Эссе по курсу “Защита информации”, кафедра радиотехники, Московский физико-технический институт (ГУ МФТИ), http://www.re.mipt.ru/infsec

Измерение количества случайности в системе

Ермолицкий Александр, 112 группа, 21 мая 2005 г.

В эссе рассмотрены основные тесты для проверки качества генераторов псевдослучайных последовательностей, основные моменты реализации этих тестов на языке C/C++. Приведены результаты измерения количества случайности настольной системы под управлением ОС Linux. Подробнее о теоретическом обосновании изложенных идей можно почерпнуть из [1]

Введение

Во многих криптографических системах необходим источник (генератор) случайных чисел (RNG - Random Number Generator). RNG используются, например, для генерации секретных криптографических ключей, генерации IV (Initial Vector) в схемах сцепления блоков при блочном шифровании передаваемых сообщений, Salt при хешировании паролей и т.д..

Стандартные генераторы случайных чисел, входящие в составе библиотек во многие языки программирования, генерируют статистически случайные последовательности чисел. Однако далеко не всегда они подходят для нужд криптографов, а это значит, что криптографическая система, основанная на таких RNG может оказаться нестойкой.

В идеале RNG должен быть основан на некотором истинно случайном физическом процессе. В качестве примеров можно привести радиоактивный распад и шумы в различных электронных приборах. На практике же приборы, основанные на подобных физических процессах, могут быть неприемлемыми из-за своей стоимости, габаритов, скорости выдачи результатов или подверженности внешним воздействиям. Поэтому во многих приложениях используются программные *псевдослучайные генераторы*, вырабатывающие *детерминированные* последовательности чисел (битов), которые близки по своим свойствам к истинно случайным генераторам.

Для разработчика криптографической системы важно убедиться в том, что используемый RNG обладает нужными свойствами (что это за свойства мы поговорим ниже). Для этого были разработаны специальные тесты для проверки RNG.

Статистические тесты на случайность

Тесты для RBG основаны на сравнении генераторов с идеалом. Что такое *идеальный* генератор случайных битов (RBG)? [1] Одно из самых простых и понятных определений гласит:

Идеальный генератор случайных битов вырабатывает такую последовательность битов, что по любой подпоследовательности выходных битов (любой длины) нельзя предсказать с вероятностью, отличающейся от 1/2, следующий выходной бит.

[1] Понятия генератор случайных чисел и битов являются, по сути, одним и тем же

Следствием этого определения является, например, то, что в последовательности битов, генерируемой идеальным RBG, число 1 и 0 совпадает с точностью до 1, а также то, что у нас нет повторяющихся подпоследовательностей. На подобных следствиях из определения идеального RBG основаны статистические тесты, представленные ниже.

Генератор псевдослучайных битов - детерминированный [2] алгоритм, который по начальной последовательности длиной k_{init} бит [3] вырабатывает последовательность длиной $l \gg k_{init}$ бит. Требования, предъявляемые к генератору псевдослучайных битов, основаны на том, что последовательности ограниченной длины, вырабатываемые *хорошим* псевдослучайным генератором, не должны по своим статистическим свойствам заметно отличаться от тех, которые генерирует идеальный генератор случайных битов. Другими словами, по последовательности битов ограниченной длины, выработанной хорошим псевдослучайным генератором, нельзя с вероятностью, заметно отличающейся от $1/2$, предсказать следующий выходной бит. При этом считается, что алгоритм работы генератора является известным, а зерно - нет.

Все существующие тесты для RBG являются вероятностными. Это значит, что результат теста звучит как "этот RBG подходит" или "этот RBG с вероятностью ошибки не более α не подходит". Это вполне естественно, т.к. любой тест может проверить лишь последовательность битов конечной длины, которая случайно может оказаться неприемлемой. Основным параметром всех тестов является α - вероятность того, что тест забракует хороший генератор. Во всех тестах я использовал значение $\alpha = 0.01$.

Поскольку RBG, основанный на физических случайных процессах, может быть подвержен внешним воздействиям или может просто сломаться, его необходимо периодически тестировать. Т.о. тест для проверки RBG должен обладать следующими свойствами:

- *Эффективность* - тест должен отбраковывать как можно больше разновидностей "плохих" генераторов.
- *Скорость* - тест должен работать достаточно быстро.

Далее представлены краткое описание и основные моменты реализации тестов для генераторов псевдослучайных последовательностей битов.

Монобитный тест

Это самый простой тест. Он основан на равенстве частот 1 и 0 в идеальном RBG. Пусть число битов в проверяемой последовательности равно L, число 1 - n_1, число 0 - n_0. В данном тесте вычисляется следующее значение: $X_1 = \frac{(n_0 - n_1)^2}{L}$. Если значение X_1 превысит некоторый порог (который зависит от α), то генератор не проходит тест. Поскольку X_1 имеет приблизительно χ^2 распределение[4] с одной степенью свободы, то в качестве порога было взято число 6.63 (т.к. $\alpha = 0.01$).

Единственной сложностью здесь является эффективный подсчет числа различных битов, т.к. в большинстве современных вычислительных архитектур нет команд для работы с

[2]Здесь *детерминированный* означает, что при одинаковом значении зерна генератор всегда будет выдавать одинаковые последовательности.

[3]Начальная последовательность называется зерном (seed).

[4]подробнее о χ^2 распределении можно прочитать в [1]

отдельными битами. Для подсчета количества единичных битов я использовал метод, описанный в [3]. Число [5] мысленно разбивается на *ячейки*, равной длины и биты в старшей и младшей половинах ячейки рассматривается как число битов в соответствующей части ячейки. Затем параллельно для всех ячеек число битов в обоих половинах ячейки суммируются и результат записывается в младшую часть. Вначале суммирование производится для ячеек размером 2 бита, потом 4 и т.д. до 32. В результате мы получаем полное число битов за логарифмическое (от длины машинного слова) время. Далее представлена несколько оптимизированная реализация этого алгоритма на языке C++:

Листинг 1 *Подсчет единичных битов в числе*

```
inline int countBits( unsigned int x )
{
    x = x - ((x >> 1) & 0x55555555);                // 2-битовые ячейки
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333); // 4-битовые
    x = (x + (x >> 4)) & 0x0F0F0F0F;                // 8-битовые
    x = x + (x >> 8);                               // 16-битовые
    x = x + (x >> 16);                              // 32-битовая ячейка
    x = x & 0x3F;   // Обнуляем старшие разряды, содержащие "мусор"
    return x;
}
```

После этого реализация самого теста достаточно проста. В дальнейшем я буду приводить не весь текст функции теста, а только основную его часть.

Листинг 2 *Реализация монобитного теста*

```
#define INT_BITS    (8 * sizeof(int))   // Число бит в случайном числе

bool monobitTest( int sequenceLength /* длина проверяемой последовательности */ )
{
    int     n1 = 0, n0 = 0; // Количество 1 и 0 в последовательности
    double  X1;             // Статистика

    for( int i = 0; i < sequenceLength; i++ )
        // Генерируем случайное число и считаем число единичных битов
        n1 += countBits( rand() );

    n0 = INT_BITS * sequenceLength - n1; // Число нулевых битов

    // Вычисляем статистическую функцию (длина последовательности считается в битах)
    X1 = (n1 - n0)*(n1 - n0) / (INT_BITS * sequenceLength);

    if( X1 > 6.63 )     // Порог при alpha = 0.01
        return false;   // Тест не пройден
    else
        return true;    // Тест пройден
}
```

[5] По-умолчанию, под словом *число* будет подразумеваться 32-битное машинное слово

Двубитный тест

В этом тесте считается не только число 1 и 0, но и число пар битов: 00, 01, 10 и 11 [6] Соответствующие числа обозначим $n_1, n_0, n_{00}, n_{01}, n_{10}, n_{11}$. Используемая функция:

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

имеющая χ^2 распределение с двумя степенями свободы. Поэтому порог для X_2 был выбран равным 9.21. Алгоритм подсчета битовых пар основан на следующем. Если мы побитово умножим число на это же число, сдвинутое на один бит, то количество единичных бит в результате даст нам количество пар 11 в исходном числе. Инвертируя один из множителей (или оба), мы получим количество пар 01 или 10 (или 00). Т.к. мы рассматриваем последовательность чисел как последовательность бит, то при сдвиге необходимо учитывать также бит предыдущего числа. Итак, алгоритм для подсчета битовых пар:

Листинг 3 *Подсчет встречаемости битовых пар*

```
for( int i = 0; i < sequenceLength; i++ )
{
    x = rand();             // Генерируем случайное число
    n1  += countBits( x );  // Число единичных битов
    n11 += countBits(  x &  ((x << 1) | prevBit) ); // Число пар 11
    n01 += countBits( ~x &  ((x << 1) | prevBit) ); // Число пар 01
    n10 += countBits(  x & ~((x << 1) | prevBit) ); // Число пар 10
    prevBit = x >> (INT_BITS - 1);  // Запоминаем последний бит
}

n0  = INT_BITS * sequenceLength - n1;   // Число нулевых битов
n00 = INT_BITS * sequenceLength - 1 - n11 - n01 - n10;  // Число пар 00
```

После этого текст самого теста тривиален.

Покерный тест

Здесь вся проверяемая последовательность разбивается на блоки длиной m бит. Этот тест, как и оба предыдущих, основан на том, что в идеальной случайной последовательности вероятность всех блоков одинакова. Пусть n_i есть число m-битных блоков, двоичное представление которых есть число i. В покерном тесте считается следующая статистическая функция:

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

где k $= \left\lfloor \frac{L}{m} \right\rfloor$ - суммарное число m-битных блоков в исследуемой последовательности. Из соображений эффективности разумно выбирать m равным степени двойки. Замечу, что при $m = 1$ покерный тест переходит в монобитный [7]. Стандарт FIPS 140-1 устанавливает $m = 4$ и $L = 20000$ бит. Поскольку X_3 имеет χ^2 распределение с $2^m - 1$ степенями свободы, для $m = 4$ мы должны выбрать порог, равный 30.6. Эффективный алгоритм подсчета встречаемости

[6] эти пары пересекаются, так что полное число пар равно $L - 1$

[7] Но при $m = 2$ не переходит в двубитный тест - т.к. m-битовые блоки не пересекаются

различных блоков основан на использовании таблицы (использование таблиц может быть целесообразно лишь при небольшом размере блоков, т.е. при $4 \leq m \leq 16$ в зависимости от приложения):

Листинг 4 *Подсчет встречаемости m-битных блоков*

```
#define BLOCK_LEN   4                   // Длина битовых блоков (m)
#define BLOCK_NUM   (1 << BLOCK_LEN)    // Число различных блоков (2^m)
#define BLOCK_MASK  (BLOCK_NUM - 1)     // Маска для выделения блока (2^m - 1)

int     blockCounter[BLOCK_NUM] = {0};  // Число различных битовых блоков
double  X3 = 0;                         // Вычисляемая статистика

// Считаем число различных блоков:
for( int i = 0; i < sequenceLength; i++ )
{
    x = rand(); // Генерируем случайное число

    for( int j = 0; j < INT_BITS / BLOCK_LEN; j++ ) // Для всех блоков числа
    {
        blockCounter[ x & BLOCK_MASK ]++; // Изменяем счетчик очередного блока
        x = x >> BLOCK_LEN;               // Переходим к следующему блоку
    }
}

// Вычисляем статистику:
for( int i = 0; i < BLOCK_NUM; i++ )
    X3 += blockCounter[i] * blockCounter[i];

X3 = X3 * BLOCK_NUM * BLOCK_LEN / sequenceLength - sequenceLength / BLOCK_LEN;
```

Серийный тест [8]

Тест определяет число вхождений серий одинаковых битов различной длины. В идеальной случайной последовательности среднее число серий длиной i равняется $l_i = \frac{(L-i+3)}{2^{i+2}}$. Пусть $n = \max_i (i : l_i \geq 5)$, число единичных и нулевых серий в проверяемой последовательности длиной i равняется B_i и G_i соответственно [9]. Вычисляемая в тесте статистика:

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

Имеет χ^2 распределение с $2n-2$ степенями свободы. Для $L = 20000$ имеем $n = 9$ и порог равный 32.0. Алгоритм вычисления длины битовых блоков основан на том, что число $2^n - 1$ содержит n единичных битов. Для произвольного числа x, содержащего серию нулевых битов длиной n в младших разрядах, $XOR(x, x-1)$ содержит серию из $n+1$ единичных битов в младших разрядах (все остальные биты нулевые). Длину выделенной таким

[8] Runs test

[9] От слов *Block* - серия единиц и *Gap* - серия нулей

образом *канонической серии* единичных битов можно считать бинарным поиском [3], либо с помощью заранее вычисленной таблицы (если длина серии мала). Здесь я использовал более эффективный комбинированный метод:

Листинг 5 *Подсчет длины серии нулевых битов в младших разрядах*

```
inline int runLength( unsigned int x )
{
    int length = 0; // Длина серии
    unsigned int y;

    // Приводим x к каноническому виду (b000...00111...11):
    x = (x ^ (x-1)) >> 1;

    // Считаем длину серии единичных битов бинарным поиском с точностью до 8:
    y = x >> 16;    if( y ) { length += 16;  x = y; }
    y = x >>  8;    if( y ) { length +=  8;  x = y; }

    // Уточняем длину с помощью заранее вычисленной таблицы:
    return length + smallRunLength[x];
}
```

Таблица `smallRunLength[]` содержит длины серий единичных битов для чисел от 0 до 255. Замечу, что мы могли бы использовать алгоритм подсчета единичных битов, однако, проведенное мной исследование показало, что он в данном случае менее эффективен, т.к. не учитывает то, что все единичные биты прижаты к правому краю.

Поскольку мы рассматриваем последовательность генерируемых чисел как последовательность бит, мы должны учитывать, что серия битов может быть разорвана (лежать на границе двух чисел). Программа подсчета различных серий, учитывающая последнее замечание, довольно длинная, поэтому здесь я не буду ее приводить. Вы можете найти ее в Интернете по адресу http://aer.nm.ru/test_rand.cpp.

Результат теста может вычисляться двумя способами: можно вычислять статистику X_4 и сравнивать ее с одним порогом, либо (стандарт FIPS 140-1) сравнивать с различными порогами каждое из чисел B_i и G_i.

Автокорреляционный тест

Данный тест основан на том, что в идеальной случайной последовательности нет (либо крайне мало) повторяющихся подпоследовательностей. В тесте считается число совпадающих битов в исходной и сдвинутой на N_{shift} бит последовательностях. Статистика:

$$X_5 = \frac{1}{\sqrt{L - N_{shift}}} \left(2 \left(\sum_{i=0}^{L-N_{shift}-1} XOR\left(b_i, b_{i+N_{shift}}\right) \right) - L + N_{shift} \right),$$

где b_i - i-ый бит последовательности. Поскольку X_5 имеет нормальное распределение с нулевым средним и дисперсией 1, то порог равен 2.33.

Листинг 6 *Вычисление функции автокорреляции*

```
int autocor( int sequenceLength, int shiftSize /* величина сдвига */ )
{
    unsigned int x, next, shifted; // Случайные числа
    int          autoCor = 0;      // Число совпадающих битов

    x = rand();

    for( int i = 1; i < sequenceLength; i++ )
    {
        next = rand(); // Генерируем случайное число

        // Вычисляем сдвинутую часть последовательности:
        shifted = (next << (INT_BITS - shiftSize)) | (x >> shiftSize);

        // Считаем число совпадающих битов:
        autoCor += countBits( x ^ shifted );
        x = next;
    }

    return autoCor;
}
```

Тест Маурера

Тест Маурера основан на предположении, что идеально случайную последовательность нельзя сколько-нибудь заметно сжать без потери информации. В этом тесте последовательность битов делится на $Q + K$ блоков размером m бит. Первые Q блоков используются для инициализации *таблицы* T, позволяющей по содержимому блока получить его последнюю позицию в последовательности. Следующие K блоков используются для вычисления статистики:

$$X_6 = \left(\frac{1}{K} \sum_{i=Q+1}^{Q+K} log\left(i - T[b_i]\right) \right) - \mu,$$

где b_i есть i-ый блок, а μ зависит от длины блока ($\mu = 7.1837$ для $m = 8$). Порог для данного теста вычисляется по формуле: $t = y\sigma$, где y зависит от α ($y = 2.58$ при $\alpha = 0.01$), а

$$\sigma \simeq \left(0.7 - \frac{0.8}{m} + \frac{\left(1.6 + \frac{12.8}{m}\right)}{K^{4/m}} \right) \sqrt{\frac{z}{K}},$$

где z зависит от m ($z = 3.238$ для $m = 8$). Подробнее об этом можно прочитать в [4]. Реализация теста довольно простая:

Листинг 7 *Реализация теста Маурера*

```
#define BLOCK_LEN   8                   // Длина битовых блоков (m)
#define BLOCK_NUM   (1 << BLOCK_LEN)    // Число различных блоков (2^m)
#define BLOCK_MASK  (BLOCK_NUM - 1)     // Маска для выделения блока (2^m - 1)

bool maurerTest( int initLength, // Длина инициализирующей последовательности
                 int workLength )// Длина проверяемой последовательности
{
    int     i, j, index = 0;
    int     indexTable[BLOCK_NUM] = {0};// Таблица последних позиций блоков
    unsigned int x;     // Случайное число
    double  X6 = 0;     // Статистика

    // Инициализация таблицы:
    for( i = 0, j = 0; i < initLength; i++ )
    {
        x = rand(); // Генерируем случайное число
        for( j = 0; j < INT_BITS / BLOCK_LEN; j++, index++ )// Для всех блоков числа
        {
            indexTable[ x & BLOCK_MASK ] = index; // Запоминаем последнее положение
            x = x >> BLOCK_LEN;                   // Переходим к следующему блоку
        }
    }

    // Вычисляем статистику:
    for( i = 0; i < workLength; i++ )
    {
        x = rand(); // Генерируем случайное число
        for( j = 0; j < INT_BITS / BLOCK_LEN; j++, index++ )// Для всех блоков числа
        {
            X6 += log( index - indexTable[ x & BLOCK_MASK ] ); // Статистика
            indexTable[ x & BLOCK_MASK ] = index; // Запоминаем последнее положение
            x = x >> BLOCK_LEN;                   // Переходим к следующему блоку
        }
    }
    // Число блоков в проверяемой последовательности:
    double K = workLength * INT_BITS / BLOCK_LEN;

    // Приводим статистику к распределению с нулевым средним (для длины блока 8 бит):
    X6 = fabs( X6 / (K * log(2.0)) - 7.1837 );

    // Вычисляем порог (для длины блока 8 бит и alpha = 0.01):
    double threshold = 2.58 * (0.6 + 3.2 / sqrt(K)) * sqrt(3.238 / K);
    return ( X6 > threshold ) ? false : true;
}
```

Выводы

Здесь я бы хотел привести результаты реального применения представленных выше тестов. Файл программы можно найти в Интернете: http://aer.nm.ru/test_rand.cpp.

Я тестировал псевдоустройство /dev/random, встроенное в ядро ОС Linux. /dev/random - это специальное псевдоустройство, собирающее различные события в драйверах мыши, клавиатуры, жесткого диска в *пул энтропии*. Для первых $L = 20000$ бит последовательности, полученной из /dev/random, были запущены все выше описанные тесты (в тесте Маурера $Q = 3200$, $K = 128000$ бит). Тестируемая последовательность записывалась в файл при помощи команды:

```
$ cat /dev/random > rand.txt
```

Таблица 1: Результаты тестов для последовательности из /dev/random

Тест	Статистика	Порог	Результат
Монобитный	1.7672	6.63	+
Двубитный	1.7675	9.21	+
Покерный	29.0112	30.6	+
Серийный	22.3277	32.0	+
Автокорреляционный	0.099	2.33	+
Тест Маурера	0.00067	0.023	+

Я протестировал также генератор псевдослучайных последовательностей из стандартной библиотеки glibc. Случайные числа были получены вызовом функции `rand()`, объявленной в стандартном заголовочном файле `stdlib.h`:

Таблица 2: Результаты тестов для функции rand() из glibc

Тест	Статистика	Порог	Результат
Монобитный	15.5682	6.63	-
Двубитный	15.5987	9.21	-
Покерный	83.136	30.6	-
Серийный	35.3664	32.0	-
Автокорреляционный	0.085	2.33	+
Тест Маурера	0.0444	0.023	-

Замечу, что rand() прошел монобитный тест с порогом, заданным в стандарте FIPS 140-1, однако, не прошел покерный и серийный тесты из этого стандарта. Как видим, использование “физических” случайных явлений позволяет получить очень хорошие случайные последовательности. RNG из библиотеки glibc, дает плохие псевдослучайные последовательности, поэтому серьезные криптографические системы должны использовать /dev/random или подобные генераторы.

Список литературы

[1] A.Menezes, P. van Oorschot, S.Vanstone “Handbook of Applied Cryptography” http://www.cacr.math.uwaterloo.ca/hac/

[2] Э.М.Габидулин, Лекции по криптологии

[3] Henry S. Warren, Jr. “Hacker’s Delight”

[4] Journal of Cryptology, vol. 5, no. 2, pp. 89-105, 1992 ftp://ftp.inf.ethz.ch/pub/crypto/publications/Maurer92a.pdf

[5] http://www.rt.com/man/random.4.html