PRNG con uno stato predicibile. Questo aggiunge un altro problema a un progettista, il quale deve assicurare che il suo sistema inizializzi il PRNG con una sequenza non predicibile.
4. **Seme compromesso**. Vorremmo che ci fossero molte più discussioni su come deve resistere uno stato compromesso. Questo è un grande problema pratico, che ha ricevuto un'attenzione relativamente piccola da parte della letteratura.
5. **Analizzare altri PRNG**. Ci sono molti PRNG che qui non sono stati analizzati. In particolare vorremmo occupare una discussione completa sulla classe dei PRNG utilizzati nei PGP e nel Cryptolib di Gutmann. Questi PRNG si adattano ai nostri modelli, ma sono molto diversi da quelli che abbiamo analizzato qui: tipicamente mantengono uno stato

considerabilmente molto più grande, nella speranza di accumulare grandi quantità di entropia.

6. **Sviluppare nuovi PRNG**. Abbiamo discusso su PRNG già esistenti. Siamo interessati a considerare nuovi progetti che propongono di resistere ai nostri attacchi.

Capitolo 5. Fortuna generator

Il *Fortuna generator* è un generatore di numeri pseudo casuali crittograficamente sicuro proposto da Bruce Schneier e Niels Ferguson. E' uno dei generatori di numeri pseudo casuali allo stato dell'arte. Prende il nome da Fortuna, la dea del caso secondo la mitologia romana.

5.1 Algoritmo Fortuna

Essenzialmente, il **Fortuna generator** è composto in tre parti: un accumulatore di entropia, un generatore di numeri pseudo casuali e da un sistema per il Seed File Managment.

1. **Entropy Accumulator.** L'accumulatore di entropia raccoglie informazioni veramente casuali da sorgenti esterne di entropia e le usa come seme per il generatore. L'algoritmo permette di usare di 256 sorgenti di entropia. L'algoritmo Fortuna è stato progettato in modo che il sistema rimarrà sicuro se un attaccante ottiene il controllo di alcune, ma non tutte, fonti di entropia. Questa resistenza è ottenuta attraverso l'uso di 'pools' di entropia (fonti di entropia), come illustrato nella Figura 1.

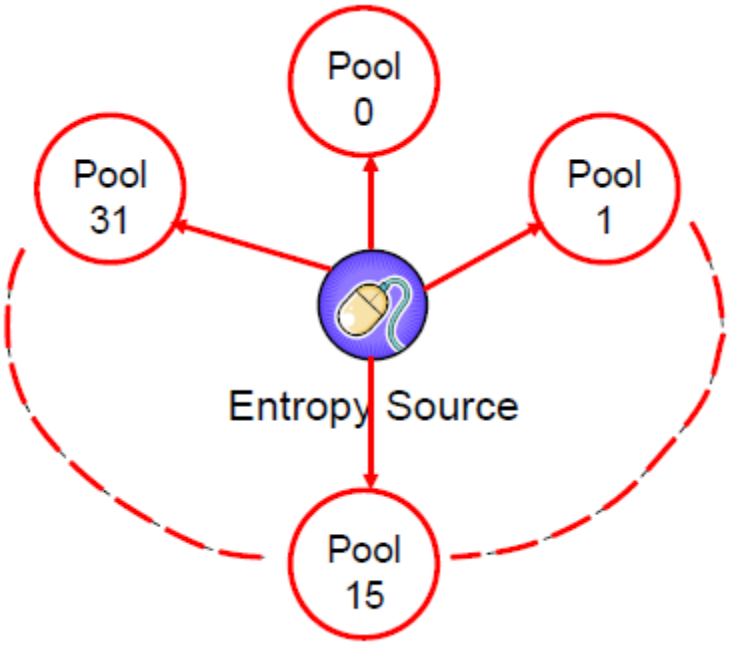


Fig. 1

2. **Generatore.** Il generatore prende un seme casuale di fissate dimensioni dall'accumulatore di entropia e produce arbitrariamente lunghe sequenze di dati pseudo casuali. Il generatore è costituito da un blocco cifrato in modalità counter encryption, come mostrato in Figura 2.

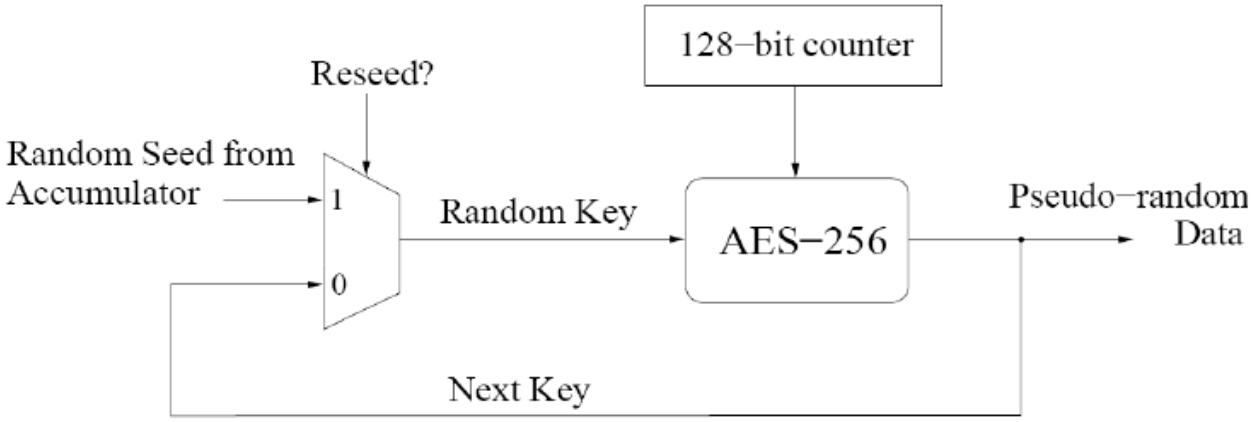


Fig. 2

3. **Seed File Manager.** Al momento dell'inizializzazione, un "seed file" fornisce un seme al generatore. Questo seme iniziale permette al generatore di produrre dati casuali. Il seed file viene letto all'avvio, e un nuovo seme viene immediatamente generato e scritto nel file. Mentre il Fortuna accumula entropia, questo dato è utilizzato per creare una migliore qualità del seme. Un nuovo seed file viene generato approssimativamente ogni 10 minuti, ma questo dipende il larga misura dall'applicazione, e dal tasso di accumulo di entropia.

Tipicamente, nell'implementazione del *Fortuna generator* vengono utilizzate come fonti di entropia il movimento del mouse, il tempo di battitura della tastiera e il rumore della scheda audio.