Liczby losowe

Marek Biskup

14 czerwca 2002

Spis treści

1 Wstęp

Niniejszy dokument powstał w ramach ćwiczeń do przedmiotu *Algorytmicze aspekty kryptografii* prowadzonego na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego w roku akademickim 2001/2002 i stanowi podsumowanie referatu wygłoszonego przez autora na tych zajęciach.

2 Generatory pseudolosowe

Liczb losowych nie można wygenerować za pomocą deterministycznego algorytmu. Weźmy deterministyczną maszynę generującą ciąg liczb losowych. Deterministyczną, tzn. generowana liczba, jak i kolejny stan maszyny jest funkcją

aktualnego stanu. Oczywiście taka maszyna ma skończoną pamięć (bo jak dotychczas nikt nie widział maszyny z nieskończoną pamięcią), więc musi przyjmować skończoną liczbę stanów. W czasie generowania kolejnych liczb któryś ze stanów musi się powtórzyć, a wtedy kolejne generowane liczby także. Taki ciąg byłby okresowy, czyli nie losowy.

Jak widać programowe generowanie liczb losowych jest trudne (bo coś, co jest „deterministyczne" nie może być „losowe"). Ponieważ programowe generatory nie generują liczb losowych, to nazywa się je generatorami pseudolosowymi. Dokładniejszą definicję podaję za [1]:

Generator pseudolosowy to deterministyczny algorytm, który mając daną sekwencję binarną długości k, daje w wyniku sekwencję binarną długości $l \gg k$, która „wydaje się" być losowa. Ciąg wejściowy jest zwany zarodkiem (ang. seed).

Aby uspokoić czytelnika, że sytuacja nie jest aż tak beznadziejna dodam, że do generowania liczb losowych można użyć przyrody, która jest często losowa, a jeszcze częściej nieprzewidywalna (nieprzewidywalność wystarcza, bo z naszego punktu widzenia ciąg, którego wartości nie możemy przewidzieć jest losowy). W dodatku generatory pseudolosowe nie są złe, bo:

- okres generowanego ciągu można dobrać na tyle duży, że praktycznie ciąg jest nieokresowy,
- istnieją algorytmy generujące wartości, których nie da się przewidzieć w odpowiednio szybkim czasie znając poprzednie wygenerowane wartości, a nie znając stanu generatora. Taki ciąg jest praktycznie losowy

3 Generatory liniowe

Generatory liniowe są najbardziej popularnymi generatorami. Ich popularność wynika w szybkości obliczeń i dobrej znajomości ich własności. W ogólności, kolejny wyraz generowanego ciągu wyraża się wzorem:

$$X_{n+1} = (a_1 X_n + a_2 X_{n-1} + \cdots + a_k X_{n-k+1} + b) \bmod m$$

gdzie $a_0, a_1, \ldots, a_k, b$ i m są ustalonymi liczbami całkowitymi. Aby zainicjować generator podaje się wartości $X_0, \ldots, X_k$. Najczęściej stosuje się $k = 1$:

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

Takie generatory nazywają się często generatorami liniowymi, kongruentnymi i oznacza się je $LCG(m, a, b, x_0)$. Nie wszystkie parametry dla LCG są dobre, a sposób ich wyznaczania to dość obszerna teoria (odsyłam do [5] i [4]). Ideałem jest, by okres ciągu był możliwie duży; w przypadku m będącego liczbą pierwszą maksymalny okres to $m - 1$.

Jak wcześniej wspomniałem, LCG są bardzo popularne i szeroko stosowane (przede wszystkim ze względu na szybkość działania i prostą implementację). Np. w ANSI-C jako funkcja `rand()` stosowany jest LCG(2^{31}, 1103515245, 12345, 12345). Jak widać z wartości b i x_0 autorzy nie mieli dobrego pomysłu na

dobranie współczynników, ale „pierwsze lepsze" okazały się dobre. LCG($2^{31}-1$, 630360016, 0, 0) jest wykorzystywany w fotranie w funkcji *RAN*. Inne LCG (różne) były stosowane w językach CUPL, BCPL, Turbo C++, TurboPascalu, na komputerach Apple, Cray (w bibliotece SPRNG), w pakiecie matematycznym Mapple. Ich dokładniejsze omówienie można zobaczyć w [5].

Warta podkreślenia jest bezużyteczność prostych generatorów liniowych w kryptografii. Weźmy dla przykładu generator z ANSI-C. Przypuśćmy, że mamy algorytm generujący klucz dla jakiegoś systemu kryptograficznego, dla którego źródłem losowości jest funkcja `rand()`, a dodatkowo algorytm generowania klucza jest znany (założenie o znajomości algorytmu jest jak najbardziej realne — tak będzie, gdy będzie dostępny kod źródłowy programu szyfrującego lub nawet plik wykonywalny — wtedy można go zdesalemblować). Aby wygenerować odpowiednio długi, losowy klucz (np. 1024 bity) algorytm korzysta wielokrotnie z funkcji `rand()`. Niestety wynik kolejnego wywołania `rand` jest jednoznacznie wyznaczony przez poprzednie. Możliwe ciągi losowe użyte w czasie działania algorytmu są jednoznacznie wyznaczone przez pierwszą wylosowaną liczbę. Widać więc, że różnych kluczy, które mogą być wygenerowane przez nasz algorytm jest co najwyżej[1] 2^{31}. Złamanie klucza może polegać na sprawdzeniu jednego z 2^{31} kluczy, a nie jednego z 2^{1024}.

W rzeczywistości jest jeszcze gorzej. Popularną metodą inicjowania generatora (funkcja `srand()`, czyli ustawianie wartości X_0) jest wykorzystanie zegara systemowego (zwykle podającego czas w sekundach). Przy takim rozwiązaniu, jeśli znamy przybliżony czas (np. dzień), kiedy klucz był generowany, możemy zmniejszyć ilość kluczy do parudziesięciu tysięcy. Wtedy szyfr może zostać złamany w ciągu paru sekund. Inicjowanie generatora jakąś skomplikowaną (ale deterministyczną) funkcją czasu (czyli dnia, godziny, sekundy, roku) nic nie pomaga w tym przypadku.

4 Generatory nieliniowe

Dużą wadą generatorów liniowych jest to, że punkty postaci $(X_i/m, X_{i+1}/m)$ układają się w regularną siatkę w kwadracie $[0,1)^2$ (ładne wykresy pokazane są w [5]).

Aby ominąć ten problem należy stosować wzory, które nie są liniowe. Generatorem opartym na takim wzorze jest odwrotny generator kongruencyjny:

$$X_{n+1} = (aX_n^{-1} + b) \bmod m$$

gdzie m jest liczbą pierwszą, a X^{-1} to odwrotność modulo m.

Innym wariantem generatora opartego na odwrotności jest:

$$X_{n+1} = (a(n+n_0)^{-1} + b) \bmod m$$

[1] Korzystając z dowolnego generatora pseudolosowego, którego stan mieści się w n bitach, można wygenerować co najwyżej 2^n kluczy.

Ten generator ma tę zaletę, że kolejna wartość nie zależy od poprzednich, co może bardzo pomóc w obliczeniach prowadzonych równolegle na wielu komputerach.

Oczywiście zastosowanie tych generatorów w kryptografii jest takie jak generatorów liniowych, czyli żadne.

5 Generatory losowe

Na szczęście, przy generowaniu liczb losowych nie musimy się zdawać tylko na algorytmy deterministyczne. Niedeterminizm można wprowadzić przez kontakt programu ze światem zewnętrznym. Do generowania liczb losowych wykorzystuje się losowe i nieprzewidywalne zjawiska fizyczne. Losowych (naprawdę losowych) zdarzeń dostarcza mechanika kwantowa. W [2] opisane jest urządzenie generujące losowe bity z prędkością 1Mbit/s wykorzystujące mechaniką kwantową.

Działanie urządzenia opiera się na fakcie, że foton przechodzi przez półprzepuszczalne lustro z prawdopodobieństwem 1/2 i z takim samym prawdopodobieństwem zostaje odbity. W dodatku to, czy przejdzie, czy nie, jest całkowicie losowe. Innym pomysłem jest przepuszczanie fotonu spolaryzowanego po kątem 45^o przez lustro polaryzacyjne (dla którego foton spolaryzowany pod kątem 0^o będzie przepuszczony, a pod kątem 90^o odbity).

Inne urządzenia wykorzystują np. rozpad radioaktywnej substancji (^{85}Kr lub ^{60}Co.

Zjawiska nieprzewidywalne to np. szum termiczny diody lub tranzystora (cokolwiek by to nie znaczyło). W praktyce można wykorzystać np. sygnał z kamery lub z mikrofonu, parametry systemu (obciążenie procesora, statystyki sieciowe), użytkownika (np. ruchy myszki, odstępy czasu między uderzeniami w klawiaturę).

Naturalne źródła losowości często dają bity z nierównym prawdopodobieństwem (w przykładzie z fotonem, jeśli polaryzator nie był ustawiony dokładnie pod kątem 45^o, ale np. 40^o, to prawdopodobieństwo przepuszczenia fotonu przez lustro będzie większe niż odbicia). Najprostsza metoda na pozbycie się tej wady jest następująca: ustawiamy wygenerowane bity parami, jeśli mamy parę różnych bitów, to bierzemy pierwszy z nich, a jeśli bity są równe, to oba odrzucamy. Widać, że w wyjściowym ciągu prawdopodobieństwa obu bitów są równe. W praktyce, aby nie tracić cennych losowych bitów, stosuje się funkcję haszującą do wylosowanych bitów.

Podane wyżej źródła losowości nie zawsze są dobre. Gutmann w [3] podaje, że szum ze zwykłego mikrofonu, podłączonego do zwykłej, 8-bitowej karty muzycznej daje tylko jeden zmieniający się bit, w dodatku zmienia się on dość przewidywalny sposób. Ruchy myszki i kody naciśniętych klawiszy nie zawsze są wykonywane lokalnie. W czasie transportu przez sieć mogą być podsłuchiwane. W dodatku docierają zwykle w partiach i odstępy między naciśniętymi klawiszami nie muszą być losowe. Ruchy myszką przy przesyłaniu przez sieć bywają kompresowane (bo zwykle ważne jest gdzie myszka dotarła, a nie którędy) aby

zmniejszyć obciążenie sieci, co również zmniejsza losowość. Sygnały z klawiatury są obsługiwane przez system operacyjny, m.in. są buforowane. Wpływ systemu operacyjnego powoduje zmniejszenie losowości odstępów między naciśnięciami klawiszy.

6 Testowanie generatorów

Nie każdy generator liczb losowych jest dobry. Niektóre mogą generować mniej losowe ciągi niż inne (np. generator, który dwa razy częściej daje zero niż jedynkę nie może być zaakceptowany). Dlatego bardzo ważne jest odpowiednie testowanie generatorów. Poniżej przedstawiam najbardziej podstawowe testy:

Test częstości
Rozważmy ciąg losowych bitów długości n i zmienną losową:

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

gdzie n_0 jest liczbą zer w tym ciągu, a n_1 liczbą jedynek. Można pokazać, że X_1 w przybliżeniu spełnia rozkład χ^2 z jednym stopniem swobody, o ile $n \geq 10$ (w praktyce stosuje się o wiele większe wartości n, np. $n \gg 10^4$). Znając rozkład potrafimy obliczyć prawdopodobieństwo określonego odchylenia zmiennej od jej wartości oczekiwanej. Jeśli dla wylosowanego ciągu dostaniemy bardzo małą wartość tego prawdopodobieństwa, to znaczy, że generator nie przeszedł tego testu. Próg prawdopodobieństwa dobieramy odpowiednio do zastosowań.

Test dwubitowy
Test jest rozwinięciem poprzedniego testu. Tym razem sprawdzamy częstość występowania par bitów. Niech n_0 to liczba zer, n_1 jedynek, n_{00}, n_{01}, n_{10}, n_{11} to liczba ciągów 00, 01, 10, 11, odpowiednio. Zmienna losowa

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

spełnia w przybliżeniu rozkład χ^2 z dwoma stopniami swobody, dla $n \geq 21$.

Te dwa testy przedstawiają ideę konstruowania wszelkich testów losowości. Powyższe testy można uogólnić na sekwencje dowolnej długości. Taki test nazywa się **testem pokerowym**. Innym testem jest **test ciągów**, który sprawdza, czy w wylosowanej sekwencji ilość spójnych ciągów zer i spójnych ciągów jedynek różnych długości jest taka jak w sekwencji losowej. **Test autokorelacji** wykrywa związek między wygenerowaną sekwencją, a tą samą sekwencją przesuniętą o parę pozycji. **Test** π polega na obliczeniu liczby π metodą *Monte Carlo* i sprawdzeniu, czy wynik jest odpowiednio bliski prawdziwej wartości. Ideą stojącą za **uniwersalnym testem Mauera** jest fakt, że losowy ciąg nie powinien dać się znacząco skompresować (bez utraty informacji). Nieco dokładniejsze opisy powyższych testów można znaleźć w [1].

Istnieje standand FIPS 140-1 dotyczący testowania generatorów. Standand określa cztery testy losowości i granice w jakich mogę mieścić się wyniki. Jednym z testów jest test częstości — generator przechodzi ten test jeśli ilość jedynek w wygenerowanej 20000 bitowej sekwencji zawiera się między 9654 a 10346. Pozostałe testy to test pokerowy, test ciągów oraz test długich ciągów, który generator przechodzi gdy w 20000 bitowej próbce losowej nie ma spójnej sekwencji jednakowych bitów długości większej niż 34.

7 Generatory w kryptografii

Liczby losowe są potrzebne w prawie każdym systemie kryptograficznym. Podstawowym zastosowaniem jest generowanie klucza. Często okazuje się, że generatory losowe używane go generacji kluczy są o wiele łatwiejsze do złamania niż same szyfry.

Gutmann [3] podaje przykłady systemów, w których nie zadbano o losowość kluczy:

- 1995, Netscape — okazało się, że szyfrowanie przeglądarki może zostać łatwo złamane. Jakkolwiek do szyfrowania używano kluczy 128 bitowych, to naprawdę losowych bitów było co najwyżej 47.
- Kerberos V4 — używał funkcji `rand()`.
- MIT-MAGIC-COOKIE — używał generatora kluczy (deterministycznego), który był inicjowany jedną z 256 wartości.

Nie każdy generator pseudolosowy nadaje się do zastosowań w kryptografii. Poniższe definicje podaję za [1]:

Generator pseudolosowy przechodzi wszystkie *wielomianowe testy statystyczne*, jeśli nie istnieje wielomianowy algorytm, który potrafi poprawnie odróżnić wygenerowanej sekwencji od prawdziwej, losowej sekwencji tej samej długości, z prawdopodobieństwem znacząco większym niż 0,5.

Ta własność wydaje się być bardzo silna, a jednocześnie bardzo pożądana w generatorach stosowanych w kryptografii.

Generator pseudolosowy przechodzi *test następnego bitu*, jeśli nie istnieje wielomianowy algorytm, który mając dane na wejściu pewną ilość bitów sekwencji z generatora potrafiłby przewidzieć kolejny bit tej sekwencji z prawdopodobieństwem znacząco większym niż 0,5.

Własność wydaje się nieco słabsza, ale również pożądana. Okazuje się, że oba warunki są równoważne (spełnienie jednego z tych warunków przez generator implikuje spełnienie drugiego). O generatorze, który spełnia te warunki mówimy, że jest **bezpieczny kryptograficznie**

Do stworzenia generatora pseudolosowego możemy wykorzystać dowolną funkcję jednokierunkową f. Najpierw ustalamy losowy zarodek s. Wygenerowaną losową sekwencją będzie $f(s)$, $f(s+1)$, ... W zależności od własności użytej funkcji, może okazać się konieczne wykorzystanie jedynie kilku bitów z każdej

wartości. Jakkolwiek taka metoda wydaje się dawać generator bezpieczny kryptograficznie, to formalnie nie zostało to dowiedzione.

Generator **ANSI X9.17** jest generatorem tego typu. Kolejna wartość jest generowana przez zaszyfrowanie poprzedniej algorytmem DES z tajnym kluczem (w rzeczywistości szyfrowana jest nie poprzednia wartość, ale jej XOR z inną liczbą, a szyfrowanie jest przeprowadzane dwa razy dla każdej wartości; szczegóły w [1]). Ten algorytm jest standardem, a jest używany do generowania kluczy do DES-a. Generatory **FIPS 186** używają SHA-1 i DES jako funkcji jednokierunkowych. Służą do generowania tajnych parametrów dla DSA (klucze prywatne i jednorazowe, do podpisywania wiadomości).

8 Generatory bezpieczne kryptograficznie

Bezpieczeństwo tych generatorów zwykle opiera się na nieistnieniu algorytmów wielomianowych (w które to nieistnienie wszyscy głęboko wierzymy) rozwiązujących pewne problemy z teorii liczb (rozkład liczb na czynniki pierwsze, logarytm dyskretny). Niestety te algorytmy są o wiele wolniejsze od prezentowanych wcześniej, niemniej zdarzają się sytuacje, w których mogą być użyteczne.

Genarator RSA polega na wygenerowaniu sekwencji powstałej przez iteracyjne szyfrowanie RSA (z pewnym tajnym kluczem) początkowej losowej wartości i wzięcie najmniej znaczącego bitu z każdej wygenerowanej w ten sposób liczby.

Generator Blum-Blum-Shub jest bardzo podobny. Bierzemy dwie duże liczby pierwsze (tajne), które przystają do 3 modulo 4 i liczymy ich iloczyn n. Jako początkową wartość bierzemy losową liczbę s względnie pierwszą z n. Generujemy ciąg $x_0 = s^2 \bmod n$, $x_{n+1} = x_n^2 \bmod n$. Jako wygenerowany ciąg bierzemy najmniej znaczący bit z każdego x_i (kolejno). Zaletą tego generatora w porównaniu do RSA jest to, że występuje tu jedynie podnoszenie do kwadratu, co jest znacznie szybsze niż podnoszenie do dużej potęgi, jak w RSA.

Można pokazać, że w obu algorytmach można wziąć więcej niż jeden najniższych bitów i generator będzie ciągle bezpieczny kryptograficznie (i będzie o wiele szybszy).

9 Rzeczywiste generatory

Gutmann w [3] podaje postulaty, które powinien spełniać generator liczb losowych stosowany w praktyce do kryptografii.

- Musi być odporny na analizę danych wejściowych (losowych danych pobieranych z otoczenia). Haker, który odkryje część danych wejściowych generatora nie może na podstawie tych informacji poznać stanu generatora (mając dany stan generatora wiadomo jaki kolejne liczby będzie generował).

- Musi być odporny na manipulację danymi wejściowymi. Haker nie może, poprzez podanie generatorowi pewnych specyficznych danych, wpłynąć na stan generatora w jakikolwiek przewidywalny sposób
- Musi być odporny na analizę danych wyjściowych. Atakujący, nie może poznać stanu generatora na podstawie generowanych przez niego liczb.

Na tych postulatach oparty jest generator zawarty z jądrze linuksa (urządzenie `/dev/random`), a także w GPG (Gnu Privacy Guard — odpowiednik PGP).

10 Generator `/dev/random` w Linuksie

Generator zbiera dane losowe z otoczenia i gromadzi w „pojemniku losowości" (ang. *random pool*) o rozmiarze 512 bajtów. Danymi zbieranymi przez generator są np. kod naciśniętego klawisza lub odstęp czasu między kolejnymi przerwaniami. Generator szacuje w czasie dodawania danych, na ile dane te były losowe i na tej podstawie ustala ilość losowych danych w pojemniku (dalej zwana *entropią*). Kodu naciśniętego klawisza nie można uznać za losowy (więc jego dodanie do pojemnika nie zwiększa entropii). Także przerwania występujące w regularnych odstępach czasu nie dają losowych danych. W tym przypadku generator oblicza różnicę poprzedniego odstępu i obecnego i na tej podstawie określa ile losowych bitów zostało dodane (tak naprawdę, to tylko odstępy między przerwaniami mogą zwiększyć entropię).

Przy dodawaniu do pojemnika 32-bitowego słowa, jest ono obracane cyklicznie o określoną ilość bitów i XOR-owane ze słowami na pozycjach i, $i+7$, $i+9$, $i+31$, $i+99$ (liczby to współczynniki pewnego wielomianu CRC; stosuje się też inne współczynniki), gdzie i to pozycja w pojemniku ostatnio wstawionego słowa. Otrzymana wartość jest znowu obracana cyklicznie (tym razem o jeden bit) i wstawiana na pozycję $i-1$. (i jest liczone modulo rozmiar pojemnika).

Przy dodawaniu słowa do pojemnika nie można wykorzystać kryptograficznych funkcji haszujących ze względu na wymaganą wydajność (słowa są dodawane przy prawie każdym przerwaniu).

Przy pobieraniu bajtu z pojemnika (o ile entropia jest większa od zera), zwracana jest zawartość pojemnika przekształcona funkcją jednokierunkową (SHA lub MD5). Następnie cała zawartość pojemnika jest przekształcana jeszcze raz za pomocą SHA lub MD5. Oczywiście zmniejszana jest także entropia. Więcej o generatorze zawartym w jądrze linuksa jest w [7] oraz w źródłach linuksa.

Bibliografia

[1] A. Menzes, P. van Oorschot, S. Vanstone: *Handbook of Applied Cryptography*, CRC Press 1996

[2] T. Jennewein, U. Acheitner, G. Weihs, H. Weinfurter, A. Zeilinger: *A Fast and Compact Quantum Number Generator*, 2001

[3] P. Gutmann: *Software Generation of Practically Strong Random Numbers*

[4] R. Wieczorkowski, R. Zieliński: *Komputerowe generatory liczb losowych*, WNT 1997

[5] J. Srzednicki: *Przegląd generatorów liczb losowych*, 2001

[6] Ł. Kwiek, P. Sadowski: *Generatory liczb pseudolsowych a kryptografia*

[7] Piotr Hoffman: *Kryptografia a generator liczb losowych w systemie Linux*, http://rainbow.mimuw.edu.pl/SO/LabLinux/WEJSCIE-WYJSCIE/PODTEMAT_1/crypto.htm