Задания СРС:

**Цель: у**глубление теоретических знаний и развитие практических навыков по обеспечению кибербезопасности, формирование компетенций в области защиты информации, освоение принципов функционирования систем защиты, а также анализ современных киберугроз и методов противодействия им.

### Лекция ә: Поточные шифры и генераторы псевдослучайных чисел. Часть 1

**Цель:**  
Закрепить знания о типах поточных шифров и ГПСЧ, проанализировать современные реализации и требования к их безопасности.

**Задание:**

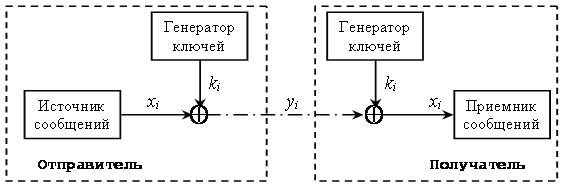
1. Повторить ключевые понятия из Лекции №2 (Часть 1):  
   – Синхронные и асинхронные поточные шифры  
   – Основные принципы генерации псевдослучайных последовательностей
2. Изучить современные поточные шифры, используемые в криптографии:  
   – A5/1 и A5/2 (использовались в GSM-связи)  
   – RC4 (применялся в WEP и TLS)  
   – Salsa20/ChaCha (современные шифры с открытым исходным кодом)
3. Выполнить аналитическое задание:  
   – Сравнить минимум два поточных шифра по следующим критериям:  
   • Скорость  
   • Стойкость к криптоанализу  
   • Применение  
   • Степень предсказуемости выходных битов
4. Подготовить:  
   – Презентацию (10–15 слайдов)  
   – Таблицу сравнения  
   – Краткий письменный вывод (0,5–1 страница)

Цель лекции: познакомиться с понятием "*поточный шифр*", а также с принципами использования генераторов псевдослучайных ключей при *потоковом шифровании*

### Поточные шифры

Блочный *алгоритм* предназначен для шифрования блоков определенной длины. Однако может возникнуть необходимость шифрования данных не блоками, а, например, по символам. **Поточный шифр (stream cipher)** выполняет преобразование входного сообщения по одному биту (или байту) за операцию. Поточный *алгоритм* шифрования устраняет необходимость разбивать сообщение на *целое число* блоков достаточно большой длины, следовательно, он может работать в реальном времени. Таким образом, если передается *поток* символов, каждый символ может шифроваться и передаваться сразу.

Работа типичного поточного шифра представлена на [рис. 7.1](https://www.intuit.ru/studies/courses/691/547/lecture/12383?page=1" \l "image.7.1).



**Рис. 7.1.**Принцип работы поточного шифра

*Генератор* ключей выдает *поток* битов ki, которые будут использоваться в качестве гаммы. Источник сообщений генерирует биты открытого текста хi, которые складываются по модулю 2 с гаммой, в результате чего получаются биты зашифрованного сообщения уi:

Описание: y_i = x_i \oplus k_i, i=1,2,…,n

Чтобы из шифротекста y1, y2,..., yn восстановить сообщение x1, x2,..., xn, необходимо сгенерировать точно такую же ключевую последовательность k1, yk,..., kn, что и при шифровании, и использовать для расшифрования формулу

Описание: x_i = y_i \oplus k_i,\  i=1,2,…,n


Обычно исходное сообщение и ключевая последовательность представляют собой независимые потоки *бит*. Таким образом, так как шифрующее (и расшифрующее) преобразование для всех поточных шифров одно и то же, они должны различаться только способом построения генераторов ключей. Получается, что *безопасность* системы полностью зависит от свойств генератора потока ключей. Если *генератор* потока ключей выдает последовательность, состоящую только из одних нулей (или из одних единиц), то зашифрованное сообщение будет в точности таким же, как и исходный *поток* битов (в случае единичных ключей зашифрованное сообщение будет инверсией исходного). Если в качестве гаммы используется один символ, представленный, например, восемью битами, то хотя зашифрованное сообщение и будет внешне отличаться от исходного, *безопасность* системы будет очень низкой. В этом случае при многократном повторении кода ключа по всей длине текста существует опасность его раскрытия статистическим методом. Поясним это на простом примере цифрового текста, закрытого коротким цифровым кодом ключа методом гаммирования.

**Пример**. Пусть известно, что исходное сообщение представляло собой двоично-десятичное число, то есть число, каждая тетрада (четыре бита) которого получена при переводе десятичной цифры 0...9 в двоичный вид. Перехвачено 24 бита зашифрованного сообщения Y, то есть шесть тетрад Y1, Y2, Y3, Y4, Y5, Y6, а именно *значение* 1100 1101 1110 1111 0000 0001. Известно, что *ключ шифрования* состоял из четырех *бит*, которые тоже представляют собой однозначное десятичное число, то есть одно и то же *значение* 0≤K≤9 использовалось для шифрования каждых четырех *бит* исходного сообщения. Таким образом, *шифрование* числа X1, X2, X3, X4, X5, X6 ключом К можно представить в виде системы уравнений:

Описание: \begin {multiple}X_1 \oplus K=1100\ \ \ X_2 \oplus K=1101\ \ \ X_3 \oplus K=1110\\
X_4 \oplus K=1111\ \ \ X_5 \oplus K=0000\ \ \ X_6 \oplus K=0001
\end{multiple}

Исходя из условия, что Хi принимает десятичные значения от 0 до 9, для поиска неизвестного К определим все возможные значения X1’; и К, сумма которых по модулю 2 приводит к результату 1100:

K = 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001

Y1 = 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100

---------------------

X1’ = 1100 1101 1110 1111 1000 1001 1010 1011 0100 0101

Так как исходное *значение* состояло из цифр от 0 до 9, то можно исключить из рассмотрения значения ключа 0000, 0001, 0010, 0011, 0110, 0111, так при сложении с ними получаются значения большие 9 в десятичном эквиваленте. Такие значения не могли присутствовать в открытом тексте. Таким образом, первый этап анализа уже позволил сократить количество возможных ключей с десяти до четырех.

Для дальнейшего поиска неизвестного К определим все возможные значения X2’; и оставшихся вариантов ключа, сумма которых по модулю 2 приводит к результату Y2 = 1101:

K = 0100 0101 1000 1001

Y2 = 1101 1101 1101 1101

---------------------

X2’ = 1001 1000 0101 0100

Видно, что этот этап не позволил отбросить ни одного из оставшихся вариантов ключа. Попытаемся это сделать, используя Y3=1110:

K = 0100 0101 1000 1001

Y3 = 1110 1110 1110 1110

---------------------

X2’ = 1010 1011 0110 0111

После проведения этого этапа становится ясно, что ключом не могли быть значения 0100 и 0101. Остается два возможных значения ключа: 1000(2)=8(10) и 1001(2)=9(10).

Дальнейший *анализ* по данной методике в данном случае, к сожалению, не позволит однозначно указать, какой же из двух полученных вариантов ключа использовался при шифровании. Однако можно считать успехом уже то, что *пространство* возможных ключей снизилось с десяти до двух. Остается попробовать каждый из двух найденных ключей для дешифровки сообщений и проанализировать смысл полученных вскрытых текстов.

В реальных случаях, когда исходное сообщение составлено не только из одних цифр, но и из других символов, использование статистического анализа позволяет быстро и точно восстановить *ключ* и исходные сообщения при короткой длине ключа, закрывающего *поток* секретных данных.

### Принципы использования генераторов псевдослучайных чисел при потоковом шифровании

Современная *информатика* широко использует псевдослучайные числа в самых разных приложениях — от методов математической статистики и имитационного моделирования до криптографии. При этом от качества используемых **генераторов псевдослучайных чисел** (ГПСЧ) напрямую зависит качество получаемых результатов.

ГПСЧ могут использоваться в качестве генераторов ключей в поточных шифрах. Целью использования *генераторов псевдослучайных чисел* является получение "бесконечного" ключевого слова, располагая относительно малой длиной самого ключа. *Генератор псевдослучайных чисел* создает последовательность битов, похожую на случайную. На самом деле, конечно же, такие последовательности вычисляются по определенным правилам и не являются случайными, поэтому они могут быть абсолютно точно воспроизведены как на передающей, так и на принимающей стороне. Последовательность ключевых символов, использующаяся при шифровании, должна быть не только достаточно длинной. Если *генератор* ключей при каждом включении создает одну и ту же последовательность битов, то взломать такую систему также будет возможно. Следовательно, *выход* генератора потока ключей должен быть функцией ключа. В этом случае расшифровать и прочитать сообщения можно будет только с использованием того же ключа, который использовался при шифровании.

Для использования в криптографических целях *генератор псевдослучайных чисел* должен обладать следующими свойствами:

1. период последовательности должен быть очень большой;
2. порождаемая последовательность должна быть "почти" неотличима от действительно случайной;
3. вероятности порождения различных значений должны быть в точности равны;
4. для того, чтобы только законный получатель мог расшифровать сообщение, следует при получении потока ключевых битов ki использовать и учитывать некоторый секретный ключ, причем вычисление числа ki+1 по известным предыдущим элементам последовательности ki без знания ключа должно быть трудной задачей.

При наличии указанных свойств последовательности псевдослучайных чисел могут быть использованы в поточных шифрах.

### Линейный конгруэнтный генератор псевдослучайных чисел

*Генераторы псевдослучайных чисел* могут работать по разным алгоритмам. Одним из простейших генераторов является так называемый **линейный конгруэнтный генератор**, который для вычисления очередного числа ki использует формулу

ki=(a\*ki-1+b)mod c,

где а, b, с — некоторые *константы*, a ki-1 — предыдущее *псевдослучайное число*. Для получения k1 задается начальное *значение* k0. Возьмем в качестве примера a=5,b=3,c=11 и пусть k0= 1. В этом случае мы сможем по приведенной выше формуле получать значения от 0 до 10 (так как с = 11). Вычислим несколько элементов последовательности:

k1 = (5 \* 1 + 3) mod 11 = 8;

k2 = (5 \* 8 + 3) mod 11 = 10;

k3 = (5 \* 10 + 3) mod 11 = 9;

k4 = (5 \* 9 + 3) mod 11 = 4;

k5 = (5 \* 4 + 3) mod 11 = 1.

Полученные значения (8, 10, 9, 4, 1) выглядят похожими на случайные числа. Однако следующее *значение* k6 будет снова равно 8:

k6 = (5 \* 1 + 3) mod 11 = 8,

а значения k7 и k8 будут равны 10 и 9 соответственно:

k7 = (5 \* 8 + 3) mod 11 = 10;

k8= (5 \* 10 + 3) mod 11 = 9.

Выходит, наш *генератор псевдослучайных чисел* повторяется, порождая периодически числа 8, 10, 9, 4, 1. К сожалению, это свойство характерно для всех линейных конгруэнтных генераторов. Изменяя значения основных параметров a, b и c, можно влиять на длину периода и на сами порождаемые значения ki. Так, например, увеличение числа с в общем случае ведет к увеличению периода. Если параметры a, b и c выбраны правильно, то *генератор* будет порождать случайные числа с максимальным периодом, равным c. При программной реализации *значение* с обычно устанавливается равным 2b-1 или 2b, где b — *длина* слова ЭВМ в битах.

Достоинством линейных конгруэнтных *генераторов псевдослучайных чисел* является их простота и высокая скорость получения псевдослучайных значений. Линейные конгруэнтные генераторы находят применение при решении задач моделирования и математической статистики, однако в криптографических целях их нельзя рекомендовать к использованию, так как специалисты по криптоанализу научились восстанавливать всю последовательность ПСЧ по нескольким значениям. Например, предположим, что противник может определить значения k0, k1, k2, k3. Тогда:

k1=(a\*k0+b) mod c

k2=(a\*k1+b) mod c

k3=(a\*k2+b) mod c

Решив систему из этих трех уравнений, можно найти a, b и c.

Для получения псевдослучайных чисел предлагалось использовать также квадратичные и кубические генераторы:

ki=(a12\*ki-1+a2\*ki-1+b) mod c

ki=(a13\*ki-1+a22\*ki-1+a3\*ki-1+b) mod c

Однако такие генераторы тоже оказались непригодными для целей криптографии по той же самой причине "предсказуемости".

### Метод Фибоначчи с запаздыванием

Известны и другие схемы получения псевдослучайных чисел.

**Метод Фибоначчи с запаздываниями** (*Lagged Fibonacci Generator*) — один из методов генерации псевдослучайных чисел. Он позволяет получить более высокое "качество" псевдослучайных чисел.

Наибольшую популярность фибоначчиевы датчики получили в связи с тем, что скорость выполнения арифметических операций с вещественными числами сравнялась со скоростью целочисленной арифметики, а фибоначчиевы датчики естественно реализуются в *вещественной арифметике*.

Известны разные схемы использования метода Фибоначчи с запаздыванием. Один из широко распространённых фибоначчиевых датчиков основан на следующей рекуррентной формуле:

Описание: 7_3

где ki — вещественные числа из диапазона [0,1], a, b — целые положительные числа, параметры генератора. Для работы фибоначчиеву датчику требуется знать max{a,b} предыдущих сгенерированных случайных чисел. При программной реализации для хранения сгенерированных случайных чисел необходим некоторый объем памяти, зависящих от параметров a и b.

**Пример**. Вычислим последовательность из первых десяти чисел, генерируемую методом Фибоначчи с запаздыванием начиная с k5 при следующих исходных данных: a = 4, b = 1, k0=0.1; k1=0.7; k2=0.3; k3=0.9; k4=0.5:

k5 = k1 - k4 = 0.7 - 0.5 = 0.2;

k6 = k2 - k5= 0.3 - 0.2 = 0.1;

k7 = k3 - k6 = 0.9 - 0.1 = 0.8;

k8 = k4 - k7 + 1 =0.5 - 0.8 + 1 = 0.7;

k9 = k5- k8 + 1 =0.2 - 0.7 + 1 = 0.5;

k10 = k6 - k9 + 1 =0.1 - 0.5 + 1 = 0.6;

k11 = k7 - k10 = 0.8 - 0.6 = 0.2;

k12 = k8 - k11 = 0.7 - 0.2 = 0.5;

k13 = k9 - k12 + 1 =0.5 - 0.5 + 1 = 1;

k14 = k10 - k13 + 1 =0.6 - 1 + 1 = 0.6.

Видим, что генерируемая последовательность чисел внешне похожа на случайную. И действительно, исследования подтверждают, что получаемые случайные числа обладают хорошими статистическими свойствами.

Для генераторов, построенных по *методу Фибоначчи* с запаздыванием, существуют рекомендуемые параметры a и b, так сказать, протестированные на качество. Например, исследователи предлагают следующие значения: (a,b) = (55, 24), (17, 5) или (97,33). Качество получаемых случайных чисел зависит от значения *константы* a: чем оно больше, тем выше *размерность* пространства, в котором сохраняется равномерность случайных векторов, образованных из полученных случайных чисел. В то же время с увеличением величины *константы* a увеличивается *объём* используемой алгоритмом памяти.

В результате значения (a,b) = (17,5) рекомендуются для простых приложений. Значения (a,b) = (55,24) позволяют получать числа, удовлетворительные для большинства криптографических алгоритмов, требовательных к качеству случайных чисел. Значения (a,b) = (97,33) позволяют получать очень качественные случайные числа и используются в алгоритмах, работающих со случайными векторами высокой размерности.

Генераторы ПСЧ, основанные на методе Фибоначчи с запаздыванием, использовались для целей криптографии. Кроме того, они применяются в математических и статистических расчетах, а также при моделировании случайных процессов. *Генератор* ПСЧ, построенный на основе метода Фибоначчи с запаздыванием, использовался в широко известной системе Matlab.

### Генератор псевдослучайных чисел на основе алгоритма BBS

Широкое распространение получил *алгоритм* генерации псевдослучайных чисел, называемый **алгоритмом BBS** (от фамилий авторов — L. Blum, M. Blum, M. Shub) или *генератором с квадратичным остатком*. Для целей криптографии этот метод предложен в 1986 году.

Он заключается в следующем. Вначале выбираются два больших простых1 числа p и q. Числа p и q должны быть оба *сравнимы* с 3 по модулю 4, то есть при делении p и q на 4 должен получаться одинаковый *остаток* 3. Далее вычисляется число M = p\* q, называемое целым числом Блюма. Затем выбирается другое случайное *целое число* х, взаимно простое (то есть не имеющее общих делителей, кроме единицы) с М. Вычисляем х0= х2mod M. х0 называется стартовым числом генератора.

На каждом n-м шаге работы генератора вычисляется хn+1= хn2 mod M. Результатом n-го шага является один (обычно младший) *бит* числа хn+1. Иногда в качестве результата принимают *бит* чётности, то есть количество единиц в двоичном представлении элемента. Если количество единиц в записи числа четное – *бит* четности принимается равным 0, нечетное – *бит* четности принимается равным 1.

**Например**, пусть p = 11, q = 19 (убеждаемся, что 11 mod 4 = 3, 19 mod 4 = 3). Тогда M = p\* q = 11\*19=209. Выберем х, взаимно простое с М: пусть х = 3. Вычислим стартовое число генератора х0:

х0 = х2 mod M = 32 mod 209 = 9 mod 209 = 9.

Вычислим первые десять чисел хi по алгоритму *BBS*. В качестве случайных *бит* будем брать младший *бит* в двоичной записи числа хi:

|  |  |  |
| --- | --- | --- |
| х1=92 mod 209= 81 mod 209= 81 | младший бит: | 1 |
| х2=812 mod 209= 6561 mod 209= 82 | младший бит: | 0 |
| х3=822 mod 209= 6724 mod 209= 36 | младший бит: | 0 |
| х4=362 mod 209= 1296 mod 209= 42 | младший бит: | 0 |
| х5=422 mod 209= 1764 mod 209= 92 | младший бит: | 0 |
| х6=922 mod 209= 8464 mod 209= 104 | младший бит: | 0 |
| х7=1042 mod 209= 10816 mod 209= 157 | младший бит: | 1 |
| х8=1572 mod 209= 24649 mod 209= 196 | младший бит: | 0 |
| х9=1962 mod 209= 38416 mod 209= 169 | младший бит: | 1 |
| х10=1692 mod 209= 28561 mod 209= 137 | младший бит: | 1 |

Самым интересным для практических целей свойством этого метода является то, что для получения n-го числа последовательности не нужно вычислять все предыдущие n чисел хi. Оказывается хn можно сразу получить по формуле

Описание: x_n=x_0^{2^n mod ((p-1)(q-1)} mod \: M

Например, вычислим х10 сразу из х0:

Описание: x_{10}=x_0^{2^{10} mod ((11-1)(19-1)} mod \ 209 = x_0^{1024 mod \ 180} mod \ 209 =\\
9^{124} \ mod \ 209=(9^4)^{31} \ mod \ 209=81^{31} \ mod \ 209 =\\
(81^{15} \ mod \ 209)(81^{16} \ mod \ 209)=((81^3)^5) \ mod \ 209)((81^4)^4) \ mod \ 209)=\\
(26^5 \ mod \ 209)(42^4 \ mod \ 209)=(144*104) \ mod \ 209=14976 \ mod \ 209=137

В результате действительно получили такое же *значение*, как и при последовательном вычислении, – 137. Вычисления кажутся достаточно сложными, однако на самом деле их легко оформить в виде небольшой процедуры или программы и использовать при необходимости.

Возможность "прямого" получения хn позволяет использовать *алгоритм* *BBS* при потоковой шифрации, например, для файлов с произвольным доступом или фрагментов файлов с записями *базы данных*.

*Безопасность* алгоритма *BBS* основана на сложности разложения большого числа М на множители. Утверждается, что если М достаточно велико, его можно даже не держать в секрете; до тех пор, пока М не разложено на множители, никто не сможет предсказать *выход* генератора ПСЧ. Это связано с тем, что задача разложения чисел вида n = pq (р и q — простые числа) на множители является вычислительно очень трудной, если мы знаем только n, а р и q — большие числа, состоящие из нескольких десятков или сотен *бит* (это так называемая *задача факторизации*).

Кроме того, можно доказать, что *злоумышленник*, зная некоторую последовательность, сгенерированную генератором *BBS*, не сможет определить ни предыдущие до нее биты, ни следующие. *Генератор* *BBS* *непредсказуем в левом направлении* и *в правом направлении*. Это свойство очень полезно для целей криптографии и оно также связано с особенностями разложения числа М на множители.

Самым существенным недостатком алгоритма является то, что он недостаточно быстр, что не позволяет использовать его во многих областях, например, при вычислениях в реальном времени, а также, к сожалению, и при *потоковом шифровании*.

Зато этот *алгоритм* выдает действительно хорошую последовательность псевдослучайных чисел с большим периодом (при соответствующем выборе исходных параметров), что позволяет использовать его для криптографических целей при генерации ключей для шифрования.

### Ключевые термины

**Stream cipher** – *поточный шифр*.

**Алгоритм BBS** – один из методов генерации псевдослучайных чисел. Название алгоритма происходит от фамилий авторов - L. Blum, M. Blum, M. Shub. *Алгоритм* может использоваться в криптографии. Для вычислений очередного числа xn+1 по алгоритму *BBS* используется формула хn+1= хn2 mod M, где M = pq является произведением двух больших простых p и q.

**Генератор псевдослучайных чисел (ГПСЧ)** – некоторый *алгоритм* или устройство, которые создают последовательность битов, внешне похожую на случайную.

**Линейный конгруэнтный генератор** псевдослучайных чисел – один из простейших ГПСЧ, который для вычисления очередного числа ki использует формулу ki=(a\*ki-1+b)mod c, где а, b, с — некоторые *константы*, a ki-1 — предыдущее *псевдослучайное число*.

**Метод Фибоначчи с запаздываниями** – один из методов генерации псевдослучайных чисел. Может использоваться в криптографии.

**Поточный шифр** – *шифр*, который выполняет *шифрование* входного сообщения по одному биту (или байту) за операцию. Поточный *алгоритм* шифрования устраняет необходимость разбивать сообщение на *целое число* блоков. *Поточные шифры* используются для шифрования данных в реальном времени.

### Краткие итоги

*Поточный шифр* – это *шифр*, который выполняет *шифрование* входного сообщения по одному биту (или байту) за операцию. Поточный *алгоритм* шифрования устраняет необходимость разбивать сообщение на *целое число* блоков. Таким образом, если передается *поток* символов, каждый символ может шифроваться и передаваться сразу. *Поточные шифры* используются для шифрования данных в режиме реального времени.

В качестве генераторов ключей в поточных шифрах могут использоваться *генераторы псевдослучайных чисел* (ГПСЧ). Целью использования ГПСЧ является получение "бесконечного" ключевого слова, располагая относительно малой длиной самого ключа. Для использования в криптографических целях *генератор псевдослучайных чисел* должен обладать некоторыми свойствами, например, период последовательности, порождаемой генератором, должен быть очень большой.

Простейшими генераторами псевдослучайных чисел являются: линейный конгруэнтный *генератор*, *генератор* по *методу Фибоначчи* с запаздываниями, *генератор* на основе алгоритма Блюм – Блюма – Шуба (*BBS*).

### Набор для практики

#### Вопросы для самопроверки

1. Чем *поточный шифр* отличается от блочного?
2. Каким образом организуется шифрование потока данных переменной длины?
3. Какие числа называют "псевдослучайными"?
4. Какими свойствами должен обладать *генератор псевдослучайных чисел* для использования в криптографических целях?
5. Какие *генераторы псевдослучайных чисел* Вы можете назвать?
6. Перечислите основные характеристики, достоинства и недостатки каждого из рассмотренных в данной лекции *генераторов псевдослучайных чисел*.