Оглавление

Раздел 1. Основы теории информации

Тема 1. Непрерывная и дискретная информация

http://www.tspu.tula.ru/ivt/old_site/umr/timoi/solovieva/Computer/sys_kod.htm

Термин **"информатика"** (франц. *informatique*) происходит от французских слов *information* (информация) и *automatique* (автоматика) и дословно означает "информационная автоматика". Широко распространён также англоязычный вариант этого термина — "*Computer science*", что означает буквально "компьютерная наука".

Информатика — это основанная на использовании компьютерной техники дисциплина, изучающая структуру и общие свойства информации, а также закономерности и методы её создания, хранения, поиска, преобразования, передачи и применения в различных сферах человеческой деятельности. В *1978* году международный научный конгресс официально закрепил за понятием "информатика" области, связанные с разработкой, созданием, использованием и материально-техническим обслуживанием систем обработки информации, включая компьютеры и их программное обеспечение, а также организационные, коммерческие, административные и социально-политические аспекты компьютеризации — массового внедрения компьютерной техники во все области жизни людей. Таким образом, информатика базируется на компьютерной технике и немыслима без нее.

Информатика — научная дисциплина с широчайшим диапазоном применения. Её основные направления:

- разработка вычислительных систем и программного обеспечения;
- теория информации, изучающая процессы, связанные с передачей, приёмом, преобразованием и хранением информации;
- методы искусственного интеллекта, позволяющие создавать программы для решения задач, требующих определённых интеллектуальных усилий при выполнении их человеком (логический вывод, обучение, понимание речи, визуальное восприятие, игры и др.);
- системный анализ, заключающийся в анализе назначения проектируемой системы и в установлении требований, которым она должна отвечать;
- методы машинной графики, анимации, средства мультимедиа;
- средства телекоммуникации, в том числе, глобальные компьютерные сети, объединяющие всё человечество в единое информационное сообщество;
- разнообразные приложения, охватывающие производство, науку, образование, медицину, торговлю, сельское хозяйство и все другие виды хозяйственной и общественной деятельности.

Термином информатика обозначают совокупность дисциплин, изучающих свойства информации, а также способы представления, накопления, обработки и передачи информации с помощью технических средств. Теоретическую основу информатики образует группа фундаментальных наук, которую в рав-

ной степени можно отнести как к математике, так и к кибернетике: теория информации, теория алгоритмов, математическая логика, теория формальных языков и грамматик, комбинаторный анализ и т. д. Кроме них информатика включает такие разделы, как архитектура ЭВМ, операционные системы, теория баз данных, технология программирования и многие другие.

Информационная технология есть совокупность конкретных технических и программных средств, с помощью которых мы выполняем разнообразные операции по обработке информации во всех сферах нашей жизни и деятельности. Иногда информационную технологию называют компьютерной технологией или прикладной информатикой. Информатика является комплексной, междисциплинарной отраслью научного знания.

Понятие **информация** является одним из фундаментальных в современной науке. Информацию наряду с веществом и энергией рассматривают в качестве важнейшей сущности мира, в котором мы живем.

Термин **информация** ведет свое происхождение от латинского слова *informatio*, означающего разъяснение, изложение, осведомленность. Информацию мы передаем друг другу в устной и письменной форме, а также в форме жестов и знаков. Любую нужную информацию мы осмысливаем, передаем другим и делаем определенные умозаключения на ее основе. Информацию мы извлекаем из учебников и книг, газет и журналов, телепередач и кинофильмов. Записываем ее в тетрадях и конспектах. В производственной деятельности информация передается в виде текстов и чертежей, справок и отчетов, таблиц и других документов. Такого рода информация может предоставляться и с помощью ЭВМ.

В любом виде информация для нас выражает сведения о ком-то или о чем-то. Она отражает происходящее или происшедшее в нашем мире, например, что мы делали вчера или будем делать завтра, как провели летний отпуск или каков будет характер будущей работы. При этом информация обязательно должна получить некоторую форму - форму рассказа, рисунка, статьи и т. д. Чертежи и музыкальные произведения, книги и картины, спектакли и кинофильмы - все это формы представления информации.

Информация, в какой бы форме она ни предоставлялась, является некоторым отражением реального или вымышленного мира. Поэтому информация - это отражение предметного мира с помощью знаков и сигналов. Стоит отметить, что абсолютно точное определение информации дать невозможно, это такое же первичное понятие, как точка или плоскость в геометрии. Выделяют два подхода к понятию информация: субъективный и кибернетический.

Субъективный подход (бытовой человеческий):

Информация - это знания, сведения, которыми обладает человек, которые он получает из окружающего мира.

Человеку свойственно субъективное восприятие информации через некоторый набор ее свойств: важность, достоверность, своевременность, доступность и т.д. В этом смысле одно и то же сообщение, передаваемое от источника к получателю, может передавать информацию в разной степени. Так, например, вы хотите сообщить о неисправности компьютера. Для инженера из группы

технического обслуживания сообщение "компьютер сломался" явно содержит меньше информации, чем сообщение "Не включается монитор", поскольку второе сообщение в большей степени снимает неопределенность, связанную с причиной неисправности компьютера.

Киберетический подход:

Между информатикой и кибернетикой существует тесная связь. Основал кибернетику в конце 1940-х гг. американский ученый Норберт Винер. Можно сказать, что кибернетика породила современную информатику, выполнила роль одного из ее источников. Сейчас кибернетика входит в информатику как составная часть. Кибернетика имеет дело со сложными системами: машинами, живыми организмами, общественными системами. Но она не стремится разобраться в их внутреннем механизме. Кибернетику интересуют процессы взаимодействия между такими системами или их компонентами. Рассматривая такие взаимодействия как процессы управления, кибернетику определяют как науку об общих свойствах процессов управления в живых и неживых системах. Для описания сложных систем в кибернетике используется модель "черного ящика". Термины "черный ящик" и "кибернетическая система" можно использовать как синонимы. Главные характеристики "черного ящика" - это входная и выходная информация. И если два таких "черных ящика" взаимодействуют между собой, то делают они это только путем обмена информацией.

Информация - это содержание последовательностей символов (сигналов) из некоторого алфавита.

В таком случае все виды информационных процессов (хранение, передача, обработка) сводятся к действиям над символами, что и происходит в технических информационных системах.

Информация передается в виде **сообщений**, определяющих форму и представление передаваемой информации. Примерами сообщений являются музыкальное произведение; телепередача; команды регулировщика на перекрестке и т.д. При этом предполагается, что имеются **"источник информации"** и **"получатель информации"**.

Сообщение от источника к получателю передается посредством какой-нибудь среды, являющейся в таком случае **"каналом передачи информации"**. Так, при передаче речевого сообщения в качестве такого канала связи можно рассматривать воздух, в котором распространяются звуковые волны.

Получение информации - это получение фактов, сведений и данных о свойствах, структуре или взаимодействии объектов и явлений окружающего нас мира.

Чтобы сообщение было передано от источника к получателю, необходима некоторая материальная субстанция - **носитель информации**.

Сигнал - сообщение, передаваемое с помощью носителя.

В общем случае сигнал - это изменяющийся во времени процесс. Такой процесс может содержать различные характеристики (например, при передаче электрических сигналов могут изменяться напряжение и сила тока).

Параметр сигнала - та из характеристик, которая используется для представления сообщений. Природа большинства физических явлений такова, что они могут принимать различные значения в определенном интервале (температура воды, скорость автомобиля и т.д.)

Непрерывный (**аналоговый**) способ представления информации - представление информации, в котором сигнал на выходе датчика будет меняться вслед за изменениями соответствующей физической величины.

Примеры непрерывной информации:

Примером непрерывного сообщения служит человеческая речь, передаваемая модулированной звуковой волной; параметром сигнала в этом случае является давление, создаваемое этой волной в точке нахождения приемника - человеческого уха.

Аналоговый способ представления информации имеет недостатки:

1. Точность представления информации определяется точностью измерительного прибора (например, точность числа отображающего напряжение в электрической цепи, зависит от точности вольтметра).
2. Наличие помех может сильно исказить представляемую информацию.

Дискретность (от лат. **discretus** – разделенный, прерывистый) – прерывность; противопоставляется непрерывности. Напр., дискретное изменение к.-л. величины во времени – это изменение, происходящее через определенные промежутки времени (скачками); система целых (в противоположность системе действительных чисел) является дискретной. Заметим, что в приведенной цитате указано на связь дискретности с системой целых чисел, и это можно считать подтверждением положения о том, что дискретные значения можно пронумеровать.

Дискретный сигнал - сигнал, параметр которого принимает последовательное во времени конечное число значений (при этом все они могут быть пронумерованы). Сообщение, передаваемое с помощью таких сигналов - **дискретным сообщением.** Информация, передаваемая источником, в этом случае также называется **дискретной информацией**.

Цифровой способ представления информации – представление информации в дискретном виде.

Примеры дискретной информации:

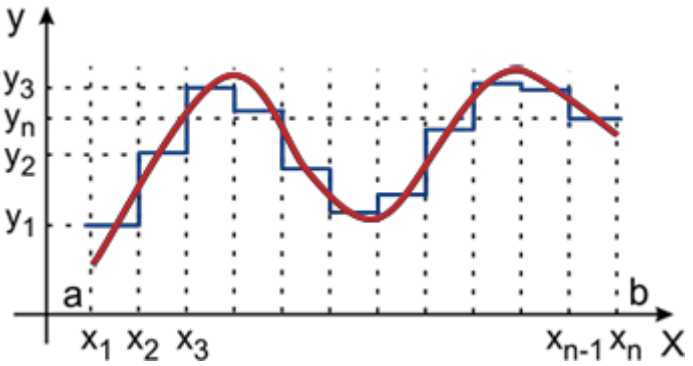
Дискретными являются показания цифровых измерительных приборов, например, вольтметра (сравните со "старыми", стрелочными приборами). Очевидным (в самом изначальном смысле этого слова!) образом дискретной является распечатка матричного принтера, а линия, проводимая графопостроителем, напротив, является непрерывной. Дискретным является растровый способ представления изображений, тогда как векторная графика по своей сути непре-

рывна. Дискретна таблица значений функции, но когда мы наносим точки из нее на миллиметровую бумагу и соединяем плавной линией, получается непрерывный график. Механический переключатель диапазонов в приемниках был сконструирован так, чтобы он принимал только фиксированные положения.
Тем не менее, все не так просто. То, что фотографии в старых газетах дискретны, видят и соглашаются все. А в современном красочном глянцевом журнале? А распечатка картинки на лазерном принтере – она дискретна или непрерывна (все-таки, она состоит из частичек специального порошка, а они маленькие, но конечные по размеру; да и сама характеристика dpi – количество точек на единицу площади наводит на сомнения в непрерывности картинки, хотя глаз упорно не видит дискретности)? Если еще в этот момент вспомнить, что твердые тела состоят из мельчайших атомов, а глаз, воспринимающий изображение, имеет чувствительные маленькие палочки и колбочки, то все вообще станет туманным и неоднозначным…

Видимо, чтобы не запутаться совсем, надо принять правило, что в тех случаях, когда рассматриваемая, величина имеет настолько большое количество значений, что мы не в состоянии их различить, то практически ее можно считать непрерывной.

Непрерывное сообщение может быть представлено непрерывной функцией, заданной на некотором отрезке **[a,b]**. Непрерывное сообщение можно преобразовать в дискретное, такая процедура называется **дискретизацией** (**оцифровывание**). Для этого из бесконечного множества значений этой функции (параметра сигнала) выбирается их определенное число, которое приближенно может характеризовать остальные значения. Один из способов такого выбора состоит в следующем. Этапы дискретизации:

Область определения функции разбивается точками $\mathbf{x_1, x_2, ..., x_n}$ на отрезки равной длины и на каждом из этих отрезков значение функции принимается постоянным и равным, например, среднему значению на этом отрезке; полученная на этом этапе **функция называется ступенчатой**. Следующий шаг - проецирование значений "ступенек" на ось значений функции (ось ординат). Полученная таким образом последовательность значений функции $\mathbf{y_1, y_2, ..., y_n}$ является дискретным представлением непрерывной функции, точность которого можно неограниченно улучшать путем уменьшения длин отрезков разбиения области значений аргумента.



Ось значений функции можно разбить на отрезки с заданным шагом и отобразить каждый из выделенных отрезков из области определения функции в соответствующий отрезок из множества значений. В итоге получим конечное множество чисел, определяемых, например, по середине или одной из границ таких отрезков.

Таким образом, любое сообщение может быть представлено как дискретное, иначе говоря, последовательностью знаков некоторого алфавита. Возможность дискретизации непрерывного сигнала с любой желаемой точностью (для возрастания точности достаточно уменьшить шаг) принципиально важна с точки зрения информатики. Компьютер - цифровая машина, т.е. внутреннее представление информации в нем дискретно. Дискретизация входной информации (если она непрерывна) позволяет сделать ее пригодной для компьютерной обработки. Существуют и другие вычислительные машины - аналоговые ЭВМ. Они используются обычно для решения задач специального характера и широкой публике практически не известны. Эти ЭВМ в принципе не нуждаются в дискретизации входной информации, так как ее внутренне представление у них непрерывно. В этом случае все наоборот - если внешняя информация дискретна, то ее перед использованием необходимо преобразовать в непрерывную.

Тема 2. Мера информации

Ярмолик В.Н., Портянко С.С., Ярмолик С.В. Криптография, стеганография и охрана авторского права. – Минск: Издательский центр БГУ, 2007. – 242с. (*Глава 3*.)

Неоценимый вклад в криптографию внес основоположник теории информации Клод Шеннон (**К.Shannon**). В 1949 К.Shannon опубликовал свои теоретические исследования по криптографии, основанные на полученных им ранее результатах в теории информации.

Одним из главных результатов по криптографии можно считать его теоретическое обоснование возможности создания **абсолютно секретных криптосистем**. Он определил теоретическую секретность шифра по неопределённости возможного исходного текста на основании полученного шифротекста. Соответственно, если вне зависимости от того, сколько шифротекста перехвачено, нельзя получить никакой информации относительно исходного текста, то шифр обладает **идеальной секретностью.**

Теория информации измеряет количество информации в сообщении по среднему количеству бит, необходимых для оптимального кодирования всех возможных сообщений. Например, поле *Gender* («пол») в базе данных содержит только один бит информации, потому что оно может быть закодировано одним битом (*Male* может быть представлено как “0”, *Female* – как “1”). Если поле будет представлено ASCII-кодами символов строк *Male* и *Female*, это потребует больше места на носителях информации, но не будет содержать в себе большего объема информации.

Количество информации в сообщении формально измеряется **энтропией** сообщения, которая базируется на понятии **количества информации**.

Пусть $X_1,...,X_n$ это n возможных сообщений, возникающих с вероятностями $p(X_1),...,p(X_n)$, и сумма этих вероятностей $p(X_i)$, $i=1,...,n$ равна 1. Тогда получение сообщения X_i можно оценить количеством полученной информации, которое вычисляется как $F(X_i)=-log_2p(X_i)=log_2(1/p(X_i))$. Очевидно, что при получении сообщения, вероятность которого крайне мала, будет получено большое количество информации, что следует из выражения $log_2(1/p(X_i))$ и наоборот для событий с большой вероятностью будет получено мало информации. Действительно, если $p(X_i)=1$, количество информации $F(X_i)=log_2(1/p(X_i))=log_21=0$. То есть, события, происходящие с вероятностью единица, не дают никакой информации.

Под **энтропией** понимают среднее количество информации при получении одного из возможных сообщений. Численно энтропия представляет собой средневзвешенное количество информации и вычисляется по формуле

$$H(X) = -\sum_{i=1}^{n} p(X_i)\log_2(p(X_i)) = \sum_{i=1}^{n} p(X_i)\log_2(1/p(X_i)).$$

Интуитивно, каждый элемент $log_2(1/p(X_i))$ в последнем выражении представляет собой число бит, необходимых для оптимального кодирования сообщения X_i. Действительно, при оптимальном кодировании сообщения (события), а в нашем случае, например, символа исходного текста необходимо использовать меньшее число бит для кодирования часто встречающегося символа, а для редко встречающегося символа – большее число бит. Тогда в среднем для кодирования сообщения, состоящего из множества символов, будет использовано оптимальное суммарное количество бит.

Поскольку $1/p(X)$ уменьшается при увеличении $p(X)$, оптимальное кодирование использует короткие коды для часто встречающихся сообщений за счёт использования длинных кодов для редких сообщений. Этот принцип применён в **коде Морзе**, где наиболее часто используемые буквы представлены самыми короткими кодами.

Код Хаффмана является оптимальным кодом, ассоциированным с буквами, словами, фразами или машинными инструкциями. Односимвольный код **Хаффмана** часто используется для минимизации больших файлов.

Пример 2.1. Пусть $n=3$, и пусть три сообщения представлены событиями A,B, и C, где $p(A)=1/2$ и $p(B)=p(C)=1/4$. Тогда $log_2(1/p(A))=log_22=1$; $log_2(1/p(B))=log_2(1/p(C))=log_24= 2$, что подтверждает наши предыдущие наблюдения о том, что для оптимального кодирования часто встречающегося сообщении нужно минимальное число бит.

Пример 2.2. Предположим, необходимо оптимально закодировать пол клиента в базе данных. Имеются две возможности (два события) *Male* и *Female* примерно с одинаковыми вероятностями $p(Male)=p(Female)=1/2$. Тогда значение энтропии будет вычисляться как

$$H(X)=p(Male)log_2(1/p(Male))+p(Female)log_2(1/p(Female))=$$
$$=(1/2)(log_22)+(1/2)(log_22)=1,$$

что подтверждает наши предыдущие наблюдения о том, что в поле базы данных пол оптимально будет закодирован одним битом. С другой стороны, независимо от того, как закодирован пол клиента в базе данных, он содержит в себе 1 бит информации.

Следующий пример иллюстрирует применение энтропии для определения содержания сообщения.

Пример 2.3. Пусть n=3, и пусть три сообщения представлены буквами A,B, и C, где $p(A)=1/2$, $p(B)=p(C)=1/4$. Тогда $H(X)=(1/2)log_2 2+2(1/4)log_2 4 =0,5+1,0=1,5$.

Оптимальное кодирование может быть достигнуто с использованием однобитного кода для кодирования A в силу того, что вероятность этого сообщения наибольшая, и двухбитных кодов для B и C. Например, A может быть закодировано битом 0, в то время как B и C могут кодироваться двумя битами каждый: 10 и 11. Применяя подобное кодирование, последовательность, состоящая из восьми букв ABCAABAC, кодируется, как 12-битная последовательность 010110010011 как показано ниже:

A	B	C	A	A	B	A	C
0	10	11	0	0	10	0	11

Среднее число бит на букву равно 12/8=1,5, что соответствует нашим предварительным наблюдениям. Действительно, при получении одного из трех сообщений A,B, или C среднее ожидаемое количество информации равняется 1,5.

Достижения в теории информации позволили формальным образом исследовать исходные тексты, представленные на конкретном языке, и использовать эти результаты для взлома криптосистем. Одним из таких методов взлома является **частотный анализ**. В соответствии с данным методом, распределение букв в криптотексте сравнивается с распределением букв в алфавите исходного сообщения. Вероятность успешного вскрытия повышается с увеличением длины криптотекста.

Пусть для заданного языка определено множество сообщений длиной N символов, тогда **частота (скорость) языка** для сообщений X длиной N определяется как

$$r=H(X)/N,$$

где величина r определяет среднее число бит информации в каждом символе сообщения.

Простейший способ определения частоты языка (**абсолютной частоты R**), основывается на предположении, что все буквы алфавита языка имеют одинаковую вероятность появления во всех возможных сообщениях так же, как и всевозможные последовательности букв алфавита в сообщениях равновероятны. Если алфавит языка содержит L букв, тогда **абсолютная частота** может быть получена так:

$$R=log_2 L,$$

Для английского языка L=26, тогда $R=log_2 L=log_2 26 \approx 4,7$.

Абсолютная частота языка $R \approx 4{,}7$ определяет максимальное число бит информации, которое может быть получено при получении одной буквы сообщения. Или, максимальное количество бит, необходимых для кодирования сообщения, представленного на английском языке.

Реальная частота английского языка значительно меньше абсолютной частоты. Это происходит потому, что английский язык, так же как и другие неискусственные языки, слишком многословный или, что то же самое, избыточный. Например, фраза «occurring frequently» («часто появляющийся») может быть сокращена на 58% как «crng frg» без потери информации.

Существуют множество различных таблиц распределений букв в том или ином языке, но, ни одна из них не содержит окончательной информации – даже порядок букв может отличаться в различных таблицах. Распределение букв очень сильно зависит от типа текста: проза, разговорный язык, технический язык и т.п. Наиболее часто встречающиеся распределения для английского и русского языка приведены в таблице 2.1 и таблице 2.2.

Таблица 2.1.

Частота букв английского языка

A	0,0804	B	0,0154	C	0,0306
D	0,0399	E	0,1251	F	0,0230
G	0,0196	H	0,0554	I	0,0726
J	0,0016	K	0,0067	L	0,0414
M	0,0253	N	0,0709	O	0,0760
P	0,0200	Q	0,0011	R	0,0612
S	0,0654	T	0,0925	U	0,0271
V	0,0099	W	0,0192	X	0,0019
Y	0,0173	Z	0,0009		

Таблица 2.2.

Частота букв русского языка

А	0,062	Л	0,053	Ц	0,004
Б	0,014	М	0,026	Ч	0,012
В	0,038	Н	0,053	Ш	0,006
Г	0,013	О	0,090	Щ	0,003
Д	0,025	П	0,023	Ы	0,016
Е	0,072	Р	0,040	Ь,Ъ	0,014
Ж	0,007	С	0,045	Э	0,003
З	0,016	Т	0,053	Ю	0,006
И	0,062	У	0,021	Я	0,018
Й	0,010	Ф	0,002		
К	0,028	Х	0,009		

Несмотря на то, что нет таблицы, которая может учесть все виды текстов, есть характерные черты общие для всех таблиц. Например, в английском языке буква *E* всегда возглавляет таблицу частот встречаемости, а *T* идет на второй позиции. *A* и *O* почти всегда третьи. Кроме того, девять букв английского языка *E*, *T*, *A*, *O*, *N*, *I*, *S*, *R*, *H* всегда имеют частоту выше, чем любые другие. Эти девять букв занимают примерно 70% английского текста. Ниже приведены соответствующие таблицы для различных языков.

Таблица 2.3.

Частота встречаемости девяти букв в различных языках

Русский		Английский		Немецкий		Французский		Итальянский		Финский	
О	0,090	E	0,125	E	0,184	E	0,159	E	0,118	A	0,121
Е	0,072	T	0,092	N	0,114	A	0,094	A	0,117	I	0,106
А	0,062	A	0,080	I	0,080	I	0,084	I	0,113	T	0,098
И	0,062	O	0,076	R	0,071	S	0,079	O	0,098	N	0,086
Н	0,053	I	0,073	S	0,070	T	0,073	N	0,069	E	0,081
Т	0,053	N	0,071	A	0,054	N	0,072	L	0,065	S	0,078
С	0,045	S	0,065	T	0,052	R	0,065	R	0,064	L	0,059
Р	0,040	R	0,061	U	0,050	U	0,062	T	0,056	O	0,055
В	0,038	H	0,055	D	0,049	L	0,053	S	0,050	K	0,052
#	0,515	#	0,699	#	0,726	#	0,741	#	0,750	#	0,736

Используя частоту встречаемости букв английского алфавита как распределение вероятностей $p(X_i)$ для вычисления энтропии получим значение частоты языка равную $r=H(1\text{-}grams)/1\approx4{,}15$. Как видим, реальные частоты встречаемости букв (1-*grams*) в английском тексте заметно уменьшили количество информации, которое они содержат.

Отметим, что в двух предыдущих случаях вычисления частоты языка была использована гипотеза отсутствия зависимости последовательностей букв в исходных текстах, хотя очевидно, что такая зависимость, безусловно, существует.

Известны статистические распределений частоты биграмм (двух символов) исходных текстов. Например, для английского языка такие сочетания букв как *TH* и *EN* возникают гораздо чаще, чем другие. Некоторые биграммы (например, OZ) никогда не возникают в сообщениях, содержащих смысловую информацию (исключение составляют акронимы). Используя подобное распределение для пар букв, частота английского языка будет иметь величину $r=H(2\text{-}grams)/2\approx3{,}62$.

Доля значащих (имеющих смысл) последовательностей букв любого языка понижается, когда длина последовательности увеличивается. Например, в английском языке можно встретить сочетание *BB* из двух букв *B*, и практически невозможно встретить триграммы *BBB*. Учитывая частоту распределения триграмм в английском языке, численное значение частоты языка примет значение $r=H(3\text{-}grams)/3\approx3{,}22$.

Частота языка (значение энтропии на один символ) будет иметь максимально близкую величину к истинному значению при использовании статистических результатов распределения N-грамм для возрастающих значений N. Когда N возрастает, энтропия на символ уменьшается, потому что для больших N количество сообщений, содержащих смысл, резко уменьшается по отношению к произвольному случайному набору из N букв. Показано, что в зависимости от структуры текста, использованного языка и других факторов реальное значение энтропии на один символ для больших значений N принимает значение из диапазона $r=1 \div 1,5$.

Как видно, реальное значение количества информации, содержащегося в одном символе сообщения, заметно меньше абсолютной частоты $R=4,7$. С другой стороны, такое их соотношение свидетельствует об избыточности языков. Формально избыточность языка с частотой r и абсолютной частотой R определена как $D=R-r$. Для $R=4,7$ и частоты $r=1$, $D=3,7$, что свидетельствует о том, что английский язык на 79% избыточен. Для $r=1,5$, $D=3,2$, предполагается избыточность на 68%.

Тема 3. Преобразование информации

http://www.tspu.tula.ru/ivt/old_site/umr/timoi/solovieva/Computer/sys_kod.htm

Информация передается в виде сообщений. **Дискретная информация** записывается с помощью некоторого конечного набора знаков, которые будем называть буквами, не вкладывая в это слово привычного ограниченного значения (типа "русские буквы" или "латинские буквы"). Буква в расширенном понимании любой из знаков, которые некоторым соглашением установлены для общения. Например, при привычной передаче сообщений на русском языке такими знаками будут русские буквы - прописные и строчные, знаки препинания, пробел; если в тексте есть числа - то и цифры.

Буква - элемент некоторого конечного множества (набора) отличных друг от друга знаков.

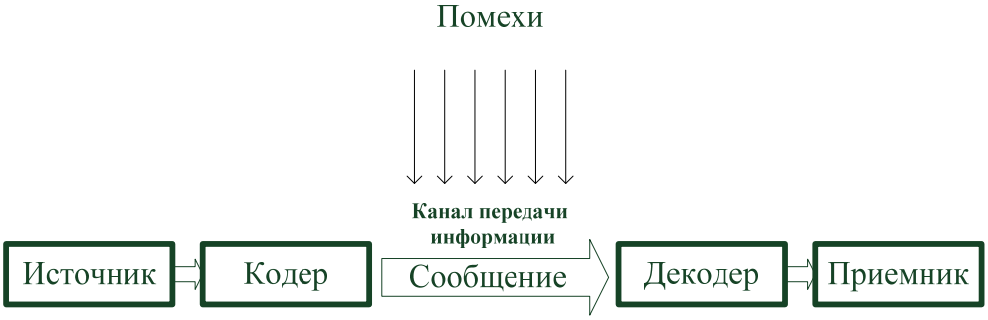
Алфавит - множество знаков (букв), в котором определен их порядок. Например, общеизвестен порядок знаков в русском алфавите: А, Б, ..., Я.

Некоторые примеры алфавитов:

1. Алфавит Морзе;
2. Алфавит клавиатурных символов;
3. Алфавит арабских цифр;
4. Алфавит шестнадцатеричных цифр (этот пример, в частности, показывает, что знаки одного алфавита могут образовываться из знаков других алфавитов);
5. Алфавит латинских букв;
6. Алфавит нотных символов.

В канале связи сообщение, составляемое из символов (букв) одного алфавита, может преобразовываться в сообщение из символов (букв) другого алфавита. Правило, описывающее однозначное соответствие букв алфавитов

при таком преобразовании, называют **кодом**. Саму процедуру преобразования сообщения называют **кодированием** (перекодировкой). Подобное преобразование сообщения может осуществляться в момент поступления сообщения от источника в канал связи (кодирование) и в момент приема сообщения получателем (декодирование). Устройства, обеспечивающие кодирование и декодирование, будем называть **кодером** (кодировщиком) и **декодером** (декодировщиком).



На рисунке представлена схема, иллюстрирующая процесс передачи сообщения в случае перекодировки, а также воздействия помех.

Рассмотрим некоторые примеры кодов.

1. Азбука **Морзе** в русском варианте (алфавиту, составленному из алфавита русских заглавных букв и алфавита арабских цифр, ставится в соответствие алфавит Морзе)

2. Код **Trifid** (знакам латинского алфавита ставятся в соответствие комбинации из трех знаков: 1, 2, 3):

A	111	J	211	S	311
B	112	K	212	T	312
C	113	L	213	U	313
D	121	M	221	V	321
E	122	N	222	W	322
F	123	O	223	X	323
G	131	P	231	Y	331
H	132	Q	232	Z	332
I	133	R	232	.	333

Код Trifid является примером, так называемого, равномерного кода (такого, в котором все кодовые комбинации содержат одинаковое число знаков - в данном случае три). Пример неравномерного кода - азбука Морзе.

3. Кодирование чисел знаками различных систем счисления.

Ранее отмечалось, что при передаче сообщений по каналам связи могут возникать помехи, способные привести к искажению принимаемых знаков. Так, например, если вы попытаетесь в ветреную погоду передать речевое сообщение

человеку, находящемуся от вас на значительном расстоянии, то оно может быть сильно искажено такой помехой, как ветер. Вообще, передача сообщений при наличии помех является серьезной теоретической и практической задачей. Ее значимость возрастает в связи с повсеместным внедрением компьютерных телекоммуникаций, в которых помехи неизбежны.

Основные проблемы, возникающие при работе с кодированной информацией, искажаемой помехами:

- установление самого факта того, что произошло искажение информации;
- выяснение того, в каком конкретно месте передаваемого текста произошло искажение информации;
- исправление ошибки, хотя бы с некоторой степенью достоверности.

Помехи в передачи информации - вполне обычное дело во всех сферах профессиональной деятельности и в быту. Один из примеров был приведен выше, другие примеры - разговор по телефону, в трубке которого "трещит", вождение автомобиля в тумане и т.д. Чаще всего человек вполне справляется с каждой из указанных выше задач, хотя и не всегда отдает себе отчет, как он это делает (т.е. неалгоритмические, а исходя из каких-то ассоциативных связей). Известно, что естественный язык (язык на котором разговаривают люди) большой избыточностью (в европейских языках - до 70%), чем объясняется большая помехоустойчивость сообщений, составленных из знаков алфавитов таких языков.

Примером, иллюстрирующим устойчивость русского язык к помехам, может служить предложение **"в словох всо глосноо зомононо боквой о"**. Здесь 26% символов "поражены", однако это не приводит к потере смысла. Таким образом, в данном случае избыточность является полезным свойством. Избыточность могла бы быть использована и при передаче кодированных сообщений в технических системах. Например, каждый фрагмент текста ("предложение") предается трижды, и верным считается та пара фрагментов, которая полностью совпала. Однако, большая избыточность приводит к большим временным затратам при передаче информации и требует большого объема памяти при ее хранении.

Впервые теоретическое исследование эффективного кодирования предпринял К.Шеннон. Информатика и ее приложения интернациональны. Это связано как с объективными потребностями человечества в единых правилах и законах хранения, передачи и обработки информации. Компьютер считают универсальным преобразователем информации. Тексты на естественных языках и числа, математические и специальные символы - одним словом все, что в быту или профессиональной деятельности может быть необходимо человеку, должно иметь возможность быть введенным в компьютер.

В силу безусловного приоритета двоичной системы счисления при внутреннем представлении информации в компьютере кодирование "внешних" символов основывается на сопоставлении каждому из них определенной группы двоичных знаков. При этом из технических соображений и из соображений удобства кодирования-декодирования следует пользоваться равномерными кодами, т.е. двоичными группами равной длины.

Попробуем подсчитать наиболее короткую длину такой комбинации с точки зрения человека, заинтересованного в использовании лишь одного естественного алфавита - скажем, английского: 26 букв следует умножить на 2 (прописные и строчные) - итого 52; 10 цифр; будем считать 10 знаков препинания; 10 разделительных знаков (три вида скобок, пробел и др.), знаки привычных математических действий, несколько специальных символов (типа %, @, &, $ и др.) - итого приблизительно 100. Точный подсчет здесь не нужен, поскольку нам предстоит решить задачу: имея, скажем равномерный код по N двоичных знаков, сколько можно образовать разных кодовых комбинаций. Ответ очевиден: $K=2^N$. Итак, при N = 6, K = 64 - явно мало, при N = 7, K = 128 - вполне достаточно. Однако, для кодирования нескольких (хотя бы двух) естественных алфавитов (плюс все отмеченные выше знаки) и этого недостаточно. Минимально достаточное значение N в этом случае 8; имея 256 комбинаций двоичных символов, вполне можно решить указанную задачу. Поскольку 8 двоичных символов составляют 1 байт, то говорят о **системах "байтового" кодирования.**
Наиболее распространены две такие системы: EBCDIC (Extended Binary Coded Decimal Interchange Code) и ASCII (American Standard Information Interchange). Первая - исторически тяготеет к "большим" машинам, вторая чаще используется на мини- и микроЭВМ (включая персональные компьютеры). Но даже 8-битная кодировка недостаточна для кодирования всех символов, которые хотелось бы иметь в расширенном алфавите. Все препятствия могут быть сняты при переходе на 16 - битную кодировку Unicode, допускающую 65536 кодовых комбинаций

Тема 4. Сжатие информации

http://book.itep.ru/2/26/ziv_261.htm

Алгоритм Зива-Лемпеля

В 1977 году Абрахам Лемпель и Якоб Зив предложили алгоритм сжатия данных, названный позднее **LZ77**. Этот алгоритм используется в программах архивирования текстов compress, lha, pkzip и arj. Модификация алгоритма LZ78 применяется для сжатия двоичных данных. Эти модификации алгоритма защищены патентами США. Алгоритм предполагает кодирование последовательности бит путем разбивки ее на фразы с последующим кодированием этих фраз.

Суть алгоритма заключается в следующем:

Если в тексте встретится повторение строк символов, то повторные строки заменяются ссылками (указателями) на исходную строку. Ссылка имеет формат <префикс, расстояние, длина>. Префикс в этом случае равен 1. Поле расстояние идентифицирует слово в словаре строк. Если строки в словаре нет, генерируется код символ вида <префикс, символ>, где поле префикс = 0, а поле символ соответствует текущему символу исходного текста. Отсюда видно, что префикс служит для разделения кодов указателя от кодов символ. Введение кодов символ, позволяет оптимизировать словарь и поднять эффективность сжа-

тия. Главная алгоритмическая проблема здесь заключатся в оптимальном выборе строк, так как это предполагает значительный объем переборов.

Рассмотрим пример с исходной последовательностью (см.также http://geeignetra.chat.ru/lempel/lempelziv.htm)

U=0010001101 (без надежды получить реальное сжатие для столь ограниченного объема исходного материала).

Введем обозначения: P[n] - фраза с номером n. C - результат сжатия.

Разложение исходной последовательности бит на фразы представлено в таблице ниже.

№ фразы	Значение	Формула	Исходная последовательность U
0	-	P[0]	0010001101
1	0	P[1]=P[0].0	0. 010001101
2	01	P[2]=P[1].1	0.01.0001101
3	010	P[3]=P[1].0	0. 01.00.01101
4	00	P[4]=P[2].1	0. 01.00.011.01
5	011	P[5]=P[1].1	0. 01.00. 011.01

P[0] - пустая строка. Символом . (точка) обозначается операция объединения (конкатенации).

Формируем пары строк, каждая из которых имеет вид (A.B). Каждая пара образует новую фразу и содержит идентификатор предыдущей фразы и бит, присоединяемый к этой фразе. Объединение всех этих пар представляет окончательный результат сжатия C. P[1]=P[0].0 дает (00.0), P[2]=P[1].0 дает (01.0) и т.д. Схема преобразования отражена в таблице ниже.

Формулы	P[1]=P[0].0	P[2]=P[1].1	P[3]=P[1].0	P[4]=P[2].1	P[5]=P[1].1
Пары	00.0=000	01.1=011	01.0=010	10.1=101	01.1=011
C	000.011.010.101.011 = 000011010101011				

Все формулы, содержащие P[0] вовсе не дают сжатия. Очевидно, что C длиннее U, но это получается для короткой исходной последовательности. В случае материала большего объема будет получено реальное сжатие исходной последовательности.

Локально адаптивный алгоритм сжатия

Этот алгоритм используется для кодирования (L,I), где L строка длиной N, а I - индекс. Это кодирование содержит в себе несколько этапов.

1. Сначала кодируется каждый символ L с использованием локально адаптивного алгоритма для каждого из символов индивидуально. Определяется вектор целых чисел R[0],…,R[N-1], который представляет собой коды для сим-

волов L[0],…,L[N-1]. Инициализируется список символов Y, который содержит в себе каждый символ из алфавита X только один раз. Для каждого i = 0,…,N-1 устанавливается R[i] равным числу символов, предшествующих символу L[i] из списка Y. Взяв Y = [‘a’,’b’,’c’,’r’] в качестве исходного и L = ‘caraab’, вычисляем вектор R: (2 1 3 1 0 3).

2. Применяем алгоритм Хафмана или другой аналогичный алгоритм сжатия к элементам R, рассматривая каждый элемент в качестве объекта для сжатия. В результате получается код OUT и индекс I.

Рассмотрим процедуру декодирования полученного сжатого текста (OUT,I).

Здесь на основе (OUT,I) необходимо вычислить (L,I). Предполагается, что список Y известен.

Сначала вычисляется вектор R, содержащий N чисел: (2 1 3 1 0 3). Далее вычисляется строка L, содержащая N символов, что дает значения R[0],…,R[N-1]. Если необходимо, инициализируется список Y, содержащий символы алфавита X (как и при процедуре кодирования). Для каждого i = 0,…,N-1 последовательно устанавливается значение L[i], равное символу в положении R[i] из списка Y (нумеруется, начиная с 0), затем символ сдвигается к началу Y. Результирующая строка L представляет собой последнюю колонку матрицы M. Результатом работы алгоритма будет (L,I). Взяв Y = [‘a’,’b’,’c’,’r’] вычисляем строку L = ‘caraab’.

Наиболее важным фактором, определяющим скорость сжатия, является время, необходимое для сортировки вращений во входном блоке. Наиболее быстрый способ решения проблемы заключается в сортировке связанных строк по суффиксам.

Для того чтобы сжать строку S, сначала сформируем строку S’, которая является объединением S с EOF, новым символом, который не встречается в S. После этого используется стандартный алгоритм к строке S’. Так как EOF отличается от прочих символов в S, суффиксы S’ сортируются в том же порядке, как и вращения S’. Это может быть сделано путем построения дерева суффиксов, которое может быть затем обойдено в лексикографическом порядке для сортировки суффиксов. Для этой цели может быть использован алгоритм формирования дерева суффиксов Мак-Крейгта. Его быстродействие составляет 40% от наиболее быстрой методики в случае работы с текстами. Алгоритм работы с деревом суффиксов требует более четырех слов на каждый исходный символ. Манбер и Майерс предложили простой алгоритм сортировки суффиксов строки. Этот алгоритм требует только двух слов на каждый входной символ. Алгоритм работает сначала с первыми i символами суффикса a за тем, используя положения суффиксов в сортируемом массиве, производит сортировку для первых 2i символов. К сожалению, этот алгоритм работает заметно медленнее.

В ряде работ предложен несколько лучший алгоритм сортировки суффиксов. В этом алгоритме сортируются суффиксы строки S, которая содержит N символов S[0,…,N-1].

Пусть k число символов, соответствующих машинному слову. Образуем строку S’ из S путем добавления k символов EOF в строку S. Предполагается, что EOF не встречается в строке S.

Инициализируем массив W из N слов W[0,…,N-1] так, что W[i] содержат символы S’[i,…,i+k-1] упорядоченные таким образом, что целочисленное сравнение слов согласуется с лексикографическим сравнением для k-символьных строк. Упаковка символов в слова имеет два преимущества: это позволяет для двух префиксов сравнить сразу k байт и отбросить многие случаи, описанные ниже.

Инициализируется массив V из N целых чисел. Если элемент V содержит j, он представляет собой суффикс S’, чей первый символ равен S’[j]. Когда выполнение алгоритма завершено, суффикс V[i] будет i-ым суффиксом в лексикографическом порядке.

Инициализируем целочисленный массив V так, что для каждого i = 0,…,N-1 : V[i]=i.

Сортируем элементы V, используя первые два символа каждого суффикса в качестве ключа сортировки. Далее для каждого символа ch из алфавита выполняем шаги 6 и 7. Когда эти итерации завершены, V представляет собой отсортированные суффиксы S и работа алгоритма завершается. Для каждого символа ch’ в алфавите выполняем сортировку элементов V, начинающихся с ch, за которым следует ch’. В процессе выполнения сортировки сравниваем элементы V путем сопоставления суффиксов, которые они представляют при индексировании массива W. На каждом шаге рекурсии следует отслеживать число символов, которые оказались равными в группе, чтобы не сравнивать их снова. Все суффиксы, начинающиеся с ch, отсортированы в рамках V. Для каждого элемента V[i], соответствующего суффиксу, начинающемуся с ch (то есть, для которого S[V[i]] = ch), установить W[V[i]] значение с ch в старших битах и i в младших битах. Новое значение W[V[i]] сортируется в те же позиции, что и старые значения.

Данный алгоритм может быть улучшен различными способами. Одним из самоочевидных методов является выбор символа ch на этапе 5, начиная с наименьшего общего символа в S и предшествующий наиболее общему.

Сжатие данных с использованием преобразования Барроуза-Вилера

Майкл Барроуз и Давид Вилер (Burrows-Wheeler) в 1994 году предложили свой алгоритм преобразования (BWT). Этот алгоритм работает с блоками данных и обеспечивает эффективное сжатие без потери информации. В результате преобразования блок данных имеет ту же длину, но другой порядок расположения символов. Алгоритм тем эффективнее, чем больший блок данных преобразуется (например, 256-512 Кбайт).

Последовательность S, содержащая N символов ({S(0),… S(N-1)}), подвергается N циклическим сдвигам (вращениям), лексикографической сортировке, а последний символ при каждом вращении извлекается. Из этих символов формируется строка L, где i-ый символ является последним символом i-го вращения. Кроме строки L создается индекс I исходной строки S в упорядоченном списке вращений. Существует эффективный алгоритм восстановления исход-

ной последовательности символов S на основе строки L и индекса I. Процедура сортировки объединяет результаты вращений с идентичными начальными символами. Предполагается, что символы в S соответствуют алфавиту, содержащему K символов.

Для пояснения работы алгоритма возьмем последовательность S= “abraca” (N=6), алфавит X = {‘a’,’b’,’c’,’r’}.

1. Формируем матрицу из N*N элементов, чьи строки представляют собой результаты циклического сдвига (вращений) исходной последовательности S, отсортированных лексикографически. По крайней мере одна из строк M содержит исходную последовательность S. Пусть I является индексом строки S. В приведенном примере индекс I=1, а матрица M имеет вид:

Номер строки	
0	aabrac
1	abraca
2	acaabr
3	bracaa
4	caabra
5	racaab

2. Пусть строка L представляет собой последнюю колонку матрицы M с символами L[0],…,L[N-1] (соответствуют M[0,N-1],…,M[N-1,N-1]). Формируем строку последних символов вращений. Окончательный результат характеризуется (L,I). В данном примере L=’caraab’, I =1.

Процедура декомпрессии использует L и I. Целью этой процедуры является получение исходной последовательности из N символов (S).

1. Сначала вычисляем первую колонку матрицы M (F). Это делается путем сортировки символов строки L. Каждая колонка исходной матрицы M представляет собой перестановки исходной последовательности S. Таким образом, первая колонка F и L являются перестановками S. Так как строки в M упорядочены, размещение символов в F также упорядочено. F=’aaabcr’.

2. Рассматриваем ряды матрицы M, которые начинаются с заданного символа ch. Строки матрицы M упорядочены лексикографически, поэтому строки, начинающиеся с ch упорядочены аналогичным образом. Определим матрицу M’, которая получается из строк матрицы M путем циклического сдвига на один символ вправо. Для каждого i=0,…, N-1 и каждого j=0,…,N-1,

$$M'[i,j] = m[i,(j-1) \bmod N]$$

В рассмотренном примере M и M’ имеют вид:

Строка	M	M’
0	aabrac	caabra

1	abraca	aabrac
2	acaabr	racaab
3	bracaa	abraca
4	caabra	acaabr
5	racaab	bracaa

Подобно M каждая строка M’ является вращением S, и для каждой строки M существует соответствующая строка M’. M’ получена из M так, что строки M’ упорядочены лексикографически, начиная со второго символа. Таким образом, если мы рассмотрим только те строки M’, которые начинаются с заданного символа ch, они должны следовать упорядоченным образом с учетом второго символа. Следовательно, для любого заданного символа ch, строки M, которые начинаются с ch, появляются в том же порядке что и в M’, начинающиеся с ch. В нашем примере это видно на примере строк, начинающихся с ‘a’. Строки ‘aabrac’, ‘abraca’ и ‘acaabr’ имеют номера 0, 1 и 2 в M и 1, 3, 4 в M’.

Используя F и L, первые колонки M и M’ мы вычислим вектор T, который указывает на соответствие между строками двух матриц, с учетом того, что для каждого j = 0,…,N-1 строки j M’ соответствуют строкам T[j] M.

Если L[j] является к-ым появлением ch в L, тогда T[j]=1, где F[i] является к-ым появлением ch в F. Заметьте, что T представляет соответствие один в один между элементами F и элементами L, а F[T[j]] = L[j]. В нашем примере T равно: (4 0 5 1 2 3).

3. Теперь для каждого i = 0,…, N-1 символы L[i] и F[i] являются соответственно последними и первыми символами строки i матрицы M. Так как каждая строка является вращением S, символ L[i] является циклическим предшественником символа F[i] в S. Из T мы имеем F[T[j]] = L[j]. Подставляя i =T[j], мы получаем символ L[T(j)], который циклически предшествует символу L[j] в S.
Индекс I указывает на строку M, где записана строка S. Таким образом, последний символ S равен L[I]. Мы используем вектор T для получения предшественников каждого символа: для каждого i = 0,…,N-1 S[N-1-i] = L[T^i[I]], где T^0[x] =x, а T^{i+1}[x] = T[T^i[x]. Эта процедура позволяет восстановить первоначальную последовательность символов S (‘abraca’).

Последовательность T^i[I] для i =0,…,N-1 не обязательно является перестановкой чисел 0,…,N-1. Если исходная последовательность S является формой Z^p для некоторой подстановки Z и для некоторого p>1, тогда последовательность T^i[I] для i = 0,…,N-1 будет также формой Z'^p для некоторой субпоследовательности Z’. Таким образом, если S = ‘cancan’, Z = ‘can’ и p=2, последовательность T^i[I] для i = 0,…,N-1 будет [2,4,0,2,4,0].

Описанный выше алгоритм упорядочивает вращения исходной последовательности символов S и формирует строку L, состоящую из последних символов вращений. Для того, чтобы понять, почему такое упорядочение приводит к более эффективному сжатию, рассмотрим воздействие на отдельную букву в обычном слове английского текста.

Возьмем в качестве примера букву “t” в слове ‘the’ и предположим, что исходная последовательность содержит много таких слов. Когда список вращений упорядочен, все вращения, начинающиеся с ‘he’, будут взаимно упорядочены. Один отрезок строки L будет содержать непропорционально большое число ‘t’, перемешанных с другими символами, которые могут предшествовать ‘he’, такими как пробел, ‘s’, ‘T’ и ‘S’.

Аналогичные аргументы могут быть использованы для всех символов всех слов, таким образом, любая область строки L будет содержать большое число некоторых символов. В результате вероятность того, что символ ‘ch’ встретится в данной точке L, весьма велика, если ch встречается вблизи этой точки L, и мала в противоположном случае. Это свойство способствует эффективной работе локально адаптивных алгоритмов сжатия, где кодируется относительное положение идентичных символов. В случае применения к строке L, такой кодировщик будет выдавать малые числа, которые могут способствовать эффективной работе последующего кодирования, например, посредством алгоритма Хафмана.

Алгоритма Хафмана.

Данный метод выделяется своей простотой. Берутся исходные сообщения **m(i)** и их вероятности появления **P(m(i)).** Сообщения упорядываются так, чтобы вероятность i-го сообщения была не больше (i+1)-го. Этот список делится на две группы с примерно равной интегральной вероятностью. Каждому сообщению из группы 1 присваивается 0 в качестве первой цифры кода. Сообщениям из второй группы ставятся в соответствие коды, начинающиеся с 1. Каждая из этих групп делится на две аналогичным образом и добавляется еще одна цифра кода. Процесс продолжается до тех пор, пока не будут получены группы, содержащие лишь одно сообщение. Каждому сообщению в результате будет присвоен код **x** с длиной **–lg(P(x)).** Это справедливо, если возможно деление на подгруппы с совершенно равной суммарной вероятностью. Если же это невозможно, некоторые коды будут иметь длину –lg(P(x))+1. Алгоритм Шеннона-Фано не гарантирует оптимального кодирования.

Смотри http://www.ics.uci.edu/~dan/pubs/DC-Sec3.html.

Раздел 2. Элементы теории чисел

Тема 5. Введение в теорию чисел

Ярмолик В.Н., Портянко С.С., Ярмолик С.В. Криптография, стеганография и охрана авторского права. – Минск: Издательский центр БГУ, 2007. – 242с. (*Глава 3.*)

Все множество **натуральных чисел** состоит их двух подмножеств: **действительных чисел** и **целых чисел.** Наиболее часто используемым подмножеством в криптографии и различных приложениях по защите информации является подмножество целых чисел. В свою очередь целые числа делятся на **простые** и **составные** числа.

Целое число d является делителем n тогда и только тогда, когда это число может быть представлено в виде произведения, то есть $n=kd$, и обозначается как $d \mid n$.

Целое число p, $p>1$ является **простым**, если его делителями являются только 1 и p. Последовательность простых чисел начинается с чисел 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113,….

Натуральные целые числа, имеющие больше двух делителей, называются **составными**. Таким образом, все натуральные целые числа делятся на простые и составные числа.

Любое целое $n>1$ может быть представлено единственным образом как произведение простых чисел в соответствующих степенях. Представление натурального целого числа в виде произведения простых чисел называется **каноническим разложением Евклида**, а процедура получения такого разложения – **разложением на простые сомножители** или **факторизацией** числа.

На настоящий момент не известны полиномиальные алгоритмы факторизации чисел, хотя и не доказано, что таких алгоритмов не существует. На этом факте базируется большинство из существующих криптосистем.

Весьма значимым с точки зрения криптографических приложений является факт того, что не известен эффективный алгоритм разложения чисел на множители, кроме того, не было получено никакой конструктивной нижней оценки временной сложности такого разложения. Более того, не известно никаких эффективных методов даже в таком простом случае, когда необходимо восстановить два простых числа p и q на основании их произведения $n=pq$. Единственной альтернативой является последовательное деление числа n на все простые числа. Поэтому получение простых чисел и в особенности больших простых чисел является одной из первоочередных задач криптографии.

Для оценки этой проблемы приведем несколько тривиальных теорем.

Теорема 5.1. (**Евклида**) Существует бесконечное множество простых чисел.

Доказательство: Предположим, что это множество конечно и состоит из простых чисел $p1, p2, p3,\ldots,pk$, тогда получим противоречие, заключающееся в том, что число

$$\left(\prod_{i=1}^{k} p_i\right)+1,$$

не делится ни на одно простое число $p1, p2, p3,\ldots,pk$, тогда как оно делится на 1 и на самого себя, а это значит, что это число простое. #

Важным выводом приведенной теоремы является тот факт, что простых чисел существует бесконечное множество.

Теорема 5.2. Для сколь угодно большого положительного целого числа $k>1$, на числовой оси существует k последовательно идущих друг за другом составных чисел.

Доказательство: Число $(k+1)!=2\times3\times4\times\ldots\times(k+1)$ делится на любое из следующих чисел $2,3,4,\ldots,(k+1)$. Тогда числа, следующие последовательно в числовом ряду целых чисел $(k+1)!+2$, $(k+1)!+3$, $(k+1)!+4,\ldots$, $(k+1)!+(k+1)$, являются составными числами вследствие того факта, что первое число делится, по крайней мере, на 2, второе на 3 и т. д. #

Приведенная теорема свидетельствует о сложности нахождения простых чисел, так как их удельный вес по сравнению с составными числами является несравненно меньшим.

Реальный подсчет простых чисел в каждой сотне целых чисел для интервала от 1 до 1000, то есть в первой сотне от 1 до 100, во второй сотне от 101 до 200 и так далее равен: 25, 21, 16, 16, 17, 14, 16, 14, 15, 14. Аналогичный подсчет для интервала от 1 000 001 до 1 001 000 равен: 6, 10, 8, 8, 7, 7, 10, 5, 6, 8. Еще меньше простых чисел в интервале от 10 000 001 до 10 001 000, а именно: 2, 6, 6, 6, 5, 4, 7, 10, 9, 6.

Количество целых чисел для произвольного интервала может быть оценено с помощью следующей теоремы.

Теорема 5.3. Отношение количества простых чисел $\pi(x)$ находящихся в интервале от 2 до x к величине равной $x/ln(x)$ стремиться к единице при x стремящемся к бесконечности, то есть

$$\lim_{x\to\infty}\frac{\pi(x)}{x/\ln(x)}=1,$$

где $ln(x)$ есть натуральный логарифм от x.

Количественно приведенная теорема иллюстрируется таблицей.

x	$\pi(x)$	$x/ln(x)$	$\pi(x)/(x/ln(x))$
1 000	168	145	1,159
10 000	1 229	1 086	1,132
100 000	9 592	8 686	1,104
1 000 000	78 498	72 382	1,084
10 000 000	664 579	620 421	1,071
100 000 000	5 761 455	5 428 681	1,061
1 000 000 000	50 847 476	48 254 942	1,054

Один из первых алгоритмов, позволяющих уменьшить сложность определения простоты целых чисел, был предложен еще во времена Евклида и базируется на использовании следующей теоремы.

Теорема 5.4. Если целое число $n>1$ не делится ни на одно из простых чисел не большее чем $(n)^{1/2}$, то это число есть простое число.

Доказательство: Пусть n есть составное число, и соответственно может быть выражено как произведение двух сомножителей $n=ab$, где $1<a<n$; $1<b<n$. Числа a и b не могут быть больше, чем $(n)^{1/2}$ одновременно. #

Алгоритм получения простых чисел описывается следующей теоремой.

Теорема 5.5. (**Эратосфена**)

1) Если в наборе целых чисел 2,3,4,..,N удалить все числа, которые делятся на первые r простых чисел 2,3,5,7,...,p_r, тогда первое (наименьшее) не удаленное число будет простым.

2) Если в наборе целых чисел 2,3,4,..,N удалить все числа, которые делятся на простые числа меньшие или равные $(N)^{1/2}$, тогда все оставшиеся числа будут простыми числами p, принадлежащими интервалу $(N)^{1/2}<p\le N$.

Доказательство:

1) Любое составное число n делится, по крайней мере, на одно простое число меньшее, чем n.

2) Каждое составное число n, такое что $(N)^{1/2}< n\le N$ делится, по крайней мере, на одно простое $p_i\le(n)^{1/2}\le(N)^{1/2}$, то есть на одно из чисел 2, 3,…, p_r ($p_r<(N)^{1/2}\le p_{r+1}$) и, следовательно, будет вычеркнуто. #

Пример 5.1. В качестве примера рассмотрим пятьдесят первых чисел числового ряда, то есть

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49											

В результате удаления всех чисел кратных 2,3,5 и 7 получим множество простых чисел: 11,13,17,19,23,29,31,37,41,43,47, которое удовлетворяет вышеприведенной теореме.

Попытки определить формальным образом простые числа приводили к отрицательному результату.

Так простые числа **Эйлера** (**Euler's**

) могут быть сгенерированы в соответствии с формулой x^2-x+41, для целых чисел x принадлежащих интервалу, $0<x<40$. Очевидно, что для больших значений x данная формула неприменима.

Простые числа **Ферма** (**Fermat's**): 3,5,17,257,65537 генерируются в соответствии с формулой 2^K+1, где $K=2^k$, для целых значений k.

Простые числа **Мерсенна** (**Mersenne's**) могут быть сгенерированы по формуле 2^n-1, для простых чисел n=2,3,5,7,13,17,19,31,61. Интерес к числам Мерсенна не ослабевает.

Наибольшее известное на данный момент простое число $M=2^{25964951}-1$ содержит 7816230 десятичных цифр. Это 42-е известное простое число Мер-

сенна было найдено 18 февраля 2005 года в проекте по распределённому поиску простых чисел Мерсенна GIMPS.

Предыдущее по величине известное простое число $M=2^{24036583}-1$ (из 7235733 десятичных цифр). Это 41-е простое число Мерсенна, также было найдено GIMPS 15 мая 2004.

Числа Мерсенна выгодно отличаются от остальных наличием эффективного **теста простоты**, носящего имя **Люка-Лемера**. Благодаря этому тесту простые числа Мерсенна давно удерживают рекорд как самые большие известные простые числа. За нахождение простого числа, состоящего из более чем 10^7 десятичных цифр, назначена награда в 100000 долларов США.

Составные числа могут быть представлены в канонической форме

$$a = \prod_{i=1}^{k} p_i^{\alpha_i},$$

где p_i – простые числа, α_i – целые числа, a – составное число.

Пример 5.2. $a=120=2^3\times3^1\times5^1$.

Общим делителем чисел $a_1,a_2,a_3,\ldots,a_n$ является целое число d, такое, что $d \mid a_1$, $d \mid a_2$, $d \mid a_3,\ldots, d \mid a_n$.

Наибольший общий делитель чисел $a_1,a_2,a_3,\ldots,a_n$ это наибольший целый делитель d, который может быть поделен любым общим делителем этих чисел $d=(a_1,a_2,a_3,\ldots,a_n)$.

Пример 5.3. (6,15,27)=3.

Теорема 5.6. Если

$$a_n = \prod_{i=1}^{k} p_i^{\gamma_i},\ a_2 = \prod_{i=1}^{k} p_i^{\beta_i} \quad \ldots a_1 = \prod_{i=1}^{k} p_i^{\alpha_i}$$

тогда наибольший общий делитель (НОД) будет равен:

$$(a_1,a_2,\ldots,a_n) = \prod_{i=1}^{k} p_i^{\min(\alpha_i,\beta_i,\ldots,\gamma_i)}.$$

Пример 5.4. Если $6=2^1\times3^1$, $15=3^1\times5^1$, $27=3^3$, тогда НОД(6,15,27)= $=2^{min\{1,0,0\}}\times3^{min\{1,1,3\}}\times5^{min\{0,1,0\}}=2^0\times3^1\times5^0=3$.

Теорема 5.7. Если

$$a_n = \prod_{i=1}^{k} p_i^{\gamma_i},\ a_2 = \prod_{i=1}^{k} p_i^{\beta_i} \quad \ldots a_1 = \prod_{i=1}^{k} p_i^{\alpha_i}$$

тогда наименьшее общее кратное (НОК) будет равно:

$$НОК(a_1,a_2,\ldots,a_n) = \prod_{i=1}^{k} p_i^{\max(\alpha_i,\beta_i,\ldots,\gamma_i)}.$$

Пример 5.5. Если $6=2^1\times3^1$, $15=3^1\times5^1$, $27=3^3$, тогда НОК(6,15,27)= $=2^{max\{1,0,0\}}\times3^{max\{1,1,3\}}\times5^{max\{0,1,0\}}=2^1\times3^3\times5^1=270$.

Алгоритм Евклида

Проблема нахождения наибольшего общего делителя является одной из часто используемых процедур в криптографии, которая в частности позволяет определить взаимную простоту целых чисел. Одним из исторически первых инструментов определения взаимной простоты является хорошо известный алгоритм Евклида. Методологической основой указанного алгоритма является следующая теорема.

Теорема 5.8. Если $a=bq+r$, тогда наибольший общий делитель чисел a,b равен наибольшему общему делителю чисел b,r, то есть $(a,b)=(b,r)$.

Доказательство:

Пусть $d=(a,b)$, тогда из утверждения $d \mid a$ и $d \mid b$ можно сделать вывод, что d является делителем bq, а также делителем разности чисел a и bq. Таким образом d является делителем $a-bq=r$.#

Теорема 5.9. **(Алгоритм Евклида)** Для любых целых чисел $a>0$ и $b>0$ таких, что $a>b$, и b не является делителем a, для некоторого s, существуют целые числа $q_0,q_1,q_2,\ldots,q_s$ и $r_1,r_2,\ldots,r_s$ такие, что $b>r_1>r_2>\ldots>r_s>0$ и $a=bq_0+r_1$, $b=r_1q_1+r_2$, $r_1=r_2q_2+r_3,\ldots$, $r_{s-2}=r_{s-1}q_{s-1}+r_s$, $r_{s-1}=r_sq_s$, и соответственно $(a,b)=r_s$

Доказательство последней теоремы следует из последовательного применения теоремы 3.8 для пар чисел (a,b), (b,r_1), $(r_1,r_2),\ldots,(r_{s-1},r_s)$, для которых выполняется равенство $(a,b)=(b,r_1)=(r_1,r_2)=\ldots=(r_{s-1},r_s)$.

В виде вычислительной процедуры алгоритм Евклида можно представить следующим кодом.

```
Алгоритм Евклида
begin
        g0:=a;
        g1:=b;
        while  gi≠0 do
        begin
                gi+1:= gi-1 mod gi;
                i:=i+1;
        end
        gcd:= gi-1              {gcd – Greatest Common Divisor (НОД)}
end
```

Пример 5.9. Определим НОД для целых чисел 1173 и 323, (1173,323) =*?*. Так как 1173=323·3+204, то НОД для целых чисел 1173 и 323 равен НОДу для чисел 323 и 204. Повторяя подобное разложение чисел: 323=204×1+119; 204=119×1+85; 119=85×1+34; 85=34×2+17; 34=17×2 окончательно получим, что (1173,323) =17.

В соответствии с приведенным выше алгоритмом Евклида получим:

***gi*+1**	**:=**	***gi*-1 *mod* *gi*;**
204	:=	1173 *mod* 323
119	:=	323 *mod* 204

85	:=	204 *mod* 119
34	:=	119 *mod* 85
17	:=	85 *mod* 34
0	:=	34 *mod* 17

Одним из существенных недостатков алгоритма Евклида является применение операции деления для его реализации, что заметно увеличивает его вычислительную сложность. Развитием алгоритма Евклида является **бинарный алгоритм.** Этот алгоритм является расширением алгоритма Евклида и основан на следующих очевидных утверждениях:

1. Если и a и b чётные числа, тогда $(a,b)=2(a/2,b/2)$;
2. Если a чётное, а b нечётное число, то $(a,b)=(a/2,b)$;
3. В соответствии с теоремой 3.8 $(a,b)=(b,a-b)$;
4. Если и a и b нечётные числа, то $a-b$ является чётным числом.

Пример 5.10. Определим НОД для целых чисел 1173 и 323, (1173,323) =?. Последовательно применяя утверждения, приведенные выше, получим следующие равенства (1173,323)=(323,850)=(323,425)=(323,102)=(323,51)=(51, 272)=(51,136)=(51,68)=(51,34)=(51,17)=(17,34)=(17,17)=17. Отметим, что в данном примере от проблемы определения НОД для чисел (1173,323) перешли к такой же проблеме для чисел (323,850) на основании утверждения 3. А последующий переход выполнен на основании утверждения 2 и так далее.

Числа $a_1,a_2,a_3,\ldots,a_n$ называются **взаимно простыми** тогда и только тогда, когда $(a_1,a_2,a_3,\ldots,a_n)=1$.

Числа $a_1,a_2,a_3,\ldots,a_n$ называются **попарно взаимно простыми** тогда и только тогда, когда для любых i и $j\neq i$ $(a_i,a_j)=1$.

Теорема 5.10. Если $(a,b)=1$, тогда для любых целых чисел n и m $(a^n,b^m)=1$. Справедливо также и обратное утверждение: если $(a^n,b^m)=1$, для любых целых чисел n и m, то выполняется равенство $(a,b)=1$.

Доказательство: Предположим, что выполняется равенство $(a,b)=1$. Тогда, если в каноническом представлении числа $a=p_1^{\alpha 1},p_2^{\alpha 2},\ldots,p_k^{\alpha k}$ для некоторого показателя степени αi выполняется неравенство $\alpha i>0$, то $\gamma i=0$ для канонического представления числа $b=p_1^{\gamma 1},p_2^{\gamma 2},\ldots,p_k^{\gamma k}$. Это следует из равенства $(a,b)=1$. Аналогично в случае, когда $n\alpha i>0$ получим $\gamma i=0$. #

Сравнения

Два целых числа a и b **сравнимы** (**конгруэнтны**) **по модулю** натурального числа m, если при делении на m они дают одинаковые остатки. Другими словами, a и b **сравнимы по модулю** m, если их разность $a - b$ делится на m. Например, 32 и 39 сравнимы по модулю 7, так как 32 = 7×4 + 4, 39 = 7×5 + 4. Утверждение «a и b сравнимы по модулю m» записывается в виде:

$$a \equiv b \ \textbf{mod}\ m.$$

Последнее выражение называется **сравнением** (**конгруэнцией)**.

Пример 5.11. Следующие равенства являются сравнениями $32 \equiv 5 \ mod\ 9$; $48 \equiv 12 \ mod\ 9$; $17 \equiv 7 \ mod\ 5$.

Числа a и b **несравнимы по модулю** m, если их разность $a - b$ не делится на m. Этот факт обозначается неравенством $a \not\equiv b \ mod\ m$. Очень часто для представления сравнения $a \equiv b \ mod\ m$ используют символ равенства (=), тогда $a = b \ mod\ m$ означает, что a и b сравнимы по модулю m, а $a \neq b \ mod\ m$, что числа не сравнимы по модулю m.

В случае, когда в сравнении $a = b \ mod\ m$ величина $b < m$, b называется **вычетом** a по модулю m.

Для сравнений существует несколько полезных лемм.

Лемма 5.1. Если $a \equiv b \ mod\ m$, тогда для любого целого k будет справедливо $ka \equiv kb \ mod\ m$.

Лемма 5.2. Если $ka \equiv kb \ mod\ m$, и $(k,m)=1$, то $a \equiv b \ mod\ m$.

Лемма 5.3. Если $ka \equiv kb \ mod\ km$, где k и m любые целые числа, то $a \equiv b \ mod\ m$.

Лемма 5.4. Если $a \equiv b \ mod\ m$, и $c \equiv d \ mod\ m$, то $a+c \equiv b+d \ mod\ m$.

***Лемма* 5.5.** Если $a_1 \equiv b_1 \ mod\ m$, и $a_2 \equiv b_2 \ mod\ m, \ldots, a_n \equiv b_n \ mod\ m$, то $a_1+a_2+a_3+\ldots+a_n \equiv b_1+b_2+b_3+\ldots+b_n \ mod\ m$.

Лемма 5.6. Если $a \equiv b \ mod\ m$, и $c \equiv d \ mod\ m$, то $a \cdot c \equiv b \cdot d \ mod\ m$.

Лемма 5.7. Если $a_1 \equiv b_1 \ mod\ m$, и $a_2 \equiv b_2 \ mod\ m, \ldots, a_n \equiv b_n \ mod\ m$, то $a_1 \cdot a_2 \cdot a_3 \cdot \ldots \cdot a_n \equiv b_1 \cdot b_2 \cdot b_3 \cdot \ldots \cdot b_n \ mod\ m$.

Лемма 5.8. Если $a \equiv b \ mod\ m$, то для любого целого $k>0$ будет справедливо $a^k \equiv b^k \ mod\ m$.

Теория сравнений эффективно используется для выполнения различных вычислений при реализации криптографических алгоритмов. Основой для реализации вычислительных процедур является так называемая **модулярная арифметика**.

Модульная арифметика основана на следующей классической лемме, используемой во многих технических приложениях.

Лемма 5.9. $(a * b) \ mod\ m = [(a \ mod\ m) * (b \ mod\ m)] \ mod\ m$, где $*$ это любая из следующих операций "+", "–" или "×".

Приведенная лемма показывает, что вычисление $(a*b) \ mod\ m$ в модульной арифметике даёт тот же результат, что и в случае обычной целочисленной арифметики, однако существенно упрощает вычисление результата $(a*b) \ mod\ m$.

Пример 5.12. $7 \cdot 9 \ mod\ 5 = [(7 \ mod\ 5) \cdot (9 \ mod\ 5)] \ mod\ 5 = 3$.

Заметим, что принцип модульной арифметики также применяется и для возведения в степень, так как операция возведения в степень равносильна повторяющемуся выполнению операции умножения.

Пример 5.13. Рассмотрим выражение $3^5 \ mod\ 7$. Возможны три очевидных алгоритма получения искомого результата. Этот результат может быть вычислен путём возведения 3 в степень 5, и затем получения результата по *mod* 7 или в соответствии с двумя алгоритмами приведенными ниже.

В первом случае число 3 последовательно умножается на предыдущий результат умножения до тех пор, пока не будет получено значение 3^5. Отметим,

что для данного примера необходимо выполнить 4 операции умножения, а для общего случая $a^z \ mod \ m$ необходимо выполнить z-1 операцию умножения. Далее необходимо выполнить операцию деления для определения искомого значения $3^5 \ mod \ 7$. Отметим неоправданно высокую вычислительную сложность такого алгоритма и, в первую очередь, в части операции возведения в степень. Значительно меньшую вычислительную сложность можно достичь, используя быстрые алгоритмы возведения в степень.

Во втором случае, при использовании быстрых алгоритмов возведения в степень, искомый результат для $3^5 \ mod \ 7$ может быть получен в следующей последовательности.

1. Величина 3 возводится в квадрат:	$3^2=3\times3=9.$
2. Используя значение 3^2, получаем:	$3^4=3^2\times3^2=9\times9=81.$
3. Вычисляется 3^5 как:	$3^5=3^4\times3=81\times3=243.$
4. Выполняя операцию деления, получим:	$3^5 \ mod \ 7=243 \ mod \ 7=5.$

В данном случае, для нашего примера $3^5 \ mod \ 7$, количество операций умножения равняется 3, а для общего случая ($a^z \ mod \ m$) необходимо не более чем $2(\lceil \boldsymbol{log}_2 z \rceil - 1)$ операций умножения. Очевидным недостатком второго алгоритма является большая размерность операндов, которые могут быть существенно больше чем величина m.

Третий алгоритм основан на использовании основополагающей леммы 3.9. Все промежуточные вычисления будут выполняться по модулю 7 или для общего случая ($a^z \ mod \ m$) по модулю m, в следующей последовательности.

1. Величина 3 возводится в квадрат по модулю 7, в результате получим $3^2 \ mod \ 7=3\cdot3 \ mod \ 7=2$.

2. Для получения $3^4 \ mod \ 7$ используем предыдущий результат, тогда, в соответствии с леммой 3.9, получим $3^4 \ mod \ 7=[(3^2 mod \ 7)\times(3^2 mod \ 7)] \ mod \ 7= =2\times2 \ mod \ 7=4$.

3. Для получения окончательного результата, выполним следующие вычисления $3^5 mod \ 7=[(3^4 mod \ 7)\times(3 \ mod \ 7)] \ mod \ 7= 4\times3 \ mod \ 7=5$.

Для общего случая ($a^z \ mod \ m$) последний алгоритм можно представить в виде следующей вычислительной процедуры.

Алгоритм быстрого возведения в степень

```
begin "возвращаем x=a^z mod m"
    a1:=a; z1:=z;
    x:=1;
    while z1≠0 do "x(a1^z1 mod m)=a^z mod m"
    begin
        while z1 mod 2=0 do
        begin "возводим в квадрат a1 пока z1 чётно"
            z1:= z1 div 2;
            a1:= (a1×a1) mod m;
        end;
```

```
            z1:= z1–1;
            x:=(x×a1) mod m; "умножение"
        end;
        fastexp:=x;
end
```

Пример 5.14. Рассмотрим вычисление $x=5^{10}\ mod\ 7=5^{(1010)}\ mod\ 7$ с использованием алгоритма быстрого возведения в степень. Здесь $10_{10}=1010_2$. В последующей диаграмме приведена последовательность вычислений в соответствии с указанным алгоритмом.

a_1:=5; z_1:=10; x:=1;
- $z_1 \neq 0$; (10≠0);
  - $z_1\ mod\ 2=0$; (10 *mod* 2=0);
    - $z_1\ div\ 2=5$; (10/2=5);
    - $a_1:= a_1 \cdot a_1\ mod\ m=4$; (5·5 *mod* 7=4);
  - $z_1\ mod\ 2 \neq 0$; (5 *mod* 2≠0);
- $z_1:= z_1-1=4$; (5-1=4);
- $x:= (x \cdot a_1)\ mod\ m\ =4$; (1·4 *mod* 7=4);
- $z_1 \neq 0$; (4≠0);
  - $z_1\ mod\ 2=0$; (4 *mod* 2=0);
    - $z_1\ div\ 2=2$; (4/2=2);
    - $a_1:= a_1 \cdot a_1\ mod\ m=2$; (4·4 *mod* 7=2);
  - $z_1\ mod\ 2=0$; (2 *mod* 2=0);
    - $z_1\ div\ 2=1$; (2/2=1);
    - $a_1:= a_1 \cdot a_1\ mod\ m=4$; (2·2 *mod* 7=4);
  - $z_1\ mod\ 2 \neq 0$; (1 *mod* 2≠0);
- $z_1:= z_1-1=0$; (1-1=0);
- $x:= (x \cdot a1)\ mod\ m\ =2$; (4·4 *mod* 7=2).

$z_1=0$; (0=0);

Таким образом, результирующее значение $x=5^{10}$ ***mod*** 7 равняется 2.

Теорема Эйлера

С понятием сравнения тесно связана величина **вычета** r ($0<r<m$) числа a по модулю m. **Вычетом** числа a по модулю m называется остаток от деления величины a на m. Согласно приведенному определению, если $a=r\ mod\ m$, ($0<r<m$), то $m \mid (a-r)$. Отсюда, справедливо равенство $a-r=qm$ или $a=qm+r$.

Если набор из m целых чисел $\{a_i\}=\{a_1,a_2,\ldots,a_m\}$ формирует полный набор чисел, сравнимых по **модулю** m с каждым значением r_i в системе вычетов $\{0,1,2,\ldots,m-1\}$, то этот набор $\{a_i\}$ называется **полной системой вычетов по модулю *m*.** Так, для любого целого a_i, выполняется сравнение $a_i=r_j\ mod\ m$ где r_j является уникальным значением неповторяющимся для остальных чисел множества $\{a_i\}$.

Пример 5.15. Набор целых чисел {16,12,19,48,65} является полной системой вычетов по модулю 5. Действительно, 16=1 *mod* 5, 12=2 *mod* 5, 19=4 *mod* 5, 48=3 *mod* 5, 65=0 *mod* 5 и мы получаем полный набор вычетов {0,1,2,3,4}.

Множество целых чисел $\{a_i\}=\{a_1,a_2,...,a_n\}$ формирует ***класс вычета по модулю m*,** если все вычеты для данного множества одинаковы и равны *r*.

Пример 5.16. Набор целых чисел {16,21,56,91,106} является классом вычета по модулю 5. Действительно, 16=1 *mod* 5, 21=1 *mod* 5, 56=1 *mod* 5, 91=1 *mod* 5, 106=1 *mod* 5 и мы имеем одинаковый вычет 1 для всех чисел исходного множества.

Большое практическое значение в криптографии имеет известная теорема Ферма.

Теорема 5.11. (***Fermat*).** Если *p* простое число и (*a*,*p*)=1, где *a* целое, тогда

$$a^{p-1}=1 \ \mathbf{mod}\ p.$$

Доказательство: Пусть для заданных взаимно простых величин *p* и *a*, (*a*,*p*)=1 существует *p*-1 положительных произведений *a*,2*a*,3*a*,...,(*p*-1)*a*. Тогда любая пара *ia*, *ja* (*i*≠*j*) этих произведений не сравнима по модулю *p*, то есть

$$ia \not\equiv ja \bmod p,$$

что следует из леммы 3.2 для сравнений, приведенной ранее. Следовательно, каждое произведение имеет свой уникальный ненулевой вычет r_i , ($1 \le r_i \le p$-1). Эти вычеты {1,2,3,...,(*p*-1)} для множества чисел {*a*,2*a*,3*a*,...,(*p*-1)*a*} расположены в произвольном порядке и формируют полную систему вычетов $\{r_\alpha,r_\beta,r_\chi,...,r_\lambda\}=\{1,2,3,...,(p-1)\}$, где $a=r_\alpha \bmod p$, $2a=r_\beta \bmod p$, $3a=r_\chi \bmod p$,...,$(p-1)a=r_\lambda \bmod p$.

На основе леммы 3.7 для сравнений $a=r_\alpha \bmod p$, $2a=r_\beta \bmod p$, $3a=r_\chi \bmod p$,...,$(p-1)a=r_\lambda \bmod p$ получим $a\times 2a\times 3a\times ...\times (p-1)\times a=r_\alpha\times r_\beta\times r_\chi\times ...\times r_\lambda \bmod p$. Учитывая, что $\{r_\alpha,r_\beta,r_\chi,...,r_\lambda\}=\{1,2,3,...,(p-1)\}$ и выполнив перестановку, будем иметь $a\times 2a\times 3a\times ...\times (p-1)\times a=1\times 2\times 3\times ...\times (p-1) \bmod p$. Окончательно $a^{p-1}\times(p-1)!=(p-1)! \bmod p$. Поскольку (*p*-1)*!* и *p* взаимно простые целые числа, то есть ((*p*-1)!,*p*)=1, тогда окончательно получаем $a^{p-1}=1 \ \mathbf{mod}\ p$.#

Отметим, что теорема Ферма справедлива только для простых чисел *p*. Обобщением данной теоремы является теорема Эйлера, которая основана на использовании функции Эйлера. Определим данную функцию для целого числа *n*.

Функцией Эйлера (Euler's) $\psi(n)$ целого числа $n\ge 1$ является количество целых чисел, которые меньше чем *n* и взаимно просты с *n*. Для небольших значений *n* эта функция принимает следующие значения: $\psi(1)=0$, $\psi(2)=1$, $\psi(3)=2$, $\psi(4)=2$, $\psi(5)=4$, $\psi(6)=2$, $\psi(7)=6$, $\psi(8)=4$, $\psi(9)=6$, $\psi(10)=4$, $\psi(11)=10$,.... Не сложно показать, что согласно приведенному определению, если *n* есть простое число, то $\psi(n)=n-1$.

Для функции Эйлера справедлив ряд утверждений и теорем. Одной и наиболее часто используемых в криптографии теорем относительно функции Эйлера является следующая теорема.

Теорема 5.12. Если $n=pq$, где p и q ($p\neq q$) простые числа, то $\psi(n)=\psi(p)\psi(q)=(p-1)(q-1)$.

Доказательство: Имеем множество $\{0,1,2,\ldots,pq-1\}$ из pq целых чисел, которые меньше чем $n=pq$. Все эти числа взаимно простые с $n=pq$ за исключением $(p-1)$ чисел $\{q,2q,3q,\ldots,(p-1)q\}$, кратных q, и $(q-1)$ чисел $\{p,2p,3p,\ldots,(q-1)p\}$, кратных p и 0. Тогда, $\psi(pq)=pq-(p-1)-(q-1)-1=pq-p-q+1=(p-1)(q-1)$.#

Пример 5.17. $\psi(10)=\psi(2\times5)=\psi(2)\times\psi(5)=1\times4=4$.

Теорема 5.13. Если p простое число, и $k>0$ целое число, то $\psi(p^k)=p^k-p^{k-1}=p^{k-1}(p-1)$.

Доказательство: Множество целых чисел, которые меньше чем p^k и не взаимно простые с p^k, включает числа $\{p,2p,3p,\ldots,(p^{k-1}-1)p\}$. Это значит, что среди p^k-1 чисел, меньших, чем p^k, исключая ноль, есть $p^{k-1}-1$ целых, не взаимно простых с p^k. Тогда, $\psi(p)=p^k-1-(p^{k-1}-1)=p^k-p^{k-1}$.#

Пример 5.18. $\psi(8)=\psi(2^3)=2^3-2^2=8-4=4$.

Теорема 5.14. Функция Эйлера является мультипликативной для произведения целых чисел, если эти числа являются взаимно простыми, то есть $\psi(n\times m)=\psi(n)\times\psi(m)$, если $(n,m)=1$.

Данная теорема позволяет сформулировать алгоритм вычисления функции Эйлера для произвольного целого числа a.

Первоначально представим число в виде канонического разложения Евклида $a=p_1^{\alpha1}\ p_2^{\alpha2}\ldots p_r^{\alpha r}$, тогда в силу того, что $p_i^{\alpha i}$ и $p_j^{\alpha j}$ являются взаимно простыми для любых $i\neq j$ используя предыдущую теорему получим $\psi(a)=\psi(p_1^{\alpha1})\psi(p_2^{\alpha2})\ldots\psi(p_r^{\alpha r})=(p_1^{\alpha1}-p_1^{\alpha1-1})(p_2^{\alpha2}-p_2^{\alpha2-1})\ldots(p_r^{\alpha r}-p_r^{\alpha r-1})=a(1-1/p_1)(1-1/p_2)\ldots(1-1/p_r)$.

Пример 5.19. $\psi(2700)=?$ $270=2^2 3^3 5^2$. $\psi(2700)=2700(1-1/2)(1-1/3)(1-1/5)=720$.

Теорема 5.15. (***Euler's***). Если $n\geq0$ положительное целое число, и $(a,n)=1$, где a – целое, то

$$a^{\psi(n)}=1\ \textbf{\textit{mod}}\ n.$$

Доказательство: Пусть $\{r_1,r_2,r_3,\ldots,r_{\psi(n)}\}$ представляет собой приведенную систему вычетов по модулю n, тогда для $(a,n)=1$ числа ar_1, ar_2, $ar_3,\ldots$, $ar_{\psi(n)}$ образуют ту же приведенную систему вычетов. То есть, $ar_1=r_\alpha\ mod\ n$, $ar_2=r_\beta\ mod\ n$, $ar_3=r_\chi\ mod\ n,\ldots,ar_{\psi(n)}=r_\lambda\ mod\ p$, где $\{r_\alpha,r_\beta,r_\chi,\ldots,r_\lambda\}$ это есть те же значения вычетов $\{r_1\ ,r_2\ ,r_3,\ldots,r_{\psi(n)}\}$, переставленные в ином порядке. Используя аналогичный подход, который применялся при доказательстве теоремы 3.11, перемножив правую и левую части сравнений $ar_1=r_\alpha\ mod\ n$, $ar_2=r_\beta\ mod\ n$, $ar_3=r_\chi\ mod\ n$, $\ldots,ar_{\psi(n)}=r_\lambda\ mod\ p$, получим: $a^{\psi(n)}r_1\ r_2\ r_3\ldots r_{\psi(n)}=r_\alpha\ r_\beta\ r_\chi\ldots r_{\psi(n)}\ mod\ n$. Учитывая, что $\{r_1,r_2,r_3,\ldots,r_{\psi(n)},m\}=1$, окончательно получим $a^{\psi(n)}=1\ \textbf{\textit{mod}}\ n$.#

Пример 5.20. $3^{10}\ mod\ 11=?$. Согласно теореме Ферма $3^{10}=1\ mod\ 11$, где $p=11$ есть простое число, и $a=3$ есть целое взаимно простое с $p=11$, то есть $(11,3)=1$.

Пример 5.21. 3^{12} *mod* 26=?. Согласно теореме Эйлера 3^{12} *mod* 26=1, где n=26, $\psi(26)=\psi(2\times13)=\psi(2)\times\psi(13)=1\times12=12$.

Линейные сравнения

Под линейным сравнением (линейной конгруэнцией) понимают выражение вида

$$ax=b \ \textbf{mod}\ n,$$

где a, b и n есть целые числа, и $b<n$, а x ($x<n$) – неизвестное целое число, удовлетворяющее линейному сравнению.

Существует три возможных случая для линейного сравнения относительно его решения x. Линейное сравнение может: 1) не иметь решений; 2) иметь одно решение; 3) иметь множество решений, удовлетворяющих этому линейному сравнению.

Теорема 5.16. Если наибольший общий делитель d чисел a и n ($d=(a,n)$) не является делителем b, то линейное сравнение $ax=b$ ***mod*** n не имеет решений.

Доказательство: Предположим противоположное, что есть решение x_0, которое удовлетворяет линейному сравнению, то есть $ax_0=b$ ***mod*** n. Согласно теореме 3.16, d является делителем a и n, откуда следует, что d должен являться делителем ax_0 и nq, так же как и делителем $ax_0–nq=b$. Здесь q есть положительное целое число. Тогда получим противоречие: по условию теоремы d не является делителем b, а в случае наличия решения x_0, d должно является делителем b. Значит, линейное сравнение $ax=b$ ***mod*** n не имеет решений, если $d=(a,n)$ не является делителем b. #

Пример 5.22. Линейное сравнение $2x=1$ *mod* 4 не имеет решения, так как наибольший общий делитель d=2 чисел a=2 и n=4 (2=(2,4)) не является делителем b=1. Действительно, ни одно из возможных целочисленных значений $x<4$ не удовлетворяет сравнению $2x=1$ *mod* 4.

Теорема 5.17. Если наибольший общий делитель d чисел a и n равен единице $(a,n)=1$, то есть a и n являются взаимно простыми числами, то линейное сравнение $ax=b$ ***mod*** n имеет одно решение.

Доказательство: Предположим, что существует полная система вычетов $\{0,1,2,\ldots,n-1\}$ по модулю n. Тогда согласно тому, что a и n взаимно простые целые числа, множество чисел $\{0\cdot a, 1\cdot a, 2\cdot a,\ldots, (n-1)\cdot a\}$ образует полную систему вычетов по модулю n. Тогда среди всех целых чисел есть одно и только одно ax_0 с вычетом равным b.#

Пример 5.18. Линейное сравнение $2x=1$ *mod* 3 имеет одно решение x_0=2, так как a=2 и n=3 являются взаимно простыми числами.

Теорема 3.17 позволяет сформулировать задачу нахождения решения линейного сравнения для случая, когда a и n являются взаимно простыми числами. Здесь возможны два случая в зависимости от величины b.

Для b=1 линейное сравнение принимает вид $ax=1$ ***mod*** n, где неизвестная величина $x=a^{-1}$ является **мультипликативной инверсной величиной** по отношению к a. Тогда $aa^{-1}=1$ ***mod*** n.

Для вычисления мультипликативной инверсной величины возьмём два линейных сравнения: исходное сравнение $ax=1$ ***mod*** n в соответствии с услови-

ем теоремы 3.17 и сравнение $1=a^{\psi(n)}$ **mod** n, соответствующее теореме Эйлера. Затем, используя лемму 3.6, перемножим левую и правую части этих сравнений. Как результат получим $ax=a^{\psi(n)}$ **mod** n. Используя лемму 3.2, разделим правую и левую части последнего равенства $ax=a^{\psi(n)}$ **mod** n на a, получим

$$x=a^{\psi(n)-1} \textbf{ mod } n,$$

что и является основным расчетным соотношением для вычисления мультипликативной инверсной величины. Для случая, когда n является простым числом $x=a^{n-2}$ **mod** n.

Пример 5.19. Найти решение линейного сравнения $3x=1$ *mod* 7.

Так как 7 это простое число, тогда $x=a^{n-2}$ **mod** $n=3^{7-2}$ **mod** $7=3^5$ **mod** $7=5$.

Пример 5.20. Найти решение линейного сравнения $4x=1$ *mod* 9.

Функция Эйлера целого числа $n=9$ вычисляется как $\psi(9)=6$. Тогда $x=a^{\psi(n)-1}$ **mod** $n=4^{6-1}$ **mod** $9=4^5$ **mod** $9=7$.

Для $b\neq 1$ линейное сравнение принимает вид $ax=b$ **mod** n, где неизвестная величина x будет вычисляться согласно соотношению

$$x=ba^{\psi(n)-1} \textbf{ mod } n,$$

а для случая когда n является простым числом – по формуле $x=ba^{n-2}$ **mod** n.

Пример 5.21. Найти решение линейного сравнения $3x=3$ *mod* 7.

Принимая во внимание, что $(3,7)=1$, а 7 есть простое число $x=ba^{n-2}$ **mod** $n=3\times 3^{7-2}$ **mod** $7=3^6$ **mod** $7=1$.

Теорема 5.18. Если наибольший общий делитель d чисел a и n является делителем числа b ($d \mid b$), то существует d решений линейного сравнения вида $ax=b$ **mod** n.

Доказательство: Согласно условию теоремы, d является делителем a, n и b. Тогда из линейного сравнения $ax=b$ **mod** n получим $a_1dx=b_1d$ **mod** n_1d. Используя лемму 3.3, получим линейное сравнение вида $a_1x=b_1$ **mod** n_1, где $(a_1,n_1)=1$. Сравнение $a_1x=b_1$ **mod** n_1 имеет одно решение x_0. Целые числа того же класса по модулю n/d будут решениями для исходного сравнения $ax=b$ **mod** n. То есть $x_1=x_0$ **mod** n, $x_2=x_0+n/d$ **mod** n, $x_3=x_0+2n/d$ **mod** $n,\ldots, x_d=x_0+((d-1)n)/d$ **mod** n.#

Пример 5.22. Найти решения для следующего линейного сравнения $6x=4$ *mod* 10.

Принимая во внимание, что $(6,10)=2$ и 2 это делитель 4, получим сравнение $3x=2$ *mod* 5. Решением последнего сравнения будет $x_0=ba^{n-2}$ **mod** $n=2\times 3^{5-2}$ **mod** $5=2\times 3^3$ **mod** $5=4$. Тогда решениями сравнения $6x=4$ *mod* 10 будут $x_1=x_0$ **mod** $n=4$ **mod** $10=4$; $x_2=x_0+n/d$ **mod** $n=4+10/2$ **mod** $10=9$.

Тема 6. Псевдослучайные числа

V.N.Yarmolik, S.N.Demidenko, "Generation and Application of Pseudorandom Sequences for Random Testing", John Wiley & Sons Ltd., Chichester, 1988, 176 p.

В.Н.Ярмолик, С.Н.Демиденко, "Генерирование и применение псевдослучайных последовательностей в системах испытаний и контроля", Наука и Техника, Минск, 1986, 200с.

В задачах активных экспериментальных исследований современных сложных технических систем с применением статистических методов важное место принадлежит генерированию сигналов возбуждения. Диктуется это не только необходимостью подачи на объект требуемого числа воздействий с заданными свойствами, но и максимальной скорости их выработки. Одним из наиболее распространенных в настоящее время методов формирования таких процессов является преобразование сигналов, получаемых с помощью, так называемых генераторов белого шума (ГБШ). В применении к цифровым методам генерирования под белым шумом понимается последовательность некоррелированных чисел или цифр, распределенных, как правило, по равномерному закону.

Известны два основных метода получения цифрового белого шума: физический — генерирование случайных двоичных чисел с помощью специальных устройств — генераторов случайных чисел (ГСЧ); математический — формирование псевдослучайных числовых последовательностей (ПСЧП) по специальным программам или с использованием генераторов псевдослучайных чисел (ГПСЧ).

Возможен также и третий путь, совмещающий в себе физический и математический принципы.

Принцип действия ГСЧ состоит в преобразовании случайного сигнала на выходе физического источника шума в импульсную последовательность с вероятностью появления импульса $p(1)= 0{,}5$.

В зависимости от способа формирования числовых последовательностей существующие ГСЧ можно разделить на четыре основных типа :

1) ГСЧ, основанные на использовании случайных состояний бистабильной схемы при периодическом включении и выключении источника питания;

2) ГСЧ с пересчетом импульсов периодической последовательности за случайный интервал времени;

3) ГСЧ с пересчетом импульсов случайной последовательности за фиксированный интервал времени;

4) ГСЧ, основанные на дискретизации непрерывного шумового сигнала по двум уровням.

Наиболее часто при разработке быстродействующих и высококачественных ГСЧ применяются последних два способа.

В генераторах с пересчетом случайных импульсов за фиксированный интервал времени равновероятность символов выходной последовательности достигается путем выполнения суммирования по модулю два. В то же время данная операция, протекающая последовательно во времени, снижает быстродействие ГСЧ, что ограничивает область их применения. Более высоким быстродействием обладают ГСЧ, основанные на дискретизации непрерывного

шумового сигнала по двум уровням. Однако данным устройствам свойственна высокая чувствительность к изменениям параметров первичного шумового сигнала, главным образом его среднего значения.

Общими и наиболее существенными недостатками, затрудняющими применение ГБШ, являются [2, 6—9] ограниченное быстродействие, определяемое первичным аналоговым источником шума; низкая стабильность основных вероятностных характеристик, объясняемая нестабильностью первичных источников, дрейфом параметров преобразующих схем, источников питания и др., что требует периодической статистической проверки качества генерируемой последовательности; сложность аппаратурной реализации, вызываемая наличием нескольких источников питания (для физического источника шума и цифровых схем), необходимостью стабилизации и фильтрации их напряжения, развязки линий питания от линий устройства, коррекции выходной последовательности и др.; невозможность воспроизведения и предсказания генерируемых последовательностей в силу их случайной природы; неоднородность структуры генераторов вследствие наличия как аналоговых, так и цифровых узлов, что затрудняет миниатюризацию ГСЧ.

Указанные недостатки физических ГСЧ явились причиной все более широкого распространения математических методов получения шумовых числовых последовательностей. Мгновенные значения таких псевдослучайных последовательностей в отличие от случайных в принципе могут быть предсказаны заранее. В то же время все оценки статистических характеристик конкретной реализации ПСЧП совпадают с оценками соответствующей ей случайной выборки. Любую статистическую характеристику псевдослучайной числовой последовательности можно получить, используя реализацию длиной в один период повторения ПСЧП. Для истинно случайной последовательности это потребовало бы бесконечно большую длину реализации. Искусственное увеличение периода ПС-сигнала неограниченно приближает его структуру к структуре одной из возможных реализаций истинно случайного процесса. Однако и при ограниченных величинах периода в определенных условиях псевдослучайные числовые последовательности могут заменить случайные. При анализе псевдослучайной реализации равной или меньшей длине периода вообще практически невозможно определить, является ли она отрезком регулярной или случайной последовательности. С другой стороны, если записать конкретную случайную реализацию на каком-либо носителе, и периодически воспроизводить ее, то получим регулярную ПСЧП.

Таким образом, с точки зрения реальных характеристик трудно установить границу между случайными и псевдослучайными числовыми последовательностями. В то же время применение ПСЧП имеет ряд существенных преимуществ: периодический характер псевдослучайного сигнала обусловливает низкий уровень дисперсии оценок, получаемых при усреднении в течение целого числа периодов; характеристики ПСЧП абсолютно стабильны и определяются алгоритмом формирования псевдослучайных чисел; последователь-

ность можно повторить с любого желаемого участка реализации, для чего не требуется сложных запоминающих устройств и др.

Наиболее часто при генерировании ПСЧП с равномерным распределением используются два метода. Первый, лежащий в основе большинства современных программных датчиков и некоторых специализированных устройств, описывается рекуррентным соотношением

$$X_k = AX_{k-1} + C(\mathbf{mod}R),\ k=1, 2, 3, \ldots ,$$

где A, C, R — постоянные числа; $X_0>0$, $A>0$, $C>0$, $R>X_0$, $R>A$, $R>C$. Данный метод получил название мультипликативного конгруэнтного (при $C=0$) или смешанного конгруэнтного (при $C>0$), а формируемая в соответствии с ним последовательность — линейной конгруэнтной.

Тема 7. Генерирование М-последовательностей

V.N.Yarmolik, S.N.Demidenko, "Generation and Application of Pseudorandom Sequences for Random Testing", John Wiley & Sons Ltd., Chichester, 1988, 176 p.

В.Н.Ярмолик, С.Н.Демиденко, "Генерирование и применение псевдослучайных последовательностей в системах испытаний и контроля", Наука и Техника, Минск, 1986, 200с.

Генераторы *М*-последовательностей являются одними из наиболее часто используемых типов генераторов псевдослучайных последовательностей, применяемых в различных приложениях. Для их реализации используется сдвиговый регистр с линейной обратной связью (***Linear Feedback Shift Register – LFSR***). Подобный генератор часто используется как источник криптографического ключа для потоковых систем шифрования. Одной из основных причин широкого применения генераторов *М*-последовательностей является сравнительно простая их реализация, как программная, так и аппаратная. Кроме того, отмечаются хорошие статистические свойства *М*-последовательностей, практически не отличающиеся от свойств случайных последовательностей. В то же время генераторы *М*-последовательностей, подобно, как и любые другие генераторы псевдослучайных последовательностей, обеспечивают воспроизводимость выходных последовательностей. Подобные генераторы (далее их будем обозначать LFSR) строятся на основании примитивных порождающих полиномов. Так, согласно примитивному порождающему полиному $\varphi(x)=1+x+x^4$, структурная схема LFSR представлена на рис.7.1.

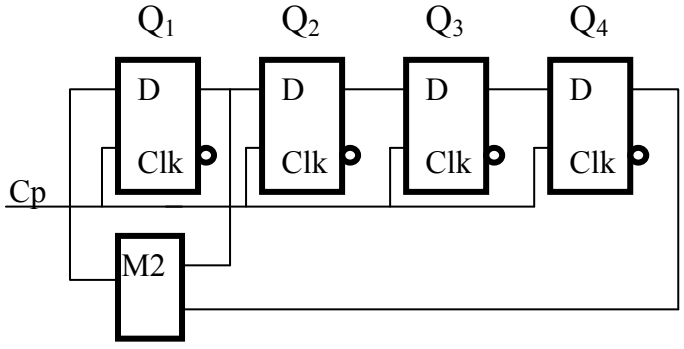


Рис.7.1. Структурная схема LFSR

Схема LFSR, приведенного на рис.7.1, состоит из четырехразрядного регистра сдвига на один разряд вправо и двухвходового сумматора по модулю два, включенного в цепь обратной связи. Аналитически функционирование приведенного генератора описывается системой уравнений:

$$Q_1(k+1)=Q_1(k)\oplus Q_4(k);$$
$$Q_2(k+1)=Q_1(k);$$
$$Q_3(k+1)=Q_2(k);$$
$$Q_4(k+1)=Q_3(k).$$

Последовательные состояния генератора приведены в таблице 7.1.

***Таблица* 7.1**.

Состояния генератора LFSR

#	$Q_1Q_2Q_3Q_4$	#	$Q_1Q_2Q_3Q_4$
0	1 0 0 0	8	1 1 0 1
1	1 1 0 0	9	0 1 1 0
2	1 1 1 0	10	0 0 1 1
3	1 1 1 1	11	1 0 0 1
4	0 1 1 1	12	0 1 0 0
5	1 0 1 1	13	0 0 1 0
6	0 1 0 1	14	0 0 0 1
7	1 0 1 0	15	1 0 0 0

Как видно из приведенной таблицы, используя начальное состояние 1000, LFSR генерирует всевозможные четырехразрядные двоичные коды кроме нулевого кода (0000). Последовательность генерируемых кодов имеет случайный характер, и её свойства не зависят от ненулевого начального состояния.

Для любой разрядности двоичных кодов существуют примитивные порождающие полиномы. Полиномы с минимальным количеством ненулевых коэффициентов для разрядностей от 1 до 28 приведены в таблице 7.2.

***Таблица* 7.2.**

Примитивные порождающие полиномы

$m=deg\varphi(x)$	$\varphi(x)$	$m=deg\varphi(x)$	$\varphi(x)$

1	$1+x$	15	$1+x+x^{15}$
2	$1+x+x^2$	16	$1+x^2+x^3+x^5+x^{16}$
3	$1+x+x^3$	17	$1+x^3+x^{17}$
4	$1+x+x^4$	18	$1+x^7+x^{18}$
5	$1+x^2+x^5$	19	$1+x+x^2+x^5+x^{19}$
6	$1+x+x^6$	20	$1+x^3+x^{20}$
7	$1+x+x^7$	21	$1+x^2+x^{21}$
8	$1+x+x^5+x^6+x^8$	22	$1+x+x^{22}$
9	$1+x^4+x^9$	23	$1+x^5+x^{23}$
10	$1+x^3+x^{10}$	24	$1+x^3+x^4+x^{24}$
11	$1+x^2+x^{11}$	25	$1+x^3+x^{25}$
12	$1+x^3+x^4+x^7+x^{12}$	26	$1+x+x^2+x^6+x^{26}$
13	$1+x+x^3+x^4+x^{13}$	27	$1+x+x^2+x^5+x^{27}$
14	$1+x+x^{11}+x^{12}+x^{14}$	28	$1+x^3+x^{28}$

Для произвольной степени $m=deg\varphi(x)$ порождающего полинома $\varphi(x)=a_0+a_1x+a_2x^2+...+a_{m-1}x^{m-1}+a_mx^m$; $a_m=a_0=1$; $a_i\in\{0,1\}$ существуют две альтернативные структуры реализации LFSR с внешними и внутренними сумматорами по модулю два. На рис.7.2 приведена структура LFSR с внешними сумматорами по модулю два.

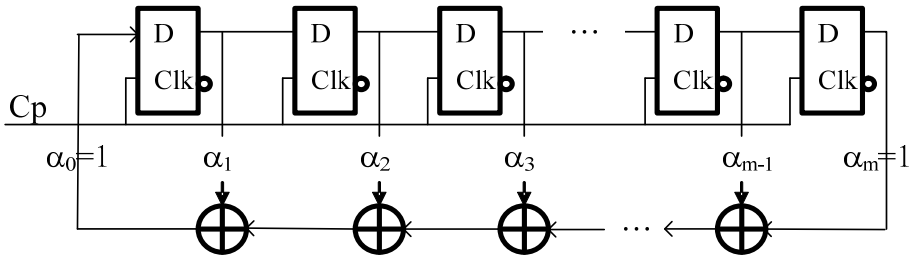


Рис.7.2. Структурная схема LFSR с внешними сумматорами по модулю два

На рис.5.4 приведена структура LFSR с внутренними сумматорами по модулю два. Альтернативная структура LFSR описывается тем же примитивным порождающим полиномом, что и предыдущая структура.

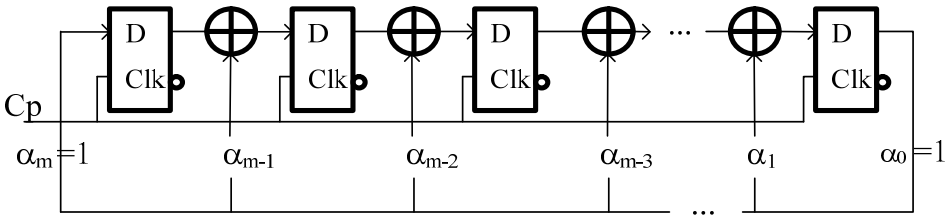
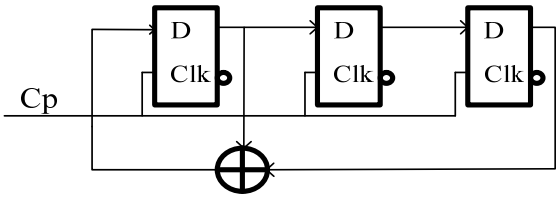


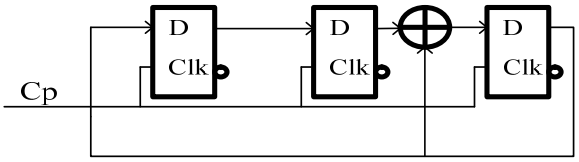
Рис.7.3. Структурная схема LFSR с внутренними сумматорами по модулю два

***Пример* 7.1.** В качестве примера LFSR с внешними и внутренними сумматорами рассмотрим случай порождающего полинома $\varphi(x)=1+x+x^3$, для которого ниже приведены соответствующие структуры и временные диаграммы формируемых последовательностей.



1 0 0
1 1 0
1 1 1
0 1 1
1 0 1
0 1 0
0 0 1

а)



1 0 0
0 1 0
0 0 1
1 0 1
1 1 1
1 1 0
0 1 1

б)

Рис.*7.4. *Структурные схемы LFSR для порождающего полинома* $\varphi(x)=1+x+x^3$ *с внешними а) и внутренними б) сумматорами по модулю два

Функционирование LFSR описывается следующим математическим соотношением, представленным системой уравнений:

$$a_1(k+1)=\sum_{i=1}^{m}\alpha_i a_i(k);$$

$$a_j(k+1)=a_{j-1}(k),\ j=\overline{2,m},k=0,1,2,...$$

Матричное описание LFSR имеет вид:

$$\begin{vmatrix} a_1(k+1) \\ a_2(k+1) \\ a_3(k+1) \\ \dots \\ a_m(k+1) \end{vmatrix} = \begin{vmatrix} \alpha_1 & \alpha_2 & \alpha_3 & \dots & \alpha_{m-1} & \alpha_m \\ 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 1 & 0 \end{vmatrix} \times \begin{vmatrix} a_1(k) \\ a_2(k) \\ a_3(k) \\ \dots \\ a_m(k) \end{vmatrix}$$

В векторной форме будем иметь

$$A(k+1) = V \times A(k),$$

где V представляет собой порождающую (генерирующую) матрицу, $A(k)$ – вектор-столбец текущего, а $A(k+1)$ – последующего состояния LFSR.

Псевдослучайные последовательности (M-последовательности) характеризуются следующими свойствами.

1.Существует $\psi(2^m-1)/m$ примитивных полиномов для заданного значения его степени m, где ψ есть функция Эйлера. Так, например, для m=3 $\psi(2^m-1)/m=\psi(2^3-1)/3=6/3=2$. Таким образом, для m=3 существует два примитивных полинома $\varphi(x)=1+x+x^3$ и $\varphi(x)=1+x^2+x^3$. Следует отметить, что функция Эйлера принимает большие значения даже для небольших значений m.

2.Для заданного полинома $\varphi(x)$ существует инверсный полином $\varphi(x)^{-1}$, который может быть получен согласно следующему отношению

$$\varphi(x)^{-1}=x^m\varphi(x^{-1}).$$

***Пример* 7.2.** Так, например, для порождающего полинома $\varphi(x)=1+x^2+x^5$ $\varphi(x)^{-1}= x^5\varphi(x^{-1}) = x^5\ (1+x^{-2}+x^{-5})= 1+x^3+x^5$.

3.Период M-последовательности зависит от степени (m) примитивного порождающего полинома и равняется $L=2^m-1$.

Отметим что LFSR, построенный на базе примитивного порождающего полинома $\varphi(x)$ степени m, генерирует всевозможные m-разрядные двоичные коды кроме кода, состоящего из всех нулей.

4.Для заданного полинома $\varphi(x)$ существует L различных M-последовательностей, каждая из которых отличается фазовым сдвигом.

***Пример* 7.3.** Для порождающего полинома $\varphi(x)=1+x+x^4$ существует 15 M-последовательностей, приведенных в таблице 7.3.

***Таблица* 7.3.**

***M*-последовательности, описываемые полиномом $\varphi(x)=1+x+x^4$**

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	0	1	0	0	1	1	0	1	0	1	1	1
1	1	0	0	0	1	0	0	1	1	0	1	0	1	1
1	1	1	0	0	0	1	0	0	1	1	0	1	0	1
1	1	1	1	0	0	0	1	0	0	1	1	0	1	0
0	1	1	1	1	0	0	0	1	0	0	1	1	0	1
1	0	1	1	1	1	0	0	0	1	0	0	1	1	0
0	1	0	1	1	1	1	0	0	0	1	0	0	1	1
1	0	1	0	1	1	1	1	0	0	0	1	0	0	1
1	1	0	1	0	1	1	1	1	0	0	0	1	0	0
0	1	1	0	1	0	1	1	1	1	0	0	0	1	0
0	0	1	1	0	1	0	1	1	1	1	0	0	0	1
1	0	0	1	1	0	1	0	1	1	1	1	0	0	0

0	1	0	0	1	1	0	1	0	1	1	1	1	0	0
0	0	1	0	0	1	1	0	1	0	1	1	1	1	0
0	0	0	1	0	0	1	1	0	1	0	1	1	1	1

5.*M*-последовательность представляет собой псевдослучайную последовательность, в которой вероятность появления нулей и единиц определяется соотношениями

$$p(a_k = 1) = \frac{2^{m-1}}{2^m - 1} = \frac{1}{2} + \frac{1}{2^{m+1} - 2};$$

$$p(a_k = 0) = \frac{2^{m-1} - 1}{2^m - 1} = \frac{1}{2} - \frac{1}{2^{m+1} - 2}.$$

Как видно из приведенных соотношений, вероятность появления нуля и единицы практически равняется 0,5.

7.Функция автокорреляции *M*-последовательности максимально близка к автокорреляционной функции идеальной случайной последовательности. Действительно, оригинальная *M*-последовательность является идентичной в 2^{m-1}-1 позициях со сдвинутой своей копией на любое число тактов, и будет отличаться в 2^{m-1} позициях.

Так, например, последовательность 000111101011001, генерируемая в соответствии с полиномом $\varphi(x)=1+x+x^4$, и ее сдвинутая на две позиции копия 011110101100100 совпадают на семи позициях и имеют различное значение на восьми позициях.

8.Свойство ***сдвига и сложения***. Для любого s ($1 \le s < L$) существует $r \ne s$ ($1 \le r < L$) такое что $\{a_k\} \oplus \{a_{k-s}\} = \{a_{k-r}\}$:

$\{a_0\}$	000111101011001
$\{a_{-2}\}$	011110101100100
$\{a_{-9}\}$	011001000111101

9.Среди L *M*-последовательностей, полученных на основании примитивного полинома $\varphi(x)$, существует единственная *M*-последовательность, для которой выполняется равенство $a_k = a_{2k}$, k=0,1,2,....

Такая *M*-последовательность называется ***характеристикой*** и получается следующим образом. Для заданного примитивного порождающего полинома строится система линейных уравнений $a_i = a_{2i}$, i=0,1,2,..., m-1. Ненулевое решение приведенной системы уравнений и есть искомая характеристическая последовательность.

***Пример* 7.4.** В качестве примера рассмотрим процедуру определения характеристической последовательности для полинома $\varphi(x)=1+x+x^4$. Для этого случая система линейных уравнений имеет форму

$a_0=a_0$;
$a_1=a_2$;
$a_2=a_4=a_0\oplus a_3$;
$a_3=a_6=a_2\oplus a_5= a_1\oplus a_1\oplus a_4= a_0\oplus a_3$.

Единственное ненулевое решение $a_0a_1a_2a_3$=0111, удовлетворяющее приведенной системе, и есть искомая характеристическая *M*-последовательность. В таблице 5.3 эта последовательность приведена под номером 2.

10.Свойство ***децимаций***. Децимация *M*-последовательности $\{a_i\}$ по индексу q, (q=1,2,3,...) означает порождение другой последовательности $\{b_j\}$ как q-тые элементы первоначальной *M*-последовательности $\{a_i\}$, то есть $b_j=a_{qj}$. Если наибольший общий делитель периода L *M*-последовательности и индекса q равен единице, то есть (L,q)=1, то период $\{b_j\}$ будет равен $L=2^m-1$, и децимация называется нормальной.

***Пример* 7.5.** В качестве примера рассмотрим нормальную децимацию *M*-последовательности, соответствующей полиному $\varphi(x)=1+x+x^4$ по индексу два. В результате получим

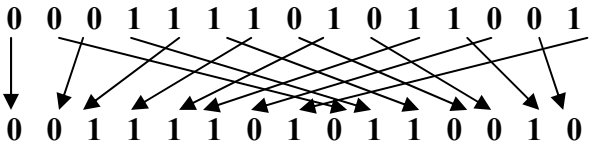


Рис.7.5. Пример децимации M-последовательности

Как видно из предыдущего примера, в результате децимации получили *M*-последовательность, описываемую тем же полиномом. В общем случае результатом децимации может быть любая *M*-последовательность, описываемая любым примитивным порождающим полиномом той же степени m.

В общем случае генераторы нелинейных последовательностей строятся с использованием нелинейных зависимостей для получения очередного выходного значения. Чаще всего для этих целей используются регистры сдвига с нелинейной обратной связью F. Структура подобного генератора включает в себя два основных блока, а именно сдвиговой регистр на один разряд вправо и комбинационную схему, реализующую нелинейную функцию, которая описывает обратную связь. Аналогично регистрам сдвига с линейной обратной связью (LFSR) подобные устройства называются ***регистрами сдвига с нелинейной обратной связью*** (***Nonlinear Feedback Shift Register – NFSR***). Общий вид подобного устройства приведен на рисунке 7.6.

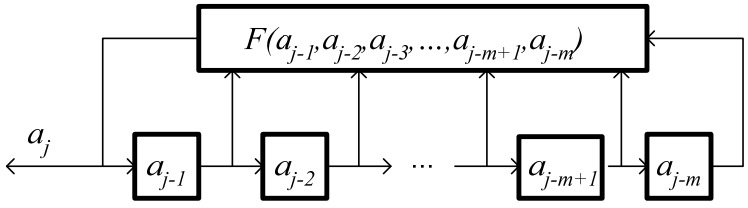


Рис.7.6. Регистр сдвига с нелинейной обратной связью

Хорошо изученной разновидностью подобных генераторов являются генераторы ***последовательностей де Брейна*** (***De Bruijn***), характерной чертой для которых является генерирование всевозможных 2^m двоичных комбинаций, где m является разрядностью регистра сдвига, на котором формируется последовательность де Брейна. Существует процедура преобразования LFSR в *NFSR*, генерирующего последовательность де Брейна. В этом случае нелинейная функция $F(a_{j-1},a_{j-2},a_{j-3},\ldots,a_{j-m+1},a_{j-m})$ образуется как сумма по модулю два функции $f(a_{j-1},a_{j-2},a_{j-3},\ldots,a_{j-m+1},a_{j-m})$, описывающей линейную обратную связь LFSR, с нелинейной функцией $g(a_{j-1},a_{j-2},a_{j-3},\ldots,a_{j-m+1})=a^*_{j-1},a^*_{j-2},a^*_{j-3},\ldots,a^*_{j-m+2},a^*_{j-m+1}$. Здесь $a^*_i=1\oplus a_i$.

Вид функции $g(a_{j-1},a_{j-2},a_{j-3},\ldots,a_{j-m+1})=a^*_{j-1},a^*_{j-2},a^*_{j-3},\ldots,a^*_{j-m+2},a^*_{j-m+1}$ свидетельствует о том, что данная нелинейная функция принимает единичное значение только на двух наборах переменных (двух состояниях регистра сдвига), а именно на наборе 000…001 и 000…000. Отметим, что в последовательности кодов, формируемых LFSR, присутствует только код 000…001, и нет больше кодов, у которых все первые m-1 бита принимают нулевое значение. Кроме того, независимо от вида конкретного примитивного порождающего полинома, функция $f(a_{j-1},a_{j-2},a_{j-3},\ldots,a_{j-m+1},a_{j-m})$, описывающая линейную обратную связь LFSR, всегда принимает единичное значение на наборе 000…001.

Таким образом, при появлении на регистре сдвига набора 000…001 обе функции g и f принимают единичное значение, а F, соответственно, нулевое, в силу чего на регистре сдвига будет получен код 000…000, который будет являться следующим набором аргументов для указанных функций g, f и F. В следующем цикле g будет равняться единице, а f – нулю. Тогда F=1, и на регистре сдвига будет получен код 100…000.

***Пример* 7.6.** Простейшим примером генератора последовательности де Брейна является генератор, представленный на рисунке 7.7.

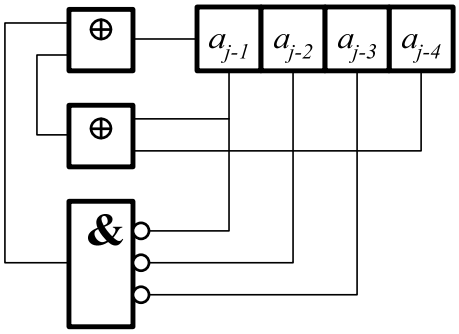


Рис.7.7. Генератор последовательности де Брейна

Данный генератор синтезирован на базе LFSR, соответствующего порождающему полиному $\varphi(x)=1\oplus x^1\oplus x^4$, линейная обратная связь которого описывается функцией $f(a_1,a_2,a_3,a_4)=a_1\oplus a_4$. Нелинейная функция $F(a_{j-1},a_{j-2},a_{j-3},\ldots,a_{j-m+1},a_{j-m})$ для данного примера принимает вид $F(a_{j-1},a_{j-2},a_{j-3},\ldots,a_{j-m+1},a_{j-m})=$ $=a_1\oplus a_4\oplus(a^*_1a^*_2a^*_3)$. Диаграмма состояний генератора приведена в таблице.

***Таблица* 7.4.**

Диаграмма состояний генератора последовательности де Брейна

#	***a*1**	***a*2**	***a*3**	***a*4**	#	***a*1**	***a*2**	***a*3**	***a*4**
1	1	0	0	0	9	1	1	0	1
2	1	1	0	0	10	0	1	1	0
3	1	1	1	0	11	0	0	1	1
4	1	1	1	1	12	1	0	0	1
5	0	1	1	1	13	0	1	0	0
6	1	0	1	1	14	0	0	1	0
7	0	1	0	1	15	0	0	0	1
8	1	0	1	0	16	0	0	0	0

Последовательность де Брейна, как видно из приведенного примера, может быть получена путем добавления нулевого символа к последовательности из 2^m-1 бит, таким образом, чтобы на периоде ее повторения образовалась серия из m нулей. Такая последовательность соответственно имеет период равный 2^m и характеризуется наличием на интервале повторения всевозможных кодов длиной m, каждый из которых встречается только один раз. Последнее свойство и определило интерес к использованию последовательностей де Брейна для различных приложений в криптографии.

Частным случаем последовательностей де Брейна являются так называемые ***последовательности Форда***, которые описываются более простым алгоритмом формирования. Последний носит рекуррентный характер и заключается в формировании очередного m-разрядного кода $X_k=(b_2,b_3,b_4,\ldots,b_{m+1})$ на основе предыдущего кода $X_{k-1}=(b_1,b_2,b_3,\ldots,b_m)$, где значение $b_{m+1}\in\{0,1\}$ определяется следующим образом. Формируется код вида $X^*_{k-1}=(b_2,b_3,b_4,\ldots,b_m,1)$ и строятся всевозможные его циклические сдвиги. Среди них выбирается код $X^{**}_{k-1}=(b_i,b_{i+1},\ldots,b_m,1,b_2,\ldots,b_{i-1})$, представляющий собой наибольшее m-разрядное число. Если $b_2=\ldots=b_{i-1}=0$, то значение b_{m+1} равно $b_1\oplus1$, а в противном случае b_{m+1} равно b_1. Отметим, что начальным кодом X_0 для формирования последовательности Форда может быть любой код, включая $X_0=(0,0,0,\ldots,0)$.

В качестве примера в таблице 7.5 приведены результаты формирования последовательности Форда для m=4.

***Таблица* 7.5.**

Диаграмма состояний генератора последовательности Форда

k	$X_k=$ $=(b_2,b_3,b_4,\ldots,b_{m+1})$	$X^{**}_{k-1}=$ $=(b_i,b_{i+1},\ldots,b_m,1,b_2,\ldots,b_{i-1})$	b_{m+1}

0	0 0 0 0	1 0 0 0	$b_1 \oplus 1 = 0 \oplus 1 = 1$
1	0 0 0 1	1 1 0 0	$b_1 \oplus 1 = 0 \oplus 1 = 1$
2	0 0 1 1	1 1 1 0	$b_1 \oplus 1 = 0 \oplus 1 = 1$
3	0 1 1 1	1 1 1 1	$b_1 \oplus 1 = 0 \oplus 1 = 1$
4	1 1 1 1	1 1 1 1	$b_1 \oplus 1 = 1 \oplus 1 = 0$
5	1 1 1 0	1 1 1 0	$b_1 = 1$
6	1 1 0 1	1 1 1 0	$b_1 = 1$
7	1 0 1 1	1 1 1 0	$b_1 \oplus 1 = 1 \oplus 1 = 0$
8	0 1 1 0	1 1 1 0	$b_1 = 0$
9	1 1 0 0	1 1 0 0	$b_1 = 1$
10	1 0 0 1	1 1 0 0	$b_1 \oplus 1 = 1 \oplus 1 = 0$
11	0 0 1 0	1 0 1 0	$b_1 \oplus 1 = 0 \oplus 1 = 1$
12	0 1 0 1	1 1 1 0	$b_1 = 0$
13	1 0 1 0	1 0 1 0	$b_1 \oplus 1 = 1 \oplus 1 = 0$
14	0 1 0 0	1 1 0 0	$b_1 = 1$
15	1 0 0 0	1 0 0 0	$b_1 \oplus 1 = 1 \oplus 1 = 0$
16	0 0 0 0	1 0 0 0	...
...	...		

Весьма близкими по свойствам к *M*-последовательностям и находящими наряду с ними широкое практическое применение являются ***последовательности Голда и Касами***. Последовательность Голда образуется путем суммирования по модулю 2 двух *M*-последовательностей, порождаемых отличными примитивными полиномами $\varphi'(x)$ и $\varphi''(x)$ одной и той же степени m. Период данной последовательности равен 2^m-1, а порождающий ее полином представляет собой произведение $\varphi'(x)$ на $\varphi''(x)$.

Более широким классом последовательностей, включающим в качестве подмножества последовательности Голда, является множество последовательностей Касами. Последние могут быть получены в результате двоичного сложения трех *M*-последовательностей $\{a_i\}$, $\{b_i\}$ и $\{c_i\}$, порождаемых полиномами $\varphi'(x)$ и $\varphi''(x)$ степени m и $\varphi^*(x)$ степени $m/2$ соответственно, где m – четно. Наиболее важное качество последовательностей Касами, а также входящих в них подмножеств последовательностей, является их высокие корреляционные свойства.

Тема 8. Анализ числовых последовательностей

Эффективность статистического моделирования систем на ЭВМ и достоверность получаемых результатов зависит от качества исходных (базовых) последовательностей псевдослучайных чисел, которые являются основой для получения стохастических воздействий на элементы моделируемой системы. Поэтому перед моделированием на ЭВМ необходимо убедиться в том, что исходная последовательность псевдослучайных чисел удовлетворяет предъявляемым к ней требованиям.

При моделировании важными характеристиками качества генератора являются длина периода p и длина отрезка апериодичности L. Длина отрезка апериодичности L псевдослучайной последовательности есть наибольшее целое число, такое, что все числа x_i в пределах этого отрезка не повторяются.

Способ экспериментального определения длины периода p и длины отрезка апериодичности L сводится к следующему:

1. Запускается программа генерации последовательности $\{x_i\}$ с начальным значением x_0 и генерируется V чисел x_i. В большинстве случаев $V – (1 – 5)10^6$. Генерируются числа и фиксируется число x_V.

2. Затем программа запускается повторно с начальным числом x_0 и при генерации очередного числа проверяется истинность события $p\{x_i = x_V\}$. Если это событие истинно $i = i_1$ и $i = i_2$ ($i_1 < i_2 < V$), то вычисляется длина периода последовательности $p = i_2 – i_1$.

3. Проводится запуск программы генерации с начальными числами x_0 и x_p. При этом фиксируется минимальный номер $i = i_3$, при котором истинно событие $p''\{x_i = x_{p+i}\}$, и вычисляется длина отрезка апериодичности L = $i_3 + p$.

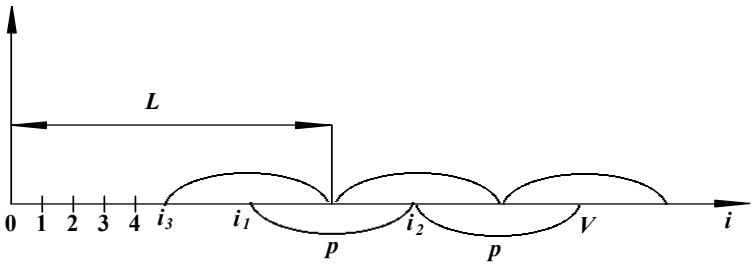


Рис. 7.8. Представление определения длин периода P и отрезка периодичности L

Применяемые в имитационном моделировании генераторы случайных чисел должны пройти тесты на пригодность.

Существуют тесты двух типов.

Эмпирические тесты – это обычный тип статистических тестов, они основаны на действительных значениях x_i, выдаваемых генератором.

Теоретические тесты не являются тестами в том смысле, в каком они предусматриваются в статистике. Однако в них используются числовые параметры, чтобы оценить генератор глобально без фактического генерирования некоторых или всех значений x_i.

Основные анализируемые характеристики генерируемых датчиком последовательностей:

- равномерность;
- стохастичность (случайность);
- независимость.

Рассмотрим методы проведения такого анализа, наиболее часто применяемые на практике.

Проверка равномерности

Проверка равномерности может быть выполнена с помощью гистограммы

относительных частот генерируемой случайной величины. Для ее построения интервал (0;1) разбивается на *m* равных частей и подсчитывается относительное число попаданий значений случайной величины в каждый интервал. Чем ближе огибающая гистограммы к прямой, тем в большей степени генерируемая последовательность отвечает требованию равномерности распределения (рис. 7.9).

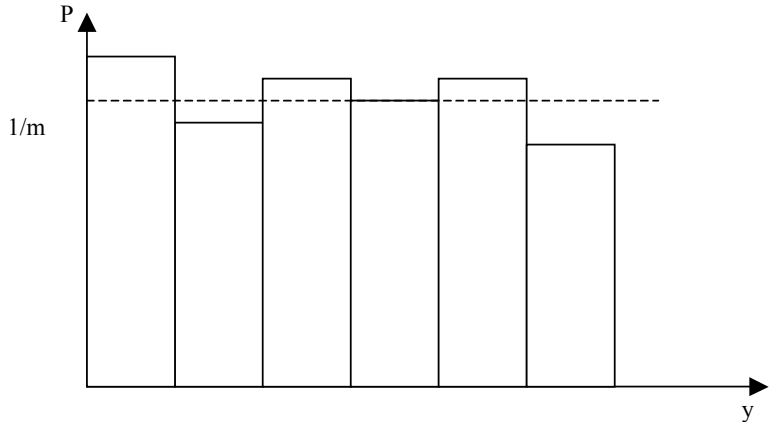


Рис. 7.9. Гистограмма относительных частот случайных чисел

Проверка стохастичности

Рассмотрим один из основных методов проверки – *метод комбинаций.*

Суть его сводится к следующему. Выбирают достаточно большую последовательность случайных чисел x_i и для нее определяют вероятность появления в каждом из x_i ровно j единиц. При этом могут анализироваться как все разряды числа, так и только *l* старших. Теоретически закон появления j единиц в *l* разрядах двоичного числа может быть описан как биномиальный закон распределения (исходя из независимости отдельных разрядов).

Тогда при длине выборки *N* ожидаемое число появлений случайных чисел x_i с j единицами в проверяемых *l* разрядах будет равно:

$$n_j = N \times C_l^j p^l(1),$$

где C_l^j – число комбинаций (сочетаний) j единиц в *l* разрядах,

$p^l(1)$ – вероятность появления единицы в двоичном разряде, $p(1)=0.5$.

Для полученной последовательности определяется эта же характеристика. Проверка соответствия реального значения теоретическому выполняется с помощью одного из статистических критериев согласия.

Проверка независимости

Проверка независимости проводится на основе вычисления корреляционного момента.

Напомним, что две случайные величины *a* и *b* называются независимыми,

если закон распределения каждой из них не зависит от того, какое значение приняла другая. Для независимых случайных величин корреляционный момент равен нулю.

Для оценки независимости элементов последовательности поступают следующим образом.

Вводят в рассмотрение дополнительную последовательность Y, в которой $y_i = x_i + t$, где t – величина сдвига последовательности Y относительно исходной последовательности X.

Вычисляют коэффициент корреляции случайных величин X и Y, для чего используются специальные расчетные соотношения.

Раздел 3. Шифрование информации

Тема 9. Потоковые системы шифрования

Ярмолик В.Н., Портянко С.С., Ярмолик С.В. Криптография, стеганография и охрана авторского права. – Минск: Издательский центр БГУ, 2007. – 242с. (*Глава 5.*)

Характерной особенностью потоковых шифраторов является использование при их реализации криптографических ключей большой размерности часто равной длине исходного текста.

Различают два типа потоковых шифров: ***синхронные*** (***synchronous***) и ***самосинхронизирующиеся*** (***self-synchronizing***)

шифраторы. В первом случае последовательность значений ключа является независимой от открытого текста и шифротекста. Понятие синхронный шифратор связано с тем, что для успешного дешифрования требуется синхронизация между последовательностью значений ключа и зашифрованным текстом. В то время как для самосинхронизирующихся шифраторов значения ключа зависят либо от исходного текста, либо от шифротекста. Обобщенная структура синхронного потокового шифратора приведена на рис.9.1.

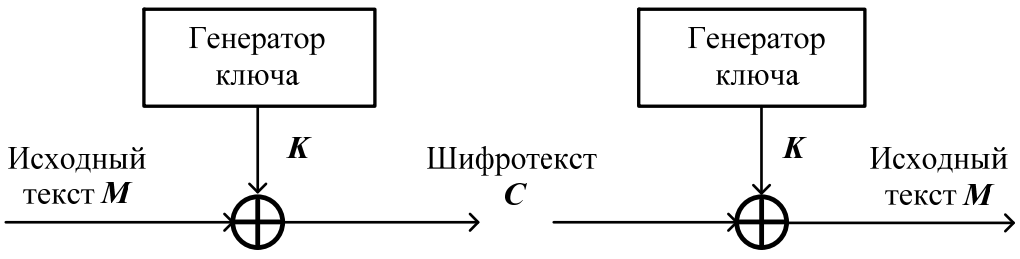


Рис.*9.1. *Структурная схема синхронного потокового шифратора

Подобная структура шифратора позволяет достичь высокой помехозащищенности и надежности функционирования. Действительно, искажение од-

ного символа в зашифрованном тексте приведет к искажению только одного символа исходного текста при дешифровании. В то же время нарушение синхронизации является причиной искажения всех последующих символов исходного текста с момента потери синхронного формирования ключа с шифротекстом. Подобная ситуация может возникнуть при удалении либо добавлении символов шифротекста.

С целью восстановления синхронизации используются самосинхронизирующиеся потоковые шифраторы. При их использовании ошибочно добавленный или удаленный бит вызывает только ограниченное количество ошибочных символов в дешифрованном тексте, после чего правильный текст восстанавливается.

Различают принципиальные отличия между симметричными блочными шифраторами (DES, IDEA, BLOWFISH, …) и потоковыми системами шифрования.

1.Блочный шифр одновременно шифрует целый блок данных, как правило, 64 бита, а в потоковых шифраторах процедура шифрования и дешифрования выполняется поразрядно.

2.В блочном шифре каждый бит зашифрованного текста является сложной функцией открытого текста, и криптографического ключа. В потоковом шифраторе (см. рис.9.1) каждый бит шифротекста получается в результате поразрядной операции сложения по модулю два очередного бита открытого текста и бита ключа.

3.Блочный шифр не требует задания начального вектора для генератора последовательности символов ключа, а в потоковом шифраторе такой вектор необходим.

4. Блочные шифры (подобно как DES) используется в коммерческом секторе, а потоковые шифры используется для специальных приложений и, как правило, широко не обсуждаются.

Основная составная часть любого потокового шифратора включает в себя алгоритм генерирования последовательности символов криптографического ключа. По сути, проблема создания эффективных симметричных блочных алгоритмов шифрования в случае потоковых шифраторов перенесена на проблему создания эффективных генераторов последовательностей ключа. Таким образом, генерирование непредсказуемых, как правило, двоичных последовательностей большой длины является одной из важных проблем классической криптографии. Для решения этой проблемы широко используются генераторы двоичных псевдослучайных последовательностей.

Обычно для генерирования последовательностей псевдослучайных чисел применяют аппаратные либо чаще всего программные генераторы, которые, хотя и называются генераторами случайных чисел, на самом деле формируют числовые последовательности, которые по своим свойствам только очень похожи на случайные числа. Наиболее часто используемым алгоритмом генерирования псевдослучайных чисел в различных приложениях является ***линейный конгруэнтный генератор***, описываемый рекуррентным соотношением

$$x_{t+1}=(ax_t+c) \bmod N,$$

где параметр x_0 называется начальным значением (вектором), величина $a \neq 0$ – множителем, параметр c – приращением, а значение N (мощность алфавита) – модулем. В случае $c=0$ приведенное соотношение определяет ***мультипликативный конгруэнтный генератор***, а в случае $c \neq 0$ – ***смешанный конгруэнтный генератор***. Существует большое множество конкретных реализаций подобного генератора в разнообразных приложениях. Так, мультипликативный конгруэнтный генератор для персональных компьютеров Pentium с 32-разрядной архитектурой Intel использует модуль $N=2^{31}-1=2147483647$. При этом рекомендуемыми значениями множителя a являются следующие числа: 16807, 630360016, 1078318381, 1203248318, 397204094, 2027812808, 1323257245, 764261123, 112817.

Наиболее часто используемые генераторы псевдослучайных чисел приведены ниже.

1. $x_{t+1}=(1176x_t+1476x_{t-1}+1776x_{t-2}) \bmod (2^{32}-5)$;
2. $x_{t+1}=(2^{13}(x_t+x_{t-1}+x_{t-2})) \bmod (2^{32}-5)$;
3. $x_{t+1}=(1995x_t+1998x_{t-1}+2001x_{t-2}) \bmod (2^{32}-849)$;
4. $x_{t+1}=(2^{19}(x_t+x_{t-1}+x_{t-2})) \bmod (2^{32}-1629)$;
5. $x_{t+1}=(5115x_t+1776x_{t-1}+1492x_{t-2}+2111111111x_{t-3}+c_t) \bmod 2^{32}$,
   $c_t=\lfloor(5115x_{t-1}+1776x_{t-2}+1492x_{t-3}+2111111111x_{t-4}+$
   $+c_{t-1})/2^{32}\rfloor$;
6. *COMBO*: $z_n=(x_n+y_n) \bmod 2^{32}$,
   $x_n=(x_{n-1}*x_{n-2}) \bmod 2^{32}$,
   $y_n=(30903y_{n-1}+c_n) \bmod 2^{16}$,
   $c_n=\lfloor(y_{n-1}+c_{n-1})/2^{16}\rfloor$

Характерной особенностью приведенных алгоритмов генерирования псевдослучайных чисел является необходимость задания начальных значений, которые могут быть использованы как сеансовые криптографические ключи.

Определяющим недостатком конгруэнтных генераторов, используемых для поточного шифрования, является их сложность реализации, которая заключается в необходимости выполнения операции умножения. Кроме того, данный класс генераторов является достаточно хорошо изученным и в силу этого легко поддающимся прогнозированию и соответственно взлому криптосистемы.

Генераторы M-последовательностей являются одними из наиболее часто используемых типов генераторов псевдослучайных последовательностей, применяемых в различных приложениях. Для их реализации используется сдвиговый регистр с линейной обратной связью (***Linear Feedback Shift Register – LFSR***). Подобный генератор часто используется как источник криптографиче-

ского ключа для потоковых систем шифрования. Одной из основных причин широкого применения генераторов *M*-последовательностей является сравнительно простая их реализация, как программная, так и аппаратная.

Как отмечалось в предыдущих разделах, генераторы *M*-последовательностей (LFSR) являются весьма эффективным средством для генерирования криптографического ключа. Обычно для этих целей можно использовать один LFSR или несколько подобных структур, желательно описываемых различными порождающими полиномами. Если длины формируемых последовательностей являются взаимно простыми и порождающие полиномы – примитивными, оказывается возможным получить последовательность бит ключа максимальной длины. ***Сеансовым ключом*** (ключом, используемым для шифрования одного сообщения) может быть начальное состояние используемых LFSR.

Очередной бит ключа формируется в соответствии с некоторой функцией, как правило, нелинейной. Эта функция называется ***комбинированной функцией*** (***combining function***), а сами генераторы ключа ***комбинированными генераторами*** (***combination generators***).

Одним из наиболее очевидных решений при построении комбинированных генераторов ключа является ***генератор Геффи*** (***Geffe generator***). В основе данного генератора лежит использование трех *LFSR* и двухвходового мультиплексора как показано на рис.9.2.

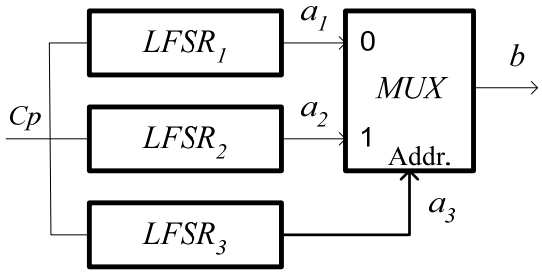


Рис.9.2. Комбинированный генератор Геффи

Два генератора *M*-последовательностей $LFSR_1$ и $LFSR_2$ являются источниками последовательностей a_1 и a_2, подаваемых на информационные входы мультиплексора, а последовательность a_3, формируемая $LFSR_3$, подается на адресный вход мультиплексора. Все три LFSR функционируют синхронно при подаче синхронизирующих импульсов *Cp*. Выходная последовательность *b* формируется в соответствии с выражением $b=(a_2a_3)+(a_1(1\oplus a_3))$, а ее период равен наименьшему общему кратному периодов трех *M*-последовательностей, генерируемых $LFSR_1$, $LFSR_2$ и $LFSR_3$. Если их порождающие полиномы имеют степени $m1$, $m2$ и $m3$, то максимальный период последовательности *b* будет равен $(2^{m1}-1)\times(2^{m2}-1)\times(2^{m3}-1)$. Обычно $m1\neq m2\neq m3$, а их величины близки и, как правило, определяются размерностью сеансового ключа.

Возможны различные модификации генератора Геффи в части увеличения количества используемых LFSR, а также режимов их функционирования.

Идея модификации режимов функционирования LFSR используется в ***генераторе Бета-Пайпера*** (***Beth-Piper Stop-and-Go generator***). В отличие от предыдущего метода, используемые в нем LFSR синхронизируются различными сигналами. Для случая трех LFSR структурная схема такого генератора приведена на рис.9.3.

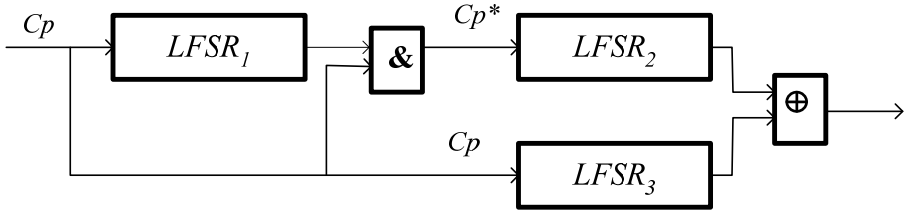


Рис.9.3.Генератор Бета-Пайпера

Как видно из приведенного рисунка, $LFSR_1$ управляет синхронизацией $LFSR_2$, а результирующая выходная последовательность определяется как сумма по модулю два выходных последовательностей, формируемых $LFSR_2$ и $LFSR_3$. Функционирование $LFSR_2$ определяет термин ***Stop-and-Go***. Действительно, в зависимости от выходного значения $LFSR_1$, данный блок ($LFSR_2$) выполняет микрооперацию сдвига либо нет, то есть, стоит либо идет. При выборе порождающих полиномов для трех LFSR генератора Бета-Пайпера руководствуются теми же требованиями, что и в случае генератора Геффи. Тогда оказывается возможным получение максимально возможной длины генерируемой последовательности.

Видоизмененная структура генератора Бета-Пайпера (***Alternating Stop-and-Go generator***) имеет большее распространение и также основана на использовании трех генераторов M-последовательностей $LFSR_1$, $LFSR_2$ и $LFSR_3$. Подобно как и в предыдущих случаях, M-последовательности, генерируемые тремя генераторами, имеют различные взаимно простые периоды. Как видно из рис.9.4, $LFSR_1$ управляет синхронизацией $LFSR_2$ и $LFSR_3$. Причем если один из них выполняет микрооперацию сдвига (идет), то второй в это время ничего не выполняет (стоит). Выходная последовательность, как и в предыдущих случаях, формируется как сумма по модулю два выходных последовательностей, формируемых $LFSR_2$ и $LFSR_3$.

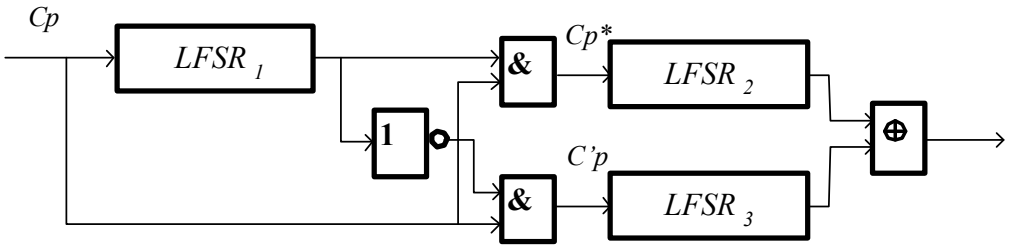


Рис.9.4. Альтернативный генератор Бета-Пайпера

Дальнейшим развитием идеи внесения нелинейных зависимостей по управлению синхронизацией является каскадная схема Голмана (***Gollmann Cascaded Key stream generator***), которая позволяет достигать желаемого качества в зависимости от количества используемых каскадов. Для случая трех каскадов подобный генератор приведен на рис.9.5.

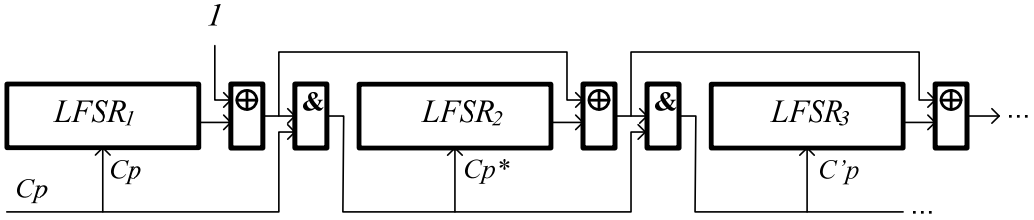


Рис.9.5. Каскадная схема Голмана

Основным достоинством приведенной схемы является возможность ее адаптации для достижения желаемого качества. Отметим, что синхронизация второго и последующих каскадов управляется предыдущими каскадами. Выполнение микрооперации сдвига либо ее отсутствие для конкретного LFSR зависит от состояний на выходах всех предыдущих генераторов *M*-последовательностей. В то же время необходимо отметить, что данная микрооперация выполняется для каждого LFSR с вероятностью 0,5.

Большое количество LFSR используется в так называемом ***пороговом генераторе*** (***Threshold generator***), приведенном на рис.9.6.

Количество (n) LFSR выбирается нечётным. Тогда нелинейная функция F формирует выходное значение 0 или 1 в зависимости от соотношения единичных и нулевых значений на выходах генераторов *M*-последовательностей.

Если количество единичных значений больше нулевых, на выходе генератора формируется единичное значение, в противном случае выходное значение равняется нулю.

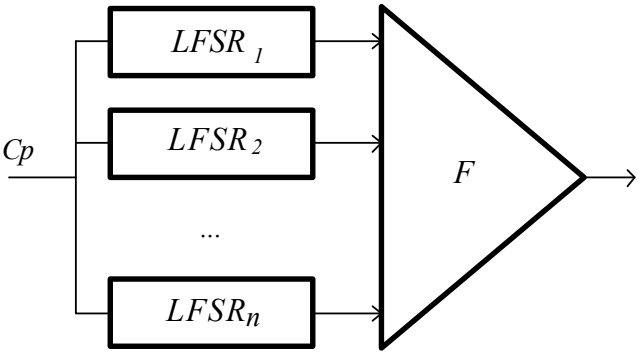


Рис.9.6. Пороговый генератор

Подобно как и в предыдущих случаях, периоды *M*-последовательностей должны быть взаимно простыми, а степени порождающих их полиномов иметь сравнимые величины и, как правило, превышающие 10.

Тема 10. Комбинированные потоковые шифрующие системы

Ярмолик В.Н., Портянко С.С., Ярмолик С.В. Криптография, стеганография и охрана авторского права. – Минск: Издательский центр БГУ, 2007. – 242с. (*Глава 5.*)

Различают ***синхронные потоковые шифраторы*** основаные на синхронном функционировании генераторов ключа у отправителя и получателя сообщения. Процедура синхронизации обеспечивается ***начальным состоянием*** (***seed***) генераторов псевдослучайных последовательностей криптографического ключа. В случае использования простейших генераторов ключа, таких как генераторы *M*-последовательностей типа LFSR, синхронизация достигается за счет установки регистров генераторов ключа у отправителя и получателя в одно и то же состояние. Начальное состояние генератора ключа часто интерпретируется как ***сеансовый ключ***, который используется только для одного сеанса передачи данных, а структура генератора – как ***долгосрочный ключ***, используемый для заданного временного промежутка использования криптосистемы. Структура синхронного потокового шифратора приведена на рис. 10.1.

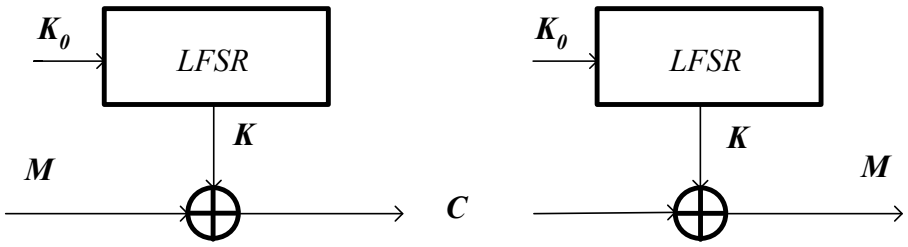


Рис.*10.1. *Синхронный потоковый шифратор

Здесь K представляет собой криптографический ключ, длина которого равняется длине шифруемого сообщения, а K_0 представляет собой сеансовый ключ.

Пример **10.1.** В качестве примера рассмотрим потоковый шифратор (рис.10.2.), использующий в качестве ключа *M*-последовательность, генерируемую в соответствии с примитивным порождающим полиномом $\varphi(x)=1+x+x^4$. Предположим, что начальное состояние генератора ключа K_0 при шифровании и при дешифровании сообщения принимает одно и то же значение равное 0010, что позволит нам достичь синхронной работы криптографической системы в целом. Отметим, что в данном случае сеансовым ключом может быть любой код из четырех символов кроме кода 0000. В соответствии с выбранным начальным состоянием обоих генераторов, у отправителя и получателя сообщения будет сгенерирована одна и та же последовательность ключа K=011110101100100….

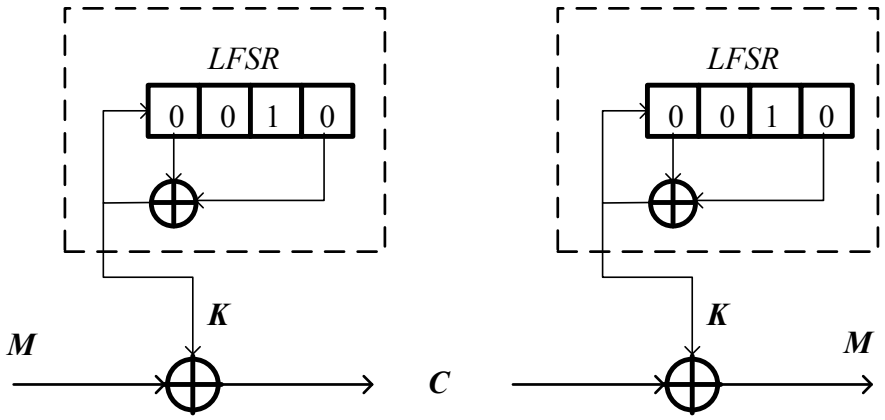


Рис.*10.2. *Синхронный потоковый шифратор, описываемый полиномом $\varphi(x)=1+x+x^4$

В качестве исходного текста используем двоичную последовательность M=101011111100001, тогда шифрование будет состоять из последовательного поразрядного сложения по модулю два. В результате получим:

$$\begin{array}{ll} M & = 101011111100001 \\ \oplus & \qquad\qquad \oplus \\ K & = 011110101100100 \\ \hline C & = 110101010000101 \end{array}$$

Получатель сообщения, получив зашифрованный текст C=11010101000 0101, выполнит обратное преобразование:

$$\begin{array}{ll} C & = 110101010000101 \\ \oplus & \qquad\qquad \oplus \\ K & = 011110101100100 \\ \hline M & = 101011111100001 \end{array}$$

К сожалению, многие генераторы псевдослучайных последовательностей криптографических ключей подвержены различного рада атакам и, зачастую, легко взламываются. В данном случае под взломом понимается получение третьей стороной структуры генератора криптографического ключа, то есть, так называемого долгосрочного ключа. В случае генератора M-последовательности, его структура однозначно описывается порождающим полиномом $\varphi(x)=a_0+a_1x+a_2x^2+\ldots+a_{m-1}x^{m-1}+a_mx^m$; $a_m=a_0=1$; $a_i\in\{0,1\}$, и для его взлома злоумышленнику необходимо получить $2m$ бит шифротекста, что почти всегда достижимо, и столько же бит соответствующего ему исходного текста, что тоже является реальной задачей. Тогда непосредственно взлом будет состоять в нахождении решения системы из m линейных уравнений, неизвестными в которой будут коэффициенты порождающего полинома, а коэффициентами – значения $2m$ бит последовательности ключа K. Такая система уравнений для общего случая имеет следующий вид

$$k_{m+1} = \alpha_1 k_m \oplus \alpha_2 k_{m-1} \oplus \alpha_3 k_{m-2} \oplus ... \oplus \alpha_{m-1} k_2 \oplus k_1;$$
$$k_{m+2} = \alpha_1 k_{m+1} \oplus \alpha_2 k_m \oplus \alpha_3 k_{m-1} \oplus ... \oplus \alpha_{m-1} k_3 \oplus k_2;$$
$$k_{m+3} = \alpha_1 k_{m+2} \oplus \alpha_2 k_{m+1} \oplus \alpha_3 k_m \oplus ... \oplus \alpha_{m-1} k_4 \oplus k_3;$$
$$\dots$$
$$k_{2m} = \alpha_1 k_{2m-1} \oplus \alpha_2 k_{2m-2} \oplus \alpha_3 k_{2m-3} \oplus ... \oplus \alpha_{m-1} k_{m+1} \oplus k_m.$$

Здесь коэффициент a_m так же как и a_0 для любого полинома равен единице.

***Пример* 10.2.** В качестве примера рассмотрим потоковый шифратор, приведенный на рис.10.2. Допустим, что злоумышленник получил доступ к $2m=2\times4=8$ битам исходного текста M=10101111 и шифротекста C=11010101. Тогда он сможет восстановить элементы ключа K, которые будут равны поразрядной сумме по модулю два исходного текста и шифротекста. Соответственно получим: $k_1=1\oplus1=0$, $k_2=0\oplus1=1$, $k_3=1\oplus0=1$, $k_4=0\oplus1=1$, $k_5=1\oplus0=1$, $k_6=1\oplus1=0$, $k_7=1\oplus0=1$, $k_8=1\oplus1=0$. Система из m=4 линейных уравнений имеет вид

$$k_5 = \alpha_1 k_4 \oplus \alpha_2 k_3 \oplus \alpha_3 k_2 \oplus \alpha_4 k_1;$$
$$k_6 = \alpha_1 k_5 \oplus \alpha_2 k_4 \oplus \alpha_3 k_3 \oplus \alpha_4 k_2;$$
$$k_7 = \alpha_1 k_6 \oplus \alpha_2 k_5 \oplus \alpha_3 k_4 \oplus \alpha_4 k_3;$$
$$k_8 = \alpha_1 k_7 \oplus \alpha_2 k_6 \oplus \alpha_3 k_5 \oplus \alpha_4 k_4.$$

Окончательно будем иметь $1=\alpha_1\oplus\alpha_2\oplus\alpha_3$; $0=\alpha_1\oplus\alpha_2\oplus\alpha_3\oplus1$; $1=\alpha_2\oplus\alpha_3\oplus1$ и $0=\alpha_1\oplus\alpha_3\oplus1$. Откуда получим $\alpha_2=0$, $\alpha_3=0$ и $\alpha_1=1$, что соответствует примененному полиному.

С целью увеличения качества шифрования можно использовать комбинированный подход применения классических симметричных и потоковых криптосистем.

***Пример* 10.3.** В качестве примера подобных решений рассмотрим случай совместного применения алгоритма шифрования DES и LFSR. В данном случае поток бит ключа формируется на выходе системы шифрования DES. Начальное значение ключа K_0 определяет исходное состояние LFSR и может использоваться как криптографический ключ для DES. Его значение удобно рассматривать как сеансовый ключ комбинированной криптосистемы, в которой очередные 64 бита ключа получаются при шифровании в соответствии с алгоритмом DES очередного состояния LFSR.

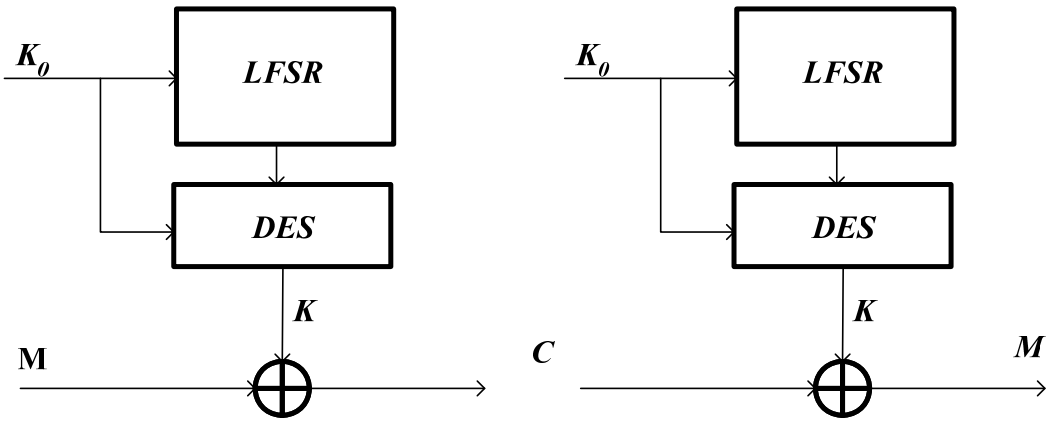


Рис.*10.3. *Комбинированная криптосистема

Могут быть использованы и другие комбинации классических криптографических систем и применения для них принципа потокового шифрования исходных сообщений.

Тема 11. Шифрования на базе эллиптических кривых

Ярмолик В.Н., Портянко С.С., Ярмолик С.В. Криптография, стеганография и охрана авторского права. – Минск: Издательский центр БГУ, 2007. – 242с. (*Глава 6.*)

Понятие эллиптических кривых является относительно новым понятием в криптографии, однако весьма активно обсуждаемым как один из кандидатов на последующее широкое применение для практических нужд защиты информации.

Эллиптические кривые (***Elliptic Curves***) в общем случае описываются кубическими уравнениями вида

$$y^2+axy+by=x^3+cx^2+dx+e,$$

где a, b, c, d и e представляют собой действительные числа, которые удовлетворяют достаточно простым требованиям. Определение эллиптических кривых включает понятие элемента O называемого ***бесконечным элементом*** (***infinite element***) или нулевым элементом (***zero element***). На следующих рисунках (11.1 и 11.2) приведены два примера эллиптических кривых.

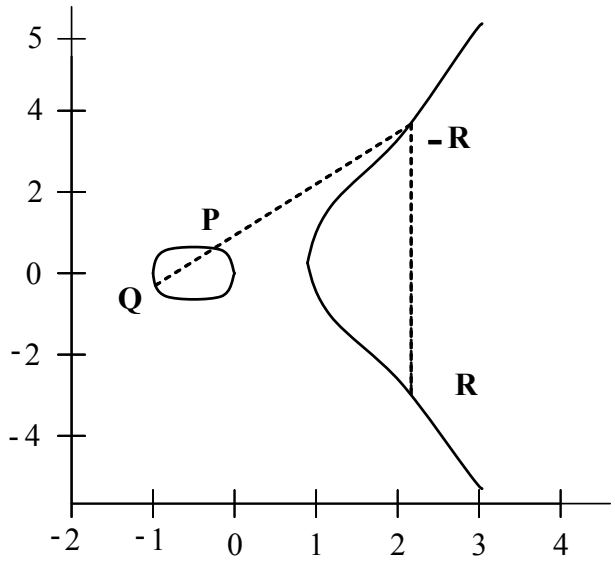


***Рис.*11.1. *Эллиптическая кривая, описываемая уравнением* $y^2=x^3-x$**

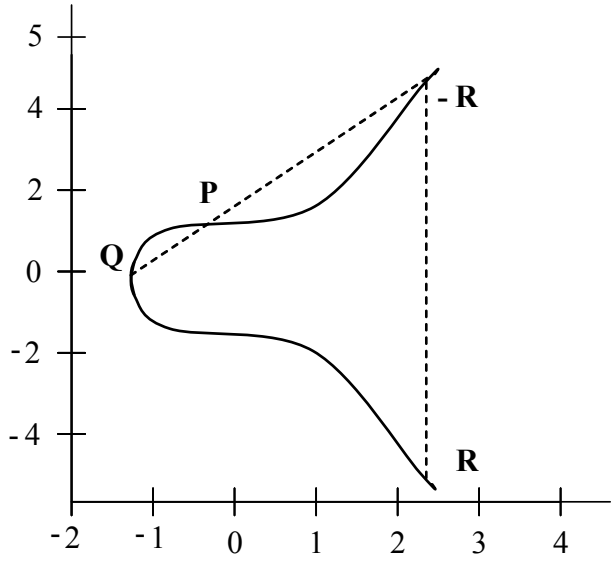


***Рис.*11.2. *Эллиптическая кривая, описываемая уравнением* $y^2=x^3+x+1$**

Эллиптические кривые над множеством действительных чисел определяются парами чисел (x,y) задающих координаты ***точек*** эллиптической кривой. На практике чаще всего применяются эллиптические кривые, описываемые уравнением

$$y^2=x^3+ax+b,$$

где x, y, a и b являются действительными числами.

Если для эллиптической кривой, описываемой уравнением $y^2=x^3+ax+b$, выполняется условие $4a^3+27b^2\neq 0$, тогда эллиптическая кривая $y^2=x^3+ax+b$ может быть использована для задания множества действительных точек, описываемых данной кривой. Это множество составляет группу точек, включающую также и специальную точку O, над которыми могут быть определены различ-

ные операции. Главным элементом эллиптической кривой является точка (***point***) $P=(x,y)$, определяемая ее координатами x и y, принадлежащими этой кривой. По определению различают точку $P=(x,y)$ эллиптической кривой и инверсную ей точку $-P=(x,-y)$.

Основополагающим свойством эллиптических кривых является следующее свойство.

Секущая прямая линия всегда пересекает эллиптическую кривую только в трех точках. Исключением является случай вертикальной секущей прямой, для которой третьей точкой является бесконечный элемент O.

Приведенное свойство позволяет определить операцию сложения двух точек эллиптической кривой. Предположим, что P и Q являются двумя различными точками эллиптической кривой и P не равняется $-Q$. Для сложения двух точек P и Q, проводится прямая линия через эти две точки. Согласно свойству эллиптических кривых, приведенному выше, эта линия пересечет эллиптическую кривую только в еще одной точке, называемой $-R$. Точке $-R$ всегда соответствует симметричная ей точка R (см. рис.11.1 и рис.11.2). Таким образом, операция сложения над группой, задаваемой точками эллиптической кривой, определяется как

$$P + Q = R.$$

Приведенное определение операции сложения является геометрической интерпретацией операции сложения точек эллиптической кривой.

Для случая, когда точка Q равняется $-P$, прямая линия является вертикальной линией, которая пересекается только в этих двух точках, поэтому в основном для этих целей и был определен бесконечный элемент (нулевой элемент) O. Тогда согласно определению, $P + (-P) = O$. Как результат данного равенства, $P + O = P$. Элемент O называется аддитивным элементом (***additive identity***) либо нулевым аддитивным элементом группы, определяемой эллиптической кривой. Все эллиптические кривые имеют аддитивный элемент. Более детально основные свойства операции сложения приведены ниже.

1.Элемент O является нулевым аддитивным элементом, для которого выполняется равенство $O=-O$. Тогда соответственно для любой точки P эллиптической кривой получим $P+O=P$.

2.Прямая линия, проведенная через точки P и $-P$ является вертикальной линией. Тогда согласно определению $P+(-P)=O$.

3.Для $P \neq Q$ и $P \neq -Q$ результатом сложения двух точек будет являться третья точка, принадлежащая эллиптической кривой, то есть $P+Q=R$.

4.В случае, когда $P = Q$, то есть когда точка P складывается сама с собой и $P \neq (x,0)$, в точке P проводится касательная линия к эллиптической кривой. Эта линия пересечет эллиптическую кривую только в одной точке $-R$. Тогда результатом удвоения точки P будет точка R симметричная точке $-R$.

Таким образом

$$P+P=2P=R.$$

5.Для случая, когда $P = (x,0)$, касательная к эллиптической кривой в этой точке будет представлять собой вертикальную линию. Тогда, согласно определению, для такой точки $2P = O$. Для случая вычисления $3P$ будем иметь $2P + P$.

Выполнив подстановку вместо $2P$ нулевого элемента O, окончательно получим $P + O = P$. Таким образом, $3P = P$. Для $P = (x,0)$ будут выполняться равенства $3P = P$, $4P = O$, $5P = P$, $6P = O$, $7P = P$ и так далее.

6.Операция умножения точки P на положительное целое число k определяется как сумма k точек P так, что $kP=P+P+P+...+P$.

Несмотря на то, что геометрическая интерпретация операции сложения точек эллиптических кривых является достаточно наглядной иллюстрацией арифметики эллиптических кривых, она не может быть использована для практической реализации вычислений. Для этих целей используется алгебраическое определение операции сложения двух точек $P=(x_P,y_P)$ и $Q=(x_Q,y_Q)$, которое однозначно вытекает из геометрической интерпретации этой операции.

1.Если P и Q являются различными точками эллиптической кривой и P не равняется $-Q$, тогда результатом операции сложения $P+Q=R$ является точка $R=(x_R,y_R)$, координаты которой вычисляются по следующим соотношениям:

$$s=(y_P-y_Q)/(x_P-x_Q);$$

$$x_R =s^2- x_P -x_Q;$$

$$y_R =-y_P+s(x_P –x_R).$$

2. Удвоение точки P будет осуществляться подобным образом:

$$s=(3x_P^2+a)/(2y_P);$$

$$x_R =s^2- 2x_P;$$

$$y_R =-y_P+s(x_P –x_R).$$

Величина a, используемая в последнем выражении, является одним из параметров эллиптической кривой, описываемой уравнением $y^2=x^3+ax+b$.

Очевидно, что даже аналитические соотношения, определяющие операцию сложения точек эллиптической кривой, не позволяют достичь высокой степени точности вычислений. Вычисления являются, как правило, медленными, а результат не точен, что неприемлемо для криптографии, где вычисления должны давать абсолютно точный результат, а процедура вычисления результата должна выполняться за реальное время.

Для практического применения в криптографии используются конечные поля (поля Галуа) $GF(M)$ и $GF(2^m)$. Так, например, конечное поле $GF(M)$ оперирует целыми числами от 0 до M-1, где M, как правило, представляет собой большое простое число, а все вычисления выполняются по модулю M.

Если в уравнении $y^2=x^3+ax+b$ параметры a и b принадлежат конечному полю $GF(M)$ и оно не содержит повторяющихся сомножителей ($4a^3 + 27b^2 \neq 0 \bmod M$), то эллиптическая кривая, описываемая таким уравнением, определяет эллиптическую группу $E_M(a,b)$ над полем Галуа $GF(M)$. Эллиптическая группа $E_M(a,b)$ также может быть описана как $y^2=x^3+ax+b \bmod M$. Она включает точки, принадлежащие эллиптической кривой $y^2=x^3+ax+b$, координаты которых являются элементами конечного поля $GF(M)$.

***Пример* 11.1**. Рассмотрим эллиптическую группу $E_M(a,b)=E_{23}(1,0)$, описываемую эллиптической кривой $y^2=x^3+x$ в конечном поле $GF(23)$.

Данной группе принадлежит точка (9,5) так как ее координаты удовлетворяют уравнению $y^2=x^3+x \bmod 23$. Действительно, подставив в уравнение $y^2 \bmod M = x^3 + x \bmod M$ значения x=9 и y=5, последовательно получим

$5^2 \bmod 23 = 9^3 + 9 \bmod 23$,

$5^2 \bmod 23 = 729 + 9 \bmod 23$,

$25 \bmod 23 = 738 \bmod 23$,

$2 = 2$.

Таким образом, точка (9,5) принадлежит эллиптической группе $E_{23}(1,0)$. Всего этой группе принадлежит 23 точки, и сама группа представляется как {*O* (0,0) (1,5) (1,18) (9,5) (9,18) (11,10) (11,13) (13,5) (13,18) (15,3) (15,20) (16,8) (16,15) (17,10) (17,13) (18,10) (18,13) (19,1) (19,22) (20,4) (20,19) (21,6) (21,17)}.

Графически эллиптическая группа $E_{23}(1,0)$ показана на рис. 11.3.

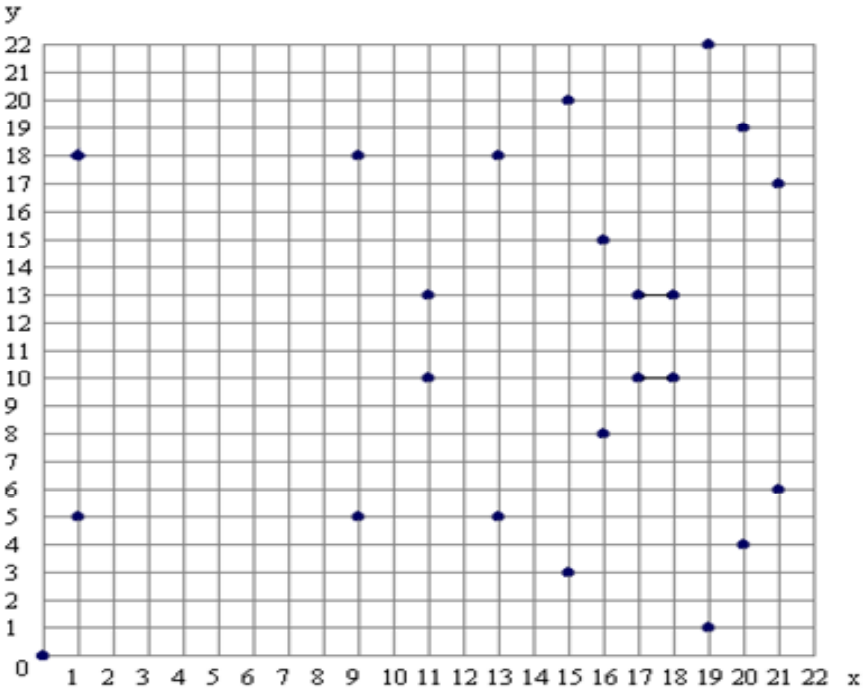


***Рис.*11.3. *Эллиптическая группа* E_{23}(1,0)**

Приведенный пример иллюстрирует структуру эллиптической группы в целом. Можно отметить две характерные черты эллиптической группы. Во-первых, их природа носит достаточно случайный характер, а во-вторых, не все целочисленные значения конечного поля *GF*(23), равно как и не все возможные точки с координатами, принадлежащими *GF*(23) входят в эллиптическую группу.

Общее количество точек эллиптической группы определяется в соответствии с утверждением Хоссего (***Hassego***):

$$M+1-2M^{1/2} \le \#E_M(a,b) \le M+1+2M^{1/2}.$$

Анализ последнего соотношения показывает, что для больших значений модуля M, количество точек эллиптической группы практически равняется величине M.

***Пример* 11.2**. В качестве второго примера рассмотрим эллиптическую группу $E_M(a,b)=E_5(0,1)$, описываемую эллиптической кривой $y^2=x^3+1$ в конечном поле $GF(5)$. Данная группа будет состоять из следующих точек $\{O, (0,1), (0,4), (2,2), (2,3), (4,0)\}$.

Для эллиптических групп также справедливы все свойства, которые касались эллиптических кривых для действительных чисел. Единственным отличием является выполнение всех операций по модулю M. Так, отрицательная точка $-P$ будет определяться как $-P = (x_P,-y_P \bmod M)$. Например, для эллиптической группы $E_5(0,1)$ и ее точки -(2,2) будем иметь $-(2,2)=(2,-2 \bmod 5)=(2,3)$.

Для эллиптических групп соотношения для определения координат точки R, являющейся суммой $P+Q$ двух исходных точек, определяется следующим образом.

1.Если P и Q являются различными точками эллиптической группы $E_M(a,b)$, и P не равняется $-Q$, тогда результатом операции сложения $P+Q=R$ является точка $R=(x_R,y_R)$, координаты которой вычисляются по следующим соотношениям:

$$s=(y_P-y_Q)/(x_P-x_Q) \bmod M;$$
$$x_R=s^2-x_P-x_Q \bmod M;$$
$$y_R=-y_P+s(x_P-x_R) \bmod M.$$

2.Удвоение точки P будет осуществляться подобным образом:

$$s=(3x_P^2+a)/(2y_P) \bmod M;$$
$$x_R=s^2-2x_P \bmod M;$$
$$y_R=-y_P+s(x_P-x_R) \bmod M.$$

Величина a является одним из параметров эллиптической группы $E_M(a,b)$ и принадлежит конечному полю $GF(M)$. Все вычисления выполняются в конечном поле по модулю M.

***Пример* 11.3.** Рассмотрим операцию сложения двух точек (0,1) и (2,2), принадлежащих эллиптической группе $E_5(0,1)$. В силу того, что $(0,1)\neq(2,2)$ и $(2,2)\neq-(0,1)$, используем первый случай для операции сложения. Тогда $s=(y_P-y_Q)/(x_P-x_Q) \bmod M=(1-2)/(0-2) \bmod 5$. Отсюда получим линейное уравнение $2s=1 \bmod 5$, решением которого является $s=3$. Далее значения координат результирующей точки $R=(x_R,y_R)$ вычисляются как $x_R=s^2-x_P-x_Q \bmod M=(3^2-0-2) \bmod 5=2$; $y_R=-y_P+s(x_P-x_R) \bmod M=-1+3(0-2) \bmod 5=-7 \bmod 5=3$. Окончательно получим точку $R=(x_R,y_R)=(2,3)$.

***Пример* 11.4.** Рассмотрим удвоение точки (2,2), принадлежащей этой же эллиптической группе $E_5(0,1)$. В силу того, что (2,2)=(2,2), используем второй случай для операции сложения. Тогда $s=(3x_P^2+a)/(2y_P) \bmod M=(3\times2^2+0)/(2\times2) \bmod 5=3$. Откуда значения координат точки $R=(x_R,y_R)$ вычисляются как $x_R=s^2-$

$2x_P \bmod M=(3^2-2\times2) \bmod 5=0$; $y_R=-y_P+s(x_P-x_R) \bmod M=-2+3(2-0) \bmod 5=4$. В результате получили точку $R=(x_R,y_R)=(0,4)$.

***Пример* 11.5.** Рассмотрим случай, когда $P=-P$. Возьмем точки (2,2) и (2,3), принадлежащие этой же эллиптической группе $E_5(0,1)$. Тогда в силу того, что (2,2)=-(2,3), результатом будет точка O. Формально это значение получается при вычислении значения $s=(y_P-y_Q)/(x_P-x_Q) \bmod M = (2-3)/(2-2) \bmod 5=\infty$. Тогда (2,2)+(2,3)=$O$.

Таблица 11.1 для операции сложения точек эллиптической группы $E_5(0,1)$ содержит результаты сложения всевозможных пар точек данной группы. Используя данную таблицу легко выполнять основные операции для эллиптических точек эллиптической группы $E_5(0,1)$.

***Таблица* 11.1.**

Операция сложения точек $E_5(0,1)$

+	O	(0,1)	(0,4)	(2,2)	(2,3)	(4,0)
O	O	(0,1)	(0,4)	(2,2)	(2,3)	(4,0)
(0,1)	(0,1)	(0,4)	O	(2,3)	(4,0)	(2,2)
(0,4)	(0,4)	O	(0,1)	(4,0)	(2,2)	(2,3)
(2,2)	(2,2)	(2,3)	(4,0)	(0,4)	O	(0,1)
(2,3)	(2,3)	(4,0)	(2,2)	O	(0,1)	(0,4)
(4,0)	(4,0)	(2,2)	(2,3)	(0,1)	(0,4)	O

Более сложные вычисления необходимо выполнить для эллиптических групп большей размерности.

***Пример* 11.6.** Предположим, имеем эллиптическую группу $E_{23}(9,17)$, описываемую эллиптической кривой $y^2=x^3+9x+17 \bmod 23$. Нетрудно убедится, что точка $P=(16,5)$ принадлежит данной группе. Действительно, $y^2 \bmod 23 =5^2 \bmod 23=2$ и $16^3+9\times16+17 \bmod 23=2$.

Вначале вычислим $2P=P+P$. Для этого последовательно определим $s=(3x_P^2+a)/(2y_P) \bmod M=(3\times16^2+9)/(2\times5) \bmod 23$. Как результат вычислений получим линейное сравнение $10s=18 \bmod 23$, решением которого в свою очередь будет $s=18\times10^{\varphi(23)-1} \bmod 23 = 18\times10^{21} \bmod 23 =11$.

Тогда $x_R=s^2-2x_P \bmod M=(11^2-2\times16) \bmod 23=20$ и $y_R=-y_P+s(x_P-x_R) \bmod M=-5+11(16-20) \bmod 23=-49 \bmod 23= -3 \bmod 23=20$. Как результат получим новую точку $Q=2P=(20,20)$.

Для вычисления $R=P+Q=3P$ последовательно выполним вычисления $s=(y_P-y_Q)/(x_Q-x_P) \bmod M=(5-20)/(20-16) \bmod 23$. Затем $4s=-15 \bmod 23 =8 \bmod 23$ и $s=8\times4^{\varphi(23)-1} \bmod 23 = 8\times4^{21} \bmod 23 = 2$.

Тогда, $x_R=s^2-x_P-x_Q \bmod M = (22-16-20) \bmod 23 = -9 \bmod 23 = 14$ и $y_R=-y_P+s(x_P-x_R) \bmod M = -5+2(14-16) \bmod 23 = -9 \bmod 23 = 14$. В результате получим точку $R=3P=(14,14)$.

Основной операцией, выполняемой при реализации криптографических алгоритмов, является операция вычисления скалярного произведения kP, которая реализуется путем последовательного выполнения операции сложения $P+P+P+\ldots$.

При последовательном выполнении подобного сложения на каждом шаге будет получаться результирующая точка, которая также должна принадлежать исходной эллиптической группе. Если операция последовательного добавления точки P к уже полученной сумме $P+P+P+...$ достаточно продолжительна то в силу того, что эллиптическая группа содержит конечное множество точек, наступит такой момент, что для некоторых целых k и l ($l>k$) наступит равенство $kP=lP$. Из последнего утверждения следует, что для некоторого целого $c=l-k$ будет выполняться равенство $cP=O$. Наименьшее значение c, для которого выполняется равенство $cP=O$, называется ***порядком*** точки P. Для эллиптической группы, рассмотренной в примере 6.9, результаты последовательного добавления $P+P+P+...$приведены в таблице 11.2.

***Таблица* 11.2.**

Результат выполнения операции P+P+P+… для E_5(0,1)

+	(0,1)	(0,4)	(2,2)	(2,3)	(4,0)
P	(0,1)	(0,4)	(2,2)	(2,3)	(4,0)
$2P$	(0,4)	(0,1)	(0,4)	(0,1)	O
$3P$	O	O	(4,0)	(4,0)	(4,0)
$4P$	(0,1)	(0,4)	(0,1)	(0,4)	O
$5P$	(0,4)	(0,1)	(2,3)	(2,2)	(0,4)
$6P$	O	O	O	O	O

Как видно из приведенной таблицы, у каждой точки различный порядок. Так, точки (0,1) и (0,4) имеют порядок $c=3$, точки (2,2) и (2,3) – порядок $c=6$, а точка (4,0) имеет $c=2$.

В реальных криптографических приложениях модуль M эллиптической группы выбирается как большое простое число, содержащее сотни знаков. Кроме того, весьма важным элементом является так называемая генерирующая точка G, порядок которой (c) должен являться также большим простым числом. В качестве примера для описания эллиптической группы возьмем эллиптическую кривую $y^2=x^3+10x+10$.

***Пример* 11.7.** На базе кривой $y^2=x^3+10x+10$ и модуля $M=23$ построим эллиптическую группу $E_{23}(10,10)$ с генерирующей точкой $G=G(x,y)=G(5,1)$. Результаты скалярного произведения генерирующей точки G на целые значения величины k приведены в таблице 11.3.

***Таблица* 11.3.**

Результаты выполнения операции kG для E_{23}(10,10)

k	1	2	3	4	5	6	7	8	9
kG	(5,1)	(8,21)	(11,5)	(10,11)	(12,8)	(7,20)	(15,19)	(9,1)	(9,22)
k	10	11	12	13	14	15	16	17	18
kG	(15,4)	(7,3)	(12,15)	(10,12)	(11,18)	(8,2)	(5,22)	(O)	(5,1)

Для выбранной генерирующей точки $G(5,1)$ ее порядок равняется 17, так как 17 является минимальным целым простым числом, для которого выполняется равенство $cP=O$.

Основой всех существующих криптографических приложения является проблема дискретного логарифма. В случае эллиптических групп эта проблема имеет следующую формулировку.

Для заданных двух точек P и Q эллиптической группы найти такое целое положительное число k, для которого выполняется равенство $kP=Q$. Величина k называется дискретным логарифмом от Q по основанию P.

Эллиптическая открытая система распределения ключей (***public key distribution algorithm***), подобно, как и алгоритм, предложенный ***W.Diffie*** и ***M.Hellman*****,** предназначается для распределения ключей по открытому каналу. Стойкость предложенного алгоритма основывается на сложности вычисления дискретного логарифма над эллиптической группой $E_M(a,b)$. Рассмотрим пару взаимно инверсных преобразований:

$$Q=kG;$$
$$k=log_G Q \pmod M \text{ над группой } E_M(a,b),$$

где k – простое целое число, а G – генерирующая точка эллиптической группы $E_M(a,b)$. Вычисление точки Q как результата скалярного произведения целого положительного k на генерирующую точку G представляет собой достаточно простую вычислительную задачу. В то же время вычисление значения k на основании точек Q и G является вычислительно трудной задачей. Для больших значений M эта задача вычислительно неразрешима. На этом факте и основана рассматриваемая эллиптическая система распределения ключей. Злоумышленник при попытке получения значения ключа столкнется с проблемой дискретного логарифма над эллиптической группой $E_M(a,b)$.

Первоначально определяются параметры эллиптической группы $E_M(a,b)$. Для этого выбирается большое простое целое число M и определяются параметры a и b эллиптической кривой, на базе которых строится эллиптическая группа $E_M(a,b)$. Затем определяется генерирующая точка $G=G(x,y)$ таким образом, что значение c, для которого выполняется равенство $cG=O$, является большим целым простым числом. Параметры эллиптической группы $E_M(a,b)$ и значение координат генерирующей точки $G=G(x,y)$ являются открытыми параметрами, доступными для всех пользователей.

Сущность эллиптической открытой системы распределения ключей состоит в том, что два пользователя A и B используя открытый канал, обмениваются криптографическим ключом, который будет известен только им двоим. Для остальных пользователей этот ключ будет закрытым. Непосредственно реализация эллиптической открытой системы распределения ключей состоит из следующих этапов.

1.Пользователь A случайным образом выбирает целое число n_A, такое, что $n_A<M$, которое является аналогом секретного ключа пользователя A. Значение

числа n_A должен знать только пользователь A. Затем A генерирует свой открытый параметр (ключ) как результат скалярного произведения n_A на G, то есть выполняет вычисление точки $P_A=n_AG$, которая является точкой эллиптической группы $E_M(a,b)$.

2.Пользователь B повторяет аналогичные действия и в результате получает свой секретный параметр n_B и соответствующий ему открытый параметр P_B.

3.По открытому каналу пользователь A высылает в адрес пользователя B свой открытый параметр P_A, а пользователь B – свой параметр P_B в адрес пользователя A.

4.Пользователь A получает значение секретного ключа K как результат скалярного произведения $K=n_AP_B$. Аналогично пользователь B генерирует значение общего с пользователем A секретного ключа K как результат вычисления $K=n_BP_A$
. В результате оба пользователя получат одно и то же значение ключа K в силу того, что для скалярного произведения целого числа на точку эллиптической группы выполняется следующее равенство $n_AP_B=n_A(n_BG)=$ $=n_B(n_AG)=n_BP_A$.

***Пример* 11.8.** Рассмотрим пример реализации эллиптической открытой системы распределения ключей на базе эллиптической группы $E_{23}(10,10)$ с генерирующей точкой $G=G(5,1)$.

1.Пользователь A случайным образом выбирает целое число $n_A=2$, для которого выполняется неравенство $n_A=2<M=23$. Затем A генерирует свой открытый параметр как результат скалярного произведения n_A на G, то есть выполняет вычисление точки $P_A=n_AG=2(5,1)=(8,21)$, которая также является точкой эллиптической группы $E_{23}(10,10)$.

2.Пользователь B повторяет аналогичные действия и в результате получает свой секретный параметр $n_B=3$ и соответствующий ему открытый параметр $P_B=n_BG=3(5,1)=(11,5)$.

3.По открытому каналу пользователь A высылает в адрес пользователя B свой открытый параметр $P_A=(8,21)$, а пользователь B – свой открытый параметр $P_B=(11,5)$ в адрес пользователя A.

4.Пользователь A получает значение секретного ключа K как результат скалярного произведения $K=n_AP_B=2(11,5)=(7,20)$. Аналогично пользователь B генерирует значение общего с пользователем A секретного ключа K как результат вычисления $K=n_BP_A$
$=3(8,21)=(7,20)$. В результате оба пользователя получат одно и то же значение ключа $K=(7,20)$.

Для реализации эллиптической криптосистемы предварительно как и для произвольной криптосистемы с открытым ключом необходимо определить эллиптическую группу $E_M(a,b)$ и генерирующую точку этой группы G. При этом сама эллиптическая группа и ее генерирующая точка являются открытыми параметрами, которые доступны всем пользователям криптосистемы. Кроме того, для всех пользователей A, B, C,...генерируются их закрытые ключи $n_A,n_B,n_C,...$,

которые представляют собой случайные целые числа меньшие чем c, где c – порядок генерирующей точки G. Далее используя эллиптическую группу $E_M(a,b)$ и ее генерирующую точку G вычисляются соответствующие им открытые ключи $P_A=n_AG$, $P_B=n_BG$, $P_C=n_CG$,... . Каждый из пользователей держит в тайне свой секретный ключ и обеспечивает доступ всем пользователям к своему открытому ключу.

Непосредственно перед процедурой шифрования отправитель кодирует свое сообщение m точкой $P_m=P_m(x,y)$ эллиптической группы $E_M(a,b)$. Здесь значения координат (x, y) и определяют сообщение m. Предположим, что пользователь A решил выслать в адрес пользователя B сообщение закодированное точкой $P_m=P_m(x,y)$. Для этого он должен выполнить следующие действия.

1.Пользователь A случайным образом генерирует целое число $k<c$, где c представляет собой порядок генерирующей точки G.

2.Используя точку G эллиптической группы $E_M(a,b)$ пользователь A вычисляет скалярное произведение kG, результатом которого является точка, принадлежащая этой же эллиптической группе.

3.На основании открытого ключа P_B пользователя B пользователь A определяет точку kP_B группы $E_M(a,b)$.

4.Пользователь А вычисляет сумму P_m+kP_B двух точек P_m и kP_B эллиптической группы $E_M(a,b)$, полученных ранее.

5.Криптограмма $C_m=(kG, P_m+kP_B)$, которая высылается в адрес получателя сообщения, состоит из двух точек эллиптической группы.

Процедура дешифрования криптограммы $C_m=(kG,P_m+kP_B)$ на стороне пользователя B будет состоять из следующих этапов.

1.Используя первую точку kG эллиптической группы $E_M(a,b)$ пользователь B вычисляет скалярное произведения своего секретного ключа n_B на точку kG. В результате получает $n_B(kG)=kP_B$.

2.Далее, используя вторую точку P_m+kP_B криптограммы $C_m=(kG,P_m+kP_B)$, пользователь B выполняет операцию сложения $(P_m+kP_B)+(-kP_B)$ и в результате получает исходное сообщение, закодированное точкой P_m. Отметим, что $-P=-P(x,y)=P(x,-y)=P(x,M-y)$.

***Пример* 11.9.** Рассмотрим пример реализации эллиптической криптосистемы на базе эллиптической группы $E_{23}(10,10)$ с генерирующей точкой $G=G(5,1)$. Предположим, что секретный ключ пользователя B равен $n_B=3$, а соответствующий ему открытый ключ $P_B=n_BG=3(5,1)=(11,5)$. Кроме того, пользователь A решил выслать в адрес B закодированное в виде точки $P_m=(15,4)$ исходное сообщение m. В результате пользователь A выполнит следующие действия.

1.Пользователь A случайным образом генерирует целое число $k=7<c=17$.

2.Используя точку $G=(5,1)$ эллиптической группы $E_{23}(10,10)$ пользователь A вычисляет скалярное произведение $kG=7(5,1)=(15,19)$.

3.На основании открытого ключа $P_B=(15,4)$ пользователя B пользователь A определяет точку $kP_B=7(11,5)=(10,11)$.

4.Пользователь А вычисляет сумму $P_m+kP_B=(15,4)+(10,11)=(11,18)$.

5.Криптограмма C_m, которая высылается в адрес получателя сообщения, состоит из двух точек эллиптической группы C_m=((15,19),(11,18)).

При получении криптограммы C_m=((15,19),(11,18)) пользователь B последовательно выполняет следующие действия.

1.Используя первую точку kG=(15,19) эллиптической группы $E_{23}(10,10)$, пользователь B вычисляет скалярное произведение своего секретного ключа n_B=3 на точку kG=(15,19). В результате он получает $n_B(kG)$=3(15,19)=(10,11).

2.Далее, используя вторую точку P_m+kP_B=(11,18) криптограммы C_m, пользователь B выполняет операцию сложения (11,18)+(-(10,11))=(11,18) +(10,-11)=(11,18)+(10,12)=(15,4). Таким образом, пользователь B получает сообщение m закодированное точкой P_m=(15,4).

Достаточно интересной представляется криптосистема, предложенная *Менезесом* и *Ванстоуном* (***Menezes-Vanstone Elliptic Curve Cryptosystem***). Ее суть, так же как и в предыдущем случае, состоит на проблеме дискретного логарифма для циклических подгрупп эллиптических групп.

Исходными открытыми данными для данной криптосистемы является эллиптическая группа $E_M(a,b)$ и ее генерирующая точка G, для которой вычисляются соответствующие открытые ключи $P_A=n_AG$, $P_B=n_BG$, $P_C=n_CG$,... для всех пользователей. Закрытые ключи n_A,n_B,n_C,...пользователей A,B,C,… представляют собой случайные целые числа и являются закрытыми параметрами.

Аналогично как и в предыдущем случае, непосредственно перед процедурой шифрования отправитель A кодирует свое сообщение m точкой $P_m=P_m(x_1,y_1)$ эллиптической группы $E_M(a,b)$ перед высылкой его в адрес пользователя B. Здесь значения координат (x_1, y_1) и определяют сообщение m.

Непосредственно процедура шифрования состоит из следующих шагов.

1.Отправитель закодированного сообщения $P_m(x_1,y_1)$ генерирует случайное целое число $k<c$, где c – порядок генерирующей точки G.

2.Пользователь A вычисляет точку $kP_B=(x_2,y_2)$, используя открытый ключ пользователя B, принадлежащую эллиптической группе $E_M(a,b)$.

3.Пользователь A вычисляет дополнительную точку $C_0=kG$.

4.Пользователь A вычисляет два целых числа c_1 и c_2 по следующим соотношениям.

$$c_1=x_1x_2 \bmod M;\ c_2=y_1y_2 \bmod M.$$

5. Пользователь A высылает в адрес пользователя B шифрограмму $C_m=(C_0,c_1,c_2)$.

При получении шифрограммы $C_m=(C_0,c_1,c_2)$ пользователь B последовательно выполняет следующие действия.

1.Пользователь B умножает первую точку C_0 криптограммы C_m на свой секретный ключ n_B. В результате получает $n_BC_0=n_BkG=kP_B=(x_2,y_2)$

2.Пользователь B получает закодированное сообщение $P_m(x_1,y_1)$ путем решения линейных сравнений $c_1=x_1x_2 \bmod M$; $c_2=y_1y_2 \bmod M$, где неизвестными являются x_1 и y_1.

***Пример* 11.10.** Рассмотрим пример реализации эллиптической криптосистемы *Менезеса-Ванстоуна* на базе эллиптической группы $E_{23}(10,10)$ с генерирующей точкой $G=G(5,1)$. Предположим, что секретный ключ пользователя B

равен n_B=9, а соответствующий ему открытый ключ $P_B=n_BG$=9(5,1)=(9,22). Закодированное сообщение m пользователем A представляется в виде точки P_m=(12,15). Перед его отправлением в адрес пользователя B, пользователь A выполняет следующие действия.

1.Отправитель A генерирует случайное целое k=5.

2.Затем пользователь A вычисляет точку kP_B=5(9,22)=(7,3).

3.Пользователь A вычисляет дополнительную точку $C_0=kG$=5(5,1)=(12,8).

4.Пользователь A вычисляет $c_1=x_1x_2 \bmod M=12\times7 \bmod 23=15$ и $c_2=y_1y_2 \bmod M=15\times3 \bmod 23=22$.

5. И, наконец, пользователь A высылает в адрес пользователя B шифрограмму $C_m=(C_0,c_1,c_2)$=((12,8),15,22).

При получении шифрограммы $C_m=(C_0,c_1,c_2)$=((12,8),15,22) пользователь B последовательно выполняет следующие действия.

1.Умножает первую точку C_0=(12,8) криптограммы C_m на свой секретный ключ n_B. В результате получает n_BC_0=9(12,8)=(7,3).

2.Пользователь B получает закодированное сообщение $P_m(x_1,y_1)$ путем решения линейных сравнений $15=7x_1 \bmod 23$; $22=3y_1 \bmod 23$, где неизвестными являются x_1 и y_1. В результате $x_1=15\times7^{21} \bmod 23=12$, а $x_2=22\times3^{21} \bmod 23=15$.

Тема 12. Современные методы шифрования информации

Ярмолик В.Н., Портянко С.С., Ярмолик С.В. Криптография, стеганография и охрана авторского права. – Минск: Издательский центр БГУ, 2007. – 242с. (*Глава 6.*)

Развитие новых технологий для создания элементной базы ЭВМ и в первую очередь нанотехнологии, подходя к атомному пределу, неизбежно столкнется с квантовыми свойствами материи, которые до настоящего времени никак не использовались в сфере информационных технологий. Применение новых свойств материи будет означать радикальную трансформацию идеологии современной вычислительной техники, основанной на известном поведении сигнала (носителя информации), и четко определяемом состоянии носителя этого сигнала.

Из всех квантовых информационных технологий, которые должны прийти на смену существующим приложениям, ближе всего к созданию приложений, пригодных для использования в реальной жизни, подошла квантовая криптография.

Технология квантовой криптографии опирается на фундаментальное свойство природы известное в физике как ***принцип неопределенности Гейзенберга***, сформулированный в 1927г., который базируется на принципиальной неопределенности поведения квантовой системы. Невозможно одновременно получить координаты и импульс частицы, невозможно измерить один параметр фотона, не исказив другой.

Благодаря этому свойству возможно построение каналов передачи данных, защищенных от подслушивания. Отправитель кодирует отправляемые данные, задавая определенные квантовые состояния, а получатель регистрирует эти состояния. Затем получатель и отправитель совместно обсуждают результаты наблюдений. При передаче данных задается и контролируется поляризация фотонов.

Впервые идея квантовой криптографии была предложена в 1984г. ***Чарльзом Беннетом*** и ***Жиль Брассаром*****,** которые предложили использовать фотоны в криптографии для получения защищенного канала передачи информации. Для кодирования нулей и единиц они предложили использовать фотоны, поляризованные в различных направлениях, и разработали простую схему квантового распределения ключей шифрования, названную ими ***BB*84**. Позднее, в 1991г. идея была развита ***Экертом***.

Сущность предложенной ими идеи заключался в том, что импульс горизонтально поляризованных фотонов проходит через горизонтальный поляризационный фильтр. Если поворачивать фильтр, то поток пропускаемых фотонов будет уменьшаться до тех пор, пока угол поворота не составит 90 градусов. Тогда ни один фотон из горизонтально поляризованного импульса не будет пропускаться вертикально поляризованным фильтром. При повороте фильтра на 45^{o} он пропустит горизонтально поляризованный фотон с вероятностью 1/2.

Реализация идеи заключается в том, что отправителем исходного сообщения свет фильтруется, поляризуется и формируется в виде коротких импульсов малой интенсивности. Поляризация каждого импульса задается отправителем произвольным образом в соответствии с одним из четырех перечисленных состояний (горизонтальная, вертикальная, лево- или право-циркулярная).

Получатель сообщения измеряет поляризацию фотонов, используя произвольную последовательность базовых состояний (ортогональная или циркулярная).

В соответствии с законами квантовой физики, с помощью измерения можно различить лишь два ортогональных состояния: если известно, что фотон поляризован либо вертикально, либо горизонтально, то путем измерения, можно установить, как именно он поляризован. То же самое можно утверждать относительно поляризации под углами 45 и 135 градусов. Однако с достоверностью отличить вертикально поляризованный фотон от фотона, поляризованного под углом 45 градусов, невозможно.

Получатель открыто сообщает отправителю, какую последовательность базовых состояний он использовал. Далее отправитель открыто уведомляет получателя о том, какие базовые состояния использованы получателем корректно. Все измерения, выполненные при неверных базовых состояниях, не совпадающих у получателя и отправителя, отбрасываются. Измерения интерпретируются согласно двоичной схеме: лево-циркулярная поляризация или горизонтальная – 1, право-циркулярная или вертикальная – 0.

Очень важным достоинством передачи информации по такому каналу, образованному потоком световых импульсов, является невозможность перехвата сообщения, то есть его подслушивания.

Примером использования квантовой криптографии на практике является ***алгоритм генерирования*** двумя пользователями ***секретного ключа***.

Целью данного алгоритма является получение отправителем и получателем секретного ключа, который будет известен только им обоим.

Первоначально отправитель формирует последовательность фотонных импульсов. Каждый из импульсов случайным образом поляризован в одном из четырех направлений. Возможна вертикальная поляризация (|), горизонтальная (–), лево-циркулярная (\) и право-циркулярная (/). Например, отправитель посылает последовательность из девяти фотонных импульсов, поляризация которых задается, как | / / – \ | | – –.

Получатель настраивает свой детектор произвольным образом (случайно) на измерение серии либо диагонально (×), либо ортогонально (+) поляризованных импульсов (мерить одновременно и те и другие невозможно).

Предположим, получатель для измерения девяти фотонных импульсов использует поляризацию в следующей последовательности + × + + × × × + ×.

Предположим, что получатель верно определил поляризацию фотонов, тогда его регистрирующее устройство будет пропускать либо не пропускать импульс фотонов, что будет означать получение либо единичного символа сообщения, либо нулевого. Если же получатель неверно выбрал поляризацию фильтра, фотон всегда будет проходить либо не проходить через фильтрующее устройство с вероятностью 0,5, так как угол поляризации по отношению к поляризации фильтра составляет 45 градусов. И это не будет зависеть от того, какая поляризация использовалась при передаче исходного сообщения: та, которая соответствует единице, или та, которая соответствовала нулю.

Таким образом, при представлении исходного текста как последовательность равновероятных двоичных символов, результаты анализа будут представляться как случайная последовательность двоичных цифр.

Для примера, когда отправитель использовал поляризацию | / / – \ | | – –, а получатель + × + + × × × + ×, результаты алгоритма генерирования секретного ключа сведены в таблицу 12.1.

***Таблица* 12.1**

Алгоритм генерирования секретного ключа

Отправитель	|	/	/	–	\	|	|	–	–
Получатель	+	×	+	+	×	×	×	+	×
Результат измерений	0	0	0	1	1	0	1	1	0
Анализатор выбран правильно	Да	Да	Нет	Да	Да	Нет	Нет	Да	Нет
Ключ	0	0		1	1			1	

В графе «*Результаты измерения*» представлены результаты физического эксперимента фильтрации потока импульсов фотонов, а конкретные значения свидетельствуют о либо не прохождении импульса фотонов либо нет через конкретный фильтр. Эти результаты являются секретной данными.

По открытому каналу связи получатель сообщает отправителю, какие анализаторы им использовались (см. строку «*Получатель*»), но не сообщает, какие результаты им были получены.

В ответ отправитель по общедоступному каналу связи сообщает получателю, какие анализаторы он выбрал правильно. Эта информация представлена в строке «*Анализатор выбран правильно*» и может быть общедоступной. Те импульсы фотонов, для которых получатель неверно выбрал анализатор, отбрасываются и не принимаются для дальнейшего рассмотрения, а результаты измерения для правильно выбранных результатов могут быть использованы обоими пользователями для криптографического ключа, так как только отправитель и только получатель будут владеть этой информацией. Отправитель – в силу того, что он формировал поляризацию сам, а получатель – принимал, используя правильную поляризацию своего анализатора.

В среднем, в половине случаев из импульсов фотонов будет получен очередной бит ключа, а в остальных случаях информация будет носить случайный характер в силу неправильно выбранной поляризации анализатора и поэтому игнорируется обоими пользователями.

Если бы злоумышленник производил перехват информации при помощи оборудования, подобного оборудованию Боба, то примерно в 50 процентах случаев он выберет неверный анализатор, не сможет определить состояние полученного им импульса фотонов, и отправит импульс фотонов получателю в состоянии поляризации, выбранной наугад. При этом в половине случаев он выберет неверную поляризацию и, таким образом, примерно в 25 процентах случаев результаты измерений получателя могут отличаться от результатов отправителя.

Для обнаружения подобного перехвата отправитель и получатель выбирают случайный участок ключа и сравнивают его по общедоступному каналу связи. Если процент ошибок велик, то он может быть отнесен на счет злоумышленника. В этом случае предыдущий эксперимент полностью аннулируется, а процедура обмена секретным ключом повторяется сначала.

Раздел 4. Запутывающее преобразование информации

Тема 13. Элементы теории сложности

М.Г. Адигеев Введение в теорию сложности. Методические указания для студентов механико-математического факультета. – Министерство образования и науки Российской Федерации Государственное образовательное учреждение высшего профессионального образования «Ростовский Государственный Университет» 2004 г.

Сложность алгоритмов

Под алгоритмом обычно понимают четко определенную последовательность действий, приводящую через конечное число шагов к результату – решению задачи, для которой разработан алгоритм.

Основные свойства, присущие любому алгоритму:

1. массовость – алгоритм предназначен для решения задачи с некоторым множеством допустимых входных данных;

2. конечность – алгоритм должен завершаться за конечное число шагов (но это количество шагов может быть разным для разных входных данных).

Задачи могут быть сформулированы по-разному (дифференциальные уравнения, задачи на графах, задачи оптимизации и т.п.). Для того чтобы можно было строить единую теорию алгоритмов, необходимо свести разные формулировки задач к какому то «единому знаменателю». Например, можно считать, что задача сводится к вычислению некоторой функции $F: X \to Y$. Ясно, что в таком виде можно сформулировать любую задачу. Но для некоторых задач функция F может быть выражена неявно. Например, для задачи поиска минимума функции φ на отрезке [0,1] имеем: X = множество функций, Y = [0,1], $F(\varphi) = x^*$: $\varphi(x^*) = \min\{\varphi(x): x \in [0,1]\}$. Не для любой задачи можно построить алгоритм. Существуют алгоритмически неразрешимые задачи. Например: задача самоприменимости машины Тьюринга, задача об остановке алгоритма. Еще один важный пример алгоритмически неразрешимой задачи – автоматическое доказательство теорем. Но даже если существует алгоритм, решающий задачу, это еще не значит, что мы сможем этим алгоритмом воспользоваться на практике для решения реальных задач. Потому что алгоритм может требовать для своей работы слишком много ресурсов. Например, если решение задачи на самых современных компьютерах займет 10^{10} лет. Очевидно, такой алгоритм для нас бесполезен. Иными словами, недостаточно существования какого-нибудь алгоритма, должен существовать алгоритм, не требующий для своей работы слишком много ресурсов.

Определение 13.1. *Количественная характеристика потребляемых ресурсов, необходимых программе или алгоритму для работы* (*успешного решения задачи*) *– это и есть* сложность алгоритма.

Основные ресурсы: время (*временна́я сложность*) и объем памяти (*ёмкостная сложность*). Наиболее важной (критической) характеристикой является *время*.

Очевидно, что для разных экземпляров задачи (для разных входных данных) алгоритму может требоваться разное количество ресурсов. С каждым экземпляром x задачи Z связывается определенное число (реже – набор чисел) $|x|$, называемое *длиной* или *размером входных данных (размером задачи)*. Размер задачи – это объем входных данных, необходимых для задания всех параметров задачи. Однако для многих задач количество времени, необходимое для решениязадачи, зависит не только от *размера* входных данных, но и от самих данных. То есть для решения задачи для двух входных данных x и y одинакового размера ($|x|=|y|$) алгоритм может тратить разное время. Поэтому для получения оценок временной сложности в зависимости от размера задачи определяют ее как *максимальное* время, затрачиваемое алгоритмом для входных данных длины n:

$T(n)=\max\{T(x): |x|=n\}$.

Эта функция называется *сложностью* алгоритма *в худшем случае*. Во многих случаях более важной для практики характеристикой алгоритма является его *сложность в среднем*. Формально сложность в среднем определяется так:

$$T_{ср}(n)= \sum \quad T(x)p(x),$$

где $p(x)$ — вероятность появления входных данных x, а суммирование ведется по всем возможным входным данным размера n. К сожалению, только для небольшого количества задач (например, для задачи сортировки) удается найти естественный способ определения вероятностей для входных данных. Поэтому при оценке сложности алгоритма обычно рассматривают его сложность в худшем случае. Более того, обычно оценивают не точное значение функции сложности $T(n)$, а *порядок роста* этой функции, т.е. находят такую функцию $f(n)$, что $T(n) = O(n)$ при $n\to\infty$.

Классы сложности

В предыдущем параграфе мы ввели понятие «сложность алгоритма». Хотелось бы аналогичным образом определить и сложность **задачи** – например, как сложность самого эффективного (по времени или ёмкости) алгоритма, решающего эту задачу (для данных размера n). К сожалению, это невозможно. Доказано, что есть задачи, для которых *не существует* самого быстрого алгоритма, потому что любой алгоритм для такой задачи можно «ускорить», построив более быстрый алгоритм, решающий эту задачу. Это утверждение называют теоремой Блюма об ускорении. Если отвлечься от технических деталей, то упрощенный вариант теоремы Блюма можно сформулировать следующим образом:

Теорема 13.1. (теорема Блюма об ускорении). *Существует такая алгоритмически разрешимая задача Z, что любой алгоритм A, решающий задачу Z, можно ускорить следующим образом: существует другой алгоритм A*, также решающий Z и такой, что* $T_{A^*}(n) \le \log T_A(n)$ *для почти всех n.*

Теорема Блюма не утверждает, что ускорение возможно для *любой* задачи. Более того, в дальнейшем мы увидим, что для задач, интересных с практической точки зрения, ускорение не возможно; для таких задач существует оптимальный (самый быстрый) алгоритм. Тем не менее, утверждение теоремы Блюма о существовании «неудобных» задач не позволяет определить универсальное (применимое ко всем задачам) понятие «оптимального алгоритма».
Поэтому в теории сложности использован другой подход – через классы сложности.

Определение 13.2. *Пусть* $f(n)$ *– некоторая функция, отображающая* $\boldsymbol{N}$ *в* $\boldsymbol{N}$. Класс сложности $C(f(n))$ *– это множество всех задач, для которых существует хотя бы один алгоритм, сложность которого не превышает* $O(f(n))$.

Это определение в некотором смысле условно – обозначение $C(f(n))$ никогда не применяют. Почему? Потому что для задания реального класса задач необходимо еще уточнить: что мы понимаем под «алгоритмом»; какая сложность (временная, емкостная или какая-нибудь еще) нас интересует.

При разных ответах на эти вопросы получатся разные классы задач, и для каждого класса используется специальное обозначение. В теории сложности под «алгоритмом» обычно понимают ту или иную разновидность машины Тьюринга. Перейдем к рассмотрению различных типов алгоритмов (машин Тьюринга) и соответствующих им классов сложности.

Детерминированные алгоритмы

В данном параграфе мы рассмотрим простейший вариант машины Тьюринга — детерминированную МТ. Прилагательное «*детерминированная*» пока будем опускать — его значение будет объяснено позже, при рассмотрении *недетерминированных* машин.

Детерминированная одноклеточная машина Тьюринга

Определение 13.3. *Алфавитом называется произвольное непустое счетное множество. Обычно рассматривают конечные алфавиты. Элементы алфавита называются* символами *или* буквами. Словом *в алфавите A называется конечная последовательность букв из этого алфавита. Количество букв в слове x называется* длиной *слова и обозначается* $|x|$. A^* *= множество всех слов над алфавитом A.* A_k *= множество всех слов длины k.*

Определение 13.4. *Машина Тьюринга* (*МТ*) *— это четверка* $M = (Q, A, S, \Pi)$, *где A — «ленточный» алфавит (содержит специально выделенный символ* $\wedge$ *— «пробел»),* $Q = \{q_0, q_1, \ldots, q_m\}$ *— алфавит состояний,* $S = \{-1, 0, +1\}$ *— алфавит сдвигов, и* Π *— программа, представляющая собой отображение* $Q \times A \to Q \times A \times S$.

Формализовав таким образом интуитивное понятие «алгоритм», мы можем четко определить временную и емкостную сложность: $T_M(x)$ — количество шагов, сделанных машиной M при обработке входа x, $S_M(x)$ — количество ячеек на ленте, на которых побывала головка машины M при обработке входа x. Если на входе x машина зацикливается, то значения $T_M(x)$ и $S_M(x)$ не определены. После этого указанным выше образом определяются функции $T_M(n)$ и $S_M(n)$.

Рассмотренное определение задает простейшую модель алгоритма — *детерминированную однолентоную* машину Тьюринга.

Многоклеточная машина

Наше определение временной и ёмкостной сложности алгоритма опирается на простейшую модель МТ. В связи с этим возникают следующие вопросы. Насколько сильно временная и ёмкостная сложности зависят от вычислительной модели? Изменится ли сложность алгоритма, если мы расширим модель — скажем, «разрешим» машине Тьюринга иметь более одной ленты?

Введем новое определение.

Определение 13.5. *Пусть k целое число,* $k \geq 1$. *k*–ленточная машина Тьюринга *— это пятерка* $M = (k, A, Q, S, \Pi)$, *где* $\Pi : Q \times A_k \to Q \times (A \times S)_k$.

Принцип работы многоленточных машин в целом такой же, как и у одноленточных. Отличия связаны с количеством лент. Очередной шаг многоленточной машины определяется символами, расположенными в текущих ячейках на *всех* лентах, т.е. набором $(q,a_1,\ldots, a_k)\in Q\times A_k$. По этому набору определяются выполняемые действия: $(q', b_1, s_1,\ldots, b_k, s_k)\in Q\times(A\times S)_k$. Эти действия также производятся на всех лентах: на i-й ленте в текущую ячейку записывается символ b_i и головка смещается в соответствии с s_i. Обратите внимание на то, что головки на лентах перемещаются независимо друг от друга. Временная сложность многоленточной машины определяется точно так же, как и для одноленточной. В определении емкостной сложности есть небольшое изменение: емкостная сложность k-ленточной МТ М на входе x $S_M(x)$ определяется как *максимум* количества ячеек, на которых при обработке входа x побывали головки машины M на всех лентах. Очевидно, что это определение согласуется с данным ранее определением для k=1. В будущем нам потребуется следующее вспомогательное утверждение (его доказательство остается в качестве упражнения).

Теорема 13.2. *Для любой машины Тьюринга выполняется неравенство $S(n)\le T(n)$, т.е. емкостная сложность не превышает временную.*

Эквивалентность машин

Очевидно, что понятие k–ленточной МТ шире, чем понятие «обычной» одноленточной машины. Но насколько увеличилась «вычислительная мощь» машин Тьюринга за счет добавления лент? Более точно, нас интересуют следующие вопросы.

1. *Качественные* отличия: есть ли такие задачи, которые можно решать с помощью k–ленточных машин (при k>1), но нельзя решить с помощью одноленточной машины?

2. *Количественные* отличия. Очевидно, что за счет использования дополнительных лент можно получить выигрыш в скорости. Как сильно можно ускорить решение задачи, добавляя ленты? Для того чтобы ответить на эти вопросы, введем следующие определения.

Определение 13.6. (k+1)*–ленточная МТ M' с программой w* симулирует *k–ленточную машину M, если для любого набора входных слов* $(x_1, x_2, \ldots, x_k)$ *результат работы M' совпадает с результатом работы M на этих же входных данных. Предполагается, что вначале слово w записано на* (k+1)*-й ленте M'. Под результатом понимается состояние первых k лент МТ в момент остановки, а если на данном входе M не останавливается, то симулирующая ее машина также не должна останавливаться на данном входе.*

Определение 13.7. (k+1)*–ленточная МТ M* называется* универсальной машиной Тьюринга для k–ленточных машин*, если для любой k- ленточной машины M существует программа w, на которой M* симулирует M.*

Обратите внимание: в определении универсальной МТ одна и та же машина M' должна симулировать разные k-ленточные машины (на разных программах w).

Рассмотрим следующую теорему.

Теорема 13.3. *Для любого k≥1 существует универсальная* (k+1)*– ленточная машина Тьюринга*.

Доказательство. Теорему докажем конструктивно, т.е. покажем, как можно построить требуемую универсальную машину **M***. Рассмотрим лишь общую схему построения, опустив сложные детали. Основная идея заключается в том, чтобы на дополнительную (k+1)-ю ленту разместить описание симулируемой машины Тьюринга и использовать это описание в процессе симулирования.

Пусть $M = (k, Q, A, S, П)$ — произвольная k–ленточная машина Тьюринга. Не нарушая общности можно считать, что алфавит A содержит символы ‘0’, ‘1’ и вспомогательный символ ‘*’ (и, возможно, какие-то другие символы). Каждое состояние $q \in Q$ можно закодировать двоичным словом фиксированной длины r ($r \geq \log_2|Q|$). Очевидно также, что сдвиги $s \in S$ можно закодировать с помощью двух бит (т.е. двоичных слов длины 2). Рассмотрим одну ячейку из табличного представления программы машины M. Пусть эта ячейка соответствует состоянию q и обозреваемым символам a_i (на i-й ленте текущим символом является a_i, i=1,…,k); и пусть машина должна в этом случае перейти в состояние q' , напечатать символы b_i и выполнить смещения s_i (i=1,…,k). Такую ячейку можно закодировать словом

$$qa_1 a_2 \ldots a_k q' \; b_1 b_2 \ldots b_k s_1 s_2 \ldots s_k .$$

Это слово состоит из $2r+4k$ символов алфавита A. В дальнейшем каждое такое слово будем называть *блоком*. Закодируем таким образом все ячейки в программе машины M и запишем их в виде последовательности блоков на (k+1)-ю ленту машины.

Для того чтобы просимулировать поведение машины M, нам необходимо не только иметь программу этой машины, но и отслеживать ее текущее состояние. Текущее состояние машины M также будем записывать на (k+1)-ю ленту. Для того чтобы различать запись программы и запись текущего состояния, будем разделять их символом ‘*’. Например, слева от символа ‘*’ расположим слово α — закодированное представление текущего состояния машины M, а справа — слово π, представляющее собой закодированное представление программы этой машины.

Процедура моделирования машины M включает в себя следующие шаги (выполняемые машиной **M***):

1. Найти в слове π блок, соответствующий «текущему состоянию» α и текущим символам на первых k лентах.
2. На первых k лентах выполнить действия, задаваемые найденным блоком, т.е. изменить символы в текущих ячейках и сместить головки.
3. Изменить «текущее состояние» симулируемой машины, записав на (k+1)-й ленте слева от символа ‘*’ вместо слова α слово α' , считанное из найденного блока.
4. Проверить, является ли «состояние» α' заключительным. Если является, то завершить работу, иначе перейти к п.1. Очевидно, что каждый из шагов 1–4

можно «запрограммировать» в виде машины Тьюринга. Искомая универсальная машина **M*** представляет собой их композицию.

Доказанная теорема позволяет нам подойти к ответу на поставленные ранее два вопроса — убедиться, что количество лент у МТ не имеет принципиального значения ни с точки зрения вычислительной мощности, ни с точки зрения временной сложности. Напомню, что «принципиальное значение» для нас имеет только различие между полиномиальной и неполиномиальной (экспоненциальной) сложностью. Часто нас интересует только «выходное» значение, расположенное на последней ленте. В этом случае имеет смысл ослабить понятие симулирования.

Определение 13.8. *Будем говорить, что машина Тьюринга M вычисляет частичную функцию $f:A^*\to A^*$, если для любого $x\in A^*$, записанного на первую ленту машины M:*

- *если f(x) определено, то M останавливается, и в момент остановки на последней ленте машины записано слово f(x);*
- *если f(x) не определено, то машина M не останавливается.*

Определение 13.9. *Будем говорить, что машины M и M' эквивалентны, если они вычисляют одну и ту же функцию. Понятие эквивалентности «слабее», чем симулирование: если машина M' симулирует машину M, то машина M' эквивалентна M; обратное, вообще говоря, неверно. С другой стороны, для симулирования требуется, чтобы у M' было как минимум столько же лент, сколько и у M, в то время как для эквивалентности это не обязательно. Именно это свойство позволяет нам сформулировать и доказать следующую теорему.*

Теорема 13.4. *Для любой k-ленточной машины M, имеющей временную сложность T(n), существует эквивалентная ей однолenточная машина M' с временной сложностью $T'(n) = O(T^2(n))$.*

Доказательство. Сначала рассмотрим идею, лежащую в основе алгоритма моделирования. В отличие от задачи симулирования k-ленточной машины с помощью (k+1)-ленточной (Теорема 3), у машины M' имеется только одна лента, с помощью которой необходимо отслеживать состояние k лент моделируемой машины M. Для этого «упакуем» все k лент машины M в одну ленту машины M'. Под «упаковкой» будем понимать взаимно однозначное отображение ячеек всех лент машины M в ячейки ленты машины M' по следующему правилу: i-й ячейке j-й ленты M соответствует ячейка с номером $2(ki+j-1)$ на ленте M'. Таким образом, для хранения входных и промежуточных данных будут использоваться ячейки с четными номерами. В ячейках с нечетными номерами будем запоминать положение головок на лентах моделируемой машины: если у машины M головка на ленте j находится в ячейке i, то на ленте машины M' в ячейке $2(ki+j-1)$ стоит символ '1', иначе — пробел. При таком способе упаковки возникает одна чисто техническая проблема: на ленте машины M' внутри записанного слова окажутся пробелы, а это про-

тиворечит определению машины Тьюринга. Для того чтобы преодолеть эту проблему, введем дополнительный символ '' (отсутствующий в алфавите машины M) и будем использовать его вместо пробела внутри слова (в том числе и в ячейках с нечетными номерами — в качестве признака того, что соответствующая ячейка машины M не является текущей).*

Таким образом, работа машины M′ состоит из трех фаз:

1. Начальное преобразование входного слова x.

2. Моделирование работы машины M.

3. Завершающее преобразование результата.

Первую фазу мы уже рассмотрели.

Во время второй фазы машина M′ моделирует каждый шаг машины M. При этом на каждом шаге M′ за счет своих состояний помнит текущее состояние машины M и номер обрабатываемой ленты. За один проход по своей ленте машина M′ выясняет, какие символы считываются каждой головкой машины M и определяет, что необходимо сделать — в какое состояние нужно перейти, что записать на каждую из лент и как сместить головки на лентах. Все это M′ «запоминает» с помощью своего состояния. Затем M′ еще раз проходит по своей ленте и выполняет необходимые преобразования. Третья фаза алгоритма представляет собой преобразование «сжатия», обратное первой фазе, с той разницей, что теперь мы оставляем только символы, записанные на последней (k-й) ленте машины M.

Очевидно, что работающая по такому алгоритму машина M′ действительно эквивалентна машине M. Для завершения доказательства нам осталось оценить временную сложность M′. Фазы 1 и 3 требуют O(S(n)) шагов, где S(n) — емкостная сложность машины M. Во время фазы 2 машина M′ должна промоделировать T(n) шагов машины M, сложность моделирования каждого шага равна O(S(n)), потому что на каждом шаге M′ дважды сканирует все слово, записанное на ее ленте. Получаем, что временная сложность машины M′ равна O(T(n)S(n)), что с учетом теоремы 2 равно $O(T_2(n))$

Определение 13.10. DTIME($f(n)$) — *это класс задач, для каждой из которых существует детерминированная МТ, решающая эту задачу с временной сложностью* $O(f(n))$.

Напомним, что в строгой формулировке под решением задачи мы понимаем вычисление некоторой функции. Доказанные теоремы (об универсальной МТ и о моделировании произвольной машины с помощью одноленточной) обеспечивают универсальность данного определения (каждая алгоритмически разрешимая задача принадлежит некоторому классу DTIME($f(n)$)).

Определение 13.11. Определим также классы:

$P = PTIME = \bigcup_{k>0} DTIME(n^k)$ — *класс задач, разрешимых за* полиномиальное *время*.

EXPTIME = $\bigcup_{k>0}$DTIME(2^(n^k)) — *класс задач, решаемых за* экспоненциальное *время*. *Здесь выражение 2^x означает «2 в степени x».*

В силу теоремы 4 определение классов PTIME и EXPTIME не зависит от количества лент в машине. Класс P — это класс эффективно решаемых задач. Алгоритм, имеющий полиномиальную временную сложность, называется *эффективным*. Задача, для которой в настоящее время не известен эффективный алгоритм или для которой доказано отсутствие такого алгоритма, называется *труднорешаемой*.

Тема 14. Оценка сложности программного обеспечения

Оценка сложности программного обеспечения в области Web-программирования Оскаленко Д.А. Тамбовский филиал ОРАГС, г. Тамбов

Доля программного обеспечения связанного с работой в сети Интернет постоянно возрастает с ростом числа Web- приложений. Оценка сложности программного обеспечения напрямую связана с нормирование трудовых затрат на разработку ПО и, соответственною, с ценой разрабатываемого ПО.

Методы нормирования затрат на программную продукцию отличаются от затрат, сложившихся в традиционных отраслях. Метод – анализ статистических данных о фактически завершенных разработках, выявление факторов, определяющих разнообразие затрат, классификация этих факторов и предоставление пользователю нормативных материалов, возможности выбора наиболее близкого ему аналога и корректировки затрат, которые произошли при разработке аналога с помощью набора коэффициентов, учитывающих факторы разнообразия.

Стадии жизненного цикла ПО	Стоимостные затраты, %	Временные затраты
Разработка требований	10	2
Проектирование	10	5
Программирование	10	5
Отладка и внедрение	20	8
Эксплуатация и сопровождение	50	80

Разработка программного обеспечения в области Web-программирования связана с определенными коммерческими рисками. Проблемы рисков присутствуют в жизненном цикле информационных систем. Компьютеризация привела к возрастанию былой сложности создаваемых систем в тысячи раз. Понятно, что нынешняя сложность измеряется не столько количеством комплектующих элементов, сколько множеством вариантов функционального поведения системы в зависимости от внешней обстановки.

Системы становятся не просто технически сложными, но и "умными", в которой функции "мозгов" реализуются программным обеспечением (ПО). А за "мозги" принято платить. Непрерывно возрастает сложность Web-приложений. Появляются геоинформационные сервисы, платежные системы, системы «дополненной реальности». Возникает проблема оценки сложности и соответственно коммерческой цены создаваемых Web-приложений.

Нами предложена модель для оценки сложности программного ПО. На первом этапе определяется степень сложности разработки-3–4 группы сложности, по каждой из групп сложности заданы характеристики, которые позволяют отнести разработку к той или иной группе:

1 группа: (высшая) интеллект и языковой интерфейс, работа в режиме реального времени (процесс обработки сопоставим по времени с требованиями), режим работы телекоммуникационный, машинная графика (разработка элементов), реализация комплекса разработок.

2 группа: поисковые алгоритмы, оптимизационные расчеты, применение сложных математических методов, настройка на изменяющиеся внешние условия.

3 группа: не встречается ничего из вышеперечисленного.

Численность исполнителей, необходимая для выполнения работ по

стадиям проектирования и по комплексам задач (задаче) в целом, определяется по формуле:

$$Ч = \frac{Тот}{Фп}$$

где Ч - численность специалистов;

Тот– общая трудоемкость разработки проекта;

Фп - плановый фонд рабочего времени одного специалиста.

Расчет общей трудоемкости , разработки проекта (Тот) производится по формуле:

$$T_{от} = \sum_{1}^{n} H_{вр}$$

где Нвр - трудоемкость работ по стадиям проектирования (от 1 до n).

В случае применения нескольких коэффициентов общий поправочный коэффициент (Коб); определяется как произведение всех применяемых коэффициентов по следующей формуле:

$Коб = К^1 * К^2 * \ldots * К^n$,

где $К^1$, $К^2$,…,$К^n$ - поправочные, коэффициенты, учитывающие влияние качественных факторов на изменение затрат времени при выполнении конкретных стадии проектирования;

Коб - общий поправочный коэффициент. (i-го вида работы).

Трудоемкость работ (Нвр) с учетом общего поправочного - коэффициента определяется по следующей формуле:

$$Нвр = \sum_{1}^{n} Нврi * Коб$$

где Нврi - базисная норма времени, определенная по нормативной таблице;

Для расчетов коэффициентов предлагаем пользоваться следующими таблицами

Кэ	**Степень новизны**			**Стадии**
	1	**2**	**3**	
Ктз	0,11	0,10	0,09	техническое задание
Кэп	0,09	0,08	0,07	эскизное проектирование
Ктп	0,11	0,09	0,07	технического проектир.
Крп	0,55	0,58	0,61	рабочего проектирования
Квн	0,14	0,15	0,16	внедрения

По степени новизны классификация по трем группам.

Кн – коэффициент новизны.

А – принципиально новые разработки.

Б – развитие параметрического ряда ПО (в известной предметной области использовалась либо новая техника, либо новые программные средства).

В – Использование знакомых средств разработки в известной предметной области.

	А	**Б**	**В**
Кн	1÷1,7	1÷0,8	0,7

Использование типовых элементов в разработке.

Кт – коэффициент типовости.

Кт	**Степень применения типовых практических решений**
0,6	>60%
0,7	40–60%
0,8	20–40%
0,9	<20%
1,0	не использовались

Коэффициент сложности Ксл.

	Ксл
1. Связь с другими программными изделиями	0,08
2. Интерактивный режим	0,06
3. Ведение сложной структуры данных	0,07
4. Наличие нескольких характеристик сложности :	
– двух	0,12
– трех	0,18
– более трех	0,26

Разнесение трудоемкости по отдельным этапам разработки:

Используются коэффициенты Тэ=То Кэ

Таким образом, предложенный расчет сложности разрабатываемого ПО состоит из нескольких этапов:

1. Сложность комплекса программ проводится классификация программ по группам сложности (3 группы) и определяются признаки, позволяющие отнести разработку к конкретной группе сложности. (увеличение затрат труда в несколько раз, по сравнению с простейшей).

2. Необходимость использования компонент создаваемого ПО для других разработок, то есть ведется разработка типового ПО.

3. Использование типовых проектных решений (ТПР) при разработке ПО.

4. Использование передовых методов организации разработки. (Структурное программирование, использование формализованных методов при распределении ресурсов, нисходящее проектирование).

5. Уровень автоматизации разработки (использование достаточно современных инструментальных средств, например систем программирования, проблемно–ориентированных систем программирования, генераторы программ, использование удачного текстового редактора для подготовки текстов и документации, средства автоматизации для отладки программ).

6. Тематическая квалификация разработчика.

7. Технологическая квалификация разработчика (опыт использования технических и технологических средств, которые применяются в данной разработке, например: язык программирования, ОС).

8. Квалификация заказчика (опыт заказчика в формулировании технического задания на аналогичные программные продукты и опыт в эксплуатации).

Предложенная методика позволяет с оценки сложности ПО перейти к стоимостным оценкам, что позволяет разработчикам и заказчикам ПО бесконфликтно разрабатывать и внедрять Web-приложения.

Тема 15. Эквивалентное преобразование алгоритмов

Тема 16. Запутывающее преобразование программ

С.Ю.Петрик Запутывающее преобразование программного обеспечения с целью его защиты. Диссертация на соискание ученой степени кандидата технических наук. – Минск, БГУИР. – 2008г.

Из-за того, что нет разработанной модели запутывающего преобразования, нет и универсальных методов, которые имели бы доказанную секретность и могли бы применяться для алгоритмов, реализованных на любом языке программирования. Поэтому на сегодняшний день ведется разработка методов для конкретных представлений программы.

Арсенал методов запутывающего преобразования достаточно широк. По традиции, методы делятся на четыре группы в зависимости от объекта преобразования (Рисунок 16.1)

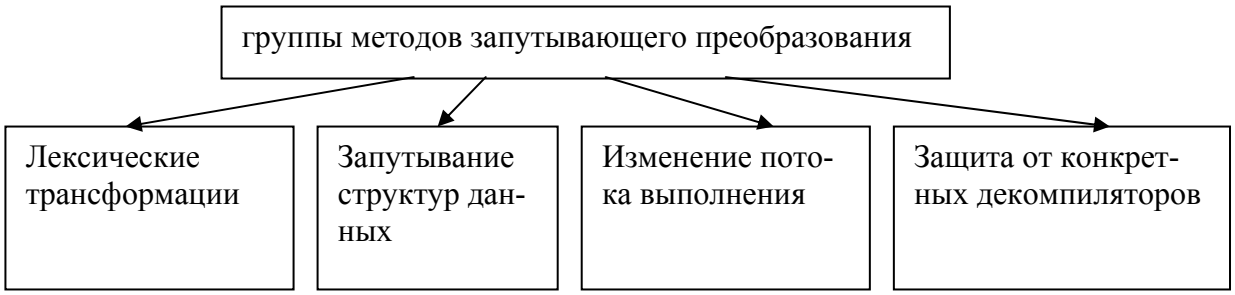


Рис. 16.1 – Группы методов запутывающего преобразования программ

Достаточно подробно методы запутывающего преобразования рассматриваются в ряде работ. Следует отметить, что хорошие результаты дает гибридное использование методов защиты программного обеспечения. И действительно запутывающее преобразование может достаточно надежно «спрятать» водяной знак или алгоритм взаимных проверок на целостность частей программы [**Ошибка! Источник ссылки не найден.**].

Рассмотрим представленные группы методов подробнее.

Лексические трансформации

Суть этого метода запутывания заключается в изменении словаря программы – использованных в ней идентификаторов.

Листинг 16.1– Исходный код до проведения запутывающего преобразования

```
private Hashtable getFrequency(string fileName)
{
    int readByte;
    FileStream fs = new
        FileStream(fileName,FileMode.Open,
                           FileAccess.Read,
                           FileShare.None);
    Hashtable frequency = new Hashtable();
    while ((readByte = fs.ReadByte()) != -1)
    {
        if (frequency[readByte] != null)
            frequency[readByte] =
                (int)(frequency[readByte]) + 1;
        else
            frequency.Add(readByte, 1);
    }
    fs.Close();
    return frequency;
}
```

В листинге 16.1 приведен один из методов класса, реализующего функции криптографического кодирования. Этот метод анализирует файл с именем fileName и на выходе возвращает хэш-таблицу frequency, содержащую коды всех символов, встреченных в файле, и их количество.

С.Ю.Петриком был разработан запутывающий кодер Obfuscation Studio. После его использования текст приведенной выше программы принял следующий вид:

Листинг 16.2 – Исходный код после проведения запутывающего преобразования

```
private Hashtable ll1lll11(string ll1llll1)
{
    int ll1lllll;
    FileStream ll1ll1ll =
        new FileStream(ll1llll1,FileMode.Open,
                                FileAccess.Read,
                                FileShare.None);
    Hashtable ll1ll1l1 = new Hashtable();
    while ((ll1lllll = ll1ll1ll.ReadByte()) != -1)
    {
        if (ll1ll1l1[ll1lllll] != null)
            ll1ll1l1[ll1lllll] =
                (int)(ll1ll1l1[ll1lllll]) + 1;
        else
            ll1ll1l1.Add(ll1lllll, 1);
    }
    ll1ll1ll.Close();
    return ll1ll1l1;
}
```

Видно, что код стал значительно более сложным для восприятия человеком (но не компьютером, подпрограмма так же компилируется и выполняется без видимых задержек. Для подтверждения этого приведенный участок программы в обоих представлениях был выполнен 10^7 раз, разницы во времени выполнения обнаружено не было).

У данного метода преобразования есть недостаток – можно выполнить автоматическое обратное преобразование. Для этого также существуют специальные средства. Однако, они только частично делают код понятней, так как очевидно, что первоначальные осмысленные имена вернуть уже невозможно.

Запутывание структур данных

Этот метод производит операции с данными: объединение нескольких скалярных переменных в одну, объединение массивов, разделение одного цельного массива на несколько отдельных и прочее.

Пример такого преобразования, также реализованного с помощью Obfuscation Studio, приведен ниже:

Листинг 16.3 – Исходный код до проведения запутывающего преобразования

```
class Source
{
    public Source()
    { }
    Random r = new Random();
    public void GetEquation()
    {
        int a, b, c;
        int D;
        double x1, x2;
        a = r.Next(10);
        b = r.Next(10) + 10;
        c = r.Next(10);
        D = ResolveX1X2(a, b, c, out x1, out x2);
        if (D >= 0)
            Console.WriteLine("x1=" + x1 +
                "; x2=" + x2);
        else
            Console.WriteLine("No real X found");
    }
    public int ResolveX1X2(int a, int b, int c,
        out double x1, out double x2)
    { //a*x*x + b*x + c = 0
        x1 = x2 = 0;
        int D;
        D = b * b - 4 * a * c;
        if (D >= 0)
        {
            x1 = (-b + Math.Sqrt(D)) / (2 * a);
            x2 = (-b - Math.Sqrt(D)) / (2 * a);
        }
        return D;
    }
}
```

Приведенная программа вычисляет значения корней квадратного уравнения, коэффициенты для которого сама же генерирует при помощи датчика псевдослучайных чисел. Преобразованный код ниже:

Листинг 16.4 – Исходный код после проведения запутывающего преобразования

```
class Test4
{
    public Test4()
    {
        FillVarArray();
    }
    object[] _1;
```

```
    private void FillVarArray()
    {
        _1 = new object[14];
        _1[0] = new Random();
    }

    public void GetEquation()
    {
        _1[1] = ((Random)(_1[0])).Next(10);
        _1[2] = ((Random)(_1[0])).Next(10) + 10;
        _1[3] = ((Random)(_1[0])).Next(10);
        _1[7] = _1[1];
        _1[8] = _1[2];
        _1[9] = _1[3];
        ResolveX1X2();
        _1[4] = _1[13];
        if ((int)_1[4] >= 0)
            Console.WriteLine("x1=" + _1[5] + "; x2=" + _1[6]);
        else
            Console.WriteLine("No real X found");
    }

    public void ResolveX1X2()
    { //a*x*x + b*x + c = 0
        _1[10] = _1[11] = 0.0;
        _1[12] = (int)_1[8] * (int)_1[8] - 4 * (int)_1[7] *
(int)_1[9];
        if ((int)_1[12] >= 0)
        {
            _1[10] = (-(int)_1[8] + Math.Sqrt((int)_1[12])) / (2 *
(int)_1[7]);
            _1[11] = (-(int)_1[8] - Math.Sqrt((int)_1[12])) / (2 *
(int)_1[7]);
        }
        _1[5] = _1[10];
        _1[6] = _1[11];
        _1[13] = _1[12];
    }
}
```

Данный метод собирает все поля класса и локальные переменные методов в один массив класса `object`, что позволяет хранить в нем данные любого типа.

Запутывание потока выполнения программы

Методы этого класса основаны на изменении программы таким образом, чтобы злоумышленнику было не очевидно, по какому алгоритму функционирует программа. Один из наиболее часто используемых методов – встраивание в код т.н. «темных» предикатов – ветвящихся конструкций. Пример такого внедрения:

Листинг 16.5 – Исходный код до проведения запутывающего преобразования

```
int method()
{ //a*x*x + b*x + c = 0
 D = b*b - 4*a*c;
 if (D > 0)
 {
   x1 = (-b + Math.Sqrt(D))/(2*a);
   x2 = (-b - Math.Sqrt(D))/(2*a);
 }
}
```

Метод решает квадратное уравнение при условии, что все используемые переменные объявлены как поля (или свойства) класса.

```
int method1()
{
 for (int i=1; i<4; i++)
 {
  if (i==3)
  {
   if (D > 0)
   {
    for (int j=1; j<8; j++)
    {
     if (j==6) x2 = x2 - D;
     if (j==4) x1 = x1 / (2*a);
     if (j==1) D = Math.Sqrt(D);
     if (j==7) x2 = x2 / (2*a);
     if (j==2) x1 = (-b);
     if (j==5) x2 = (-b);
     if (j==3) x1 = x1 + D;
    }
   }
  }
  if (i==2) D = D - 4*a*c;
  if (i==1) D = b*b;
 }
}
```

Запутывающее преобразование разбило математическую операцию на несколько более мелких и изменило порядок операций, встроив в код специальные предикат (переменную `j`), который определяет последовательность выполнения операторов.

Анализ методов запутывающего преобразования

Дальнейшие исследования понятия запутывающего преобразования привели к пониманию необходимости создания четкой теоретической основы этой науки.

Одной из основных проблем остается сложность оценки эффективности запутывающего преобразования и оценки возможности запутывания определенной программы.

Вводятся следующие понятия:

Действенность (potency) трансформации. Пусть трансформация T такова, что она при применении переводит программу P в представление P'. Величина $E(P)$ определяет сложность программы P по какой-либо метрике. Тогда мера возрастания сложности программы P после применения трансформации T будет определяться как

$$T_{potency} = E(P') / E(P) - 1 \qquad (16.1)$$

и трансформация считается действенной, если $T_{potency} > 0$

Другими словами, действенность трансформации определяет, насколько сложнее стало понять программу взломщику.

Устойчивость (resilience) трансформации. Пусть трансформация T такова, что она при применении переводит программу P в представление P'.

Под устойчивостью понимается комбинация двух мер:

а) затраты программиста – время, необходимое для создания автоматического обратного преобразователя (deobfuscator), который мог бы эффективно снизить действенность трансформации T;

б) затраты обратного преобразователя – время работы обратного преобразователя и затраченная при этом память для эффективного снижения действенности трансформации T.

Вводятся следующие уровни устойчивости в порядке возрастания: тривиальная, слабая, сильная, полная, однонаправленная.

Устойчивость трансформации обозначается $T_{resilience}$ и выражается следующим образом:

$T_{resilience}(P)$ = *однонаправленная,* если по P' невозможно восстановить P (обычно при удалении информации из P), иначе

$T_{resilience}(P) = Resilience(T_{затраты\ программиста}, T_{затраты\ обратного\ преобразователя})$, где функция Resilience определяется матрицей, рисунок 16.2. Трансформация называет межпроцессной, если она затрагивает взаимодействие между процессами, межпроцедурной – при влиянии на взаимодействие между отдельными процедурами, глобальной – если изменяет граф выполнения программы, локальной – если влияет на отдельные блоки программы.

Рисунок 16.2 – Устойчивость трансформации

Стоимость (cost) трансформации. Пусть трансформация T такова, что она при применении переводит программу P в представление P'. Тогда T_{cost} – увеличение времени выполнения и затраченного места программой P' относительно P. Стоимость может принимать следующие значения:

$$T_{cost} = \begin{cases} \textit{очень дорогая.} \text{ Если выполнение P' требует экспоненциально больше ресурсов, чем P.} \\ \textit{дорогая.} \text{ Если выполнение P' требует } O(n^p),\ p > 1 \text{ ресурсов} \\ \textit{дешевая.} \text{ Если требуется } O(n) \text{ ресурсов} \\ \textit{бесплатная.} \text{ Если требуется } O(1) \text{ ресурсов} \end{cases}$$

Для определения действенности трансформации необходимо использование метрик. Некоторое их количество предложено ранее, однако недостаточное для комплексного описания сложности программы. Поиск адекватных метрик – одно из направлений в исследовании запутывающего преобразования.

Еще одним открытым вопросом остается предсказание возможности запутывания программы до непосредственно преобразования. В работе проводится исследование оценки потенциальной возможности преобразования для одного из лексических методов – замены имен идентификаторов, для других групп методов запутывания исследования также ведутся.

Уместным видится сравнение запутывающего преобразования с другими методами защиты интеллектуальной собственности, например шифрованием. Делается вывод о том, что в отличие от шифрования, которое базируется на ма-

тематической модели и имеет доказанные уровни защищенности (proven security), запутывающее преобразование представляет собой набор методов, относящихся к категории «условной защиты» (fuzzy security). Так алгоритм шифрования RSA силен настолько, насколько слабы, на сегодняшний день, способы факторизации чисел. Методы же запутывания не дают никаких гарантий защиты, и знание алгоритма их действия приводит обычно к разработке противодействия (естественно, что это не относится к однонаправленным методам).

В дальнейшем было проведено исследование и доказано, что наряду с классом потенциально незапутываемых программ существует множество программ для которых запутывающее преобразование возможно.

В результате исследований получен результат, который отвергает возможность идеально спрятать в коде программы хотя бы один байт информации. Это важно для понимания того, что даже если программа не относится к классу незапутываемых программ, описанных в известных источниках, все равно разбор программы – вопрос времени и знания методов, которые использовались для преобразования.

Кроме того, ведутся исследования в области сокрытия факта преобразования программы. Суть состоит в том, чтобы запутывание проходило аналогично применению стеганографических методов: программа должна выглядеть естественно и факт применение автоматического преобразователя не должен бросаться в глаза. Примером такой техники является использование метода замены имен идентификаторов, когда имена заменяются не на бессмысленные последовательности символов, а на слова, имеющие определенный смысл и семантически связанные между собой, но контекст использования этих понятий должен скрывать контекст работы реальной программы.

Методы оценки запутанности программы

Для оценки сложности программы можно использовать два основных метода. Первый заключается в математическом описании запутанного представления программы и незапутанного. Усилия, которые должны быть затрачены на то, чтобы перевести программу из одного математического представления в другое и есть сложность программы. Как было сказано выше, модели, которая бы позволила описать такие преобразования и такие представления еще не создано, поэтому чаще используется оценка сложности по метрикам.

Метрика представляет собой функцию, характеризующую единицу изучения программы. Для процедурного дизайна такой единицей является процедура или их совокупность, для объектно-ориентированного – класс. Для оценки метрик существуют так называемые мета-метрики, описанные в Таблице 16.1.

Таблица 16.1. Мета-метрики

Мета-метрика	Описание
Измерительная шкала (Measurment scale)	Существуют различные шкалы измерения, такие как: • именная • порядковая

	• интервальная • относительная • абсолютная Обычно значительно проще оперировать метриками, значения которых можно представить в виде конкретных числовых значений (по абсолютной шкале).
Независимость измерений (Measurements independence)	Метрики, которые зависят только от единицы измерения, но не от способа измерения, являются предпочтительными. Пример метрики, не удовлетворяющий этому условию – количество строк программы: строки комментария можно считать или не считать и т.д.
Автоматизация (Automation)	Определяет, поддается ли метрика автоматическому подсчету, полуавтоматическому или может быть измерена только экспертным путем
Цена реализации (Value of implementation)	Определяет, является ли метрика независимой от реализации, т.е. можно ли ее определить до реализации в конкретном языке и стиле программирования
Простота (Simplicity)	Эта мета-метрика определяет, насколько проста метрика с точки зрения ее определения и понимания.
Точность (Accuracy)	Определяет, действительно ли метрика измеряет заявленную характеристику и как полно она это делает.

Составлены таблицы, показывающие, какое значение мета-метрик соответствует большинству существующих метрик.

Задача отображения сложности понимания исходного кода программы в одной единственной метрике была оставлена как не имеющая решения с точки зрения здравого смысла. Можно провести простую аналогию: чтобы описать пустую картонную коробку, необходимы такие величины как ширина, длина, высота и масса; первые три можно представить в виде интегральной оценки объема, однако объем не может дать никакого представления о массе коробки. Очевидно, что в некоторых случаях необходимы знания о геометрии коробки, однако для многих задач вполне хватит двух характеристик: объема и массы.

Была поставлена задача: из существующего множества метрик отобрать только те, которые отличаются репрезентативностью и описывают сложность программы максимально полно. Фактически, для программы необходимо отобрать такие метрики, которые скажут о ней не меньше, чем объем и масса говорят о картонной коробке. Решение этой задачи – еще одно направление, которое связано с запутывающим преобразованием.

Характеристика существующих запутывающих преобразователей

За последние годы появилось достаточно много запутывающих преобразователей для программ, написанных для платформы .NET (Aspose Obfuscator, Decompiler.NET Obfuscator, Demeanor Obfuscator, Deploy.NET, Desaware QND Obfuscator, Dotfuscator, DotNet Protector, Dynu .NET Obfuscator, Goliath .NET Obfuscator, Lesser Software LSW IL-Obfuscator, Obfuscating .NET, Salamander .NET Obfuscator, Skater .NET Obfuscator, Spices .NET Obfuscator, Xenocode

Obfuscator), однако многие из них выполняют только символьное запутывание – изменение имен идентификаторов, их относят к преобразователям первого типа. Более функциональные – преобразователи второго типа – выполняющие запутывание потока выполнения и данных. Опишем самые популярные из преобразователей: Dotfuscator Professional Edition (разработка компании PreEmptive Solution), Salamander .NET obfuscator (Remotesoft) и Spices.Net (9Rays.Net).

Dotfuscator Professional Edition

Этот запутывающий преобразователь достаточно хорошо знаком разработчикам для платформы .NET, т. к. его упрощенная версия (Community Edition) встроена в студию разработки Visual Studio .NET. Кроме встроенной версии этого запутывающего преобразователя существует и отдельная версия, не привязанная к Visual Studio (Рисунок 16.3).

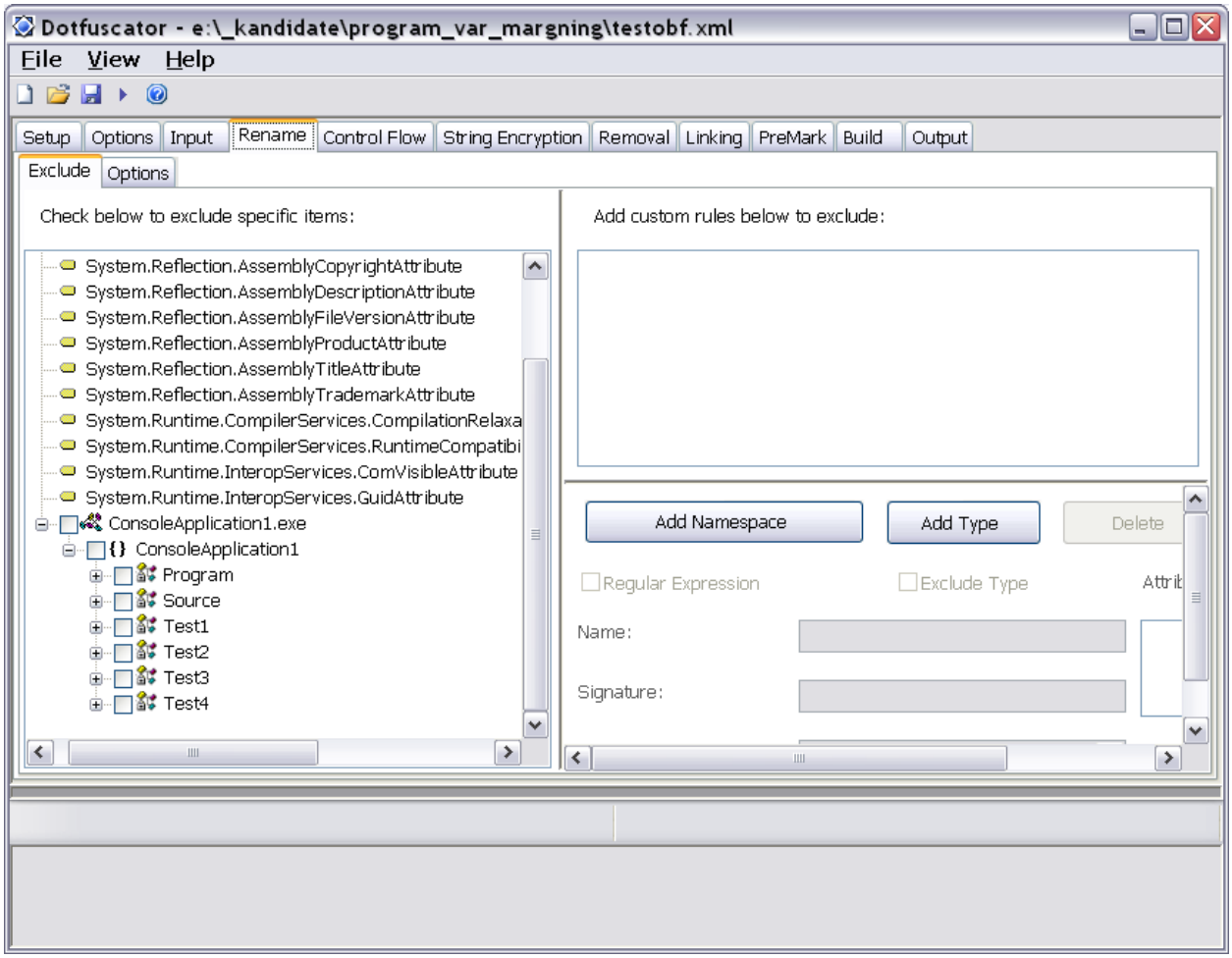


Рисунок 16.3 – Dotfuscator Professional Edition

Обе версии (встроенная и отдельная) предоставляют эквивалентные возможности.

Детализированная документация к продукту позволяет пользователю получить исчерпывающую информацию по работе с запутывающим преобразователем.

Dotfuscator – полнофункциональный запутывающий преобразователь, он поддерживает как лексический метод запутывания программ, так и различные методики изменения потока выполнения. Одной из отличительных черт этого преобразователя является использование им так называемого «overload induction engine» метода переименования. Его суть заключается в том, что одно и то же имя имеют как можно больше сущностей в программе.

Например, структура программы имеет следующий вид:

Листинг 16.6 – Исходный код до проведения запутывающего преобразования

```
namespace program_simple_test
{
	internal class Program
	{
		// Methods
		public Program();
		private static void Main(string[] args);
	}

	internal class Source
	{
		// Methods
		public Source();
		public void GetEquation();
		public int ResolveX1X2(int a, int b, int c, out double
x1, out double x2);

		// Fields
		private Random r;
	}
}
```

После использования метода «overload induction» структура примет следующий вид:

Листинг 16.7 – Исходный код после проведения запутывающего преобразования

```
internal class a
{
	// Methods
	public a();
	private static void a(string[] A_0);
}

internal class b
{
	// Methods
	public b();
	public void a();
```

```
        public int a(int A_0, int A_1, int A_2, out double A_3, out
double A_4);

        // Fields
        private Random a;
}
```

Как видно, практически все сущности получили название «a», что усложняет для человека (но не для компьютера) понимание программы.

Другой важной особенностью Dotfuscator является его способность шифровать строки таким образом, чтобы в программе нигде не было текста, который смог бы прочитать и понять человек. Ниже приведен пример шифрования:

Листинг 16.8 – Исходный код до проведения запутывающего преобразования

```
public static void a()
{
     a a1 = new a();
     Console.WriteLine("Enter password: ");
     string text1 = Console.ReadLine();
     if (!text1.Equals(a1.a))
     {
          Console.WriteLine("Incorrect password.");
     }
     else
     {
          Console.WriteLine("Correct password.");
     }
     Console.ReadLine();
}
```

Листинг 16.9 – Исходный код после проведения запутывающего преобразования

```
public static void a()
{
     int num1 = 13;
     a a1 = new a();
     Con-
sole.WriteLine(a("\uf3b5\ud6b7\uceb9\uccbd\ue0bf\ub2c1\ua5c3\ub5c5
\ubbc7\ubdc9\ua3cb\ubccd\ub4cf\ue8d1\uf4d3", num1));
     string text1 = Console.ReadLine();
     if (!text1.Equals(a1.a))
     {         Con-
sole.WriteLine(a("\uffb5\ud6b7\ud3bb\uccbd\ub2bf\ua7c1\ua7c3\ub2c5
\ue8c7\ubac9\uadcb\ubdcd\ua3cf\ua5d1\ubbd3\ua4d5\ubcd7\uf4d9",
num1));
     }
     else
     {         Con-
sole.WriteLine(a("\uf5b5\ud7b7\uc8b9\ucebb\ua3bf\ub6c1\ue4c3\ub6c5
\ua9c7\ub9c9\ubfcb\ub9cd\ubfcf\ua0d1\ub0d3\uf8d5", num1));
```

```
        }
        Console.ReadLine();
}
```

Дешифрование строк осуществляется методом «a». При кажущейся стойкости такого подхода необходимо отметить его относительную слабость: для дешифрования строк нужно всего лишь декомпилировать и применить метод «a» для всех строк в программе, что вернет ее к исходному виду.

Кроме возможностей по запутыванию кода Dotfuscator предоставляет возможность внесения в программу водяного знака при помощи отдельной вспомогательной программы Premark.

По желанию пользователя Dotfuscator может сгенерировать специальный файл в формате XML, в котором будут описаны запутывающие преобразования, одной из основных частей такого файла является карта переименований, в которой каждому новому (запутанному) имени сопоставляется исходное имя.

Salamander .NET obfuscator

Другой чрезвычайно популярный запутывающий преобразователь – Salamander .NET obfuscator – предоставляет не только функцию запутывания, но и возможность декомпиляции сборок, что позволяет просмотреть результат преобразования программы (Рисунок 16.4)

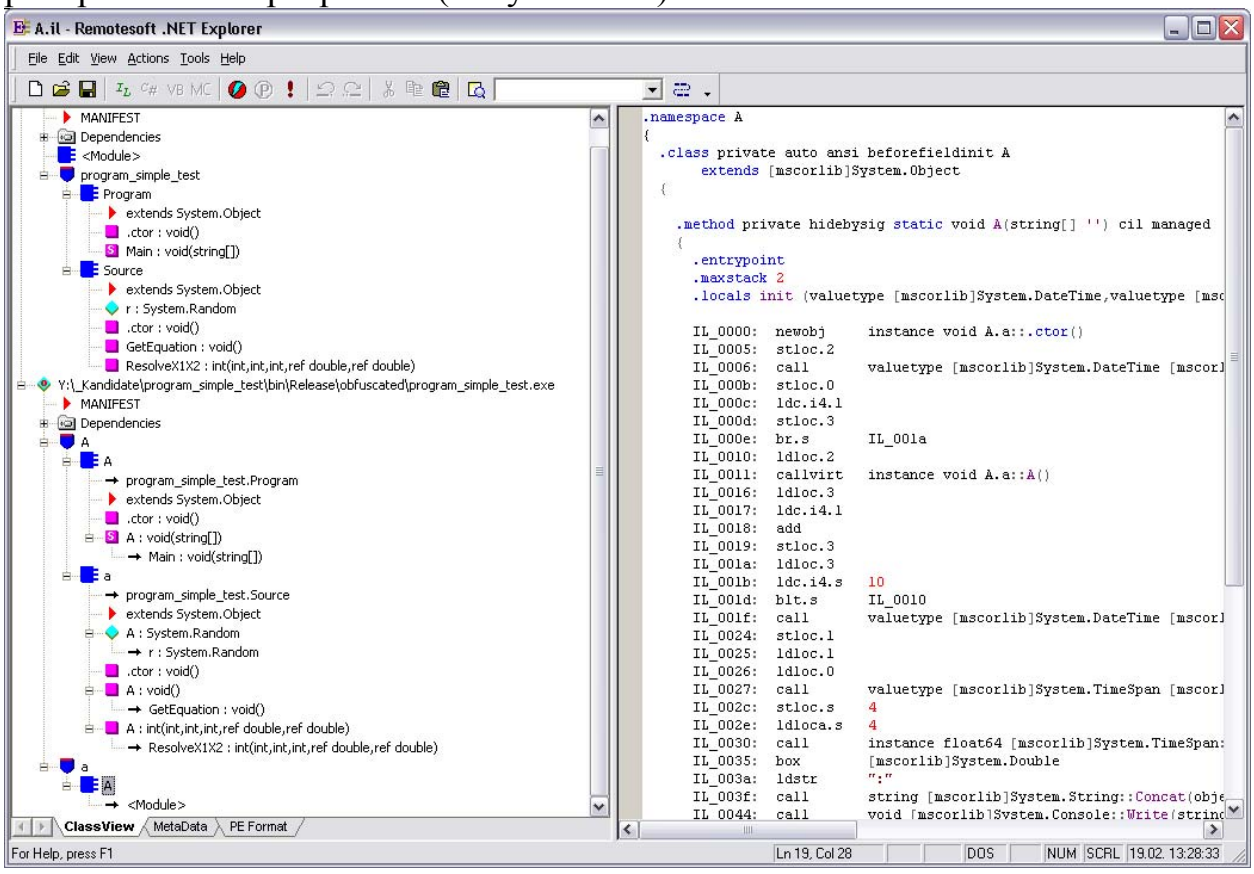


Рисунок 16.4 – Salamander .NET obfuscator

Кроме непосредственно запутанного и исходного кода Salamander .NET obfuscator позволяет исследовать метаданные и структуру PE-файла (Рисунок 16.5)

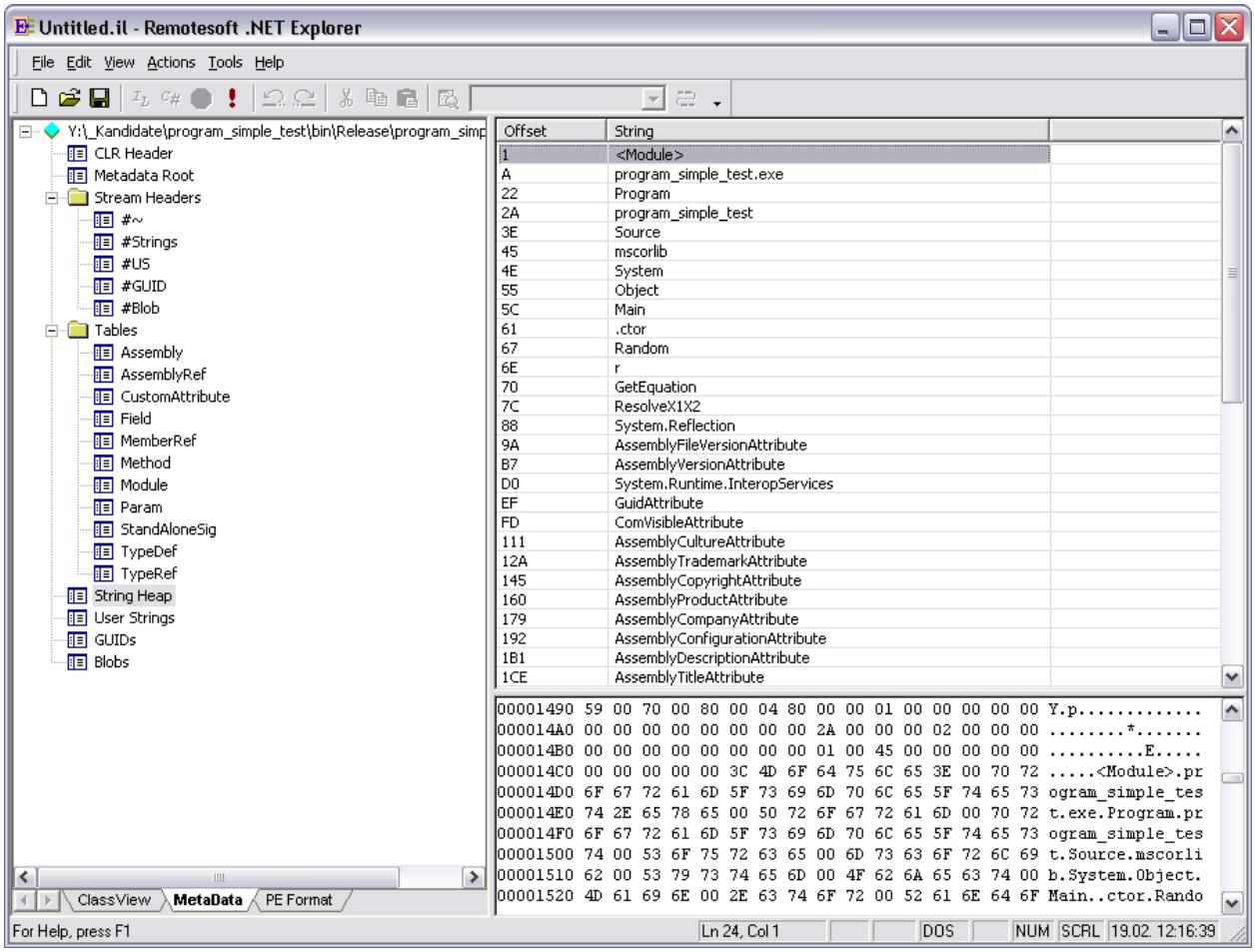


Рисунок 16.5 – Salamander .NET obfuscator

Запутывающий преобразователь предоставляет довольно широкие возможности по настройке и реализует практически все популярные методы запутывания программ, в том числе и перегрузка имен, о которой сказано выше. Метод, который не реализован – шифрование строк, что является существенным недостатком.

Salamander .NET obfuscator позволяет создавать карты соответствий между исходными и запутанными именами переменных.

Spices.NET

Многофункциональный запутывающий кодер с возможностью декомпиляции и внедрения водяных знаков – Spices.NET – предоставляет очень широкие возможности по настройке методов запутывания и шифрования (Рисунок 16.6)

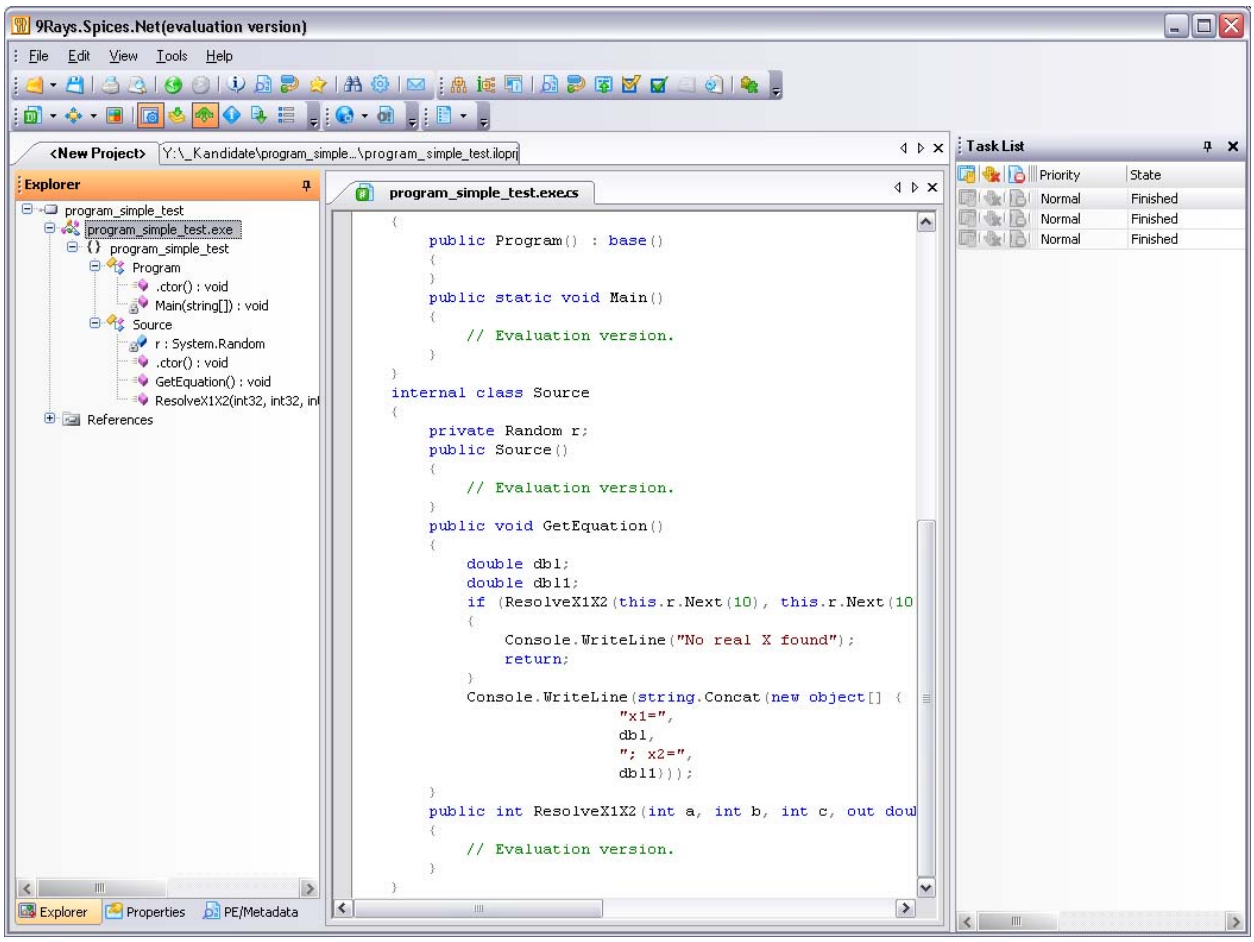


Рисунок 16.6 – Spices.NET

С точки зрения процесса запутывания Spices .Net предоставляет следующие возможности:

- Защита от дизассемблеров ILDASM, Anakrino, Reflector, Remotesoft Salamander Decompiler
- Два режима обработки строк: скрытие строк в "обертках" и шифрование строк.
- Несколько режимов переименования классов, методов, и пространств имен.
- Использование технологии генерации наиболее коротких имен для замены.
- Реализация cross-obfuscation - работа с набором сборок.

Надо сказать, что Spices .Net - это один из немногих преобразователей, проверяющих сборку после запутывания. Однако, к сожалению, это не гарантирует работоспособность запутанного приложения, так, во время изучения этого запутывающего преобразователя при определенной настройке запутывающих методов была получена неработоспособная сборка, что говорит о недостаточной надежности запутывающего преобразователя.

Сравнение существующих запутывающих преобразователей.

Как правило, запутывающие преобразователи оцениваются с точки зрения их функциональности. Так, для приведенных выше преобразователей со-

ставлена таблица, позволяющая сделать некоторые выводы о возможностях этих запутывающих преобразователей:

Таблица 16.2 Сравнение различных запутывающих преобразователей

Возможности \ Продукты	Dotfuscator for .NET Professional Edition v3.0	Salamander .NET Obfuscator v2.0.1	Spices.Obfuscator v5.1.2.4
Entity renaming	x	x	x
Control of Naming Conventions	x	x	x
Overloaded Renaming	x	x	x
Control Flow Obfuscation	x	x	x
Removal of Unused Members	x	x	–
String Encryption	x	–	x
Include/Exclude Members	x	x	x
Rules-based Configuration	x	–	x
Declarative Obfuscation	x	x	x
Strong Name Re-signing	x	x	x
Incremental Obfuscation	x	x	x
Breaks ILDASM	–	x	xКонец формы
Начало формы Software WatermarkingКонец формы	x	–	x

В приведенной таблице приведен список основных функций, поддерживаемых современными запутывающими преобразователями, рассмотрим их подробнее.

Entity renaming – лексический метод запутывания, рассмотренный выше.

Control of Naming Conventions – управление переименованием, выбор конкретной стратегии переименования (использование для этого непечатных символов или длинных строк).

Overloaded Renaming – перегрузка имен, т.е. стратегия переименования, при использовании которой максимальное количество сущностей программы получают одно и то же имя.

Control Flow Obfuscation – изменение потока выполнения. Каждый из запутывающих кодеров реализует свои методы (чаще всего запатентованные) для этой цели. Например, в Spices.Obfuscator используется технология Spices.Anonymizer, которая в данный момент находится в состоянии патентования.

Removal of Unused Members – удаление неиспользуемых методов, действительно, запутывающие преобразователи часто стремятся скрыть свое негативное влияние на объем программы за счет некоторых простых оптимизационных преобразований.

String Encryption – шифрование строк, пример использования этого метода показан выше в описании запутывающего преобразователя Dotfuscator.

Include/Exclude Members – возможность включать и исключать сущности из списка подлежащих запутыванию.

Rules-based Configuration – конфигурация, основанная на правилах, указывающих, например, какой метод использовать раньше другого.

Declarative Obfuscation – начиная с версии .NET 2.0 компания Microsoft встроила в библиотеку классов спецальные атрибуты, при помощи которых можно указать, какие сущности и в каком объеме подвергать запутыванию. Эти атрибуты должны интерптерироваться запутывающими преобразователями.

Strong Name Re-signing – возможность подписать сборку сильным именем.

Incremental Obfuscation – подразумевает возможность создания специальных карт переименований сущностей, которые впоследствии могут быть применены для повторного запутывания некоторых частей приложения во избежание конфликтов с ранее запутанными частями.

Breaks ILDASM – означает изменение сборки таким образом, чтобы ее невозможно было дизассемблировать при помощи стандартной программы ILDASM.

Software Watermarking – внедрение запутывающим преобразователем водяных знаков.

Как видно, все три рассматриваемые запутывающие преобразователи поддерживают практически все основные возможности запутывания и некоторые дополнительные свойства. Однако качество преобразования оценить только из количества функций не представляется возможным.

С другой стороны, разработчики в документации к своим продуктам и на своих сайтах не приводят теоретического обоснования использования тех или иных методов, а их алгоритмы являются закрытыми и защищенными патентами. Это приводит к тому, что пользователи запутывающих преобразователей не только не могут оценить, насколько эффективно они защитили свое программное обеспечение, но не могут быть окончательно уверенными, что эта защита не нарушила функциональности их продуктов. Как уже было упомянуто выше, в процессе изучения преобразователя Spices.Obfuscator довольно простая сборка была запутана таким образом, что потеряла возможности выполнения.

Кроме того, декларируемая возможность внедрения в программу водяных знаков, например, в Spices.Obfuscator, является примитивным добавлением атрибута сборки с говорящим именем `SoftwareWatermark,` который никак не защищен и распознается при помощи декомпилятора .NET Reflector.

Еще одним важным недостатком существующих запутывающих преобразователей заключается в том, что они не оценивают потери во времени выполнения программы и в ее объеме, что может быть весьма существенным для использующего запутывание своих продуктов разработчика.

Тема 17. Доказательство с нулевым знанием

Поводом для данной темы может послужить давняя криптографическая проблема обмена сведениями взаимно недоверяющих друг другу объектов с

помощью средств обмена сообщениями, где каждый объект хочет раскрыть другому как можно меньше своих секретов. Например, предположим, что объект ***A*** хочет убедить объект ***B*** в том, что он знает разложение на множители (факторизацию) RSA модуля $n=p*q$. Конечно, ***A*** может просто показать p и q (это называется *доказательством с максимальным раскрытием*). Однако ***A*** по «криптографической» причине не хочет так поступать. Так как же ***A*** продолжить убеждать ***B*** без непосредственного раскрытия факторизации? Другими словами, является ли возможным для ***A*** доказать ***B***, что он знает факторизацию n, но не раскрывая чего-либо вне этого утверждения?

Доказательства с нулевым знанием идеально подходят для идентификации владельца ***A*** кредитной карточки (удостоверения личности (*ID* card), электронного счета), у которого имеется номер *PIN* или пароль S, позволяя, таким образом, ***A*** доказать продавцу ***B*** знание S, не раскрыв при этом ни одного бита S.

Протокол идентификации Feige-Fiat-Shamir

В данном протоколе доказывающий ***A*** хочет доказать проверяющему (верификатору) ***B*** знание некоторого секрета S_A. Аббревиатура *TTP* будет означать третье доверенное лицо.

Подготовка:

1. *TTP* выбирает RSA модуль $n=p*q$ и делает его открытым для всех, но p и q хранятся в секрете. Также выбирается параметр $\alpha \in \mathbf{N}$.
2. ***A*** и ***B*** выбирают, соответственно, секретные значения S_A, $S_B \le n-1$. На их основе вычисляются $T_A=S_A^2 \pmod n$ и $T_B=S_B^2 \pmod n$. Затем *TTP* регистрирует S_A и S_B как закрытые ключи, где T_A и T_B открытые ключи соответственно ***A*** и ***B***.

Протокол доказательства с нулевым знанием:

1. ***A*** выбирает число $m \in \mathbf{N}$, называемое вручением (*commitment)*, причем $m \le n-1$, и отправляет B число $w=m^2 \bmod n$, называемое свидетельством (*witness)*.
2. ***B*** выбирает бит $c \in \{0,1\}$, называемый вызовом *(challenge),* и отправляет его ***A***.
3. ***A*** вычисляет значение $r=m\ S_A^c \pmod n$, называемое ответом *(response)*, и отправляет его ***B***.
4. ***B*** вычисляет $r^2 \pmod n$ и выполняет проверку равенства
$$r^2=w\ T_A^c \pmod n.$$
Если оно выполняется, то параметр α принимает значение $\alpha-1$ и протокол выполняется снова с шага 1 при условии, что $\alpha>0$. Если $\alpha=0$, тогда выполнение протокола заканчивается и ***B*** считает доказательство правильным **(**принимает доказательство). Если же неравенство не выполняется, т.е. $r^2 \neq w\ T_A^c \pmod n$, то протокол завершается и ***B*** считает доказательство неверным (отклоняет доказательство).

Предположим, что мошенник ***E*** пытается сыграть роль ***A***. Тогда ***E*** мог бы обмануть, выбрав любое значение $m \le n-1$ и отправив ***B*** свидетельство $w=m^2$

mod n. Если ***B*** затем отправляет вызов *c=0*, то ***E*** отправил бы правильный ответ *r=m* и этап 4 будет пройден. Но если в качестве вызова будет послан *c=1*, тогда ***E***, который не знает корень квадратный S_A от T_A , не сможет ответить правильно. Следовательно, вероятность, что мошенничество ***E*** не будет обнаружено после первой итерации, равна ½, после второй – ¼, т.д. Чтобы уменьшить вероятность подобного обмана к допустимой вероятности $2^{-\alpha}$, выполняются повторения протокола достаточно большое число раз α. Таким образом, протокол Feige-Fiat-Shamir является доказательством с нулевым знанием значения корня квадратного из T_A, т.е. секретного значения S_A.

Пример 17.1.

Подготовка:

1. RSA модуль $n=p*q=5\times7=35$ выбран *TTP* и открыт для всех, но $p=5$ и $q=7$ секретны. Выбран параметр $\alpha=2$.
2. ***A*** выбрал секретное значение $S_A=11$.

$T_A=S_A^2(mod\ n)=11^2\ mod\ 35=16$. *TTP* зарегистрировал $S_A=11$ как закрытый ключ, а $T_A=16$ как открытый ключ.

Протокол доказательства с нулевым знанием:

1. ***A*** выбирает вручение m=3, причем m=3≤ n-1=34, и отправляет ***B*** свидетельство $w=m^2 \bmod n=3^2 \bmod 35= 9$**.**
2. ***B*** выбирает вызов c=0 и отправляет его ***A***.
3. ***A*** вычисляет ответ $r=m\ S_A^c(\bmod n)=3\times11^0 \bmod 35 = 3$ и отправляет его ***B***.
4. ***B*** вычисляет $r^2(\bmod n)=3^2 \bmod 35=9$. Так как результат $wT_A^c(\bmod n)=9\times16^0 \bmod 35=r^2=9$, то α принимает значение α-1=2-1=1 и если $\alpha=1>0$, то протокол выполняется с шага 1.
5. ***A*** выбирает вручение m=23, причем m=23≤ n-1=34, и отправляет ***B*** свидетельство $w=m^2 \bmod n=23^2 \bmod 35= 4$.
6. ***B*** выбирает вызов c=1 и отправляет его ***A***.
7. ***A*** вычисляет ответ $r=m\ S_A^c(\bmod n)=23\times11^1 \bmod 35 = 8$ и отправляет его ***B***.
8. **B** вычисляет $r^2(\bmod n)=8^2 \bmod 35=29$. Так как результат $wT_A^c(\bmod n)=4\times16^1 \bmod 35=29=r^2=29$, то α принимает значение α-1=1-1=0. Так как $\alpha=0$, то выполнение протокола завершается и ***B*** считает доказательство верным.

Этот протокол является представителем большого класса *трехшаговых протоколов доказательства с нулевым знанием* (***Three-move, Zero-knowledge Protocols)***. В таких протоколах ***A*** отправляет свидетельство, ***B*** отвечает вызовом, ***A*** отправляет ответ. Одно повторение этих «шагов» называется *раундом*. В данном протоколе существует скрытый фактор случайности, когда ***A*** выбирает закрытый параметр вручение *m* из некоторого предопределенного набора. Затем ***A*** вычисляет связанное с ним (открытое) свидетельство *w*. Эта начальная случайность гарантирует, что итерации протокола последовательны и независимы, именно это отличает один раунд протокола от другого. Также это устанавливает набор «вопросов», на которые ***A*** сможет ответить, поэтому после-

дующие ответы ***A*** ограничены. Данный вариант протокола означает, что только доказывающий ***A,*** зная секрет *S*, способен отреагировать на все вызовы правильными ответами, из которых ни один не дает информации об *S*.

Процесс, в котором ***A*** определяет набор вопросов, а ***B*** выбирает вопросы, например (1)-(2) в протоколе Feige-Fiat-Shamir, является случаем так называемого протокола ***разрезать и выбрать***. Эта терминология происходит из классического протокола честного деления чего-либо между двумя людьми. Таким образом, ***A*** делит объект пополам, а ***B*** выбирает одну из половин себе, оставляя другую половину для ***A.*** Также процедура (2)-(3) шагов протокола Feige-Fiat-Shamir, учитывающая свидетельство из шага (1), является случаем ***протокола вызова-ответа***. В подобных протоколах из соображений безопасности необходимо, чтобы ***A*** отвечал на один вызов для данного свидетельства, а свидетельства не повторялись.

Главное ограничение предыдущего алгоритма заключается в выполнении двух требований при выборе значений открытых ключей T_A и T_B.

1. значения T_A и T_B должны удовлетворять равенству $x^2=Tj \bmod n$.
2. для Tj должно существовать инверсное значение $Tj^{-1} \bmod m$.

Рассмотрим случай для $n=35$.

Все возможные значения $Tj \in \{1,4,9,11,14,15,16,21,25,29,30\}$; необходимо упомянуть, что значения $Tj \in \{14,15,21,25,30\}$ не имеют инверсных значений.

Расширением алгоритма является применение пользователем ***A*** нескольких как открытых, так и закрытых ключей. Тогда пользователь ***A*** вычисляет *ответ*:

$r=m \times (S_{A1}^{c1} \times S_{A2}^{c2} \times S_{A3}^{c3} \times \ldots \times S_{Ak}^{ck}) (mod\ n)$, учитывая как k секретных ключей, так и случайную строку бит $c1,c2,c3,\ldots,ck$, и отправляет его ***B***.

Пользователь ***B*** в этом случае вычисляет:

$r^2 (mod\ n)$ и $w \times (T_{A1}^{c1} \times T_{A2}^{c2} \times T_{A3}^{c3} \times \ldots \times T_{Ak}^{ck}) (mod\ n)$, а затем выполняет сравнение

$r^2 (mod\ n) = w \times (T_{A1}^{c1} \times T_{A2}^{c2} \times T_{A3}^{c3} \times \ldots \times T_{Ak}^{ck}) (mod\ n)$.

Протокол идентификации Guillou-Quisquater

Пусть ***A*** – это электронная пластиковая карта (***smart card***), которая хочет доказать свою законность (***validity***) устройству контроля (верификатору) ***B.*** Идентификация ***A*** представляет собой длинную строку *I* – открытый ключ для данной карты. Открытые параметры протокола: $n=p*q$, где p и q – секретные; значение *V* и *I*. Закрытым является параметр *G*, значение которого удовлетворяет равенству:

$$I \times G^V = 1 \bmod n$$

Smart-карта ***A*** должна доказать, что она знает значение *G*.

Выполняется протокол:

1. ***A*** генерирует целое число r, где $1<r \le n-1$, вычисляет значение $T=r^V \bmod n$ и отправляет его ***B***.
2. ***B*** генерирует целое число d, где $1<d \le n-1$, и отправляет его ***A***.
3. ***A*** вычисляет $D=r \times G^d \bmod n$ и отправляет ***B***.
4. ***B*** вычисляет $T'=D^V \times I^d \bmod n$

В силу следующих соотношений:

$T'=D^V \times I^d \bmod n=(rG^d)^V \times I^d \bmod n=r^V(I \times G^V)^d \bmod n=r^V \bmod n=T$

это равенство истинно для случая $I \times G^V=1 \bmod n$.