Generatori di numeri pseudorandom

"la generazione controllata della casualità"

a.a 2009/2010

Giorgia Rossi, Fabio Bottoni, Giacomo Albanese

CORSO DI LAUREA MAGISTRALE

INGEGNERIA DELLE TECNOLOGIE DELLA COMUNICAZIONE E DELL'INFORMAZIONE

Progetto di crittografia

Prof.ssa

Francesca Merola

Sommario

Cap 1:

Cap 2:

Cap 3:

Cap 4:

Cap 5:

Cap 6:

Capitolo 1

1.1 Introduzione

Il progetto scelto e svolto in questa tesina tratterà l'esame approfondito dei generatori di numeri pseudo casuali, anche detti PRNG, che rappresentano una categoria di strumenti che continuano ad acquisire importanza nello sviluppo odierno di nuove tecnologie. Risulta sempre più evidente infatti, la necessità di poter sfruttare sequenze di numeri random per molteplici applicazioni.

Nell'ambito della tematica progetto appena esposta, il gruppo di lavoro ha deciso di focalizzare l'attenzione delle proprie ricerche su tre argomenti principali:

si svolgerà innanzitutto una panoramica sui PRNG per introdurne prima il concetto che vi è alla base, per poi approfondire la definizione sia da un punto di vista generale che più in specifico sui generatori adatti a lavorare in ambito crittografico(CSPRNG). Sono ovviamente presenti alcuni esempi dei primi algoritmi usati per generare numeri casuali in maniera deterministica. In seguito per fornire una giusta e completa idea di come lavorano tali algoritmi, saranno analizzate tutte le caratteristiche e i criteri di bontà che sono ovviamente valutati tramite degli specifici test. Tale sezione del progetto termina con una

rapida discussione sulle differenti applicazioni in cui questi algoritmi trovano impiego.

Nella seconda parte abbiamo analizzato un particolare generatore di numeri pseudo-casuali "crittograficamente sicuro": Fortuna. L'analisi di questo generatore è preceduta da un paragrafo che illustra alcuni possibili attacchi che un hacker può effettuare; in questo modo abbiamo voluto evidenziare l'obiettivo di Fortuna di aumentare la sicurezza del sistema.

Dopo aver fornito le conoscenze teoriche basilari l'ultima parte del progetto è stata realizzata con un approccio sostanzialmente diverso rispetto alle sezioni precedenti. Infatti si è deciso di approfondire l'argomento da un punto di vista pratico andando a studiare come poter effettivamente realizzare dei generatori di numeri casuali. Si approfondirà anche l'abissale differenza che distingue i PRNG da i TRNG.

1.2 Importanza dei numeri casuali

Diverse situazioni necessitano della generazione di numeri casuali, come nel caso delle chiavi crittografiche. Generare sequenze di numeri realmente casuali è molto dispendioso da un punto di vista temporale. Tale motivo solitamente porta alla decisione di optare per l' utilizzo di sequenze che sembrano casuali ma in realtà sono generate in maniera deterministica. Le sequenze di numeri pseudo casuali sono tali da avvicinarsi molto in termini di caratteristiche e proprietà a quelle realmente randomiche. Ovviamente per quanto si possano produrre stringhe di numeri sempre più "somiglianti" a delle vere sequenze casuali, non si potrà mai arrivare ad ottenere numeri che siano effettivamente non deterministici, poiché idealmente questi sono infiniti e non influenzabili da

qualunque tipo di fattore esterno. Si capisce che ciò non è possibile, anche dal solo fatto che l'output di un generatore è sicuramente regolato sia dal tipo di funzione generatrice, sia dall'input che si introduce in ingresso .

1.3 Concetto base di un generatore di numeri pseudo casuali (PRNG)

L'idea del generatore di numeri pseudo casuali è nata, come accennato al paragrafo precedente, dalla necessità di poter lavorare con stringhe random da produrre in modo semplice e veloce.

Una definizione iniziale di PRNG dice che è un algoritmo in grado di generare sequenze di numeri che approssimano le proprietà di quelle di numeri realmente casuali. L'input che viene introdotto nel generatore definisce il suo stato iniziale, ed è anche detto seme del PRNG. Il seme è l'unico elemento realmente casuale, e quello che effettivamente realizza il generatore di numeri pseudo casuali non è altro che una espansione del seme in una stringa più lunga, mediante un processo deterministico.

Poiché l'output è prodotto da una specifica funzione si deduce che l'introduzione dello stesso seme nel PRNG genererà sempre la stessa sequenza di uscita. In base allo stesso asserto si capisce che la stringa emessa non potrà essere infinita. La massima lunghezza della sequenza prima che questa cominci a ripetersi è definita come *periodo* del PRNG, e solitamente si misura in bit. Se il seme introdotto ha lunghezza n misurata in bit, allora il periodo massimo sarà 2^n.

1.4 Definizione formale di PRNG

Un PRNG è un algoritmo deterministico in grado di prendere una sequenza random in ingresso di lunghezza k e restituirne in uscita una di lunghezza l>>k che sembra casuale, ovvero:

Def

Siano k ed l interi positivi tali che l≥k+1. Un generatore di bit(k,l) è una funzione f: $(Z_2)^k \rightarrow (Z_2)^l$ che presenta un tempo computazionale polinomiale (come una funzione di k). L'imput è $s_0 \in (Z_2)^k$ è detto *seme*, l'output $f(s_0) \in (Z_2)^l$ è detto *stringa generata*. È sempre richiesto che l sia una funzione polinomiale di k.

1.5 Sicurezza di un PRNG

Le caratteristiche fondamentali che deve avere un generatore pseudo random sono due:

- Deve essere veloce, ovvero eseguibile in tempi polinomiali.
- Deve essere sicuro.

Risulta molto spesso arduo trovare un compromesso fra questi obbiettivi. L'LFSR (che vedremo in seguito) per esempio è molto veloce ma altrettanto manchevole per quanto concerne la sicurezza.

Per poter parlare di quanto un PRNG sia o meno sicuro, dobbiamo innanzitutto dare una dare una definizione di sicurezza applicata ai generatori in esame. Intuitivamente un PRNG si definisce sicuro se in un tempo polinomiale (che sia

quindi funzione di k o equivalentemente polinomiale in l) è impossibile distinguere il flusso di bit in uscita da uno realmente casuale.

Una prima semplice strategia di distinzione è la seguente.

In media un reale flusso casuale (di lunghezza l) di bit (0e1) ha un numero di 1 pari ad $l/2$, poiché risulta uniformemente distribuito. Invece un generico flusso, sempre di lunghezza l, di bit pseudo casuali, conterrà un numero di 1 pari a $2l/3$.

Quindi sarà molto probabile che un flusso in esame sia pseudo casuale se è verificata la relazione

$$l_1 > \frac{l/2 + 2l/3}{2} = 7l/12$$

Dove l_1 rappresenta il numero di 1 contenuti nella stringa esaminata.

Def

Supponiamo di avere due distribuzioni di probabilità p_0 e p_1 e di volerne determinare la distinguibilità. Le distribuzioni p_0 e p_1 appartengono all'insieme $(\mathrm{Z}_2)^{\mathrm{l}}$ di tutte le stringhe di lunghezza l. Posto $j = 0{,}1$ e $z^l \in (\mathrm{Z}_2)^{\mathrm{l}}$, diremo che $p_j(z^l)$ identifica la probabilità che la stringa z^l abbia proprio la distribuzione di probabilità p_j.

Data poi una funzione dst: $(\mathrm{Z}_2)^{\mathrm{l}} \to \{0{,}1\}$ e posto un $\varepsilon > 0$, si definisce per $j = 0{,}1$

$$E_{dst} = \sum_{\{z^l \in (\mathrm{Z}_2)^{\mathrm{l}} : \mathrm{dst}(z^l) = 1\}} p_j(z^l)$$

e diremo che la funzione dst è ε-distinguente di p_0 e p_1 se vale

$$|E_{dst}(p_0) - E_{dst}(p_1)| \geq \varepsilon$$

con ε piccolo a piacere.

Quindi p_0 e p_1 saranno ε-distinguibili se esiste una funzione ε-distinguente. Tale funzione deve avere un tempo computazionale polinomiale in l.

Alla base di questa definizione appena esposta vi è l'idea che la funzione dst cerchi di decidere quale distribuzione di probabilità (p_0 o p_1) sia quella che più si avvicina alla distribuzione che ha effettivamente prodotto z^l. L'output della dst sarà proprio p_0 o p_1.

La quantità $E_{dst}(p_j)$ rappresenta invece un valore medio degli output di dst su la distribuzione di probabilità p_j con $j = 0,1$.

1.6 Caratteristiche e problemi di un PRNG

Le caratteristiche fondamentali che una sequenza di numeri deve possedere per poter essere definita casuale sono 2, ovvero distribuzione uniforme ed indipendenza.

Solitamente le sequenze sono binarie, per cui per distribuzione uniforme si intende che il numero di 0 e 1 sia circa lo stesso, mentre il secondo requisito stabilisce che non sia possibile trovare qualche relazione che ci permetta di legare numeri adiacenti della stringa tramite una relazione.

Se prendiamo per esempio la sequenza delle cifre decimali

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9$$

Si nota subito come questa abbia una distribuzione uniforme ma che purtroppo non verifichi la seconda caratteristica, poiché ogni numero è legato al precedente dalla relazione

$$n_{i+1} = n_i + 1$$

Riferendoci a quanto appena detto si capisce che i PRNG siano soggetti a vari problemi:

- Spesso il periodo del generatore risulta più corto di quanto in realtà ci si aspetti, a seconda del tipo di seme posti in ingresso.
- Mancanza di uniformità nella distribuzione dei numeri della sequenza.
- Alti valori di correlazione fra i valori generati in output.
- Bassa distribuzione dimensionale della sequenza in uscita.

Per poter verificare la presenza o meno di tali problemi esistono degli specifici test a cui vengono ripetutamente sottoposti i PRNG per valutarne la bontà e le prestazioni (di tale argomento ne riparleremo nel capitolo 3).

1.7 Possibili applicazioni

Le applicazioni dei generatori di numeri pseudo random sono molte, le più importanti sono quelle nell'ambito della simulazione e della crittografia. Nell'ambito della simulazione sono usati per creare campioni su cui poter effettuare delle statistiche. Il metodo più conosciuto è indubbiamente il Metodo Monte Carlo, è usato per trarre stime attraverso simulazioni. Si basa su un algoritmo che genera una serie di numeri tra loro incorrelati, che seguono la distribuzione di probabilità che si suppone abbia il fenomeno da indagare.

In ambito crittografico invece i generatori pseudo casuali acquistano una diversa rilevanza. Infatti lo scopo fondamentale è di generare sequenze che non possano essere determinate da un possibile "nemico".Un pratico esempio che dimostra l'utilità dei PRNG è il seguente.

Il One-Time-Pad teoricamente è in grado di raggiungere la sicurezza perfetta, ma in pratica risulta poco agibile in quanto prevede l'uso di una chiave lunga quanto il testo in chiaro, la quale deve essere trasmessa lungo un canale sicuro.

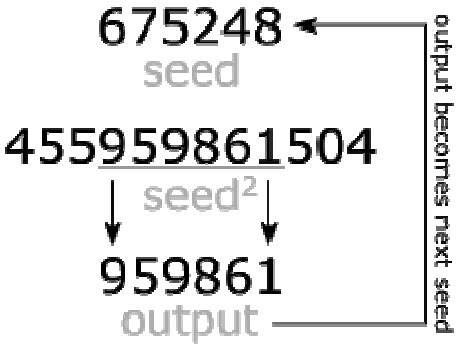
Se pensiamo al seme come ad una chiave e al PRNG come un generatore di chiavi, basterà che Alice trasmetta solo il seme a Bob e poi entrambi saranno in grado di ottenere la stessa sequenza in uscita da usare come chiave (lunga quanto il testo in chiaro). Ovviamente bisogna tenere presente che nel momento in cui usiamo una sequenza deterministica si perde comunque la sicurezza perfetta.

Capitolo 2

2.1 Primi metodi

Uno dei primi metodi proposti per generare numeri in maniera deterministica, fu il quello suggerito nel 1946 da John Von Neumann. Tale algoritmo era noto come *Middle Square Method.*

L'idea di Von Neumann era estremamente semplice. Si doveva prendere valore iniziale che fosse un qualunque numero di 10 cifre e di elevarlo al quadrato. Le 10 cifre nel mezzo del risultato sarebbero andate a costituire il secondo numero pseudo casuale della sequenza in uscita. Quanto appena estratto diventava il nuovo valore in ingresso. Iterando tale processo Von Neumann era in grado di generare una sequenza di numeri pseudo random di 10 cifre. Ovviamente il ragionamento appena spiegato valeva con numeri anche di lunghezze diverse da quelli considerati da Von Neumann. Osservando la figura sottostante è immediato capire il meccanismo del MSM.



Purtroppo le problematiche legate a questo metodo sono molte. Fra tutte spicca il fatto che se durante l'iterazione del processo le 10 cifre mediane sono tutte 0,

il generatore produrrà in uscita sempre e solo numeri di 10 cifre nulle, inoltre se metà della sequenza introdotta in ingresso fosse composta di 0, i seguenti output decrescerebbero fino a 0.

Nonostante ciò questo metodo si dimostrò comunque estremamente rapido.

2.2 Mersenne twister

Nel 1997 ci fu un notevole balzo in avanti nella tecnologia legata a gli algoritmi di generazione deterministica dei numeri. La proposta di un nuovo metodo detto *Mersenne twister*, ad opera di Makoto Matsumoto e Takuji Nishimura, risolse molti problemi che vi erano stati fino ad allora con i PRNG precedenti.

Le caratteristiche di questo metodo che gli hanno permesso di diventare uno dei più sfruttati sono diverse. Prima fra tutti c'è la peculiarità di avere un periodo a dir poco colossale, di $2^{19937} - 1$. Tale periodo molto grande non è una casualità, è stato infatti progettato appositamente per ovviare ai problemi dovuti a dei periodi brevi. In secondo luogo è il Mersenne twister è in grado di produrre punti equamente distribuiti in spazi fino a 623 dimensioni (per valori di 32bit) ed inoltre è uno dei generatori più veloci. L'algoritmo originale tuttavia non è considerato adatto per applicazioni crittografiche, anche se sono state proposte delle sue varianti che invece sarebbero in grado di lavorare in ambito crittografico.

Se pensiamo di avere una serie di sequenze di numeri pseudo random di lunghezza fissa k, possiamo supporre di sfruttare tali sequenze per individuare dei punti su uno spazio k dimensionale. Nel caso in cui i numeri generati dal PRNG fossero realmente casuali, i punti trovati sarebbero distribuiti in maniera

uniforme su tutto lo spazio. Questo è uno dei tanti criteri per valutare la bontà di un generatore pseudo casuale.

Nel capitolo 6 sarà analizzato inoltre un terzo e importante algoritmo di generazione: LCG.

Capitolo 3

3.1 Una classe particolare di PRNG: i CSPRNG

Che cosa è un CSPRNG? Per cosa stanno le lettere dell'acronimo?

Tali domande sono fondamentali e da queste partiremo per aprire il nuovo capitolo interamente dedicato a questi particolari generatori.

Un algoritmo adatto a lavorare in ambito crittografico è detto *Crittograficamente Sicuro*.

La differenza fra PRNG e CSPRNG non è sempre semplice da spiegare, poiché questa seconda tipologia di generatori deve essere in grado di resistere a attacchi ed avere una struttura particolare. Per poter essere abbastanza certi che un determinato PRNG sia crittograficamente sicuro è richiesto un tempo estremamente lungo, necessario ad effettuare tutti i possibili attacchi, con la speranza che in futuro non se ne trovino di più forti.

I requisiti minimi di un CSPRNG sono :

- Una sequenza di numeri prodotta in uscita da un algoritmo deterministico deve essere indistinguibile da una stringa realmente casuale. Quanto detto è verificabile con una serie di test a cui è sottoposto il generatore in esame.
- Deve essere impossibile per un qualunque attaccante, a partire da una data sottosequenza, poter risalire ai valori passati o futuri di output del PRNG.

- Deve essere impossibile per un qualunque attaccante, a partire da un dato stato interno, poter risalire ai valori passati o futuri di output del PRNG.

3.2 Test statistici

Vediamo ora una breve ma sostanziale panoramica dei test più importanti da effettuare su un PRNG per valutarne la bontà.

Tali test prendono in esame una delle possibili sequenze in uscita dal generatore e valutano se questa ha o meno delle caratteristiche proprie delle stringhe di numeri realmente casuali. Poiché tali test sono statistici, il loro responso non potrà mai essere certo, anche se, ovviamente, più saranno i test che passa un algoritmo, maggiore sarà la probabilità che sia effettivamente un generatore di numeri pseudo casuali. Basandomi sulle direttive del NIST(National Institute of Security and Tecnology) i test sono:

Frequency (Monobits) Test

Lo scopo di questo test è verificare se il numero di “1” e “0” in una sequenza sono circa gli stessi che ci si aspetterebbe per una sequenza veramente casuale. Il test valuta la “distanza” della frequenza degli “1” e “0” rispetto a ½, cioè il numero di zeri e di uno che dovrebbero essere contenuti in una sequenza realmente casuale.

Discrete Fourier Transform (Spectral) Test

Questo test focalizza la sua attenzione sull'altezza di picco nella FFT. Lo scopo è di rilevare le caratteristiche periodiche (ad esempio, schemi ripetitivi) nella sequenza testata, che possa indicare una deviazione dal comportamento normale nel caso puramente casuale.

Maurer's Universal Statistical Test

Lo scopo del test è di rilevare se la sequenza possa essere notevolmente compressa senza perdita di informazioni. Una sequenza eccessivamente comprimibile è considerata non casuale.

Next bit test

Dati i primi *k* bit di una sequenza casuale, non deve essere possibile, in un tempo polinomiale, predire il *(k* +1) esimo bit con probabilità di successo superiore a $1/2$. Nel 1982 Andrew Yao ha dimostrato che un generatore in grado di passare il next bit test passerà anche tutte le altre prove statistiche di casualità a tempo polinomiale .

Runs Test

Questo test si propone di valutare la frequenza delle "runs" di varie lunghezze. Con il termine "run" indichiamo una sottostringa di lunghezza variabile k, i cui k elementi siano tutti identici e che è collegata, all'inizio e al termine, al resto della sequenza tramite dei valori opposti. Vediamo un esempio in basso di una run di lunghezza 4:

… 1, 0, 1, **0**, **0**, **0**, **0**, 1,0 ….

In particolare per determinare che la frequenza delle "runs" sia come ci si aspetta da una vera stringa casuale, si fa molta attenzione a dove tale frequenza cresce e dove diminuisce.

3.3 Alcuni CSPRNG

Fra i generatori crittograficamente sicuri quelli più conosciuti, nonché più sfruttati sono:

- I cifrari a flusso.
- I cifrari a blocco.
- Fra i cifrari appositamente disegnati per avere delle caratteristiche specifiche per la sicurezza in ambito crittografico ricordiamo l'algoritmo di Yarrow, il Blum Blum Shub ed il Fortuna.

Di tutti quelli appena elencati si è deciso di approfondirne uno in particolare, il Fortuna. In questo modo si è cercato di spiegare il funzionamento base di uno dei CSPRNG più importanti e dare un'idea di come lavorano in generale i CSPRNG, ricordando d'altra parte che ogni algoritmo ha delle differenze di funzionamento rispetto a gli altri.

Capitolo 4

4.1 Possibili attacchi ad un CSPRNG

La generazione di numeri pseudo-casuali a partire da un seme costituisce un problema relativamente semplice da risolvere; la vera difficoltà è come ottenere un seme casuale e come mantenerlo segreto nelle situazioni del mondo reale.

In qualsiasi momento, il PRNG è caratterizzato da uno stato interno e la richiesta di dati casuali è soddisfatta tramite un algoritmo "crittograficamente sicuro" che genera dati pseudo-casuali.

Tra le varie modalità di attacco ad un PRNG la più semplice è l'attacco diretto, nel quale l'hacker cerca di ricostruire lo stato interno esaminando l'output.

In un PRNG tradizionale (non crittograficamente sicuro), qualora l'hacker riuscisse ad acquisire lo stato interno del generatore, potrebbe seguire tutti gli output e tutti gli aggiornamenti allo stato stesso.

Per ripristinare la sicurezza di un PRNG, una volta che il suo stato sia stato compromesso, utilizziamo i generatori di numeri realmente casuali a disposizione, in grado di fornirci una certa quantità di entropia.

In questo caso facciamo riferimento al concetto di entropia per misurare il grado di casualità.

Considerando, ad esempio, una parola di 32 bit completamente casuale, avremo 32 bit di entropia. D'altra parte, se la parola di 32 bit potesse assumere solo 4

valori (ognuno caratterizzato da una probabilità del 25% di presentarsi) sarà caratterizzata da soli 2 bit di entropia.

Supponiamo di avere a disposizione una o più fonti di numeri realmente casuali, in grado di fornirci una certa quantità di entropia ("evento") in momenti imprevedibili. Nonostante abbiamo introdotto tale quantità di entropia nello stato interno del PRNG, resta sempre possibile una linea di attacco.

L'hacker effettua al PRNG frequenti richieste di dati casuali, in modo che la quantità totale di entropia introdotta tra due sue richieste sia limitata ad un numero ridotto di bit; in questo modo l'hacker può semplicemente provare tutti i possibili input casuali e risalire al nuovo stato interno.

I dati casuali generati dal PRNG fornirebbero all'hacker la prova di aver raggiunto la soluzione corretta.

La miglior difesa contro questo particolare attacco consiste nel raggruppare gli eventi in arrivo che contengono entropia: sommiamo l'entropia finché non ne accumuliamo abbastanza da introdurla nello stato interno senza che un hacker possa risalirvi.

Per capire qual è la quantità minima di entropia da accumulare è necessario utilizzare una funzione di stima. Effettuare tale stima dell'entropia risulta particolarmente difficile nelle situazioni reali perché essa dipende fortemente dalla quantità di informazioni che l'hacker possiede o può procurarsi.

Il generatore Fortuna illustrato in seguito risolve il problema relativo alla definizione della stima dell'entropia eliminandola del tutto.

4.2 Fortuna

Fortuna è un generatore di numeri pseudo-casuali crittograficamente sicuro, proposto da Bruce Schneier e Niels Ferguson all'interno del libro "Practical Cryptography".

Il nome "Fortuna" deriva dalla divinità romana del caso e del destino.

Il generatore si articola in tre parti:

1) generatore: prende in ingresso un seme (dati realmente casuali) di lunghezza fissa e genera quantità arbitrarie di dati pseudo-casuali

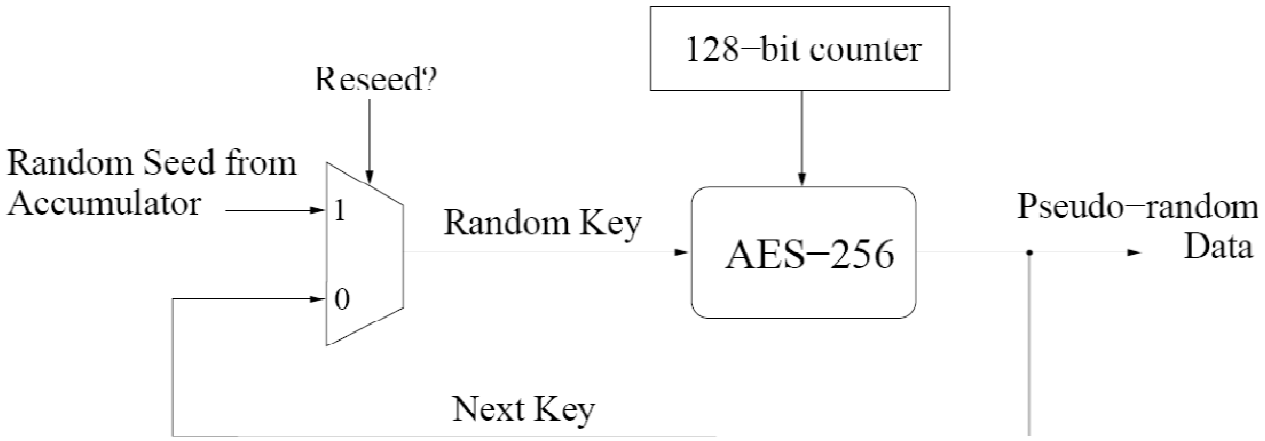
2) accumulatore: raccoglie e capitalizza l'entropia da varie fonti e a volta cambia il seme del generatore

3) controllo sul file del seme: assicura che il PRNG generi dati pseudo-casuali anche al primo avvio del computer

Capitolo 5

5.1 Generatore

Il generatore è realizzato tramite un cifrario a blocchi (AES); l'idea alla base è utilizzare il codice in modalità contatore, cifrando I valori successivi in uscita da un contatore.



Lo stato interno del generatore è caratterizzato da una chiave di 256 bit e da un contatore di 128 bit.

Il PRNG Fortuna, è un generatore di numeri pseudo-casuali crittograficamente sicuro, dunque deve rispettare due criteri:

- passare il "next-bit test": data una sequenza di numeri pseudo-casuali di l bit non dovrebbe essere possibile prevedere il (l+1)-esimo bit con una probabilità di successo maggiore del 50%

- resistere all'attacco “state compromise extension”: qualora l'hacker riuscisse ad acquisire lo stato interno del PRNG dovrebbe essere impossibile ricostruire la sequenza di cifre pseudo-casuali precedente

Per rispettare i criteri di un PRNG crittograficamente sicuro (in particolare il secondo), dopo ogni richiesta di dati casuali, generiamo 256 bit supplementari di numeri pseudo casuali e li utilizziamo come nuova chiave (next key) del codice a blocchi.

Quando cambiamo la chiave del codice a blocchi abbiamo l’accortezza di non riazzerare il contatore. In questo modo evitiamo problemi nel caso di cicli brevi: riazzerando, infatti, il contatore qualora il valore della chiave si dovesse mai ripetere, anche la chiave successiva sarebbe la stessa.

L’aggiornamento del seme (reseed) avviene:

dopo ogni richiesta di dati pseudo casuali da parte di un particolare utente

dopo aver generato 220 dati pseudo casuali, in modo da preservare le proprietà statistiche dell’output

La velocità del generatore di Fortuna è legata alla velocità del codice a blocchi sul quale si basa: su una CPU di un computer dovrebbe impiegare meno di 20 cicli di clock per byte generato.

5.2 Accumulatore

L’accumulatore raggruppa dati casuali reali da varie fonti e li utilizza per aggiornare il seme del generatore.Tra le fonti di casualità si considerano eventi

imprevedibili quali i tasti digitati, i movimenti del mouse e tutte le fonti cronometrabili in modo pratico (orario esatto della pressione dei tasti, delle risposte di dischi rigidi e stampanti,..).

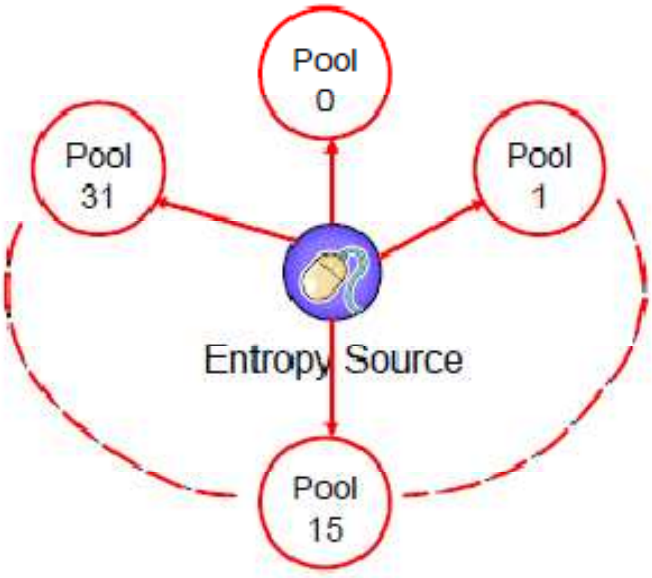
Identifichiamo ogni fonte con un numero univoco nell'intervallo 0,…,255; gli implementatori possono scegliere se allocare tali numeri in modo statico o dinamico.

Infine concateniamo in un'unica stringa le sequenze di byte (evento) provenienti dalle diversi fonti.

Per variare il seme del generatore, raggruppiamo gli eventi in un pool abbastanza grande da impedire ad un eventuale hacker di predire i possibili valori.

L'accumulatore di Fortuna prevede 32 pool: P_0, P_1,…, P_{31} ciascuno contenente concettualmente una stringa di byte di lunghezza illimitata.

Ogni fonte distribuisce ciclicamente i propri eventi casuali sui vari pool, in questo modo si assicura che l'entropia sia distribuita in maniera uniforme sui vari pool.



Il generatore riceve un nuovo seme ogni volta che il pool P_0 è sufficientemente lungo. A seconda della variazione numero r, in esso saranno inclusi uno o più pool: la regola vuole che il pool P_i sia incluso se 2^i è un divisore di r. In questo modo P_0 è utilizzato in ogni variazione, P_1 ogni due, P_2 ogni quattro,..

Una volta che un pool è utilizzato in una variazione è riazzerato.

Nel caso in cui l'hacker abbia poche informazioni riguardo le fonti casuali utilizzate non sarà in grado di prevedere P_0 alla successiva variazione del seme.

D'altra parte qualora l'hacker abbia sufficienti informazioni riguardo P_0, quando viene utilizzato P_1

in una variazione del seme la quantità di dati imprevedibili è doppia per l'hacker.

In definitiva, indipendentemente da quanti eventi possa conoscere l'hacker, se esiste anche un'unica fonte di eventi casuali per lui imprevedibili, ci sarà sempre un pool che immagazzina sufficiente entropia.

Questo sistema si adatta automaticamente alle situazioni e costituisce il vero pregio di Fortuna.

5.3 Gestione del file del seme

Il PRNG Fortuna finora accumula entropia e genera dati casuali dopo il primo rinnovo del seme. Tuttavia, se riavviamo una macchina, dobbiamo attendere che

le fonti di casualità generino un numero di eventi sufficiente a effettuare il primo rinnovo, e soltanto dopo avremo a disposizione dati casuali.

La soluzione consiste nell'impiego di un file del seme, un file contenente entropia e mantenuto dal PRNG.

Dopo un riavvio , il PRNG legge questo file e ne utilizza l'entropia per entrare in uno stato indefinibile.

In linea generale il file viene riscritto poco prima dello spegnimento della macchina; d'altra parte poiché alcuni computer potrebbero spegnersi in modo non regolare è importante garantire l'aggiornamento regolare del file del seme da parte del PRNG dopo l'accumulo di una quantità di entropia sufficiente.

Capitolo 6

6.1 Differenze sostanziali fra PRNG e TRNG

I numeri casuali sono utili per molti scopi,come generare chiavi in crittografia,simulare e modellare fenomeni complessi e fornire tutte le possibili combinazioni degli ingressi nel testing di un qualunque sistema.

Le prime macchine di un certo rilievo furono costruite infatti per studiare la risposta delle centrali telefoniche alla variazione della domanda. Se si parla di singoli numeri ,un numero random è estratto da un set di possibili valori ,ognuno dei quali è equiprobabile,cioè si ha distribuzione uniforme. Se invece parliamo di una sequenza random di numeri,ognuno di questi deve essere statisticamente indipendente dagli altri.

Con l'aumento dei computers ,i programmatori hanno riconosciuto il bisogno d'introdurre casualità nei programmi .

Fare ciò non è semplice,infatti un computer segue le istruzioni pedissequamente ed è totalmente prevedibile(se non si comporta in questo modo è considerato rotto).Esistono due approcci per generare numeri casuali usando un computer:PRNG e TRNG. Ognuno di questi ha i suoi pro e i suoi contro.

PRNG:sono algoritmi che usano formule matematiche per produrre sequenze di numeri che appaiono random. Un buon esempio di PRNG è il metodo congruenziale lineare che verrà illustrato a breve. Gran parte della ricerca in

questo settore si è focalizzata sui PRNG e i moderni algoritmi sono talmente buoni che i numeri appaiono come se fossero veramente random. La principale differenza tra PRNG e TRNG è facile da capire se mettiamo a confronto i numeri generati da un computer e quelli trovati dal lancio di un dado. Mentre i PRNG usano formule matematiche o liste calcolate precedentemente, i TRNG lavorano come se rendessero il computer capace di sostituirsi a un giocatore nel lancio di un dado, o più comunemente usano qualche altro fenomeno fisico che è più facile da collegare a un computer di quanto lo sia un dado. I PRNG sono molto efficienti ,infatti possono produrre molti numeri in pochissimo tempo ,e deterministici ,cioè una certa sequenza di numeri può essere riprodotta in un secondo momento se di questa è conosciuto lo starting point.L'efficienza è un importante caratteristica se l'applicazione che li deve utilizzare ha bisogno di molti numeri e lo è anche il determinismo. I PRNG moderni hanno un periodo così lungo che può essere ignorato per la maggior parte delle applicazioni. I PRNG non sono stilizzatissimi in applicazioni in cui è importante che i numeri siano realmente imprevedibili come nel data encryption (tuttavia esistono algoritmi molto buoni).

TRNG:estraggono la casualità da fenomeni fisici e la elaborano al calcolatore .Si utilizzano anche fenomeni molto semplici come le piccole variazioni nei movimenti di un mouse. In pratica è importante scegliere bene la sorgente di casualità da utilizzare. Una sorgente radioattiva può essere una valida scelta,infatti gli istanti di tempo a cui essa decade sono totalmente impredicibili e sono di facile rilevazione ed acquisizione con un computer.

Un altro fenomeno da considerare è il rumore atmosferico di facile acquisizione anche con un semplice apparecchio radio.

Si possono sfruttare inoltre fenomeni come il rumore termico e l 'effetto fotoelettrico,cioè l 'emissione di elettroni da una superficie solitamente metallica

quando questa viene colpita da una radiazione elettromagnetica avente una certa frequenza. Questi sono tutti fenomeni completamente imprevedibili.

Un generatore di numeri casuali(che sfrutta i predetti fenomeni)contiene tipicamente un amplificatore per portare il prodotto dei fenomeni fisici a dimensione macroscopica ed un trasduttore per convertire l'uscita in un segnale digitale.

Sono anche usati fenomeni macroscopici come le carte da gioco,i dadi o la stessa ruota della roulette.

La loro imprevedibilità può essere giustificata dalla teoria dei sistemi dinamici instabili e dalla teoria del caos.

Queste teorie suggeriscono che anche se i fenomeni macroscopici sono deterministici sotto le premesse della meccanica newtoniana, i sistemi del mondo reale evolvono in un modo che in pratica non può essere predetto perché occorrerebbe conoscere le condizioni iniziali con un accuratezza che cresce esponenzialmente nel tempo.

Characteristic	**Pseudo-Random Number Generators**	**True Random Number Generators**
Efficiency	Excellent	Poor
Determinism	Determinstic	Nondeterministic
Periodicity	Periodic	Aperiodic

Application	**Most Suitable Generator**
Lotteries and Draws	TRNG
Games and Gambling	TRNG
Random Sampling (e.g., drug screening)	TRNG
Simulation and Modelling	PRNG
Security (e.g., generation of data encryption keys)	TRNG
The Arts	Varies

6.2 Metodo della congruenza lineare (LCG)

Tale metodo permette, dato un valore iniziale x 0 detto seme, di ottenere una sequenza di numeri pseudo-casuali mediante l'applicazione ripetuta della seguente formula:

$$x_{i+1} = (a * x_i + c) \pmod m$$

dove:

a è un coefficiente intero strettamente positivo detto moltiplicatore

c è un coefficiente intero non negativo detto incremento

m è un coefficiente intero strettamente positivo detto modulo

x i è il generico numero della sequenza

MOD è l'operazione modulo, cioè a (MOD b) rappresenta il resto della divisione intera tra a e b.

Il metodo prende il nome dalla seguente definizione:

due numeri x e y si dicono congrui modulo m, e scriveremo x =y (mod m), se essi differiscono per un multiplo intero di m.

Nel nostro caso x i+1 sarà congruo modulo m ad (a ∗ x i + c).

Il metodo è detto moltiplicativo se c=0, misto se c ≠0; se a=1 il metodo è detto additivo.

Facciamo un esempio banale ipotizzando le seguenti assegnazioni:

a=3

c=5

m=11

Se x_0=3 la sequenza che si ottiene applicando la formula della congruenza modulo m è la
seguente:
3, 3, 3, 3, …, cioè una sequenza assolutamente non casuale.
Le cose cambiano se scegliamo x_0=1; la sequenza ottenuta è allora la seguente:
1, 8, 7, 4, 6, 1, 8, 7, 4, 6, 1, …
Possiamo notare che i primi 5 numeri vengono riprodotti interamente!
Se x_0=2 si ottiene
2, 0, 5, 9, 10, 2, 0, 5, 9, 10, 2, …;
ancora una sequenza di 5 numeri ripetuta.
Se modifichiamo invece a assegnandogli il valore 12, e poniamo x_0=1 allora si ottiene:

1, 6, 0, 5, 10, 4, 9, 3, 8, 2, 7, 1, 6, 0, 5, …
una sequenza di lunghezza 11 e cioè pari a m.
E' da sottolineare che in ogni caso i numeri ottenuti sono compresi tra 0 e 10 e cioè tra 0 e
m-1.
(Se si desiderano numeri compresi tra 0 e 1 sarà sufficiente dividere il numero x i per m ad
ogni passaggio di riapplicazione della formula)
Da tutto ciò si possono ricavare le seguenti osservazioni:
La lunghezza massima raggiungibile dalla sequenza generata, senza ripetizione, vale m.

Particolari scelte di a e c possono ridurre notevolmente la lunghezza utile della sequenza.

Il valore di x_0 può essere determinante nella lunghezza della sequenza.

E' allora necessario individuare dei criteri per assegnare ad a, c, m e al seme dei valori in

modo che la sequenza riprodotta sia la più lunga possibile.

Alcuni studiosi hanno approfondito tale aspetto e hanno individuato i seguenti criteri

necessari e sufficienti che garantiscono l'ottimalità del metodo:

I parametri c e m devono essere coprimi cioè MCD(c,m) = 1

ogni divisore primo di m deve dividere (a-1)

se m è multiplo di 4, anche (a-1) lo deve essere.

In ogni caso, per rendere più aleatorio il processo, il seme viene fissato in modo hardware, prelevandone il valore da un contatore interno al computer usato normalmente per altri scopi, oppure ne viene richiesto il valore all'inizio del processo di generazione.

6.3 Possibili problemi con i PRNG

Vediamo un esempio delle conseguenze generate da un PRNG progettato male in applicazioni di sicurezza.

Nel Maggio 2008 una falla di sicurezza è stata scoperta nel generatore di numeri pseudo-casuali PRNG di OpenSSL.

OpenSSL è uno dei più usati software di crittografia che permette la creazione di connessioni di rete sicure ed è incluso in molti programmi per computer famosi come il browser web Mozilla Firefox. Il problema ha interessato tutte le chiavi crittografiche create dal settembre 2006.

Questa vulnerabilità fu causata dalla rimozione di due linee di codice dalla versione originale della libreria di OpenSSL.

Queste linee servivano alla libreria per raccogliere alcuni dati di entropia necessari al fine di inizializzare il PRNG utilizzato per creare le chiavi private sulle quali sono basate le connessioni sicure.

Senza questa entropia,l'unico dato dinamico utilizzato era il PID del software.

In Linux,il PID può essere un numero tra 1 e 32768 che è una gamma di valori troppo limitata se utilizzata per inizializzare il PRNG e causa la generazione di numeri prevedibili.

Perciò la segretezza delle connessioni di rete create con queste chiavi era completamente compromessa.

L'impatto di tutto ciò poteva essere molto grave,essendo OpenSSL comunemente utilizzato per proteggere le password ed offrire privacy e sicurezza.

Ogni chiave privata creata con questa versione di OpenSSL era debole e doveva essere rimpiazzata.

Questo voleva dire che ogni dato cifrato con queste chiavi poteva essere decifrato senza un grande sforzo.

6.4 Realizzazione pratica di un RNG

Vediamo come fare in pratica a realizzare un vero generatore di numeri casuali. Si parte dal presupposto che realizzare un'autentica sequenza non ripetitiva

abbiamo visto essere inattuabile utilizzando solo un computer poiché quest'ultimo è un'automa a stati finiti e prevedibile a un certo livello.

Il generatore Lava(così si chiama) usa una sorgente caotica,ovvero un sistema fisico dominato dal caos in cui una piccola sollecitazione a un certo istante di tempo ha un notevole impatto sullo stato futuro del sistema ,esso infatti sarà totalmente scorrelato dallo stato disturbato .

Come funziona:

Physical Sourceof Chaos *A/D* Digital Blender *Out* Random Numbers

1)Si digitalizza la sorgente caotica,catturando un istante dell'attività della sorgente .

2)Digital Blender:l'istantanea digitale contenente sia dati strutturati che caotici fa da input all' algoritmo Digital Blender.

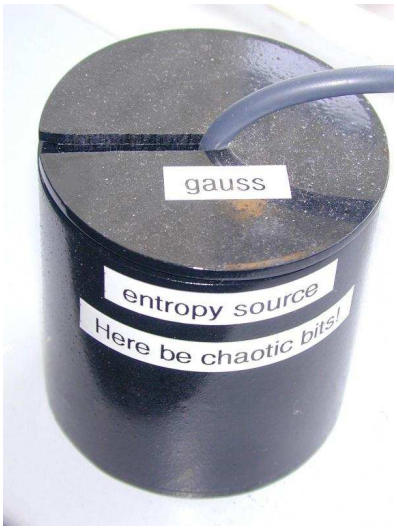
Qui la combinazione di n differenti operazioni "SHA 1 hash" in parallelo ed n differenti operazioni xor-rotate distruggono la porzione di dati strutturati dell'istantanea digitale e producono dati random uniformemente distribuiti.

3)L'insieme di dati random sono raccolti e adattati a seconda dell'applicazione che ne farà uso.

In dettaglio:

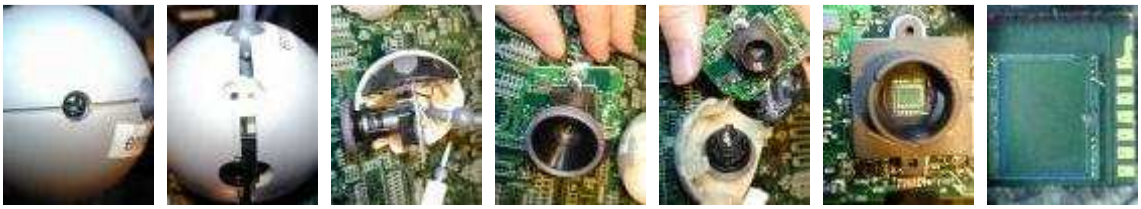
Step 1:Digitalizzazione della sorgente caotica

Com'è fatta la sorgente?....si incapsula un sensore CCD in modo che questo misuri solamente il rumore di fondo in uno spazio totalmente privo di luce.



Tale spazio si crea tramite una black box che fa in modo che la luce non giunga al sensore.

Si può utilizzare anche una webcam dato che la maggior parte di esse hanno u chip CCD al loro interno.



La funzione del sensore è di trasformare l'immagine catturata in dati digitali(pixels).

Ad esempio, si può avere una quantità di dati relativi ad un'immagine ,pari a 19200 pixels nel dominio YUV.

Si può massimizzare l'entropia della sorgente caotica collegando questa al calcolatore e settandone alcuni parametri,come il guadagno.

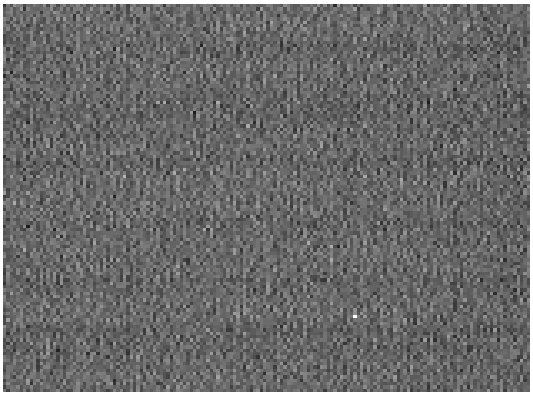
In tal modo riusciamo ad aumentare il rumore che essa produce.

Nell'ambiente privo di luce in cui inseriamo la camera le crominanze U e V non mostrano alti livelli di caoticità(non c'è colore al buio).

La luminanza è invece rumore che useremo per generare la sequenza random di numeri.

Si possono vedere i dati raccolti relativi alla luminanza attraverso due tipi di immagine:una in scala di grigi dove maggiore è il valore di luminanza,più bianco è il pixel,l'altra a colori dove i pixels con i colori più accesi corrispondono ai valori più alti di luminanza.

Webcam luminance data(2x magnification)



In sostanza il sistema che sto descrivendo utilizza solo i dati relativi alla luminanza per ogni CCD frame.

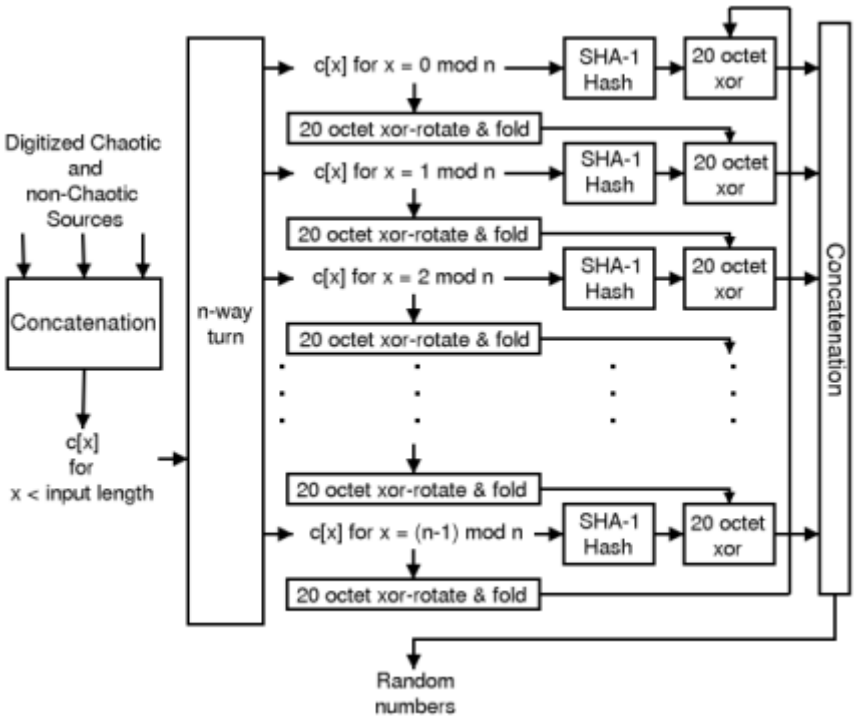
La scala di grigi è utile per vedere la distribuzione del pattern in generale,mentre quella a colori mostra bene le differenze tra pixels adiacenti.

Si osserva che i dati di luminanza hanno delle deboli strisce uniformi e intorno a queste un sacco di rumore;questo perché il CCD produce segnali sia caotici che non.

Step2: Digital Blender

Questo algoritmo ha lo scopo di eliminare la parte di dati non caotici.

Com'è costituito il DigitalBlender?



C'è un "n-way turn",n diverse SHA-1 operazioni hash in parallelo e n operazioni di xor-rotate.

L'n-way turn converte i dati digitalizzati ottenuti dal CCD in n sets di dati.

Ognuno di questi n entra nel blocco SHA-1 che produce in uscita 20 ottetti di dati.

Ognuno degli n entra anche nel blocco xor-rotate che produce anch'esso 20 ottetti di dati.

I 20 ottetti di dati prodotti da un blocco xor rotate sono messi in xor coi 20 ottetti prodotti dal blocco SHA-1 successivo così da produrre 20 ottetti di numeri random.

Ad esempio l'uscita dal secondo xor rotate è messa in xor con l'uscita dl terzo SHA-1(i blocchi in totale sono n).

Questo processo continua fino all'ultimo xor-rotate messo in xor con il primo SHA-1 per avere gli ultimi 20 ottetti di numeri random.

In totale questo algoritmo produce n*20 ottetti di numeri random.

Come scelgo n?

Dipende dalla quantità di dati ottenuti dalla sorgente caotica e dall'output rate factor chiamato alfa.

Più grande è alfa ,maggiore sarà l'output per un dato input.

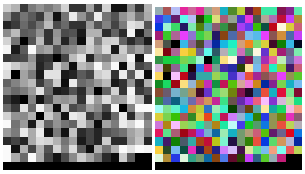
Assumendo 19200 ottetti di input, la seguente tabella mostra il numero di ottetti generati in funzione di alfa(ricordando che n è funzione di alfa):

input length	**alpha rate**	**n-way level**	**octets generated**
19200	16.00	71	1420
19200	12.00	61	1220
19200	8.00	49	980
19200	6.00	43	860
19200	4.00	35	700
19200	3.00	31	620
19200	2.00	25	500
19200	1.05	23	460
19200	1.00	17	340
19200	0.05	13	260
19200	0.25	11	220
19200	0,086806	7	140
19200	0,434028	5	100

Ipotizziamo di utilizzare un 17 way turn,una volta che questo è completo,l'algoritmo fa 17 differenti operazioni SHA-1 hash e 17 operazioni di xor rotate.

Quando le rispettive uscite di questi blocchi sono messe in xor e concatenate vengono prodotti 340 ottetti di numeri random.

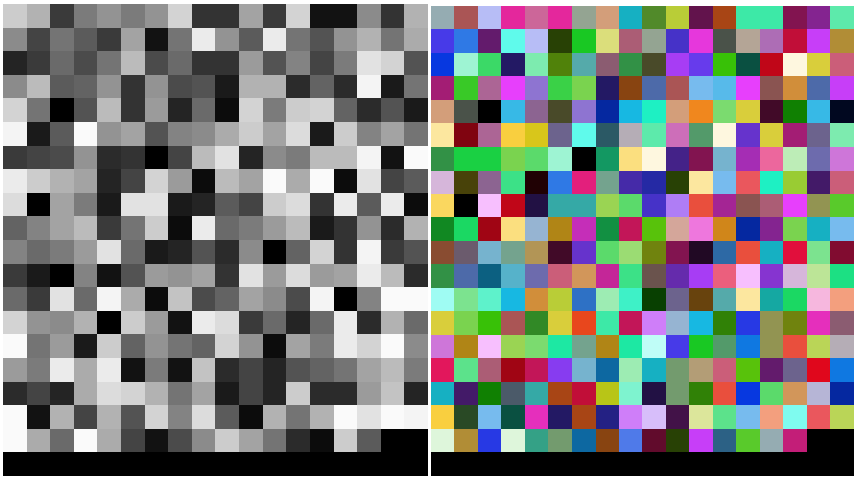
Random numbers in image form(2x magnification)



È facile notare come allo stesso livello d'ingrandimento l'immagine in uscita sia molto più piccola di quella in ingresso all'algoritmo;questo perché per un alfa uguale a 1 ad esempio,ci sono circa 56 ottetti di pixels di luminanza consumati per ogni ottetto di numeri random prodotto.

Vediamo tali immagini ingrandite :

(16x magnification)



Nella maggior parte delle applicazioni di solito non serve una grande quantità di numeri casuali tutti insieme ma serve metterli in una certa forma(a seconda dell'applicazione per la quale vengono utilizzati),ad esempio per simulare la

somma di due numeri estratti dal lancio di due dadi a 6 facce oppure un vero/falso o un blocco di dati random per testare un sistema in tutte le condizioni di input.
Nel modificare i dati(per adattarli all'applicazione finale) è importare cercare di mantenere la loro distribuzione uniforme.
Altri esempi di generatori inventati di recente:

Il QRBG usa un led ed un sensore che rileva l'emissione di fotoni ,esso sfrutta la luce prodotta da un semiconduttore.
Con l'ausilio di avanzati chip che registrano il fenomeno si può costruire un database contenente un'autentica sequenza random.
Un altro sistema di una società britannica genera randomness basandosi su tutto quello che và dall'analisi del vento solare alle nuvole di venere,dalle emissioni di giove ad altri eventi cosmici.

6.5 Caso pratico: il generatore in Excel

Excel: Due funzioni:CASUALE e CASUALE.TRA(;)

- CASUALE:
  genera numero con virgola tra 0 e 1.
- CASUALE.TRA(;) :
  genera numero in un range che posso specificare.

Esempio:
voglio generare un numero casuale tra 20 e 45;in una cella Excel scrivo =CASUALE.TRA(20;45)

Java: Utilizzo la classe java.util.Random ma va bene solo per casi semplici,cioè nessun requisito di sicurezza.

Per la crittografia si usa la classe java.security.SecureRandom; fornisce un RNG crittograficamente molto valido.

```
SecureRandom random = new SecureRandom();
 byte bytes[] = new byte[20];
 random.nextBytes(bytes);
```