Генераторы случайных и псевдослучайных последовательностей. Статистические тесты. Криптографически безопасные генераторы псевдослучайных последовательностей

Лекция

Ревизия: 0.1

Последняя модификация: Vlad Kovtun

© NRJETIX 2000 - 2008

Дата последней модификации: 12.12.2010 16:23:00

История изменений

27.09.2009 – Версия 0.1.   Первичный документ. Ковтун В.Ю.

Содержание

История изменений 2

Содержание 3

Лекция 2. Случайные и псевдослучайные последовательности 4

- Вопросы 4
- Случайные числа. Генераторы случайных чисел 4
  - Генераторы псевдослучайных последовательностей 4
  - Использование стандартных функций языков высокого уровня 5
  - Конгруэнтный генератор псевдослучайных чисел 5
  - Линейные регистры с обратной связью (Linear Feedback Shift Registers) 7
  - Модифицированные LFSR 8
- Криптографически стойкие датчики случайных чисел. Методы получения настоящих случайных чисел 9
  - Криптографически стойкие датчики случайных чисел 9
  - Системно-теоретический подход 9
  - Сложно-теоретический подход 10
  - Информационно-теоретический подход. 11
  - Рандомизированный подход 11
  - Генераторы настоящих случайных чисел 12
  - Отклонения и корреляции 12
  - Распределение случайности с помощью односторонней хэш-функции 13
- Статистические тесты 13

Литература 16

Лекция 2. Случайные и псевдослучайные последовательности

Вопросы

1. Генераторы случайных и псевдослучайных последовательностей.

2. Криптографически безопасные генераторы псевдослучайных последовательностей.

2. Статистические тесты.

Случайные числа. Генераторы случайных чисел

Известно, что при реализации криптографических преобразований, используют различные случайные первичные состояния либо целые последовательности. Отсюда следует, что стойкость криптопреобразования напрямую зависит от алгоритма формирования случайных чисел и последовательностей, точнее от их степени случайности.

Современные компиляторы обладают собственной реализацией генератора псевдослучайных последовательностей, однако с криптографической точки зрения они являются непригодными.

Основная сложность генерации последовательности псевдослучайных чисел на компьютере в том, что компьютеры детерминистичны по своей сути. Компьютер может находиться только в конечном количестве состояний (количество состояний огромно, но все-таки конечно). Следовательно любой датчик случайных чисел по определению периодичен. Все периодическое – предсказуемо, т.е. не случайно.

Лучшее, что может произвести компьютер – это псевдослучайная последовательность. Период такой последовательности должен быть таким, чтобы конечная последовательность разумной длины не была периодической. Относительно короткие непериодические подпоследовательности должны быть как можно более неотличимы от случайных последовательностей, в частности, соответствовать различным критериям случайности.

Генератор последовательности псевдослучаен, если он выглядит случайным, т.е. проходит все статистические тесты случайности (начиная с 2-го критерия – см. Кнут. т.2, стр. 52 и далее). Для криптографических приложений статистической случайности недостаточно, хотя это и необходимое свойство.

Последовательность называется **криптографически надежной псевдослучайной последовательностью (КНПСП)**, если она непредсказуема, т.е. вычислительно неосуществимо предсказать следующий бит, имея полное знание алгоритма (или аппаратуры) и всех предшествующих битов потока.

КНПСП тоже подвержены криптоанализу – как любой алгоритм шифрования. Сформулируем наиболее сильное определение.

Генератор последовательности называется случайным, если он не может быть достоверно воспроизведен, т.е. дважды запуская генератор с абсолютно одинаковыми исходными данными (по крайней мере, на пределе человеческих возможностей), мы получим случайные различные последовательности.

Вопрос существования случайных последовательностей – философский вопрос. Мы знаем, что на микроуровне случайность существует (квантовая механика), но сохранит ли эта случайность при переходе на макроуровень. Дополнительное свойство случайной последовательности: **случайная последовательность не может быть сжата**. КНПСП не может быть сжата (на практике).

Теперь рассмотрим категории генераторов подробнее.

Генераторы псевдослучайных последовательностей

В большинстве алгоритмов шифрования, особенно потоковых шифрах, используются генераторы ключевой последовательности. Генератор ключевой последовательности выдает поток битов, который выглядит случайными, но в действительности является детерминированным и может быть в точности воспроизведен на стороне получателя. Чем больше генерируемый поток похож на случайный, тем больше времени потребуется криптоаналитику для взлома шифра.

Однако, если каждый раз при включении генератор будет выдавать одну и ту же последовательность, то взлом криптосистемы будет тривиальной задачей.

Например, в случае использования потокового шифрования, перехватив два шифрованных текста, злоумышленник может сложить их по модулю 2 и получить два исходных текста, сложенных также по модулю 2. Такую систему раскрыть очень просто. Если же в руках противника окажется пара «исходный текст – шифрованный текст», то задача вообще становится тривиальной.

Поэтому все генераторы случайных последовательностей имеют зависимость от ключа. В этом случае простой криптоанализ будет невозможным.

Структуру генератора ключевой последовательности можно представить в виде конечного автомата с памятью, состоящего из трех блоков:

- блока памяти, хранящего информацию о состоянии генератора,
- выходной функции, генерирующей бит ключевой последовательности в зависимости от состояния,
- функции переходов, задающей новое состояние, в которое перейдет генератор на следующем шаге.

В настоящее время насчитывается несколько тысяч различных вариантов генераторов псевдослучайных чисел.

Рассмотрим основные методы получения псевдослучайных последовательностей, которые наиболее подходят для компьютерной криптографии.

Использование стандартных функций языков высокого уровня

Функция Rand() выдает псевдослучайное число в указанном диапазоне (способ задания диапазона отличается в различных языках). Начальная инициализация генератора случайных чисел происходит при помощи системного вызова специальной функции. Для языка C – это функция srand, в Object Pascal задание начального значения реализовано через свойство RandSeed. Часто значение, используемое в качестве начального, называют затравкой (начальным значением) генератора.

В реальных криптосистемах эта возможность не используется, так как обладает низкой криптостойкостью в силу своей доступности.

Конгруэнтный генератор псевдослучайных чисел

Линейный конгруэнтный генератор псевдослучайных чисел

Линейный конгруэнтный генератор (ЛКГ) – это последовательность чисел от 0 до $m-1$, удовлетворяющая следующему рекуррентному выражению $X_{k+1} = X_k a + b \bmod m$, X_0 - начальное значение, a – множитель, b – приращение, m – модуль. У такого генератора период меньше m. Если a, b, m правильно выбраны, то генератор является генератором максимальной длины и имеет период m (например $\gcd(m,b)=1$).

В таблице 15.1 из [8] приведен список хороших констант для ЛКГ. Они все дают ЛКГ максимальной длины и, что даже важнее, проходят Кнутовский спектральный тест случайности для размерностей 2, 3, 4, 5 и 6. [8]. Таблица 15.1 организована по максимальному значению ЛКГ, не переполняющему слово определенной длины. Преимущества ЛКГ в том, что они быстрые и требуют мало операций для производства одного бита.

Если инкремент b равен нулю, то есть генератор имеет вид:

$$X_{k+1} = X_k a \bmod m,$$

получим самую простую последовательность, которую можно предложить для генератора с равномерным распределением. При соответствующем выборе констант a может принимать значения 75 либо 16 807 и m принимать значения $2^{31}-1$ = 2 147 483 647 получим генератор с максимальным периодом повторения. Эти константы были предложены учеными Парком и Миллером, поэтому генератор вида: $X_{k+1} = 75 X_k \bmod (2^{31}-1)$, называется **генератором Парка-Миллера**.

Основным преимуществом линейных конгруэнтных генераторов является их быстрота, за счет малого количества операций на байт и простота реализации. К сожалению, такие генераторы в криптографии используются достаточно редко, поскольку являются предсказуемыми.

К сожалению, ЛКГ не может быть использованы для построения поточных шифров – они предсказуемы, впервые были взломаны Joan Boyar. Она также взломала квадратичные генераторы.

Нелинейные конгруэнтные генераторы

Иногда используют квадратичные и кубические конгруэнтные генераторы, которые обладают большей стойкостью к взлому.

Квадратичный конгруэнтный генератор имеет вид

$$X_{k+1} = \left(aX_k^2 + bX_k + c\right) \bmod m.$$

Кубический конгруэнтный генератор задается как

$$X_{k+1} = \left(aX_k^3 + bX_k^2 + cX_k + d\right) \bmod m.$$

Для увеличения размера периода повторения конгруэнтных генераторов часто используют их объединение (суперпозицию) посредством нелинейного преобразования (функции). При этом криптографическая безопасность не уменьшается, но такие генераторы обладают лучшими характеристиками в некоторых статистических тестах.

Пример такого объединения для 32-битовой архитектуры может быть реализован так:

```
// Long  - 32-bit integer
static long s1 = 1;
static long s2 = 1;
//MODMULT: s * b mod m, m = a * b + c, 0 <= c < m
#define MODMULT(a, b, c, m, s) \
q = s/a; s = b*(s-a*q)-c*q; \
if (s < 0) s+=m;

double combinedLCG (void)
{
	long q;
	long z;
	MODMULT (53668, 40014, 12211, 2147483563L, s1)
	MODMULT (52774, 40692, 3791, 2147483399L, s2)
	z = s1 - s2;
	if (z < 1) z += 2147483562;
	return z * 4.656613e-10;
}
void InitLCG (long InitS1, long InitS2)
{
	s1 = InitS1;
	s2 = InitS2;
}
```

Этот генератор работает при условии, что архитектура компьютера позволяет представлять все целые числа между −231e+85 и 231e−249. Переменные s1 и s2 глобальные и содержат текущее состояние генератора. Перед первым вызовом их необходимо проинициализировать при помощи функции InitLCG. Для переменной s1 начальное значение должно лежать в диапазоне между 1 и 2 147 483 562, для переменной s2 – между 1 и 2147483398. Период такого генератора близок к 1018. Функция combinedLCG возвращает действительное псевдослучайное число в диапазоне (0, 1). Она объединяет линейные конгруэнтные генераторы с периодами 215e−405, 215e-1041 и 215e−1111, и ее период равен произведению этих трех простых чисел.

Линейные регистры с обратной связью (Linear Feedback Shift Registers)

Линейный регистр с обратной связью (LFSR) состоит из двух частей: регистра сдвига и последовательностью ответвления (tap sequence) – см. рис. 1.

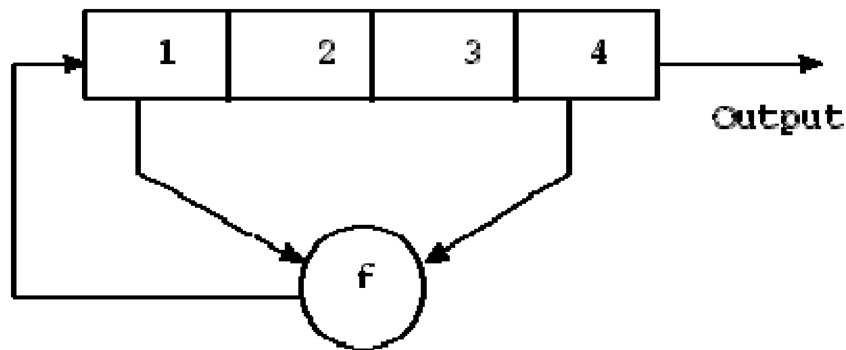

Рис. 1. Линейный регистр сдвига (LFSR)

В этой схеме регистр сдвига есть последовательность битов. Как только нам нужен следующий бит (иногда это называется **clock pulse** – тактовым импульсом – т.к. схема часто реализуется в аппаратном виде), все биты регистра сдвига сдвигаются направо и LFSR выдает наиболее значимый бит. При этом наименьший значимый бит определяется посредством вычисления XOR от прочих битов регистра, согласно последовательности ответвления.

Теоретически, n-битный LFSR может сгенерировать псевдослучайную последовательность длиной $2^n - 1$ бит перед зацикливанием. Для этого регистр сдвига должен побывать во всех $2^n - 1$ внутренних состояниях (кстати, количество состояний именно $2^n - 1$, а не 2^n, т.к. регистр сдвига, состоящий из нулей, вызовет бесконечную последовательность нулей, что не особо удобно).

Только некоторые tap sequences проходят через все $2^n - 1$ состояний, LFSR с такими tap sequences называются **LFSR максимальной длины**.

Пример: LFSR длиной 4 бита, tapped at 1st & 4th bits (см. Рис.1).

Теорема. Для того чтобы LFSR был LFSR максимальной длины, необходимо и достаточно, чтобы полином, образованный из элементов tap sequence плюс единица был примитивным полиномом по модулю 2 (на самом деле, примитивный полином – это генератор поля $\mathbf{GF}(2^n)$).

Пусть p - простое и пусть r_1, r_2, …, r_t являются различными простыми множителями для $p^m - 1$, тогда неприводимый полином $f(x) \in \mathbf{Z}_p[x]$ степени m является **примитивным полиномом**, если и только если, для каждого i, $1 \le i \le t$ выполняется условие:

$$x^{(p^m-1)/r_i} \ne 1(\mathrm{mod}\, f(x)).$$

Полином неприводим, если он не может быть выражен как произведение двух других полиномов, за исключением произведения 1 на самого себя.

Можем представить полином $x^{32} + x^7 + x^5 + x^3 + x^2 + x + 1$ в виде (32, 7, 5, 3, 2, 1, 0).

Этот полином легко превратить в LFSR максимальной длины (последнее число всегда 0). Числа, кроме последнего нуля, обозначают tap sequence. В этом примере, числа означают, что взяв 32-битный регистр сдвига и генерируя новый бит путем XOR'а 32-го, 7-го, 5-го, 3-го, 2-го, 1-го бит, мы получим LFSR максимальной длины: Приведем программу на языке C для этого LFSR:

```
Int LFSR()

{

      Static unsigned long ShiftRegister = 1;

      ShiftRegister = (((( ShiftRegister>>31)

      ^( ShiftRegister>>6)
```

```
        ^( ShiftRegister>>4)
        ^( ShiftRegister>>2)
        ^( ShiftRegister>>1)
        ^( ShiftRegister)
        &0x00000001)
        <<31
        | ShiftRegister>>1);
        return ShiftRegister & 0x00000001;
}
```

Код несколько усложняется, когда регистр сдвига длиннее, чем размер слова, но незначительно. В таблице 15.2 приведено большое количество полиномов, т.к. LFSR очень часто используется для потоковой криптографии и хочется, чтобы люди использовали различные полиномы. Кстати, любая запись в таблице представляет собой 2 разных примитивных полинома, т.к. если $p(x)$ – примитивный, то и $x^n p\left(\frac{1}{x}\right)$ - примитивный. Например, если $(a, b, 0)$ – примитивный, то и $(a, a-b, 0)$ – примитивный. Если $(a, b, c, d, 0)$ - примитивный, то и $(a, a-d, a-c, a-b, 0)$ – примитивный и т.д.

Примитивные трехчлены особенно удобны, т.к. только 2 бита регистра сдвига должны быть преобразованы посредством операции XOR (отXORены), но при этом они и более уязвимы к атакам. LFSR – неплохие генераторы случайных чисел, но имеют неприятные свойства. Последовательности бит - линейны, что делает их бесполезными для шифрации. Для LSFR длины n внутреннее состояние можно узнать по n выходным битам генератора. Даже если схема обратной связи неизвестна, то достаточно $2n$ выходных бита, чтобы определить ее. Большие случайные числа, сгенерированы из последовательных битов LFSR, сильно коррелированны и иногда даже не совсем случайны. Тем не менее, LFSR достаточно часто используется как базовые алгоритмы шифрования.

Модифицированные LFSR

Можно переписать программу, реализующую LFSR для того, чтобы не возиться с битовыми операциями, иногда такую модификацию **называют конфигурацией Галуа**.

```
#define mask 0x80000057
Static unsigned long ShiftRegister = 1;
Void seed_LFSR(unsigned long seed)
{
        If (seed == 0) seed = 1;
        ShiftRegister = seed;
}
int modified_LFSR(void)
{
        if(ShiftRegister & 0x00000001)
        {
        ShiftRegister = (ShiftRegister ^ mask >> 1) | 0x8000000;
        return 1;
        }
        else
        {
        ShiftRegister >>= 1;
        }
}
```

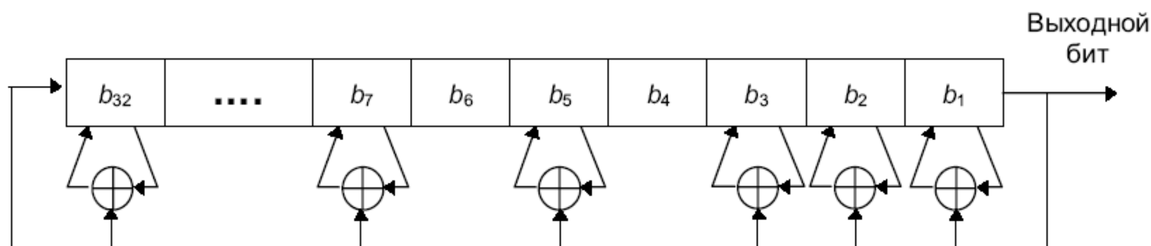

Рис. 2. LFSR Галуа

Криптографически стойкие датчики случайных чисел. Методы получения настоящих случайных чисел

Криптографически стойкие датчики случайных чисел

Основная идея криптографически стойких ДСЧ в том, что они идеально подходят для потоковых шифров. Выход таких ДСЧ неотличим (точнее, должен быть неотличим) от настоящих ДСЧ. С другой стороны, они детерминистичны, т.е. достаточно произвести XOR выхода ДСЧ с потоком исходного текста.

Известно 4 подхода к конструированию CSPRSG:

- системно-теоретический подход;
- сложностно-теоретический подход;
- информационно-теоретический подход;
- рандомизированный подход.

Эти подходы различаются в своих предположениях о возможностях криптоаналитика, определении криптографического успеха и понятия надежности. Большая часть исследований в этой области теоретически, хотя среди многих непрактичных ДСЧ существуют и удачные варианты.

Системно-теоретический подход

В этом подходе, криптограф создает генератор ключевого потока, у которого есть проверяемые свойства – период, распределение битов, линейная сложность и т.д. криптограф изучает также различные методы криптоанализа и оптимизирует ДСЧ против этих атак. Этот подход выработал набор критериев для потоковых шифров. Они были сформулированы Рюппелем [761]:

- Большой период, отсутствие повторений.
- Критерий линейной сложности: повышенная линейная сложность, локальная линейная сложность (**Линейная сложность ДСЧ** – это длина кратчайшего LFSR, которая может сгенерировать выход генератора; линейная сложность есть мера случайности ДСЧ);
- Статистические критерии, такие как идеальное распределение:
    - Перемешивание: любой бит ключевого потока должен быть сложным преобразованием всех или большинства битов ключа.
    - Рассеивание: избыточность в подструктурах должна рассеиваться;
    - Нелинейные критерии (расстояние до линейных функций, критерий лавинообразности и т.д.)

В общем, этот список критериев годится не только для потоковых шифров, созданных в рамках системно-теоретического подхода, но и для всех потоковых шифров. Более того, эти критерии годятся и для всех блочных шифров. Но при системно-теоретическом подходе потоковые шифры создаются таким образом, чтобы напрямую удовлетворять вышеописанным критериям.

Основная проблема подобных криптосистем в том, что для них трудно доказать какие-либо факты об их криптостойкости. Дело в том, что для всех этих критериев не была доказана их необходимость или достаточность. Потоковый шифр может удовлетворять всем этим принципам и все-таки оказаться нестойким.

С другой стороны, взлом каждой такой системы – отдельная задача. Если бы таких шифров было много, то криптоаналитикам могло бы и не захотеться их атаковать. В конце концов, потоковые шифры во многом похожи на блочные шифры – для них нет доказательств стойкости. Существует набор известных способов атаки, но стойкость к ним ничего не гарантирует.

Рассмотрим некоторые примеры построения генераторов на основе этого подхода.

Каскад Голлманна

Каскад Голлмана состоит из серии LFSR'ов, причем такт каждого следующего LSFR контролируется предыдущим (т.е. если выход LFSR-1 равняется 1 во время $t-1$, то LFSR-1 меняет свое состояние на следующее). Выход последовательности LFSR есть выход всего генератора. Если все LFSR – длины l, то линейная сложность системы с n LFSR равна $l(2^l-1)^{n-1}$.

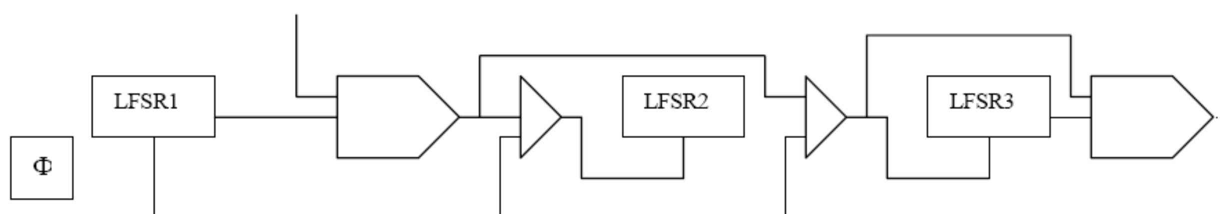

Рис. 3. Каскад Голлмана

Альтернирующий (перемежающийся) Stop-and-Go генератор

В этом генераторе используется 3 LFSR различной длины. LFSR-2 меняет состояние, если выход LFSR-1 равен 1; LFSR-3 меняет состояние в противном случае. Результат генератора есть XOR LFSR-2, LFSR-3 [404]. У этого генератора большой период и большая линейная сложность.

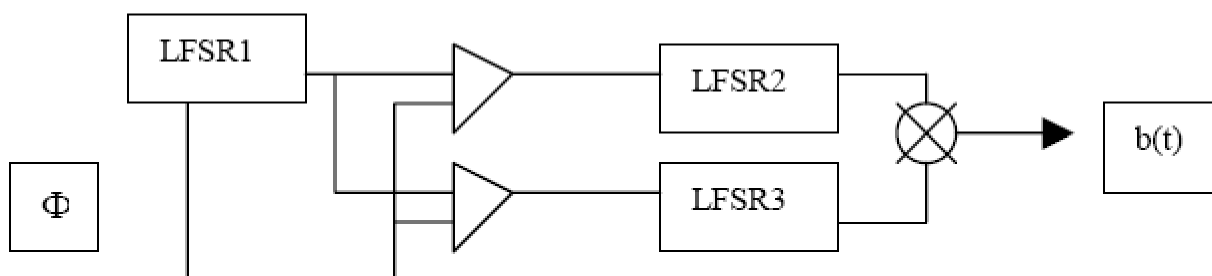

Рис. 4. Альтернирующий Stop-and-Go генератор

Сложно-теоретический подход

В этом подходе, криптограф пытается использовать теорию сложности для доказательства стойкости генератора, следовательно генераторы являются более сложными; они основаны на тех же проблемах, что применяются в криптографии с открытым ключом (по реализации они - медленные).

Рассмотрим примеры.

Генератор Шамира

Данный генератор основывается на схеме RSA. Шамиром блока показано, что предсказание выхода такого генератора равносильно взлому RSA, потенциальное смещение выхода было продемонстрировано в некоторых работах [1401, 200].

Генератор Blum-Micali

Стойкость алгоритма определяется трудностью решения задачи дискретного логарифма [200]. Пусть g - просто, и p - простое. Ключ используется в качестве начального значения x_0, последовательность формируется согласно рекуррентному выражению $x_{i+1} = g^{x_i} \bmod p$. Выходом генератора является 1, если $x_i < \frac{p-1}{2}$, и 0 в противном случае. Если p достаточно велико, чтобы вычисление дискретного логарифма $\bmod p$ стало физически невозможным, то этот генератор – безопасен. Дополнительные теоретические результаты можно найти в работах [1627, 986, 985, 1237]

Генератор RSA

Берем параметр $N = pq$, где p, q – простые и начальное значение $x_0 < N$.

$x_{i+1} = x^e \bmod N$.

Результат генератора – наименьший значимый бит x_i. Стойкость этого генератора соизмерима со стойкостью RSA. Если N достаточно большое, то генератор надежен.

Генератор Blum Blum Shub

На данный момент – самый простой и эффективный. Назван по фамилиям изобретателей или сокращенно BBS. Теория этого генератора – квадратичные вычеты по модулю n.

Найдем два больших простых числа p, q, дающие при делении на 4 остаток 3. Произведение $n = pq$ называется числом Блюма. Выберем x: $\gcd(n, x) = 1$. Вычислим начальное значение генератора: $x_0 = x^2 \bmod n$.

Заметим, что i-ым элементом псевдослучайной последовательности является наименьший значимый бит x_i, где $x_i = x_{i-2}^2 \bmod n$.

Интересно, что нет необходимости проходить через $(i-1)$ состояние для того, чтобы получить i-й бит. Если мы знаем p, q, то мы можем рассчитать его сразу: b_i - младший значащий бит x_i, где $x_i = x_0^{(2^i) \bmod ((p-1)(q-1))}$. Такой подход позволяет использовать котптографический сильный генератор псевдослучайных чисел в качестве потоковой криптосистемы для файлов с произвольным доступом. Безопасность этой схемы основана на сложности разложения числа n на множители. Можно опубликовать n, это позволит любому формировать биты такой последовательности, однако криптоаналитик не сможет разложить n на множители и предсказать выход генератора, например: «С вероятностью 0,51 на выходе будет 1».

Более того, BBS генератор непредсказуем налево и направо (т.е. имея часть последовательности нельзя предсказать предыдущий или последующий бит). Кроме того, можно использовать не только наименьший значащий бит, но и l битов x_i (где l – длина x_i). Предложенный генератор – медленный, однако может применяться для высоконадежных приложений.

Информационно-теоретический подход.

Предположим, у криптоаналитика есть бесконечные компьютерные ресурсы и время. Тогда единственный надежный метод – одноразовая лента (аналог одноразового блокнота).

Рандомизированный подход

В этом подходе задача в том, чтобы увеличить число битов, с которыми необходим о работать криптоаналитику (не увеличивая при этом ключ). Этого можно достичь путем использования больших случайных общедоступных строк. Ключ будет обозначать, какие части этих строк необходимо использовать для шифровки/дешифровки. Тогда криптоаналитику придется использовать метод грубой силы на случайных строках.

Стойкость этого метода может быть выражена в терминах среднего числа битов, которые придется изучить криптоаналитику, прежде чем шансы определить ключ станут выше простого угадывания.

Ueli Maurer описал такую схему [574]. Вероятность взлома такого алгоритма зависит от объема памяти, доступного криптоаналитику (но не зависит от его вычислительных ресурсов). Чтобы эта схема стала практичной, требуется около 100 битовых последовательностей по 20 10 бит каждая. Оцифровка поверхности Луны – один из способов получения такого количества битов.

Генераторы настоящих случайных чисел

Иногда даже CSPRNG недостаточно. Например, генерация ключей. Если врагу удастся разузнать алгоритм CSPRNG и секретную информацию, используемую для генерации ключа, то ключ у него в руках. Если же использовать для этой цели настоящий генератор случайных чисел, то никто (даже мы) не сможет воспроизвести соответствующую битовую последовательность. Итак, наша цель – произвести числа, которые невозможно воспроизвести.

Таблица RAND

В 1955 Rand Corporation опубликовала книжку с миллионом случайных цифр.

Использование таймера компьютера

Если необходим один случайный бит (или даже несколько), то можно взять наименьший значимый бит таймера. Это может плохо работать под Unix'ом из-за различий синхронизацией, но на PC будет нормально. Нельзя получать таким образом много битов. Например, если каждый вызов процедуры генерации бита занимает четное число тиков таймера, то мы будем получать одни и те же значения, если нечетное – то последовательность 101010… Даже если зависимость не настолько очевидна, результирующая строка будет далека от случайности.

Использование случайного шума

Самый лучший способ получить случайное число – это обратиться к естественной случайности реального мира – радиоактивный распад, шумные диоды и т.п. В принципе, элемент случайности есть и в компьютерах:

- время дня;
- загруженность процессора;
- время прибытия сетевых пакетов и т.п.

Проблема не в том, чтобы найти источники случайности, но в том, чтобы сохранить случайность при измерениях. Например, это можно делать так: найдем событие, случающееся регулярно, но случайно (шум превышает некоторый порог). Измерим время между первым событием и вторым, затем между вторым и третьим. Если $t_{1,2} > t_{2,3}$, то выдадим 1; если $t_{1,2} \le t_{2,3}$, то выдадим 0. Затем повторим процесс.

Отклонения и корреляции

Существенной проблемой таких систем является наличие отклонений и корреляций в сгенерированной последовательности. Сами процессы могут быть случайными, но проблемы могут возникнуть в процессе измерений. Как с этим бороться?

1) XOR'еньем: если случайный бит смещен к 0 на величину e e, то вероятность появления 0 может быть записана как $\mathbf{P}(0) = 0{,}5 + e$. XOR'енье двух битов даст: $\mathbf{P}(0) = (0{,}5 + e)^2 + (0{,}5 - e)^2 = 0{,}5 + 2e^2$. XOR'енье четырех битов: $\mathbf{P}(0) = 0{,}5 + 8e^4$ и т.д. Процесс сходится к равновероятным 0 и 1.

2) Данный метод уничтожает все смещения со случайным в остальном источника. Рассмотрим пару битов. Если это одинаковые биты, то отбросим их и рассмотрим следующую пару. Если биты различны, то берем первый бит.

Потенциальная проблема обоих методов в том, что при наличии корреляции между соседними битами данные методы увеличивают смещен. Один из способов избежать

этого – использовать различные источники случайных чисел (XOR’ить биты 4-х различных источников или смотреть на пары битов из разных источников).

Сам по себе факт наличия смещения у генератора случайных чисел не означает его непригодность. Например, рассмотрим проблему генерации 112-битного ключа для строенного DES’а. Пусть у Алисы есть только генератор со смещением к нулю: $\mathbf{P}(0) = 0{,}55$; $\mathbf{P}(1) = 0{,}45$ (=> всего 0,99277 битов энтропии на бит ключа по сравнению с 1 для идеального генератора). Тогда Маллет при попытке взлома ключа может оптимизировать процедуру поиска ключа методом грубой силы, пытаясь начать с наиболее вероятного ключа (00…0) и заканчивая наименее вероятным (11…1). Из-за смещения, Маллет может ожидать найти ключ в среднем за 109^2 попыток. Если бы смещения не было, то потребовалось бы 111^2 попыток. Выигрыш есть, но несущественный.

Распределение случайности с помощью односторонней хэш-функции

Наконец, на последовательность случайных битов можно применять односторонняя хеш-функция. Это как бы дистиллирует энтропию из входных данных. Это также помогает избавиться от смещений и корреляций низшего порядка.

Например, если нам надо 128 случайных битов для ключа, то можно сгенерировать большое количество случайных битов и применить к ним хорошую одностороннюю хеш-функцию. Если только в исходной последовательности было 128 бит энтропии, т.е. будет идеально случаен. Это будет так, даже когда каждый бит по отдельности содержит значительно меньше 1 бита энтропии.

Статистические тесты

В криптографии под понятием "**случайного числа**" понимают число, которое нельзя предсказать до момента его генерации. Однако существует и такое требование – такое число нельзя угадать, зная последующие (так называемая "атака из будущего"). Тут надо немного пояснить. В криптографии оперируют не одиночными числами, а последовательностью, серией. Генератор случайных чисел непрерывно работает, выдавая постоянную последовательность битов, а они уже программно или аппаратно далее интерпретируются в зависимости от контекста. Допустим, у нас есть последовательность случайных чисел 2, 5, x, 7 (для простоты – это десятичные числа). Криптоаналитик хочет узнать случайное число x. Говоря о стойкости серии случайных чисел, подразумевают, что:

- нет аналитической зависимости между последовательно сгенерированными числами;
- зная предыдущие числа (в нашем случае – 2 и 5), криптоаналитик не может найти следующие число x (атака из прошлого);
- зная последующие числа (в нашем случае – 7), нельзя установить предшествующие (атака из будущего);
- вероятность появления любого числа в последовательности одинаковая.

Поскольку на практике используют генерации двоичной последовательности, то вероятность появления каждого бита - 1/2 в степени n, где n - разрядность чисел. Пример: имеется последовательность 32-разрядных чисел m, вероятность того, что число $m+1$ будет таким, как предсказал криптоаналитик, равняется $\frac{1}{2^{32}}$. Если перебирать в секунду 1000 чисел, то это равнозначно одному совпадению за 268 дней круглосуточной работы.

Но не всякую последовательность можно назвать случайной. Для исследования алгоритмов реализации генераторов есть несколько тестов, которые определяют, случайна или нет данная последовательность. Поскольку мы имеем дело с вероятностными процессами, то суждение о случайности или нет такой последовательности также будет верным (неверным) с некоторой вероятностью. На практике для каждого теста есть свое распределение вероятности (своя статистика) и берется какое-то значение, обычно на краях диапазона, например 0,01% (так называемое критическое значение). Далее делают тест и рассчитывают вероятность – если она превышает критическое значение, тестовая последовательность признается неслучайной. Пример: в результате теста вероятность того, что числа случайные, равна

0,40%, значит, числа неслучайные, вероятность превышает критическое значение. При вероятности <0,01% числа признаются случайными, а тест пройденным.

В таком подходе возможны две ошибки, называемые ошибками первого и второго рода.

Ошибка первого рода (она обозначается как "a") возникает, когда тестовая последовательность чисел является истинно случайной, но тест признает ее неслучайной. Это ложное срабатывание. Ошибка второго рода случается, когда тест признает случайными данные, которые на самом деле неслучайные.

Ошибки первого рода называются еще уровнем достоверности теста, который может быть вычислен заранее. Обычно принимают уровень достоверности 0,01 – из ста истинно случайных последовательностей одна признается неслучайной.

Ошибки второго рода записывают как "β", и они означают, что исследуемые числа обладают скрытой закономерностью, а значит, порождающий их генератор выдает "плохую" последовательность. Вычислить этот коэффициент гораздо труднее, а его влияние очень большое – если ошибки первого рода приводят всего лишь к отсеиванию некоторой части чисел, то ошибки второго рода могут повлиять на стойкость шифра.

И последнее. Для оценки тестов применяют отдельный коэффициент, так называемый $\mathbf{P}$ ($\mathbf{P}$-value). $\mathbf{P}$-value – это вероятность того, что некий абстрактный идеальный генератор случайных чисел сгенерировал бы последовательность менее случайную, чем исследуемая. Когда $\mathbf{P}=0$, это значит, что последовательность чисел неслучайна, а когда $\mathbf{P}=1$, то последовательность близка к совершенно случайной. На практике значение P должно быть больше, чем уровень достоверности теста. Например, при $\mathbf{P}>0{,}01$ проверяемая последовательность случайна в 99% случаев (если достоверность теста a=0,01, то одна из ста последовательностей будет неслучайной).

Для двоичных последовательностей мы делаем еще некоторые предположения:

- Монотонность. Вероятность появления 1 или 0 каждый раз одинаковая – 1/2.
- Масштабируемость. Если вся последовательность случайна (проходит тест на случайность), то должна быть случайной и любая случайно выбранная подстрока. Если серия из 1000 чисел случайна, то серия чисел от 500-го до 531-го числа также должна быть случайной и проходить тест.

Национальным институтом стандартов и технологий (NIST) разработаны 16 специальных тестов для определения случайных чисел. Имеется программная реализация в виде NIST Statistical Test Suite для платформы Unix. Она распространяется в виде исходного кода и содержит как инструменты командной строки, так и графические утилиты. Загрузить код можно отсюда: http://www.csrc.nist.gov/rng/sts-1.5.tar (объем 8,7 Мб). Также есть набор дополнительных данных и утилит, в частности генератор псевдослучайных чисел Blum-Blum-Shub. Загрузить можно отсюда: http://www.csrc.nist.gov/rng/sts.data.tar (объем 43,8 Мб). Кстати, обратите внимание на размер – образцы случайных последовательностей плохо или вообще не сжимаются (один из тестов так и называется – тест на сжимаемость). Последовательность случайных чисел вообще нельзя сжать, так как при попытке построения словаря на таких данных его размер совпадает с размером самих данных – ведь там нет повторяющихся чисел.

Рассмотрим тесты подробнее. Замечу, что непрохождение хоть одного автоматически отменяет все последующие тесты – числа неслучайны, и продолжать проверку нет смысла.

- **Частотный тест** (монобитный тест на частоту, Frequency (Monobits) Test). В этом тесте исследуется доля 0 и 1 в последовательности и насколько она близка к идеальному варианту – равновероятной последовательности. Для теста надо иметь не менее 100 бит данных.
- **Блочный тест на частоту** (Test for Frequency within a Block). Последовательность разбивается на блоки длиной M бит, и для каждого рассчитывается частота появления единиц и насколько она близка к эталонному значению – M/2. Когда M=1, длина блока 1 бит и тест равнозначен предыдущему. Длина тестовой последовательности не менее 100 бит, длина блока больше 20 бит.
- **Тест на серийность** (Runs Test). В тесте находятся все серии битов – непрерывные последовательности одинаковых битов – и их распределение

сравнивается с ожидаемым распределением таких серий для случайной последовательности. Длина последовательности 100 и более бит.

- **Тест на максимальный размер серии единиц**. Исследуется длина наибольшей непрерывной последовательности единиц и сравнивается с длиной такой цепочки для случайной последовательности.
- **Матрично-ранговый тест** (Random Binary Matrix Rank Test). Цель теста – проверка линейной зависимости между подстроками фиксированного размера – матрицами 32x32 бита. Длина последовательности – не менее 38 912 бит, или 38 матриц.
- **Спектральный тест** (тест дискретным преобразованием Фурье). Цель теста – обнаружить повторяющиеся блоки или последовательности.
- **Тест с неперекрывающимися непериодическими шаблонами** (Non-overlapping (Aperiodic) Template Matching Test). Показывает число заранее заданных битовых строк (шаблонов) в последовательности.
- **Тест на перекрывающиеся периодические шаблоны** (Overlapping (Periodic) Template Matching Test). Показывает количество заранее определенных шаблонов (периодичных битовых последовательностей) в тестовой последовательности.
- **Универсальный статистический тест** (Maurer's Universal Statistical Test). Показывает число бит между двумя шаблонами и служит для определения сжимаемости последовательности.
- **Комплексный тест Lempel-Ziv** (Lempel-Ziv Complexity Test). Цель теста – определить число четных слов в последовательности и таким образом определить сжимаемость последовательности.

Кроме этих тестов есть еще несколько, но мы просто перечислим их без описания:

- линейный тест (Linear Complexity Test);
- серийный тест (Serial Test);
- приближенный тест на энтропию (Approximate Entropy Test);
- суммирующий тест (Cumulative Sum (Cusum) Test);
- тест на случайные отклонения (Random Excursions Test и Random Excursions Variant Test).

Как видите, получение случайных чисел совершенно нетривиальная задача, и тут есть место не только изящным инженерным решениям (для аппаратных генераторов), но и для кропотливого труда математиков и аналитиков. Плохой генератор, который дает неслучайные числа, может сильно ослабить защищенность криптосистемы, поэтому разработчики тестов предпочитают перестраховаться – лучше назвать неслучайным истинно случайное число, чем наоборот.

Рассмотрим еще один набор тестов для анализа свойств псевдослучайных последовательностей **Diehard**.

Тесты Diehard — это набор статистических тестов для измерения качества набора случайных чисел. Они были разработаны Джорджем Марсальей (англ. *George Marsaglia*) в течение нескольких лет и впервые опубликованы на CD-ROM, посвящённом случайным числам. Вместе они рассматриваются как один из наиболее строгих известных наборов тестов.

- **Дни рождения** (Birthday Spacings) — выбираются случайные точки на большом интервале. Расстояния между точками должны быть асимптотически распределены по Пуассону. Название этот тест получил на основе парадокса дней рождения.
- **Пересекающиеся перестановки** (Overlapping Permutations) — анализируются последовательности пяти последовательных случайных чисел. 120 возможных перестановок должны получаться со статистически эквивалентной вероятностью.
- **Ранги матриц** (Ranks of matrices) — выбираются некоторое количество бит из некоторого количества случайных чисел для формирования матрицы над {0,1}, затем определяется ранг матрицы. Считаются ранги.
- **Обезьяньи тесты** (Monkey Tests) — последовательности некоторого количества бит интерпретируются как слова. Считаются пересекающиеся слова в потоке.

Количество «слов», которые не появляются, должны удовлетворять известному распределению. Название этот тест получил на основе теоремы о бесконечном количестве обезьян.

- **Подсчёт единичек** (Count the 1's) — считаются единичные биты в каждом из последующих или выбранных байт. Эти счётчики преобразуется в «буквы», и считаются случаи пятибуквенных «слов».
- **Тест на парковку** (Parking Lot Test) — случайно размещаются единичные окружности в квадрате 100×100. Если окружность пересекает уже существующую, попытаться ещё. После 12 000 попыток, количество успешно «припаркованных» окружностей должно быть нормально распределено.
- **Тест на минимальное расстояние** (Minimum Distance Test) — 8000 точек случайно размещаются в квадрате 10 000×10 000, затем находится минимальное расстояние между любыми парами. Квадрат этого расстояния должен быть экспоненциально распределён с некоторой медианой.
- **Тест случайных сфер** (Random Spheres Test) — случайно выбираются 4000 точек в кубе с ребром 1000. В каждой точке помещается сфера, чей радиус является минимальным расстоянием до другой точки. Минимальный объём сферы должен быть экспоненциально распределён с некоторой медианой.
- **Тест сжатия** (The Squeeze Test) — 2^{31} умножается на случайные вещественные числа в диапазоне [0,1) до тех пор, пока не получится 1. Повторяется 100 000 раз. Количество вещественных чисел необходимых для достижения 1 должно быть распределено определённым образом.
- **Тест пересекающихся сумм** (Overlapping Sums Test) — генерируется длинная последовательность на [0,1). Добавляются последовательности из 100 последовательных вещественных чисел. Суммы должны быть нормально распределены с характерной медианой и сигмой.
- **Тест последовательностей** (Runs Test) — генерируется длинная последовательность на [0,1). Подсчитываются восходящие и нисходящие последовательности. Числа должны удовлетворять некоторому распределению.
- **Тест игры в кости** (The Craps Test) — играется 200 000 игр в кости, подсчитываются победы и количество бросков в каждой игре. Каждое число должно удовлетворять некоторому распределению.

Литература

1. Криптографическая защита информации в АСУ СН. Курс лекций. В.И. Долгов. ХВУ. 1998.

2. Криптографическая защита информации в информационных системах. Курс лекций. И.Д. Горбенко. ХНУРЭ. 2002.

3. Теория информации. Курс лекций. В.И. Долгов. ХВУ. 1998.

4. Брюс Шнайер. Прикладная криптография. 2-ое издание. Протоколы, алгоритмы и исходные тексты на языке C. Доступно: http://nrjetix.com/r-and-d/lectures

5. Национальный институт стандартов и технологий. Web-сайт: National Institute of Standards and Technology

6. Средство для проверки генераторов случайных чисел пригодных для криптографических целей. Доступно: NIST STS Software

7. Руководство по работе со средством для статистического анализа случайных последовательностей. Доступно: Guide to the Statistical Tests.

8. Д. Кнут. Искусство программирования. Т.2