Криптографические основы безопасности
Информация о курсе

Курс предполагает изучение методологических и алгоритмических основ и стандартов криптографической защиты информации.
Значительное внимание уделяется изучению широко используемых криптографических алгоритмов симметричного и асимметричного шифрования, а также криптографических хэш-функций. В частности рассмотрены основные алгоритмы симметричного шифрования: DES, AES, ГОСТ 28147, Rijndael, Blowfish, IDEA и другие, а также режимы их использования; рассмотрены алгоритмы шифрования с открытым ключом RSA, Диффи-Хеллмана, DSS, ГОСТ 3410; рассмотрены алгоритмы асимметричного шифрования с использованием эллиптических кривых; определено понятие сильной криптографической хэш-функции и рассмотрены хэш-функции MD5, SHA-1, SHA-2, ГОСТ 3411.

Цель
Изучение методологических и алгоритмических основ и стандартов криптографической защиты информации.

Предварительные знания
1. Понятие стека протоколов TCP/IP.
2. Начальные сведения о теории вычетов, дискретном логарифмировании.

Основные понятия и определения 3
Модель сетевой безопасности 5
Криптография 10
Сеть Фейштеля 13
Криптоанализ 14
Используемые критерии при разработке алгоритмов 16
Алгоритм DES 16
Алгоритм тройной DES 21
Алгоритм Blowfish 23
Алгоритм IDEA 24
Алгоритм ГОСТ 28147 31
Режимы выполнения алгоритмов симметричного шифрования 33
Создание случайных чисел 37
Разработка Advanced Encryption Standard (AES) 42
Алгоритм Rijndael 69
Алгоритм RC6 82
Основные требования к алгоритмам асимметричного шифрования 83
Алгоритм RSA 87
Алгоритм обмена ключа Диффи-Хеллмана 93
Хэш-функции 95
Коды аутентификации сообщений - MAC 116
Требования к цифровой подписи 118
Прямая и арбитражная цифровые подписи 119
Стандарт цифровой подписи DSS 122
Отечественный стандарт цифровой подписи ГОСТ 3410 126
Математические понятия 127
Аналог алгоритма Диффи-Хеллмана обмена ключами 131
Алгоритм цифровой подписи на основе эллиптических кривых ECDSA 131

Шифрование/дешифрование с использованием эллиптических кривых.....132
Алгоритмы распределения ключей с использованием третьей доверенной стороны.....133

1. Лекция: Введение

Основные понятия и определения

За несколько последних десятилетий требования к информационной безопасности существенно изменились. До начала широкого использования автоматизированных систем обработки данных безопасность информации достигалась исключительно физическими и административными мерами. С появлением компьютеров стала очевидной необходимость использования автоматических средств защиты файлов данных и программной среды. Следующий этап развития автоматических средств защиты связан с появлением распределенных систем обработки данных и компьютерных сетей, в которых средства сетевой безопасности используются в первую очередь для защиты передаваемых по сетям данных. В наиболее полной трактовке под средствами сетевой безопасности мы будем иметь в виду меры предотвращения нарушений безопасности, которые возникают при передаче информации по сетям, а также меры, позволяющие определять, что такие нарушения безопасности имели место. Именно изучение средств сетевой безопасности и связанных с ними теоретических и прикладных проблем, составляет основной материал курса.

Термины "безопасность информации" и "защита информации" отнюдь не являются синонимами. Термин "безопасность" включает в себя не только понятие защиты, но также и *аутентификацию*, аудит, обнаружение проникновения.

Перечислим некоторые характерные проблемы, связанные с безопасностью, которые возникают при использовании компьютерных сетей:

1. Фирма имеет несколько офисов, расположенных на достаточно большом расстоянии друг от друга. При пересылке конфиденциальной информации по общедоступной сети (например, Internet) необходимо быть уверенным, что никто не сможет ни подсмотреть, ни изменить эту информацию.
2. Сетевой администратор осуществляет удаленное управление компьютером. Пользователь перехватывает управляющее сообщение, изменяет его содержание и отправляет сообщение на данный компьютер.
3. Пользователь несанкционированно получает доступ к удаленному компьютеру с правами законного пользователя, либо, имея право доступа к компьютеру, получает доступ с гораздо большими правами.
4. Фирма открывает Internet-магазин, который принимает оплату в электронном виде. В этом случае продавец должен быть уверен, что он отпускает товар, который действительно оплачен, а покупатель должен иметь гарантии, что он, во-первых, получит оплаченный товар, а во-вторых, номер его кредитной карточки не станет никому известен.
5. Фирма открывает свой сайт в Internet. В какой-то момент содержимое сайта заменяется новым, либо возникает такой поток и такой способ обращений к сайту, что сервер не справляется с обработкой запросов. В результате обычные посетители сайта либо видят информацию, не имеющую к фирме никакого отношения, либо просто не могут попасть на сайт фирмы.

Рассмотрим основные понятия, относящиеся к информационной безопасности, и их взаимосвязь.

Собственник определяет множество **информационных ценностей**, которые должны быть защищены от различного рода *атак*. *Атаки* осуществляются *противниками* или *оппонентами*, использующими различные *уязвимости* в защищаемых ценностях. Основными нарушениями безопасности являются раскрытие информационных ценностей (потеря *конфиденциальности*), их неавторизованная модификация (потеря *целостности*) или неавторизованная потеря доступа к этим ценностям (потеря *доступности*).

Собственники информационных ценностей анализируют *уязвимости* защищаемых ресурсов и возможные *атаки*, которые могут иметь место в конкретном окружении. В результате такого анализа определяются *риски* для данного набора информационных ценностей. Этот анализ определяет выбор контрмер, который задается политикой безопасности и обеспечивается с помощью *механизмов* и *сервисов безопасности*. Следует учитывать, что отдельные *уязвимости* могут сохраниться и после применения *механизмов* и *сервисов безопасности*. ***Политика безопасности*** определяет согласованную совокупность *механизмов* и *сервисов безопасности*, адекватную защищаемым ценностям и окружению, в котором они используются.

На рис.1.1 показана взаимосвязь рассмотренных выше понятий информационной безопасности.

Рис. 1.1. Взаимосвязь основных понятий безопасности информационных систем

Дадим следующие определения:

Уязвимость - слабое место в системе, с использованием которого может быть осуществлена *атака*.

Риск - вероятность того, что конкретная *атака* будет осуществлена с использованием конкретной *уязвимости*. В конечном счете, каждая организация должна принять решение о допустимом для нее уровне *риска*. Это решение должно найти отражение в политике безопасности, принятой в организации.

Политика безопасности - правила, директивы и практические навыки, которые определяют то, как информационные ценности обрабатываются, защищаются и

распространяются в организации и между информационными системами; набор критериев для предоставления *сервисов безопасности*.

Атака - любое действие, нарушающее безопасность информационной системы. Более формально можно сказать, что *атака* - это действие или последовательность связанных между собой действий, использующих *уязвимости* данной информационной системы и приводящих к нарушению политики безопасности.

Механизм безопасности - программное и/или аппаратное средство, которое определяет и/или предотвращает *атаку*.

Сервис безопасности - сервис, который обеспечивает задаваемую политикой безопасность систем и/или передаваемых данных, либо определяет осуществление *атаки*. *Сервис* использует один или более механизмов безопасности.

Рассмотрим модель сетевой безопасности и основные типы *атак*, которые могут осуществляться в этом случае. Затем рассмотрим основные типы *сервисов* и *механизмов безопасности*, предотвращающих такие *атаки*.

Модель сетевой безопасности

Классификация сетевых атак

В общем случае существует информационный поток от отправителя (файл, пользователь, компьютер) к получателю (файл, пользователь, компьютер):

Рис. 1.2. Информационный поток

Все *атаки* можно разделить на два класса: *пассивные* и *активные*.

I. Пассивная атака

Пассивной называется такая ***атака***, при которой *противник* не имеет возможности модифицировать передаваемые сообщения и вставлять в информационный канал между отправителем и получателем свои сообщения. Целью *пассивной атаки* может быть только прослушивание передаваемых сообщений и анализ трафика.

Рис. 1.3. Пассивная атака

II. Активная атака

Активной называется такая ***атака***, при которой *противник* имеет возможность модифицировать передаваемые сообщения и вставлять свои сообщения. Различают следующие типы *активных атак*:

1. **Отказ в обслуживании** - ***DoS-атака (Denial of Service)***

Отказ в обслуживании нарушает нормальное функционирование сетевых сервисов. *Противник* может перехватывать все сообщения, направляемые определенному адресату. Другим примером подобной *атаки* является создание значительного трафика, в результате чего сетевой сервис не сможет обрабатывать запросы законных клиентов. Классическим примером такой *атаки* в сетях TCP/IP является SYN-атака, при которой нарушитель посылает пакеты, инициирующие установление TCP-соединения, но не посылает пакеты, завершающие установление этого соединения. В результате может произойти переполнение памяти на сервере, и серверу не удастся установить соединение с законными пользователями.

Рис. 1.4. DoS-атака

2. **Модификация потока данных** - ***атака "man in the middle"***

   Модификация потока данных означает либо изменение содержимого пересылаемого сообщения, либо изменение порядка сообщений.

Рис. 1.5. Атака "man in the middle"

3. **Создание ложного потока (фальсификация)**

   Фальсификация (нарушение аутентичности) означает попытку одного субъекта выдать себя за другого.

Рис. 1.6. Создание ложного потока

4. **Повторное использование**

   Повторное использование означает пассивный захват данных с последующей их пересылкой для получения несанкционированного доступа - это так называемая ***replay-атака***. На самом деле *replay-атаки* являются одним из вариантов фальсификации, но в силу того, что это один из наиболее распространенных вариантов *атаки* для получения несанкционированного доступа, его часто рассматривают как отдельный тип *атаки*.

Рис. 1.7. Replay-атака

Перечисленные *атаки* могут существовать в любых типах сетей, а не только в сетях, использующих в качестве транспорта протоколы TCP/IP, и на любом уровне модели OSI. Но в сетях, построенных на основе TCP/IP, *атаки* встречаются чаще всего, потому что, во-первых, Internet стал самой распространенной сетью, а во-вторых, при разработке протоколов TCP/IP требования безопасности никак не учитывались.

Сервисы безопасности

Основными *сервисами безопасности* являются следующие:

Конфиденциальность - предотвращение *пассивных атак* для передаваемых или хранимых данных.

Аутентификация - подтверждение того, что информация получена из законного источника, и получатель действительно является тем, за кого себя выдает. В случае передачи единственного сообщения *аутентификация* должна гарантировать, что получателем сообщения является тот, кто нужно, и сообщение получено из заявленного источника. В случае установления соединения имеют место два аспекта. Во-первых, при инициализации соединения *сервис* должен гарантировать, что оба участника являются требуемыми. Во-вторых, *сервис* должен гарантировать, что на соединение не воздействуют таким образом, что *третья сторона* сможет маскироваться под одну из легальных сторон уже после установления соединения.

Целостность - *сервис*, гарантирующий, что информация при хранении или передаче не изменилась. Может применяться к потоку сообщений, единственному сообщению или отдельным полям в сообщении, а также к хранимым файлам и отдельным записям файлов.

Невозможность отказа - невозможность, как для получателя, так и для отправителя, отказаться от факта передачи. Таким образом, когда сообщение отправлено, получатель может убедиться, что это сделал легальный отправитель. Аналогично, когда сообщение пришло, отправитель может убедиться, что оно получено легальным получателем.

Контроль доступа - возможность ограничить и контролировать доступ к системам и приложениям по коммуникационным линиям.

Доступность - результатом *атак* может быть потеря или снижение доступности того или иного сервиса. Данный *сервис* предназначен для того, чтобы минимизировать возможность осуществления *DoS-атак*.

Механизмы безопасности

Перечислим основные *механизмы безопасности*:

Алгоритмы симметричного шифрования - алгоритмы шифрования, в которых для шифрования и дешифрования используется один и тот же ключ или ключ дешифрования легко может быть получен из ключа шифрования.

Алгоритмы асимметричного шифрования - алгоритмы шифрования, в которых для шифрования и дешифрования используются два разных ключа, называемые открытым и закрытым ключами, причем, зная один из ключей, вычислить другой невозможно.

Хэш-функции - функции, входным значением которых является сообщение произвольной длины, а выходным значением - сообщение фиксированной длины. *Хэш-функции* обладают рядом свойств, которые позволяют с высокой долей вероятности определять изменение входного сообщения.

Модель сетевого взаимодействия

Модель безопасного сетевого взаимодействия в общем виде можно представить следующим образом:

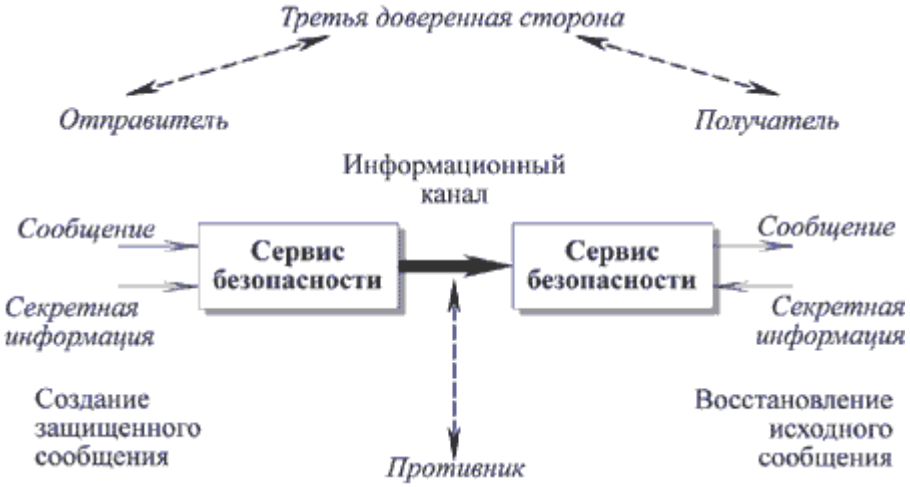

Рис. 1.8. Модель сетевой безопасности

Сообщение, которое передается от одного участника другому, проходит через различного рода сети. При этом будем считать, что устанавливается логический информационный канал от отправителя к получателю с использованием различных коммуникационных протоколов (например, TCP/IP).

Средства безопасности необходимы, если требуется защитить передаваемую информацию от *противника*, который может представлять угрозу *конфиденциальности*, *аутентификации*, *целостности* и т.п. Все технологии повышения безопасности имеют два компонента:

1. Относительно безопасная передача информации. Примером является шифрование, когда сообщение изменяется таким образом, что становится нечитаемым для *противника*, и, возможно, дополняется кодом, который основан на содержимом сообщения и может использоваться для *аутентификации* отправителя и обеспечения *целостности* сообщения.
2. Некоторая секретная информация, разделяемая обоими участниками и неизвестная *противнику*. Примером является ключ шифрования.

Кроме того, в некоторых случаях для обеспечения безопасной передачи бывает необходима *третья доверенная сторона* (third trusted party - TTP). Например, *третья сторона* может быть ответственной за распределение между двумя участниками секретной информации, которая не стала бы доступна *противнику*. Либо *третья сторона* может использоваться для решения споров между двумя участниками относительно достоверности передаваемого сообщения.

Из данной общей модели вытекают три основные задачи, которые необходимо решить при разработке конкретного *сервиса безопасности*:

1. Разработать алгоритм шифрования/дешифрования для выполнения безопасной передачи информации. Алгоритм должен быть таким, чтобы *противник* не мог расшифровать перехваченное сообщение, не зная секретную информацию.
2. Создать секретную информацию, используемую алгоритмом шифрования.
3. Разработать протокол обмена сообщениями для распределения разделяемой секретной информации таким образом, чтобы она не стала известна *противнику*.

Модель безопасности информационной системы

Существуют и другие относящиеся к безопасности ситуации, которые не соответствуют описанной выше модели сетевой безопасности. Общую модель этих ситуаций можно проиллюстрировать следующим образом:

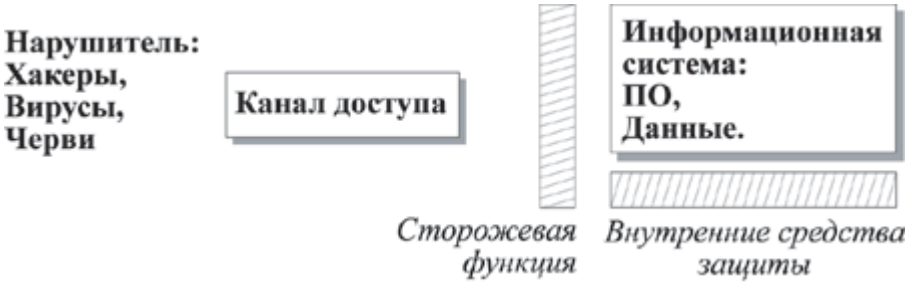

Рис. 1.9. Модель безопасности информационной системы

Данная модель иллюстрирует концепцию безопасности информационной системы, с помощью которой предотвращается нежелательный доступ. Хакер, который пытается осуществить незаконное проникновение в системы, доступные по сети, может просто получать удовольствие от взлома, а может стараться повредить информационную систему и/или внедрить в нее что-нибудь для своих целей. Например, целью хакера может быть получение номеров кредитных карточек, хранящихся в системе.

Другим типом нежелательного доступа является размещение в вычислительной системе чего-либо, что воздействует на прикладные программы и программные утилиты, такие как редакторы, компиляторы и т.п. Таким образом, существует два типа *атак*:

1. Доступ к информации с целью получения или модификации хранящихся в системе данных.
2. *Атака* на сервисы, чтобы помешать использовать их.

Вирусы и черви - примеры подобных *атак*. Такие *атаки* могут осуществляться как с помощью дискет, так и по сети.

Сервисы безопасности, которые предотвращают нежелательный доступ, можно разбить на две категории:

1. Первая категория определяется в терминах сторожевой функции. Эти *механизмы* включают процедуры входа, основанные, например, на использовании пароля, что позволяет разрешить доступ только авторизованным пользователям. Эти *механизмы* также включают различные защитные экраны (firewalls), которые предотвращают *атаки* на различных уровнях стека протоколов TCP/IP, и, в частности, позволяют предупреждать проникновение червей, вирусов, а также предотвращать другие подобные *атаки*.
2. Вторая линия обороны состоит из различных внутренних мониторов, контролирующих доступ и анализирующих деятельность пользователей.

Одним из основных понятий при обеспечении безопасности информационной системы является понятие ***авторизации*** - определение и предоставление прав доступа к конкретным ресурсам и/или объектам.

В основу безопасности информационной системы должны быть положены следующие основные принципы:

1. Безопасность информационной системы должна соответствовать роли и целям организации, в которой данная система установлена.
2. Обеспечение информационной безопасности требует комплексного и целостного подхода.
3. Информационная безопасность должна быть неотъемлемой частью системы управления в данной организации.
4. Информационная безопасность должна быть экономически оправданной.
5. Ответственность за обеспечение безопасности должна быть четко определена.
6. Безопасность информационной системы должна периодически переоцениваться.
7. Большое значение для обеспечения безопасности информационной системы имеют социальные факторы, а также меры административной, организационной и физической безопасности.

2. Лекция: Алгоритмы симметричного шифрования. Часть 1

Криптография

Основные понятия

Рассмотрим общую схему симметричной, или традиционной, *криптографии*.

Рис. 2.1. Общая схема симметричного шифрования

В процессе шифрования используется определенный алгоритм шифрования, на вход которому подаются исходное незашифрованное сообщение, называемое также ***plaintext***, и ключ. Выходом алгоритма является зашифрованное сообщение, называемое также ***ciphertext***. Ключ является значением, не зависящим от шифруемого сообщения. Изменение ключа должно приводить к изменению зашифрованного сообщения.

Зашифрованное сообщение передается получателю. Получатель преобразует зашифрованное сообщение в исходное незашифрованное сообщение с помощью алгоритма дешифрования и того же самого ключа, который использовался при шифровании, или ключа, легко получаемого из *ключа шифрования*.

Незашифрованное сообщение будем обозначать P или M, от слов *plaintext* и message. Зашифрованное сообщение будем обозначать C, от слова ciphertext.

Безопасность, обеспечиваемая традиционной *криптографией*, зависит от нескольких факторов.

Во-первых, криптографический алгоритм должен быть достаточно сильным, чтобы передаваемое зашифрованное сообщение невозможно было расшифровать без ключа, используя только различные статистические закономерности зашифрованного сообщения или какие-либо другие способы его анализа.

Во-вторых, безопасность передаваемого сообщения должна зависеть от секретности ключа, но не от секретности алгоритма. Алгоритм должен быть проанализирован специалистами, чтобы исключить наличие слабых мест, при наличии которых плохо скрыта взаимосвязь между незашифрованным и зашифрованным сообщениями. К тому же при выполнении этого условия производители могут создавать дешевые аппаратные чипы и свободно распространяемые программы, реализующие данный алгоритм шифрования.

В-третьих, алгоритм должен быть таким, чтобы нельзя было узнать ключ, даже зная достаточно много пар (зашифрованное сообщение, незашифрованное сообщение), полученных при шифровании с использованием данного ключа.

Клод Шеннон ввел понятия *диффузии* и *конфузии* для описания *стойкости алгоритма* шифрования.

Диффузия - это рассеяние статистических особенностей незашифрованного текста в широком диапазоне статистических особенностей зашифрованного текста. Это достигается тем, что значение каждого элемента незашифрованного текста влияет на значения многих элементов зашифрованного текста или, что то же самое, любой элемент зашифрованного текста зависит от многих элементов незашифрованного текста.

Конфузия - это уничтожение статистической взаимосвязи между зашифрованным текстом и ключом.

Если X - это исходное сообщение и K - криптографический ключ, то зашифрованный передаваемый текст можно записать в виде

$$Y = E_K[X].$$

Получатель, используя тот же ключ, расшифровывает сообщение

$$X = D_K[Y]$$

Противник, не имея доступа к K и X , должен попытаться узнать X, K или и то, и другое.

Алгоритмы симметричного шифрования различаются способом, которым обрабатывается исходный текст. Возможно шифрование блоками или шифрование потоком.

Блок текста рассматривается как неотрицательное целое число, либо как несколько независимых неотрицательных целых чисел. Длина блока всегда выбирается равной степени двойки. В большинстве блочных алгоритмов симметричного шифрования используются следующие типы операций:

- Табличная подстановка, при которой группа битов отображается в другую группу битов. Это так называемые ***S-box***.
- Перемещение, с помощью которого биты сообщения переупорядочиваются.
- Операция сложения по модулю 2, обозначаемая XOR или $\oplus$.
- Операция сложения по модулю 2^{32} или по модулю 2^{16}.
- Циклический сдвиг на некоторое число битов.

Эти операции циклически повторяются в алгоритме, образуя так называемые ***раунды***. Входом каждого *раунда* является выход предыдущего *раунда* и ключ, который получен по определенному алгоритму из ключа шифрования K. Ключ *раунда* называется ***подключом***. Каждый алгоритм шифрования может быть представлен следующим образом:

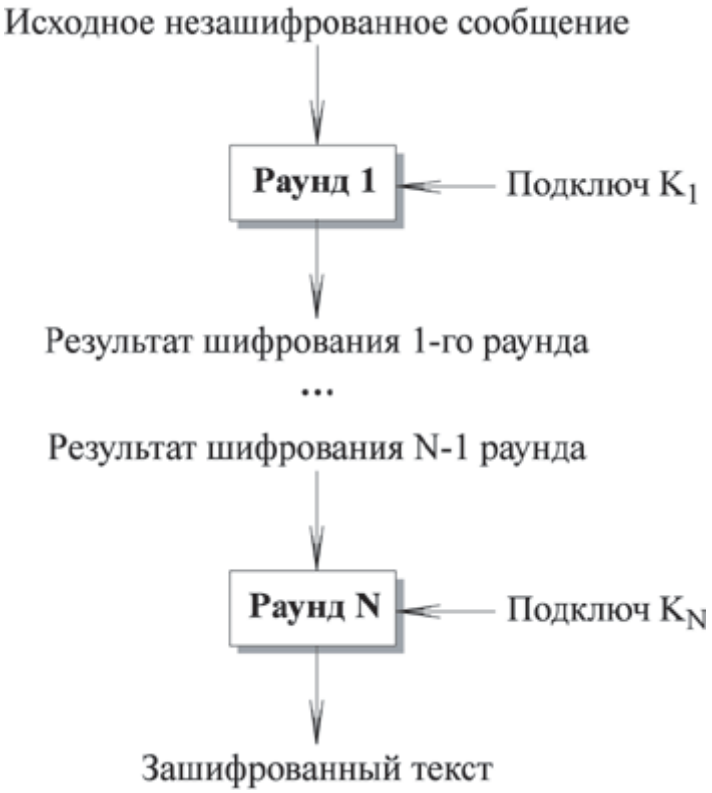

Рис. 2.2. Структура алгоритма симметричного шифрования

Области применения

Стандартный алгоритм шифрования должен быть применим во многих приложениях:

- Шифрование данных. Алгоритм должен быть эффективен при шифровании файлов данных или большого потока данных.
- Создание случайных чисел. Алгоритм должен быть эффективен при создании определенного количества случайных битов.
- Хэширование. Алгоритм должен эффективно преобразовываться в одностороннюю хэш-функцию.

Платформы

Стандартный алгоритм шифрования должен быть реализован на различных платформах, которые, соответственно, предъявляют различные требования.

- Алгоритм должен эффективно реализовываться на специализированной аппаратуре, предназначенной для выполнения шифрования/дешифрования.
- Большие процессоры. Хотя для наиболее быстрых приложений всегда используется специальная аппаратура, программные реализации применяются чаще. Алгоритм должен допускать эффективную программную реализацию на 32-битных процессорах.
- Процессоры среднего размера. Алгоритм должен работать на микроконтроллерах и других процессорах среднего размера.

- Малые процессоры. Должна существовать возможность реализации алгоритма на смарт-картах, пусть даже с учетом жестких ограничений на используемую память.

Дополнительные требования

Алгоритм шифрования должен, по возможности, удовлетворять некоторым дополнительным требованиям.

- Алгоритм должен быть простым для написания кода, чтобы минимизировать вероятность программных ошибок.
- Алгоритм должен иметь плоское пространство ключей и допускать любую случайную строку битов нужной длины в качестве возможного ключа. Наличие слабых ключей нежелательно.
- Алгоритм должен легко модифицироваться для различных уровней безопасности и удовлетворять как минимальным, так и максимальным требованиям.
- Все операции с данными должны осуществляться над блоками, кратными байту или 32-битному слову.

Сеть Фейштеля

Блочный алгоритм преобразовывает n-битный блок незашифрованного текста в n-битный блок зашифрованного текста. Число блоков длины n равно 2^n. Для того чтобы преобразование было обратимым, каждый из таких блоков должен преобразовываться в свой уникальный блок зашифрованного текста. При маленькой длине блока такая подстановка плохо скрывает статистические особенности незашифрованного текста. Если блок имеет длину 64 бита, то он уже хорошо скрывает статистические особенности исходного текста. Но в данном случае преобразование текста не может быть произвольным в силу того, что ключом будет являться само преобразование, что исключает эффективную как программную, так и аппаратную реализации.

Наиболее широкое распространение получили *сети Фейштеля*, так как, с одной стороны, они удовлетворяют всем требованиям к алгоритмам симметричного шифрования, а с другой стороны, достаточно просты и компактны.

Сеть Фейштеля имеет следующую структуру. Входной блок делится на несколько равной длины подблоков, называемых ветвями. В случае, если блок имеет длину 64 бита, используются две ветви по 32 бита каждая. Каждая ветвь обрабатывается независимо от другой, после чего осуществляется циклический сдвиг всех ветвей влево. Такое преобразование выполняется несколько циклов или *раундов*. В случае двух ветвей каждый *раунд* имеет структуру, показанную на рисунке:

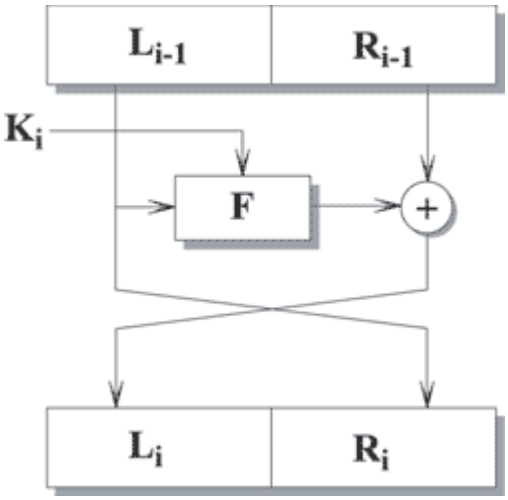

Рис. 2.3. I-ый раунд сети Фейштеля

Функция F называется образующей. Каждый *раунд* состоит из вычисления функции F для одной ветви и побитового выполнения операции XOR результата F с другой ветвью. После этого ветви меняются местами. Считается, что оптимальное число *раундов* - от 8 до 32. Важно то, что увеличение количества *раундов* значительно увеличивает криптостойкость алгоритма. Возможно, эта особенность и повлияла на столь активное распространение *сети Фейштеля*, так как для большей криптостойкости достаточно просто увеличить количество *раундов*, не изменяя сам алгоритм. В последнее время количество *раундов* не фиксируется, а лишь указываются допустимые пределы.

Сеть Фейштеля является обратимой даже в том случае, если функция F не является таковой, так как для дешифрования не требуется вычислять F^{-1}. Для дешифрования используется тот же алгоритм, но на вход подается зашифрованный текст, и ключи используются в обратном порядке.

В настоящее время все чаще используются различные разновидности *сети Фейштеля* для 128-битного блока с четырьмя ветвями. Увеличение количества ветвей, а не размерности каждой ветви связано с тем, что наиболее популярными до сих пор остаются процессоры с 32-разрядными словами, следовательно, оперировать 32-разрядными словами эффективнее, чем с 64-разрядными.

Основной характеристикой алгоритма, построенного на основе *сети Фейштеля*, является функция F. Различные варианты касаются также начального и конечного преобразований. Подобные преобразования, называемые забеливанием (whitening), осуществляются для того, чтобы выполнить начальную рандомизацию входного текста.

Криптоанализ

Процесс, при котором предпринимается попытка узнать X, K или и то, и другое, называется криптоанализом. Одной из возможных атак на алгоритм шифрования является лобовая атака, т.е. простой перебор всех возможных ключей. Если множество ключей достаточно большое, то подобрать ключ нереально. При длине ключа n бит количество возможных ключей равно 2^n. Таким образом, чем длиннее ключ, тем более стойким считается алгоритм для лобовой атаки.

Существуют различные типы атак, основанные на том, что противнику известно определенное количество пар незашифрованное сообщение - зашифрованное сообщение. При анализе зашифрованного текста противник часто применяет статистические методы анализа текста. При этом он может иметь общее представление о типе текста, например, английский или русский текст, выполнимый файл конкретной ОС, исходный текст на некотором конкретном языке программирования и т.д. Во многих случаях криптоаналитик имеет достаточно много информации об исходном тексте. Криптоаналитик может иметь возможность перехвата одного или нескольких незашифрованных сообщений вместе с их зашифрованным видом. Или криптоаналитик может знать основной формат или основные характеристики сообщения. Говорят, что криптографическая схема абсолютно безопасна, если зашифрованное сообщение не содержит никакой информации об исходном сообщении. Говорят, что криптографическая схема вычислительно безопасна, если:

1. Цена расшифровки сообщения больше цены самого сообщения.
2. Время, необходимое для расшифровки сообщения, больше срока жизни сообщения.

Дифференциальный и линейный криптоанализ

Рассмотрим в общих чертах основной подход, используемый при *дифференциальном* и *линейном криптоанализе*. И в том, и в другом случае предполагается, что известно достаточно большое количество пар (незашифрованнный текст, зашифрованный текст).

Понятие *дифференциального криптоанализа* было введено Эли Бихамом (Biham) и Ади Шамиром (Shamir) в 1990 году. Конечная задача ***дифференциального криптоанализа*** - используя свойства алгоритма, в основном свойства *S-box*, определить *подключ раунда*. Конкретный способ *дифференциального криптоанализа* зависит от рассматриваемого алгоритма шифрования.

Если в основе алгоритма лежит *сеть Фейштеля*, то можно считать, что блок m состоит из двух половин - m_0 и m_1. *Дифференциальный криптоанализ* рассматривает отличия, которые происходят в каждой половине при шифровании. (Для алгоритма *DES* "отличия" определяются с помощью операции XOR, для других алгоритмов возможен иной способ). Выбирается пара незашифрованных текстов с фиксированным отличием. Затем анализируются отличия, получившиеся после шифрования одним *раундом* алгоритма, и определяются вероятности различных ключей. Если для многих пар входных значений, имеющих одно и то же отличие X, при использовании одного и того же *подключа* одинаковыми (Y) оказываются и отличия соответствующих выходных значений, то можно говорить, что X влечет Y с определенной вероятностью. Если эта вероятность близка к единице, то можно считать, что *подключ раунда* найден с данной вероятностью. Так как *раунды* алгоритма независимы, вероятности определения *подключа* каждого *раунда* следует перемножать. Как мы помним, считается, что результат шифрования данной пары известен. Результаты *дифференциального криптоанализа* используются как при разработке конкретных *S-box*, так и при определении оптимального числа *раундов*.

Другим способом криптоанализа является ***линейный криптоанализ***, который использует линейные приближения преобразований, выполняемых алгоритмом шифрования. Данный метод позволяет найти ключ, имея достаточно большое число пар (незашифрованный текст, зашифрованный текст). Рассмотрим основные принципы, на которых базируется *линейный криптоанализ*. Обозначим

$P[1], \ldots, P[n]$ - незашифрованный блок сообщения.

$C[1], \ldots, C[n]$ - зашифрованный блок сообщения.

$K[1], \ldots, K[m]$ - ключ.

$A[i, j, \ldots, k] = A[i] \oplus A[j] \oplus \ldots \oplus A[k]$

Целью *линейного криптоанализа* является поиск линейного уравнения вида

$$P[\alpha_1, \alpha_2, \ldots, \alpha_a] \oplus C[\beta_1, \beta_2, \ldots, \beta_b] = K[\gamma_1, \ldots, \gamma_c]$$

Выполняющееся с вероятностью $p \neq 0.5$. α_i, β_i и γ_i - фиксированные позиции в блоках сообщения и ключе. Чем больше p отклоняется от 0.5, тем более подходящим считается уравнение.

Это уравнение означает, что если выполнить операцию XOR над некоторыми битами незашифрованного сообщения и над некоторыми битами зашифрованного сообщения, получится бит, представляющий собой XOR некоторых битов ключа. Это называется линейным приближением, которое может быть верным с вероятностью p.

Уравнения составляются следующим образом. Вычисляются значения левой части для большого числа пар соответствующих фрагментов незашифрованного и зашифрованного блоков. Если результат оказывается равен нулю более чем в половине случаев, то полагают, что $K[\gamma_1, \ldots, \gamma_c] = 0$. Если в большинстве случаев получается 1, полагают, что $K[\gamma_1, \ldots, \gamma_c] = 1$. Таким образом получают систему уравнений, решением которой является ключ.

Как и в случае *дифференциального криптоанализа*, результаты *линейного криптоанализа* должны учитываться при разработке алгоритмов симметричного шифрования.

Используемые критерии при разработке алгоритмов

Принимая во внимание перечисленные требования, обычно считается, что алгоритм симметричного шифрования должен:

- Манипулировать данными в больших блоках, предпочтительно размером 16 или 32 бита.
- Иметь размер блока 64 или 128 бит.
- Иметь масштабируемый ключ до 256 бит.
- Использовать простые операции, которые эффективны на микропроцессорах, т.е. исключающее или, сложение, табличные подстановки, умножение по модулю. Не должно использоваться сдвигов переменной длины, побитных перестановок или условных переходов.
- Должна быть возможность реализации алгоритма на 8-битном процессоре с минимальными требованиями к памяти.
- Использовать заранее вычисленные *подключи*. На системах с большим количеством памяти эти *подключи* могут быть заранее вычислены для ускорения работы. В случае невозможности заблаговременного вычисления *подключей* должно произойти только замедление выполнения. Всегда должна быть возможность шифрования данных без каких-либо предварительных вычислений.
- Состоять из переменного числа итераций. Для приложений с маленькой длиной ключа нецелесообразно применять большое число итераций для противостояния дифференциальным и другим атакам. Следовательно, должна быть возможность уменьшить число итераций без потери безопасности (не более чем уменьшенный размер ключа).
- По возможности не иметь слабых ключей. Если это невозможно, то количество слабых ключей должно быть минимальным, чтобы уменьшить вероятность случайного выбора одного из них. Тем не менее, все слабые ключи должны быть заранее известны, чтобы их можно было отбраковать в процессе создания ключа.
- Задействовать *подключи*, которые являются односторонним хэшем ключа. Это дает возможность использовать большие парольные фразы в качестве ключа без ущерба для безопасности.
- Не иметь линейных структур, которые уменьшают комплексность и не обеспечивают исчерпывающий поиск.
- Использовать простую для понимания разработку. Это дает возможность анализа и уменьшает закрытость алгоритма.

Большинство блочных алгоритмов основано на использовании *сети Фейштеля*, все имеют плоское пространство ключей, с возможным исключением нескольких слабых ключей.

Алгоритм DES

Принципы разработки

Самым распространенным и наиболее известным алгоритмом симметричного шифрования является ***DES*** (Data Encryption Standard). Алгоритм был разработан в 1977 году, в 1980 году был принят NIST (National Institute of Standards and Technology США) в качестве стандарта (FIPS PUB 46).

DES является классической *сетью Фейштеля* с двумя ветвями. Данные шифруются 64-битными блоками, используя 56-битный ключ. Алгоритм преобразует за несколько *раундов* 64-битный вход в 64-битный выход. Длина ключа равна 56 битам. Процесс шифрования состоит из четырех этапов. На первом из них выполняется начальная перестановка (IP) 64-битного исходного текста (забеливание), во время которой биты переупорядочиваются в соответствии со стандартной таблицей. Следующий этап состоит из 16 *раундов* одной и той же функции, которая использует операции сдвига и подстановки. На третьем этапе левая и правая половины выхода последней (16-й) итерации меняются местами. Наконец, на четвертом этапе выполняется перестановка IP^{-1} результата, полученного на третьем этапе. Перестановка IP^{-1} инверсна начальной перестановке.

Рис. 2.4. Общая схема DES

Справа на рисунке показан способ, которым используется 56-битный ключ. Первоначально ключ подается на вход функции перестановки. Затем для каждого из 16 *раундов подключ* K_i является комбинацией левого циклического сдвига и перестановки. Функция перестановки одна и та же для каждого *раунда*, но *подключи* K_i для каждого *раунда* получаются разные вследствие повторяющегося сдвига битов ключа.

Шифрование

Начальная перестановка

Начальная перестановка и ее инверсия определяются стандартной таблицей. Если `M` - это произвольные 64 бита, то `X = IP (M)` - переставленные 64 бита. Если применить обратную функцию перестановки $Y = IP^{-1} (X) = IP^{-1} (IP(M))$, то получится первоначальная последовательность битов.

Последовательность преобразований отдельного раунда

Теперь рассмотрим последовательность преобразований, используемую в каждом *раунде*.

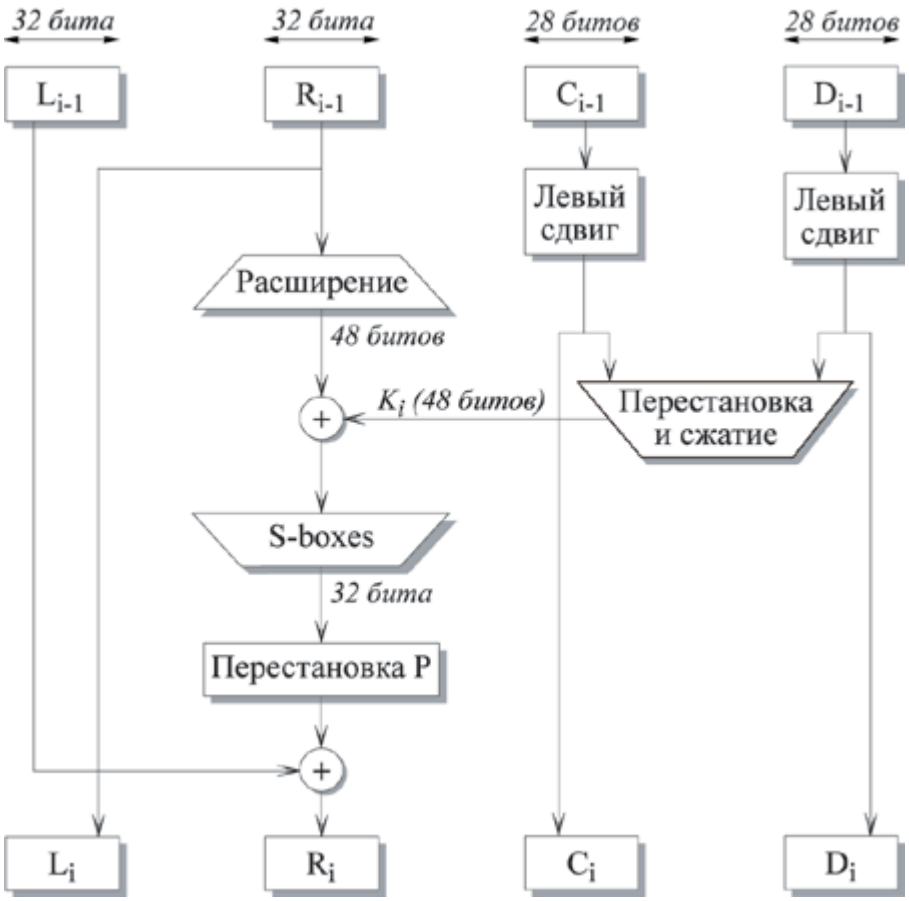

Рис. 2.5. I-ый раунд DES

64-битный входной блок проходит через 16 *раундов*, при этом на каждой итерации получается промежуточное 64-битное значение. Левая и правая части каждого промежуточного значения трактуются как отдельные 32-битные значения, обозначенные `L` и `R`. Каждую итерацию можно описать следующим образом:

$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$$

Где $\oplus$ обозначает операцию `XOR`.

Таким образом, выход левой половины L_i равен входу правой половины R_{i-1}. Выход правой половины R_i является результатом применения операции XOR к L_{i-1} и функции F, зависящей от R_{i-1} и K_i.

Рассмотрим функцию F более подробно.

R_i, которое подается на вход функции F, имеет длину 32 бита. Вначале R_i расширяется до 48 битов, используя таблицу, которая определяет перестановку плюс расширение на 16 битов. Расширение происходит следующим образом. 32 бита разбиваются на группы по 4 бита и затем расширяются до 6 битов, присоединяя крайние биты из двух соседних групп. Например, если часть входного сообщения

```
. . . efgh ijkl mnop . . .
```

то в результате расширения получается сообщение

```
. . . defghi hijklm lmnopq . . .
```

После этого для полученного 48-битного значения выполняется операция XOR с 48-битным *подключом* K_i. Затем полученное 48-битное значение подается на вход функции подстановки, результатом которой является 32-битное значение.

Подстановка состоит из восьми *S-boxes* , каждый из которых на входе получает 6 бит, а на выходе создает 4 бита. Эти преобразования определяются специальными таблицами. Первый и последний биты входного значения *S-box* определяют номер строки в таблице, средние 4 бита определяют номер столбца. Пересечение строки и столбца определяет 4-битный выход. Например, если входом является 011011, то номер строки равен 01 (строка 1) и номер столбца равен 1101 (столбец 13). Значение в строке 1 и столбце 13 равно 5, т.е. выходом является 0101.

Далее полученное 32-битное значение обрабатывается с помощью перестановки P, целью которой является максимальное переупорядочивание битов, чтобы в следующем *раунде* шифрования с большой вероятностью каждый бит обрабатывался другим *S-box*.

Создание подключей

Ключ для отдельного *раунда* K_i состоит из 48 битов. Ключи K_i получаются по следующему алгоритму. Для 56-битного ключа, используемого на входе алгоритма, вначале выполняется перестановка в соответствии с таблицей Permuted Choice 1 (PC-1). Полученный 56-битный ключ разделяется на две 28-битные части, обозначаемые как C_0 и D_0 соответственно. На каждом *раунде* C_i и D_i независимо циклически сдвигаются влево на 1 или 2 бита, в зависимости от номера *раунда*. Полученные значения являются входом следующего *раунда*. Они также представляют собой вход в Permuted Choice 2 (PC-2), который создает 48-битное выходное значение, являющееся входом функции $F(R_{i-1}, K_i)$.

Дешифрование

Процесс дешифрования аналогичен процессу шифрования. На входе алгоритма используется зашифрованный текст, но ключи K_i используются в обратной последовательности. K_{16} используется на первом *раунде*, K_1 используется на последнем *раунде*. Пусть выходом i-ого *раунда* шифрования будет $L_i||R_i$. Тогда соответствующий вход (16-i)-ого *раунда* дешифрования будет $R_i||L_i$.

После последнего *раунда* процесса расшифрования две половины выхода меняются местами так, чтобы вход заключительной перестановки IP^{-1} был $R_{16}||L_{16}$. Выходом этой стадии является незашифрованный текст.

Проверим корректность процесса дешифрования. Возьмем зашифрованный текст и ключ и используем их в качестве входа в алгоритм. На первом шаге выполним начальную перестановку IP и получим 64-битное значение $L^d_0||R^d_0$. Известно, что IP и IP^{-1} взаимнообратны. Следовательно

$$L^d_0||R^d_0 = IP\ (\text{зашифрованный текст})$$
$$\text{Зашифрованный текст} = IP^{-1}(R_{16}||L_{16})$$
$$L^d_0||R^d_0 = IP(IP^{-1}(R_{16}||L_{16})) = R_{16}||L_{16}$$

Таким образом, вход первого *раунда* процесса дешифрования эквивалентен 32-битному выходу 16-ого *раунда* процесса шифрования, у которого левая и правая части записаны в обратном порядке.

Теперь мы должны показать, что выход первого *раунда* процесса дешифрования эквивалентен 32-битному входу 16-ого *раунда* процесса шифрования. Во-первых, рассмотрим процесс шифрования. Мы видим,что

$$L_{16} = R_{15}$$
$$R_{16} = L_{15} \oplus F(R_{15}, K_{16})$$

При дешифровании:

$$L^d_1 = R^d_0 = L_{16} = R_{15}$$
$$R^d_1 = L^d_0 \oplus F(R^d_0, K_{16}) =$$
$$= R_{16} \oplus F(R^d_0, K_{16}) =$$
$$= (L_{15} \oplus F(R_{15}, K_{16})) \oplus F(R_{15}, K_{16})$$

XOR имеет следующие свойства:

$$(A \oplus B) \oplus C = A \oplus (B \oplus C)$$
$$D \oplus D = 0$$
$$E \oplus 0 = E$$

Таким образом, мы имеем $L^d_1 = R_{15}$ и $R^d_1 = L_{15}$. Следовательно, выход первого *раунда* процесса дешифрования есть $L_{15}||R_{15}$, который является перестановкой входа 16-го *раунда* шифрования. Легко показать, что данное соответствие выполняется все 16 *раундов*. Мы можем описать этот процесс в общих терминах. Для i-ого *раунда* шифрующего алгоритма:

$$L_i = R_{i-1}$$
$$R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$$

Эти равенства можно записать по-другому:

$$R_{i-1} = L_i$$
$$L_{i-1} = R_i \oplus F(R_{i-1}, K_i) = R_i \oplus F(L_i, K_i)$$

Таким образом, мы описали входы i-ого *раунда* как функцию выходов.

Выход последней стадии процесса дешифрования есть $R_0||L_0$. Чтобы входом IP^{-1} стадии было $R_0||L_0$, необходимо поменять местами левую и правую части. Но

$$IP^{-1}(R_0||L_0) = IP^{-1}(IP\ (\text{незашифрованный текст})) = \text{незашифрованный текст}$$

Т.е. получаем незашифрованный текст, что и демонстрирует возможность дешифрования *DES*.

Проблемы DES

Так как длина ключа равна 56 битам, существует 2^{56} возможных ключей. На сегодня такая длина ключа недостаточна, поскольку допускает успешное применение лобовых атак. Альтернативой *DES* можно считать *тройной DES*, IDEA, а также алгоритм Rijndael, принятый в качестве нового стандарта на алгоритмы симметричного шифрования.

Также без ответа пока остается вопрос, возможен ли криптоанализ с использованием существующих характеристик алгоритма *DES*. Основой алгоритма являются восемь таблиц подстановки, или *S-boxes*, которые применяются в каждой итерации. Существует опасность, что эти *S-boxes* конструировались таким образом, что криптоанализ возможен для взломщика, который знает слабые места *S-boxes*. В течение многих лет обсуждалось как стандартное, так и неожиданное поведение *S-boxes*, но все-таки никому не удалось обнаружить их фатально слабые места.

Алгоритм тройной DES

В настоящее время основным недостатком *DES* считается маленькая длина ключа, поэтому уже давно начали разрабатываться различные альтернативы этому алгоритму шифрования. Один из подходов состоит в том, чтобы разработать новый алгоритм, и успешный тому пример - IDEA. Другой подход предполагает повторное применение шифрования с помощью *DES* с использованием нескольких ключей.

Недостатки двойного DES

Простейший способ увеличить длину ключа состоит в повторном применении *DES* с двумя разными ключами. Используя незашифрованное сообщение P и два ключа K_1 и K_2, зашифрованное сообщение C можно получить следующим образом:

$$C = E_{k2}\ [E_{k1}\ [P]]$$

Для дешифрования требуется, чтобы два ключа применялись в обратном порядке:

$$P = D_{k1}\ [D_{k2}\ [C]]$$

В этом случае длина ключа равна $56 * 2 = 112$ бит.

Атака "встреча посередине"

Для приведенного выше алгоритма двойного *DES* существует так называемая атака "встреча посередине". Она основана на следующем свойстве алгоритма. Мы имеем

$$C = E_{k2}\ [E_{k1}\ [P]]$$

Тогда

$$X = E_{k1}\ [P] = D_{k2}\ [C].$$

Атака состоит в следующем. Требуется, чтобы атакующий знал хотя бы одну пару незашифрованный текст и соответствующий ему зашифрованный текст: (P, C). В этом случае, во-первых, шифруется P для всех возможных 2^{56} значений K_1. Этот результат запоминается в таблице, и затем таблица упорядочивается по значению X. Следующий шаг состоит в дешифровании C, с применением всех возможных 2^{56} значений K_2. Для каждого выполненного дешифрования ищется равное ему значение в первой таблице. Если соответствующее значение найдено, то считается, что эти ключи могут быть правильными, и они проверяются для следующей известной пары незашифрованный текст, зашифрованный текст.

Если известна только одна пара значений незашифрованный текст, зашифрованный текст, то может быть получено достаточно большое число неверных значений ключей. Но если противник имеет возможность перехватить хотя бы две пары значений (незашифрованный текст - зашифрованный текст), то сложность взлома двойного DES фактически становится равной сложности взлома обычного *DES*, т.е. 2^{56}.

Тройной DES с двумя ключами

Очевидное противодействие атаке "встреча посередине" состоит в использовании третьей стадии шифрования с тремя различными ключами. Это поднимает стоимость лобовой атаки до 2^{168}, которая на сегодняшний день считается выше практических возможностей. Но при этом длина ключа равна $56 * 3 = 168$ бит, что иногда бывает громоздко.

В качестве альтернативы предлагается метод тройного шифрования, использующий только два ключа. В этом случае выполняется последовательность зашифрование-расшифрование-зашифрование (EDE).

$$C = E_{K1}\ [D_{K2}\ [E_{K1}\ [P]]]$$

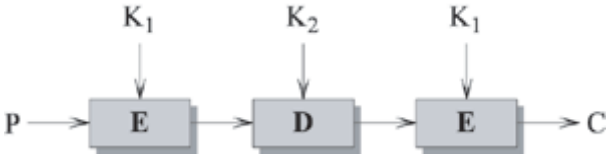

Рис. 2.6. Шифрование тройным DES

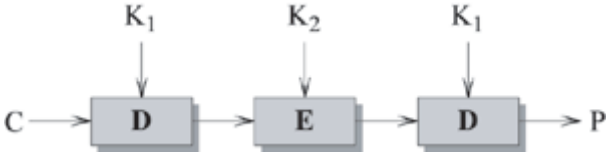

Рис. 2.7. Дешифрование тройным DES

Не имеет большого значения, что используется на второй стадии: шифрование или дешифрование. В случае использования дешифрования существует только то преимущество, что можно *тройной DES* свести к обычному одиночному *DES*, используя $K_1 = K_2$:

$$C = E_{K1}\ [D_{K1}\ [E_{K1}\ [P]]] = E_{K1}\ [P]$$

Тройной DES является достаточно популярной альтернативой *DES* и используется при управлении ключами в стандартах ANSI X9.17 и ISO 8732 и в PEM (Privacy Enhanced Mail).

Известных криптографических атак на *тройной DES* не существует. Цена подбора ключа в *тройном DES* равна 2^{112}.

3. **Лекция: Алгоритмы симметричного шифрования. Часть 2**

Алгоритм Blowfish

Blowfish является сетью Фейштеля, у которой количество итераций равно 16. Длина блока равна 64 битам, ключ может иметь любую длину в пределах 448 бит. Хотя перед началом любого шифрования выполняется сложная фаза инициализации, само шифрование данных выполняется достаточно быстро.

Алгоритм предназначен в основном для приложений, в которых ключ меняется нечасто, к тому же существует фаза начального рукопожатия, во время которой происходит аутентификация сторон и согласование общих параметров и секретов. Классическим примером подобных приложений является сетевое взаимодействие. При реализации на 32-битных микропроцессорах с большим кэшем данных *Blowfish* значительно быстрее DES.

Алгоритм состоит из двух частей: расширение ключа и шифрование данных. Расширение ключа преобразует ключ длиной, по крайней мере, 448 бит в несколько массивов *подключей* общей длиной 4168 байт.

В основе алгоритма лежит сеть Фейштеля с 16 итерациями. Каждая итерация состоит из перестановки, зависящей от ключа, и подстановки, зависящей от ключа и данных. Операциями являются XOR и сложение 32-битных слов.

Blowfish использует большое количество *подключей*. Эти ключи должны быть вычислены заранее, до начала любого шифрования или дешифрования данных. Элементы алгоритма:

1. P - массив, состоящий из восемнадцати 32-битных *подключей*:

   $P_1, P_2, \ldots, P_{18}$.

2. Четыре 32-битных *S-boxes* с 256 входами каждый. Первый индекс означает номер *S-box*, второй индекс - номер входа.
3. $S_{1,0}, S_{1,1}, \ldots S_{1,255}$;
4. $S_{2,0}, S_{2,1}, \ldots S_{2,255}$;
5. $S_{3,0}, S_{3,1}, \ldots S_{3,255}$;
6. $S_{4,0}, S_{4,1}, \ldots S_{4,255}$;

Метод, используемый для вычисления этих *подключей*, будет описан ниже.

Шифрование

Входом является 64-битный элемент данных X, который делится на две 32-битные половины, X_l и X_r.

$X_l = X_l$ XOR P_i
$X_r = F(X_l)$ XOR X_r
Swap X_l and X_r

Функция F

Разделить X_l на четыре 8-битных элемента A, B, C, D.

$$F(X_l) = ((S_{1,A} + S_{2,B} \bmod 2^{32}) \text{ XOR } S_{3,C}) + S_{4,D} \bmod 2^{32}$$

Дешифрование отличается от шифрования тем, что P_i используются в обратном порядке.

Генерация подключей

Подключи вычисляются с использованием самого *алгоритма Blowfish*.

1. Инициализировать первый P-массив и четыре *S-boxes* фиксированной строкой.
2. Выполнить операцию XOR P_1 с первыми 32 битами ключа, операцию XOR P_2 со вторыми 32 битами ключа и т.д. Повторять цикл до тех пор, пока весь P-массив не будет побитово сложен со всеми битами ключа. Для коротких ключей выполняется конкатенация ключа с самим собой.
3. Зашифровать нулевую строку *алгоритмом Blowfish*, используя *подключи*, описанные в пунктах (1) и (2).
4. Заменить P_1 и P_2 выходом, полученным на шаге (3).
5. Зашифровать выход шага (3), используя *алгоритм Blowfish* с модифицированными *подключами*.
6. Заменить P_3 и P_4 выходом, полученным на шаге (5).
7. Продолжить процесс, заменяя все элементы P-массива, а затем все четыре *S-boxes*, выходами соответствующим образом модифицированного *алгоритма Blowfish*.

Для создания всех *подключей* требуется 521 итерация.

Алгоритм IDEA

IDEA (International Data Encryption Algorithm) является блочным симметричным алгоритмом шифрования, разработанным Сюдзя Лай (Xuejia Lai) и Джеймсом Массей (James Massey) из швейцарского федерального института технологий. Первоначальная версия была опубликована в 1990 году. Пересмотренная версия алгоритма, усиленная средствами защиты от дифференциальных криптографических атак, была представлена в 1991 году и подробно описана в 1992 году.

IDEA является одним из нескольких симметричных криптографических алгоритмов, которыми первоначально предполагалось заменить DES.

Принципы разработки

IDEA является блочным алгоритмом, который использует 128-битовый *ключ для шифрования* данных блоками по 64 бита.

Целью разработки *IDEA* было создание относительно стойкого криптографического алгоритма с достаточно простой реализацией.

Криптографическая стойкость

Следующие характеристики *IDEA* характеризуют его криптографическую стойкость:

1. **Длина блока**: длина блока должна быть достаточной, чтобы скрыть все статистические характеристики исходного сообщения. С другой стороны, сложность реализации криптографической функции возрастает экспоненциально в соответствии с размером блока. Использование блока размером в 64 бита в 90-е годы означало достаточную силу. Более того, использование режима шифрования CBC говорит о дальнейшем усилении этого аспекта алгоритма.

2. **Длина ключа**: длина ключа должна быть достаточно большой для того, чтобы предотвратить возможность простого перебора ключа. При длине ключа 128 бит *IDEA* считается достаточно безопасным.
3. **Конфузия**: зашифрованный текст должен зависеть от ключа сложным и запутанным способом.
4. **Диффузия**: каждый бит незашифрованного текста должен влиять на каждый бит зашифрованного текста. Распространение одного незашифрованного бита на большое количество зашифрованных битов скрывает статистическую структуру незашифрованного текста. Определить, как статистические характеристики зашифрованного текста зависят от статистических характеристик незашифрованного текста, должно быть непросто. *IDEA* с этой точки зрения является очень эффективным алгоритмом.

В *IDEA* два последних пункта выполняются с помощью трех операций. Это отличает его от DES, где все постороено на использовании операции `XOR` и маленьких нелинейных *S-boxes*.

Каждая операция выполняется над двумя 16-битными входами и создает один 16-битный выход. Этими операциями являются:

1. Побитовое исключающее `OR`, обозначаемое как $\oplus$.
2. Сумма целых по модулю 2^{16} (по модулю `65536`), при этом входы и выходы трактуются как беззнаковые 16-битные целые. Эту операцию обозначим как `+`.
3. Умножение целых по модулю $2^{16} + 1$ (по модулю `65537`), при этом входы и выходы трактуются как беззнаковые 16-битные целые, за исключением того, что блок из одних нулей трактуется как 2^{16}. Эту операцию обозначим как `•`.

Эти три операции являются несовместимыми в том смысле, что:

1. Не существует пары из трех операций, удовлетворяющих дистрибутивному закону. Например

   ```
   a • (b + c) <> (a • b) + (a • c)
   ```

2. Не существует пары из трех операций, удовлетворяющих ассоциативному закону. Например

   ```
   a + (b ⊕ c) <> (a + b) ⊕ c
   ```

Использование комбинации из этих трех операций обеспечивает комплексную трансформацию входа, делая криптоанализ более трудным, чем в таком алгоритме как DES, основанном исключительно на функции `XOR`.

Шифрование

Рассмотрим общую схему шифрования *IDEA*. Как и в любом алгоритме шифрования, здесь существует два входа: незашифрованный блок и ключ. В данном случае незашифрованный блок имеет длину 64 бита, ключ имеет длину 128 бит.

Алгоритм IDEA состоит из восьми *раундов*, за которыми следует заключительное преобразование. Алгоритм разделяет блок на четыре 16-битных подблока. Каждый *раунд* получает на входе четыре 16-битных подблока и создает четыре 16-битных выходных подблока. Заключительное преобразование также получает на входе четыре 16-битных подблока и создает четыре 16-битных подблока. Каждый *раунд* использует шесть 16-битных ключей, заключительное преобразование использует четыре *подключа*, т.е. всего в алгоритме используется 52 *подключа*.

Рис. 3.1. Алгоритм IDEA

Последовательность преобразований отдельного раунда

Рассмотрим последовательность преобразований отдельного *раунда*.

Одним из основных элементов алгоритма, обеспечивающих диффузию, является структура, называемая MA (умножение/сложение):

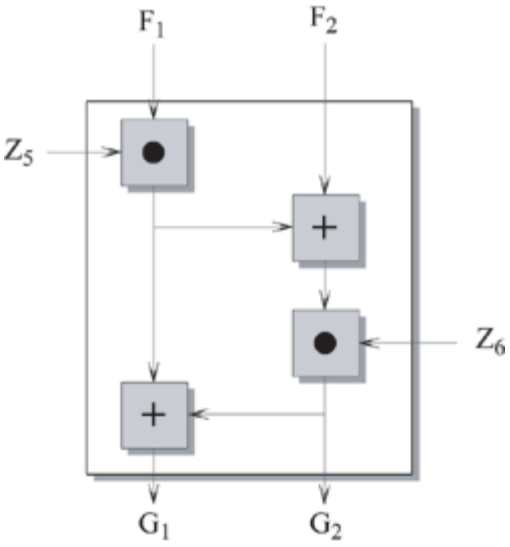

Рис. 3.2. Структура MA (умножение/сложение)

На вход этой структуре подаются два 16-битных значения и два 16-битных *подключа*, на выходе создаются два 16-битных значения. Исчерпывающая компьютерная проверка показывает, что каждый бит выхода этой структуры зависит от каждого бита входов незашифрованного блока и от каждого бита *подключей*. Данная структура повторяется в алгоритме восемь раз, обеспечивая высокоэффективную диффузию.

Раунд начинается с преобразования, которое комбинирует четыре входных подблока с четырьмя *подключами*, используя операции сложения и умножения. Четыре выходных блока этого преобразования комбинируются, используя операцию XOR для формирования двух 16-битных блоков, которые являются входами МА структуры. Кроме того, МА структура имеет на входе еще два *подключа* и создает два 16-битных выхода.

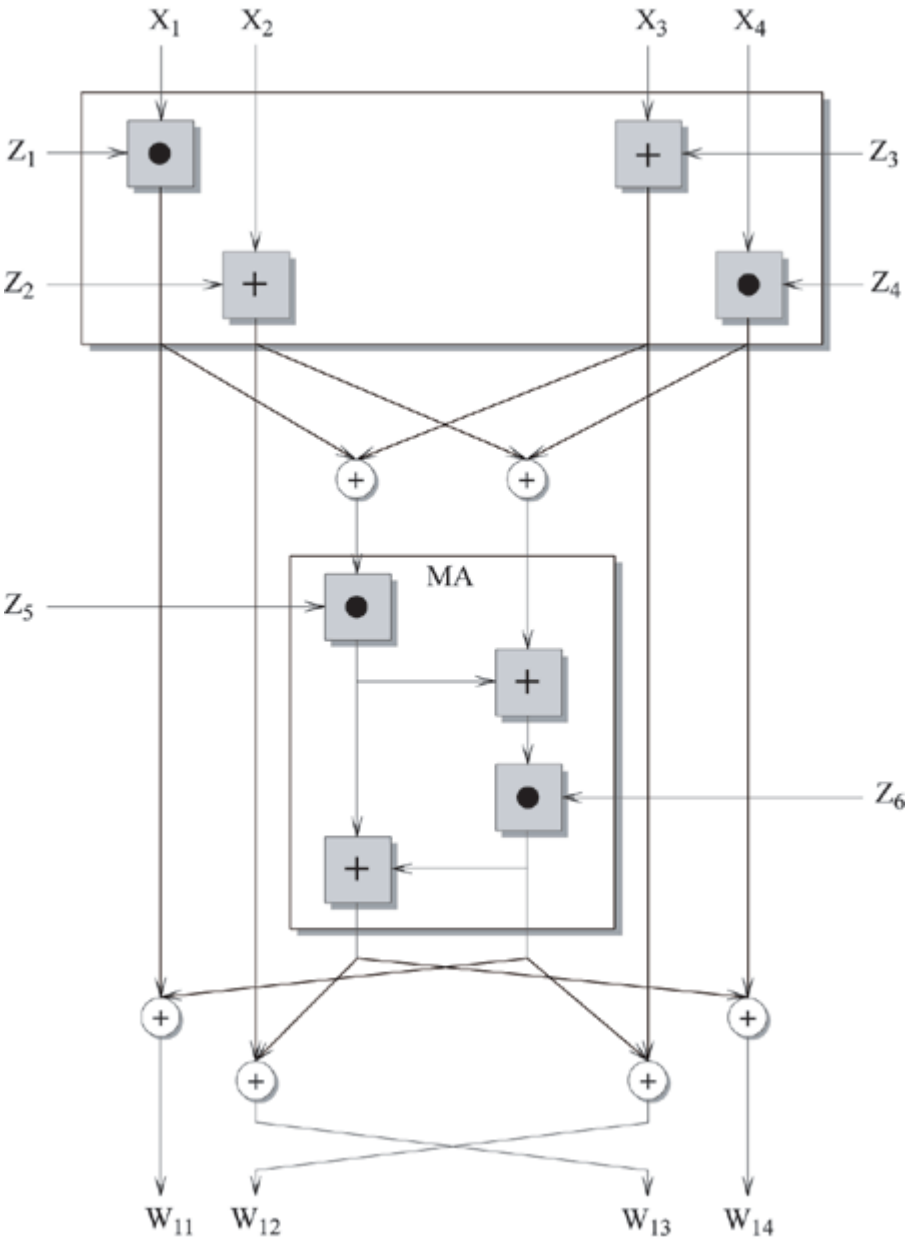

Рис. 3.3. I-ый раунд IDEA

В заключении четыре выходных подблока первого преобразования комбинируются с двумя выходными подблоками MA структуры, используя XOR для создания четырех выходных подблоков данной итерации. Заметим, что два выхода, которые частично создаются вторым и третьим входами (X_2 и X_3), меняются местами для создания второго и третьего выходов (W_{12} и W_{13}). Это увеличивает перемешивание битов и делает алгоритм более стойким для дифференциального криптоанализа.

Рассмотрим девятый *раунд алгоритма*, обозначенный как заключительное преобразование. Это та же структура, что была описана выше. Единственная разница состоит в том, что второй и третий входы меняются местами. Это сделано для того, чтобы дешифрование имело ту же структуру, что и шифрование. Заметим, что девятая

стадия требует только четыре входных *подключа*, в то время как для первых восьми стадий для каждой из них необходимо шесть входных *подключей*.

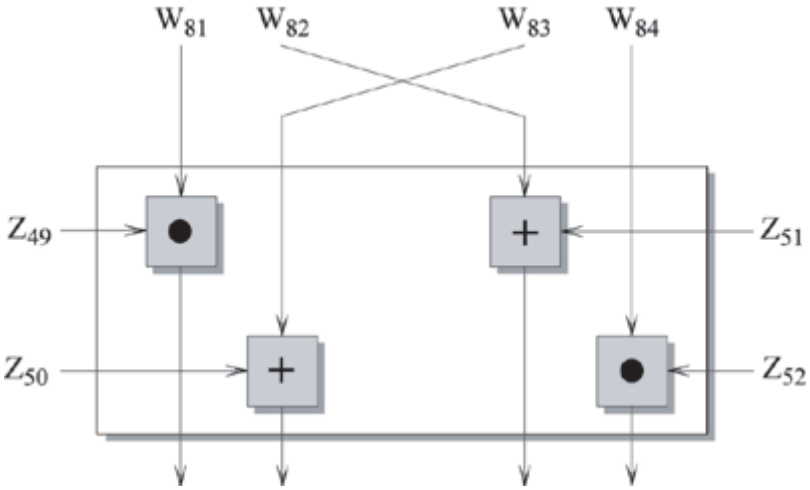

Рис. 3.4. Заключительное преобразование

Создание подключей

Пятьдесят два 16-битных *подключа* создаются из 128-битного *ключа шифрования* следующим образом. Первые восемь *подключей*, которые обозначим как Z_1, Z_2, ..., Z_8, получаются непосредственно из ключа, при этом Z_1 равен первым 16 битам, Z_2 равен следующим 16 битам и т.д. Затем происходит циклический сдвиг ключа влево на 25 битов, и создаются следующие восемь *подключей*. Эта поцедура повторяется до тех пор, пока не будут созданы все 52 *подключа*.

Заметим, что каждый первый *подключ раунда* получен из своего подмножества битов ключа. Если весь ключ обозначить как $Z_{[1..128]}$, то первыми ключами в восьми *раундах* будут:

$$Z_1 = Z_{[1..16]} \quad Z_{25} = Z_{[76..91]}$$
$$Z_7 = Z_{[97..112]} \quad Z_{31} = Z_{[44..59]}$$
$$Z_{13} = Z_{[90..105]} \quad Z_{37} = Z_{[37..52]}$$
$$Z_{19} = Z_{[83..98]} \quad Z_{43} = Z_{[30..45]}$$

Хотя на каждом *раунде* за исключением первого и восьмого используются только 96 битов *подключа*, множество битов ключа на каждой итерации не пересекаются, и не существует отношения простого сдвига между *подключами* разных *раундов*. Это происходит потому, что на каждом *раунде* используется только шесть *подключей*, в то время как при каждой ротации ключа получается восемь *подключей*.

Дешифрование

Процесс дешифрования аналогичен процессу шифрования. Дешифрование состоит в использовании зашифрованного текста в качестве входа в ту же самую структуру *IDEA*, но с другим набором ключей. Дешифрующие ключи $U_1, \ldots, U_{52}$ получаются из шифрующих ключей следующим образом:

1. Первые четыре *подключа* i-ого *раунда* дешифрования получаются из первых четырех *подключей* (10-i)-го *раунда* шифрования, где стадия заключительного преобразования считается 9-м *раундом*. Первый и четвертый ключи дешифрования эквивалентны мультипликативной инверсии по модулю $(2^{16} + 1)$ соответствующих первого и четвертого *подключей* шифрования. Для *раундов* со

2 по 8 второй и третий *подключи* дешифрования эквивалентны аддитивной инверсии по модулю (2^{16}) соответствующих третьего и второго *подключей* шифрования. Для *раундов* 1 и 9 второй и третий *подключи* дешифрования эквивалентны аддитивной инверсии по модулю (2^{16}) соответствующих второго и третьего *подключей* шифрования.

2. Для первых восьми *раундов* последние два *подключа* i *раунда* дешифрования эквивалентны последним двум *подключам* ($9 - i$) *раунда* шифрования.

Для мультипликативной инверсии используется нотация Z_j^{-1}, т.е.:

$$Z_j \cdot Z_j^{-1} = 1 \bmod (2^{16} + 1)$$

Так как $2^{16} + 1$ является простым числом, каждое ненулевое целое $Z_j \leq 2^{16}$ имеет уникальную мультипликативную инверсию по модулю ($2^{16} + 1$). Для аддитивной инверсии используется нотация ($-Z_j$), таким образом, мы имеем: $-Z_j + Z_j = 0 \bmod (2^{16})$

Для доказательства того, что алгоритм дешифрования с соответствующими *подключами* имеет корректный результат, рассмотрим одновременно процессы шифрования и дешифрования. Каждый из восьми *раундов* разбит на две стадии преобразования, первая из которых называется трансформацией, а вторая шифрованием.

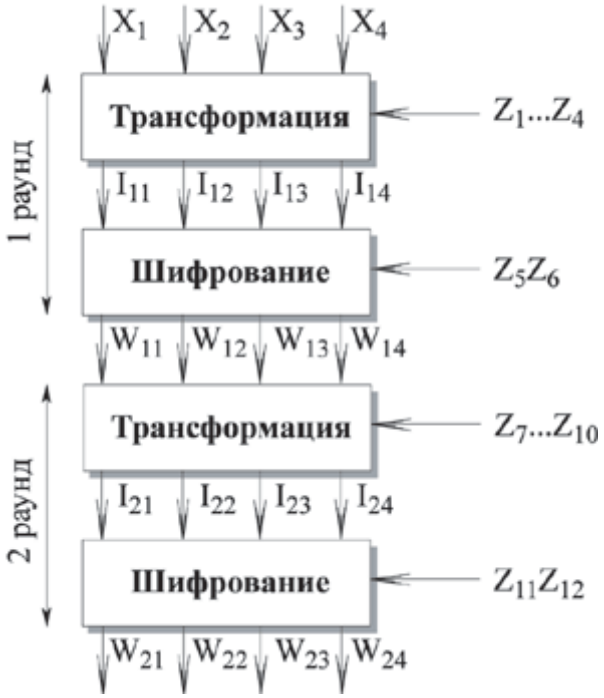

Рис. 3.5. Шифрование IDEA

Рис. 3.6. Дешифрование IDEA

Рассмотрим преобразования, выполняемые в прямоугольниках на обоих рисунках. При шифровании поддерживаются следующие соотношения на выходе трансформации:

$$Y_1 = W_{81} \bullet Z_{49} \quad Y_3 = W_{82} + Z_{51}$$
$$Y_2 = W_{83} + Z_{50} \quad Y_4 = W_{84} \bullet Z_{52}$$

Первая стадия первого *раунда* процесса дешифрования поддерживает следующие соотношения:

$$J_{11} = Y_1 \bullet U_1 \quad J_{13} = Y_3 + U_3$$
$$J_{12} = Y_2 + U_2 \quad J_{14} = Y_4 \bullet U_4$$

Подставляя соответствующие значения, получаем:

$$J_{11} = Y_1 \bullet Z_{49}^{-1} = W_{81} \bullet Z_{49} \bullet Z_{49}^{-1} = W_{81}$$
$$J_{12} = Y_2 + -Z_{50} = W_{83} + Z_{50} = W_{83} + Z_{50} + -Z_{50} = W_{83}$$
$$J_{13} = Y_3 + -Z_{51} = W_{82} + Z_{51} + -Z_{51} = W_{82}$$
$$J_{14} = Y_4 \bullet Z_{52}^{-1} = W_{84} \bullet Z_{52} \bullet Z_{52}^{-1} = W_{84}$$

Таким образом, выход первой стадии процесса дешифрования эквивалентен входу последней стадии процесса шифрования за исключением чередования второго и третьего блоков. Теперь рассмотрим следующие отношения:

$$W_{81} = I_{81} \oplus MA_R(I_{81} \oplus I_{83}, I_{82} \oplus I_{84})$$
$$W_{82} = I_{83} \oplus MA_R(I_{81} \oplus I_{83}, I_{82} \oplus I_{84})$$
$$W_{83} = I_{82} \oplus MA_L(I_{81} \oplus I_{83}, I_{82} \oplus I_{84})$$
$$W_{84} = I_{84} \oplus MA_L(I_{81} \oplus I_{83}, I_{82} \oplus I_{84})$$

Где $MA_R(X, Y)$ есть правый выход MA структуры с входами X и Y , и $MA_L(X, Y)$ есть левый выход MA структуры с входами X и Y. Теперь получаем

$$\begin{aligned} V_{11} = J_{11} &\oplus MA_R(J_{11} \oplus J_{13}, J_{12} \oplus J_{14}) = \\ & W_{81} \oplus MA_R(W_{81} \oplus W_{82}, W_{83} \oplus W_{84}) = \\ & I_{81} \oplus MA_R(I_{81} \oplus I_{83}, I_{82} \oplus I_{84}) \oplus \\ & MA_R[I_{81} \oplus MA_R(I_{81} \oplus I_{83}, I_{82} \oplus I_{84}) \oplus I_{83} \oplus \\ & MA_R(I_{81} \oplus I_{83}, I_{82} \oplus I_{84}), I_{82} \oplus \end{aligned}$$

$$MA_L(I_{81} \oplus I_{83},\ I_{82} \oplus I_{84}) \oplus I_{84} \oplus$$
$$MA_L(I_{81} \oplus I_{83},\ I_{82} \oplus I_{84})\ ] =$$
$$I_{81} \oplus MA_R(I_{81} \oplus I_{83},\ I_{82} \oplus I_{84}) \oplus$$
$$MA_R(I_{81} \oplus I_{83},\ I_{82} \oplus I_{84}) =$$
$$I_{81}$$

Аналогично мы имеем

$$V_{12} = I_{83}$$
$$V_{13} = I_{82}$$
$$V_{14} = I_{84}$$

Таким образом, выход второй стадии процесса дешифрования эквивалентен входу предпоследней стадии процесса шифрования за исключением чередования второго и третьего подблоков. Аналогично можно показать, что

$$V_{81} = I_{11}$$
$$V_{82} = I_{13}$$
$$V_{83} = I_{12}$$
$$V_{84} = I_{14}$$

Наконец, так как выход трансформации процесса дешифрования эквивалентен первой стадии процесса шифрования за исключением чередования второго и третьего подблоков, получается, что выход всего процесса шифрования эквивалентен входу процесса шифрования.

Алгоритм ГОСТ 28147

Алгоритм ГОСТ 28147 является отечественным стандартом для алгоритмов симметричного шифрования. ***ГОСТ 28147*** разработан в 1989 году, является блочным алгоритмом шифрования, длина блока равна 64 битам, длина ключа равна 256 битам, количество *раундов* равно 32. Алгоритм представляет собой классическую сеть Фейштеля.

$$L_i = R_{i-1}$$
$$R_i = L_i \oplus f\ (R_{i-1},\ K_i)$$

Функция F проста. Сначала правая половина и i-ый *подключ* складываются по модулю 2^{32}. Затем результат разбивается на восемь 4-битовых значений, каждое из которых подается на вход *S-box*. *ГОСТ 28147* использует восемь различных *S-boxes*, каждый из которых имеет 4-битовый вход и 4-битовый выход. Выходы всех *S-boxes* объединяются в 32-битное слово, которое затем циклически сдвигается на 11 битов влево. Наконец, с помощью XOR результат объединяется с левой половиной, в результате чего получается новая правая половина.

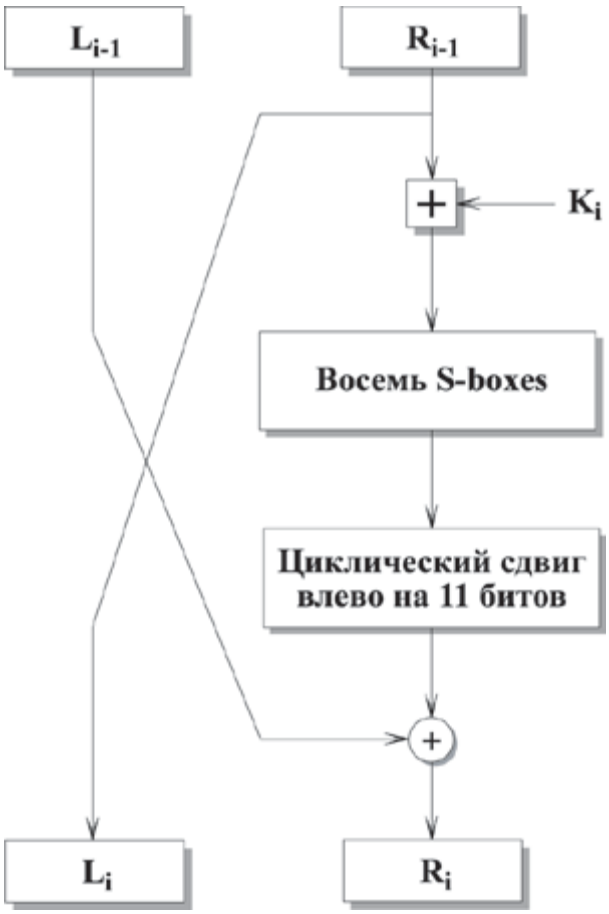

Рис. 3.7. I-ый раунд ГОСТ 28147

Генерация ключей проста. 256-битный ключ разбивается на восемь 32-битных *подключей*. Алгоритм имеет 32 *раунда*, поэтому каждый *подключ* используется в четырех *раундах* по следующей схеме:

Раунд	1	2	3	4	5	6	7	8
Подключ	1	2	3	4	5	6	7	8
Раунд	9	10	11	12	13	14	15	16
Подключ	1	2	3	4	5	6	7	8
Раунд	17	18	19	20	21	22	23	24
Подключ	1	2	3	4	5	6	7	8
Раунд	25	26	27	28	29	30	31	32
Подключ	8	7	6	5	4	3	2	1

Считается, что стойкость *алгоритма ГОСТ 28147* во многом определяется структурой *S-boxes*. Долгое время структура *S-boxes* в открытой печати не публиковалась. В настоящее время известны *S-boxes*, которые используются в приложениях Центрального Банка РФ и считаются достаточно сильными. Напомню, что входом и выходом *S-box* являются 4-битные числа, поэтому каждый *S-box* может быть представлен в виде строки чисел от 0 до 15, расположенных в некотором порядке. Тогда порядковый номер числа будет являться входным значением *S-box*, а само число - выходным значением *S-box*.

1-ый S-box	4	10	9	2	13	8	0	14
	6	11	1	12	7	15	5	3
2-ой S-box	14	11	4	12	6	13	15	10

	2	3	8	1	0	7	5	9
3-ий S-box	5	8	1	13	10	3	4	2
	14	15	12	7	6	0	9	11
4-ый S-box	7	13	10	1	0	8	9	15
	14	4	6	12	11	2	5	3
5-ый S-box	6	12	7	1	5	15	13	8
	4	10	9	14	0	3	11	2
6-ой S-box	4	11	10	0	7	2	1	13
	3	6	8	5	9	12	15	14
7-ой S-box	13	11	4	1	3	15	5	9
	0	10	14	7	6	8	2	12
8-ой S-box	1	15	13	0	5	7	10	4
	9	2	3	14	6	11	8	12

Основные различия между DES и *ГОСТ 28147* следующие:

- DES использует гораздо более сложную процедуру создания *подключей*, чем *ГОСТ 28147*. В *ГОСТ* эта процедура очень проста.
- В DES применяется 56-битный ключ, а в *ГОСТ 28147* - 256-битный. При выборе сильных *S-boxes ГОСТ 28147* считается очень стойким.
- У *S-boxes* DES 6-битовые входы и 4-битовые выходы, а у *S-boxes ГОСТ 28147* 4-битовые входы и выходы. В обоих алгоритмах используется по восемь *S-boxes*, но размер *S-box ГОСТ 28147* существенно меньше размера *S-box* DES.
- В DES применяются нерегулярные перестановки P, в *ГОСТ 28147* используется 11-битный циклический сдвиг влево. Перестановка DES увеличивает лавинный эффект. В *ГОСТ 28147* изменение одного входного бита влияет на один *S-box* одного *раунда*, который затем влияет на два *S-boxes* следующего *раунда*, три *S-boxes* следующего и т.д. В *ГОСТ 28147* требуется 8 *раундов* прежде, чем изменение одного входного бита повлияет на каждый бит результата; DES для этого нужно только 5 *раундов*.
- В DES 16 *раундов*, в *ГОСТ 28147* - 32 *раунда*, что делает его более стойким к дифференциальному и линейному криптоанализу.

Режимы выполнения алгоритмов симметричного шифрования

Для любого симметричного блочного алгоритма шифрования определено четыре режима выполнения.

ECB - Electronic Codebook - каждый блок из 64 битов незашифрованного текста шифруется независимо от остальных блоков, с применением одного и того же *ключа шифрования*. Типичные приложения - безопасная передача одиночных значений (например, криптографического ключа).

CBC - Cipher Block Chaining - вход криптографического алгоритма является результатом применения операции XOR к следующему блоку незашифрованного текста и предыдущему блоку зашифрованного текста. Типичные приложения - общая блокоориентированная передача, аутентификация.

CFB - Cipher Feedback - при каждом вызове алгоритма обрабатывается J битов входного значения. Предшествующий зашифрованный блок используется в качестве входа в алгоритм; к J битам выхода алгоритма и следующему незашифрованному блоку из J битов применяется операция XOR, результатом которой является следующий

зашифрованный блок из J битов. Типичные приложения - потокоориентированная передача, аутентификация.

OFB - Output Feedback - аналогичен *CFB*, за исключением того, что на вход алгоритма при шифровании следующего блока подается результат шифрования предыдущего блока; только после этого выполняется операция XOR с очередными J битами незашифрованного текста. Типичные приложения - потокоориентированная передача по зашумленному каналу (например, спутниковая связь).

Режим ECB

Данный режим является самым простым режимом, при котором незашифрованный текст обрабатывается последовательно, блок за блоком. Каждый блок шифруется, используя один и тот же ключ. Если сообщение длиннее, чем длина блока соответствующего алгоритма, то оно разбивается на блоки соответствующей длины, причем последний блок дополняется в случае необходимости фиксированными значениями. При использовании данного режима одинаковые незашифрованные блоки будут преобразованы в одинаковые зашифрованные блоки.

ECB-режим идеален для небольшого количества данных, например, для шифрования ключа сессии.

Существенным недостатком *ECB* является то, что один и тот же блок незашифрованного текста, появляющийся более одного раза в сообщении, всегда имеет один и тот же зашифрованный вид. Вследствие этого для больших сообщений *ECB* режим считается небезопасным. Если сообщение имеет много одинаковых блоков, то при криптоанализе данная закономерность будет обнаружена.

Режим CBC

Для преодоления недостатков *ECB* используют способ, при котором одинаковые незашифрованные блоки преобразуются в различные зашифрованные. Для этого в качестве входа алгоритма используется результат применения операции XOR к текущему незашифрованному блоку и предыдущему зашифрованному блоку.

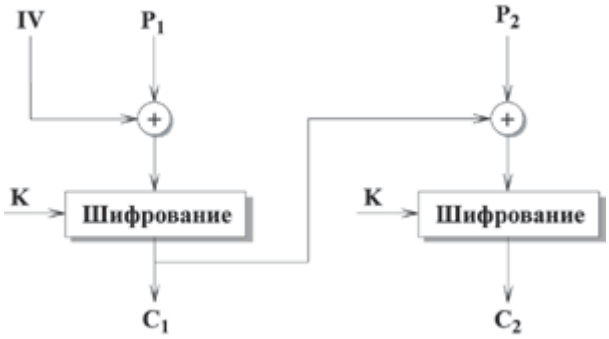

Рис. 3.8. Шифрование в режиме CBC

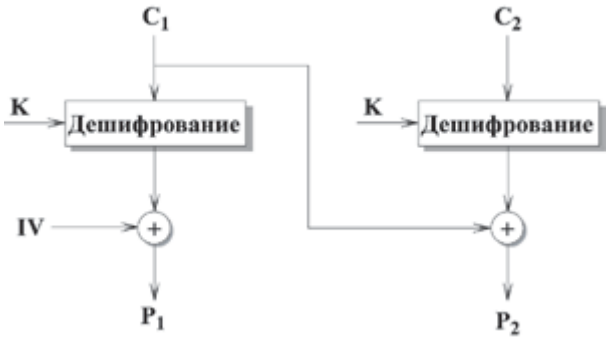

Рис. 3.9. Дешифрование в режиме CBC

Для получения первого блока зашифрованного сообщения используется инициализационный вектор (IV), для которого выполняется операция XOR с первым блоком незашифрованного сообщения. При дешифровании для IV выполняется операция XOR с выходом дешифрирующего алгоритма для получения первого блока незашифрованного текста.

IV должен быть известен как отправителю, так и получателю. Для максимальной безопасности IV должен быть защищен так же, как ключ.

Режим CFB

Блочный алгоритм предназначен для шифрования блоков определенной длины. Однако можно преобразовать блочный алгоритм в поточный алгоритм шифрования, используя последние два режима. Поточный алгоритм шифрования устраняет необходимость разбивать сообщение на целое число блоков достаточно большой длины, следовательно, он может работать в реальном времени. Таким образом, если передается поток символов, каждый символ может шифроваться и передаваться сразу, с использованием символьно ориентированного режима блочного алгоритма шифрования.

Одним из преимуществ такого режима блочного алгоритма шифрования является то, что зашифрованный текст будет той же длины, что и исходный.

Будем считать, что блок данных, используемый для передачи, состоит из J бит; обычным значением является J=8. Как и в режиме *CBC*, здесь используется операция XOR для предыдущего блока зашифрованного текста и следующего блока незашифрованного текста. Таким образом, любой блок зашифрованного текста является функцией от всего предыдущего незашифрованного текста.

Рассмотрим шифрование. Входом функции шифрования является регистр сдвига, который первоначально устанавливается в инициализационный вектор IV. Для левых J битов выхода алгоритма выполняется операция XOR с первыми J битами незашифрованного текста P1 для получения первого блока зашифрованного текста C_1. Кроме того, содержимое регистра сдвигается влево на J битов, и C_1 помещается в правые J битов этого регистра. Этот процесс продолжается до тех пор, пока не будет зашифровано все сообщение.

При дешифровании используется аналогичная схема, за исключением того, что для блока получаемого зашифрованного текста выполняется операция XOR с выходом алгоритма для получения незашифрованного блока.

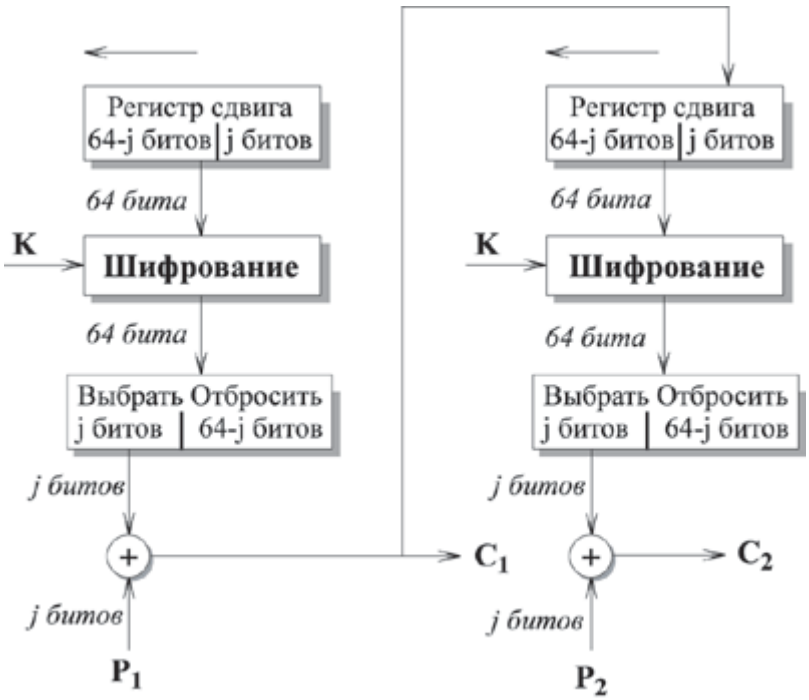

Рис. 3.10. Шифрование в режиме CFB

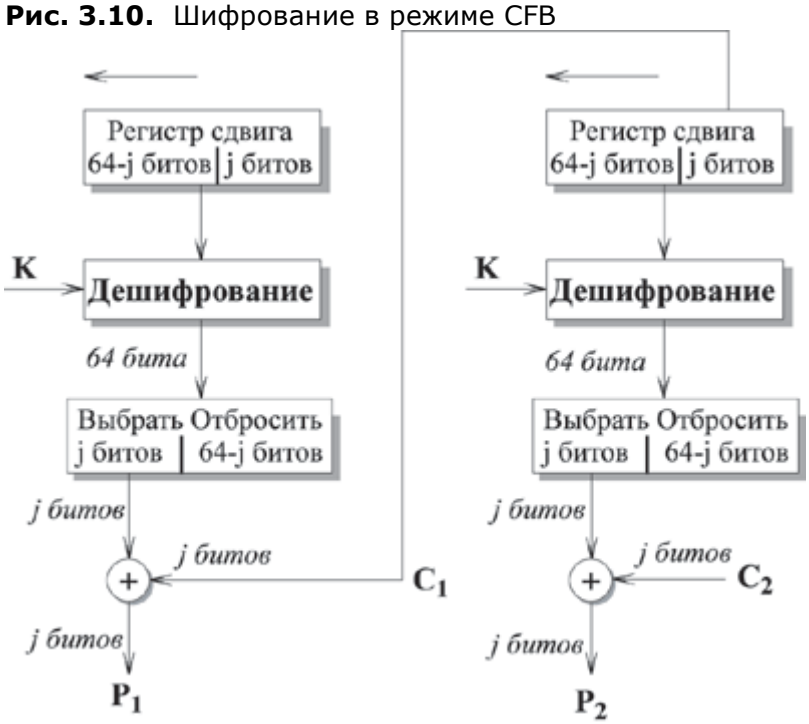

Рис. 3.11. Дешифрование в режиме CFB

Режим OFB

Данный режим подобен режиму *CFB*. Разница заключается в том, что выход алгоритма в режиме *OFB* подается обратно в регистр, тогда как в режиме *CFB* в регистр подается результат применения операции XOR к незашифрованному блоку и результату алгоритма.

Основное преимущество режима *OFB* состоит в том, что если при передаче произошла ошибка, то она не распространяется на следующие зашифрованные блоки, и тем самым сохраняется возможность дешифрования последующих блоков. Например, если появляется ошибочный бит в C_i, то это приведет только к невозможности

дешифрования этого блока и получения P_i. Дальнейшая последовательность блоков будет расшифрована корректно. При использовании режима *CFB* C_i подается в качестве входа в регистр и, следовательно, является причиной последующего искажения потока.

Недостаток *OFB* в том, что он более уязвим к атакам модификации потока сообщений, чем *CFB*.

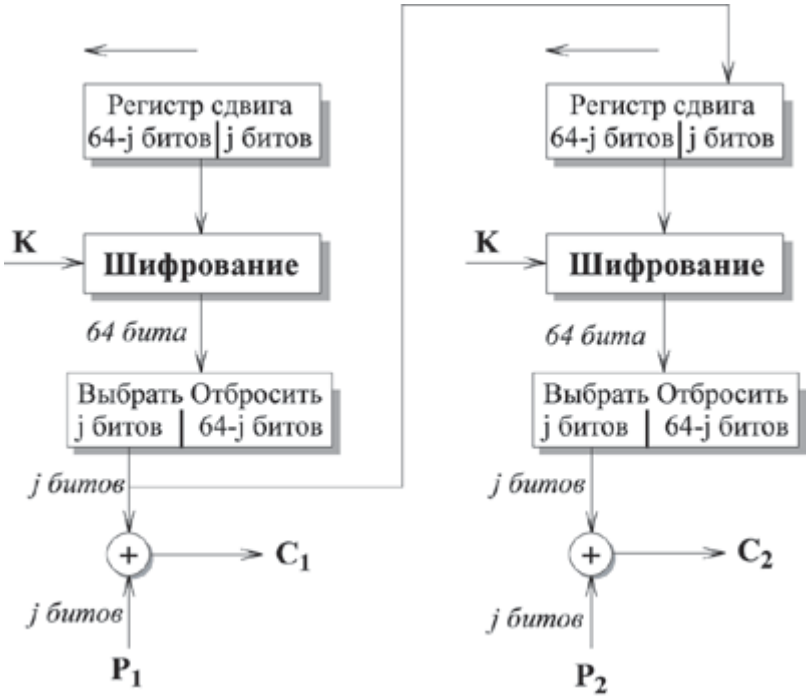

Рис. 3.12. Шифрование в режиме OFB

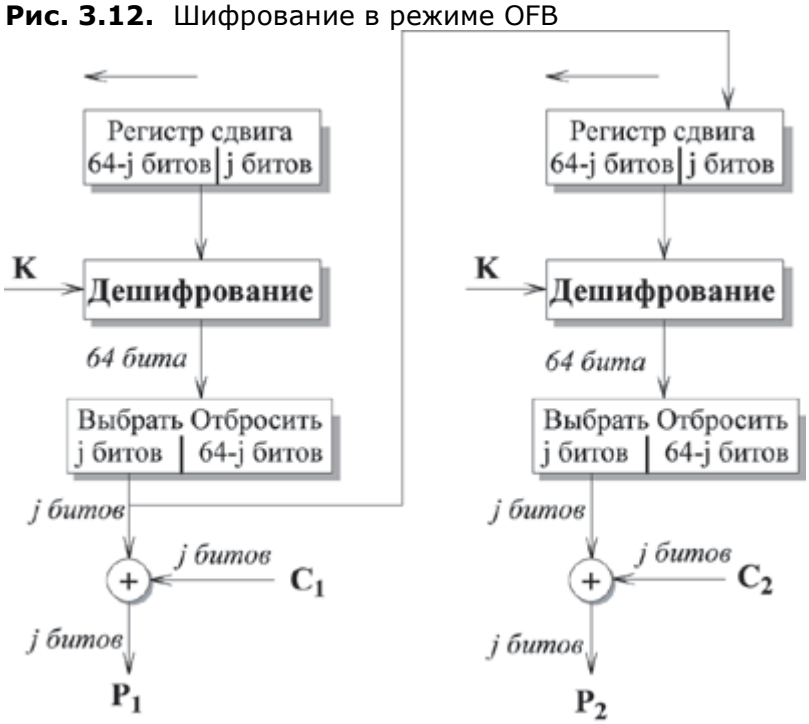

Рис. 3.13. Дешифрование в режиме OFB

Создание случайных чисел

Случайные числа играют важную роль при использовании криптографии в различных сетевых приложениях, относящихся к безопасности. Сделаем краткий обзор требований, предъявляемых к случайным числам в приложениях сетевой безопасности, а затем рассмотрим несколько способов создания случайных чисел.

Требования к случайным числам

Большинство алгоритмов сетевой безопасности, основанных на криптографии, используют случайные числа. Например:

1. Схемы взаимной аутентификации. В большинстве сценариев аутентификации и распределения ключа используются nonces для предотвращения атак повтора (replay-атак). Применение действительно случайных чисел в качестве nonces не дает противнику возможности вычислить или угадать nonce.
2. Ключ сессии, созданный KDC или кем-либо из участников.

Двумя основными требованиями к последовательности случайных чисел являются случайность и непредсказуемость.

Случайность

Обычно при создании последовательности *псевдослучайных чисел* предполагается, что данная последовательность чисел должна быть случайной в некотором определенном статистическом смысле. Следующие два критерия используются для доказательства того, что последовательность чисел является случайной:

1. Однородное распределение: распределение чисел в последовательности должно быть однородным; это означает, что частота появления каждого числа должна быть приблизительно одинаковой.
2. Независимость: ни одно значение в последовательности не должно зависеть от других.

Хотя существуют тесты, показывающие, что последовательность чисел соответствует некоторому распределению, такому как однородное распределение, теста для "доказательства" независимости нет. Тем не менее, можно подобрать набор тестов для доказательства того, что последовательность является зависимой. Общая стратегия предполагает применение набора таких тестов до тех пор, пока не будет уверенности, что независимость существует.

Непредсказуемость

В приложениях, таких как взаимная аутентификация и генерация ключа сессии, нет жесткого требования, чтобы последовательность чисел была статистически случайной, но члены последовательности должны быть непредсказуемы. При "правильной" случайной последовательности каждое число статистически не зависит от остальных чисел и, следовательно, непредсказуемо. Однако правильные случайные числа на практике используются достаточно редко, чаще последовательность чисел, которая должна быть случайной, создается некоторым алгоритмом. В данном случае необходимо, чтобы противник не мог предугадать следующие элементы последовательности, основываясь на знании предыдущих элементов и используемого алгоритма.

Источники случайных чисел

Источники действительно случайных чисел найти трудно. Физические генераторы шумов, такие как детекторы событий ионизирующей радиации, газовые разрядные

трубки и имеющий течь конденсатор могут быть такими источниками. Однако эти устройства в приложениях сетевой безопасности применяются ограниченно. Проблемы также вызывают грубые атаки на такие устройства. Альтернативным решением является создание набора из большого числа случайных чисел и опубликование его в некоторой книге. Тем не менее, и такие наборы обеспечивают очень ограниченный источник чисел по сравнению с тем количеством, которое требуется приложениям сетевой безопасности. Более того, хотя наборы из этих книг действительно обеспечивает статистическую случайность, они предсказуемы, так как противник может получить их копию.

Таким образом, шифрующие приложения используют для создания случайных чисел специальные алгоритмы. Эти алгоритмы детерминированы и, следовательно, создают последовательность чисел, которая не является статистически случайной. Тем не менее, если алгоритм хороший, полученная последовательность будет проходить много тестов на случайность. Такие числа часто называют ***псевдослучайными числами***.

Рассмотрим несколько алгоритмов генерации случайных чисел.

Генераторы псевдослучайных чисел

Первой широко используемой технологией создания случайного числа был алгоритм, предложенный Лехмером, который известен как метод линейного конгруента. Этот алгоритм параметризуется четырьмя числами следующим образом:

m	Модуль (основание системы)	m > 0
a	Множитель	$0 \leq a < m$
c	Приращение	$0 \leq c < m$
X0	Начальное значение или зерно (seed)	$0 \leq X_0 < m$

Последовательность случайных чисел $\{X_n\}$ получается с помощью следующего итерационного равенства:

$$X_{n+1} = (a\ X_n + c) \bmod m$$

Если m, a и c являются целыми, то создается последовательность целых чисел в диапазоне $0 \leq X_n < m$.

Выбор значений для a, c и m является критичным для разработки хорошего генератора случайных чисел.

Очевидно, что m должно быть очень большим, чтобы была возможность создать много случайных чисел. Считается, что m должно быть приблизительно равно максимальному положительному целому числу для данного компьютера. Таким образом, обычно m близко или равно 2^{31}.

Существует три критерия, используемые при выборе генератора случайных чисел:

1. Функция должна создавать полный период, т.е. все числа между 0 и m до того, как создаваемые числа начнут повторяться.
2. Создаваемая последовательность должна появляться случайно. Последовательность не является случайной, так как она создается детерминированно, но различные статистические тесты, которые могут применяться, должны показывать, что последовательность случайна.
3. Функция должна эффективно реализовываться на 32-битных процессорах.

Значения a, c и m должны быть выбраны таким образом, чтобы эти три критерия выполнялись. В соответствии с первым критерием можно показать, что если m является простым и $c = 0$, то при определенном значении a период, создаваемый функцией, будет равен $m-1$. Для 32-битной арифметики соответствующее простое значение $m = 2^{31} - 1$. Таким образом, функция создания *псевдослучайных чисел* имеет вид:

$$X_{n+1} = (a\ X_n)\ \mathrm{mod}\ (2^{31} - 1)$$

Только небольшое число значений a удовлетворяет всем трем критериям. Одно из таких значений есть $a = 7^5 = 16807$, которое использовалось в семействе компьютеров IBM 360. Этот генератор широко применяется и прошел более тысячи тестов, больше, чем все другие генераторы *псевдослучайных чисел*.

Сила алгоритма линейного конгруента в том, что если сомножитель и модуль (основание) соответствующим образом подобраны, то результирующая последовательность чисел будет статистически неотличима от последовательности, являющейся случайной из набора $1, 2, \ldots, m-1$. Но не может быть случайности в последовательности, полученной с использованием алгоритма, независимо от выбора начального значения X_0. Если значение выбрано, то оставшиеся числа в последовательности будут предопределены. Это всегда учитывается при криптоанализе.

Если противник знает, что используется алгоритм линейного конгруента, и если известны его параметры ($a = 7^5,\ c = 0,\ m = 2^{31} - 1$), то, если раскрыто одно число, вся последовательность чисел становится известна. Даже если противник знает только, что используется алгоритм линейного конгруента, знания небольшой части последовательности достаточно для определения параметров алгоритма и всех последующих чисел. Предположим, что противник может определить значения X_0, X_1, X_2, X_3. Тогда :

$$X_1 = (a\ X_0 + c\ )\ \mathrm{mod}\ m$$
$$X_2 = (a\ X_1 + c\ )\ \mathrm{mod}\ m$$
$$X_3 = (a\ X_2 + c\ )\ \mathrm{mod}\ m$$

Эти равенства позволяют найти a, c и m.

Таким образом, хотя алгоритм и является хорошим генератором *псевдослучайной последовательности чисел*, желательно, чтобы реально используемая последовательность была непредсказуемой, поскольку в этом случае знание части последовательности не позволит определить будущие ее элементы. Эта цель может быть достигнута несколькими способами. Например, использование внутренних системных часов для модификации потока случайных чисел. Один из способов применения часов состоит в перезапуске последовательности после N чисел, используя текущее значение часов по модулю m в качестве нового начального значения. Другой способ состоит в простом добавлении значения текущего времени к каждому случайному числу по модулю m.

Криптографически созданные случайные числа

В криптографических приложениях целесообразно шифровать получающиеся случайные числа. Чаще всего используется три способа.

Циклическое шифрование

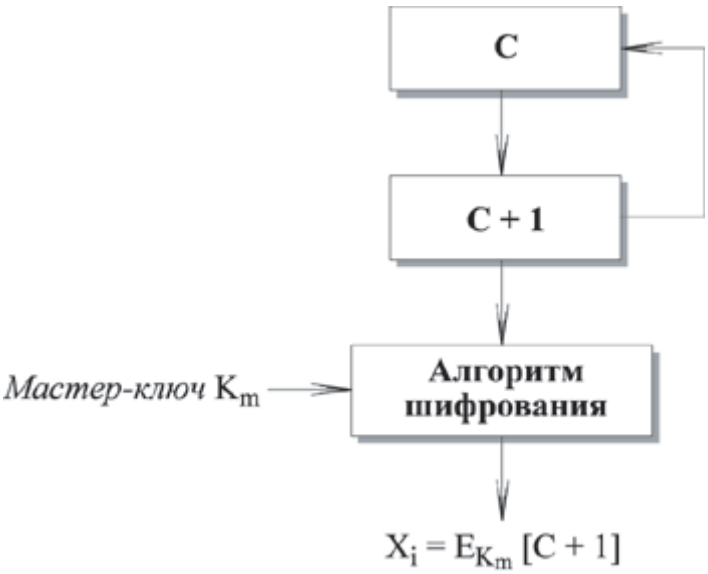

Рис. 3.14. Циклическое шифрование

В данном случае применяется способ создания ключа сессии из мастер-ключа. Счетчик с периодом N используется в качестве входа в шифрующее устройство. Например, в случае использования 56-битного ключа DES может применяться счетчик с периодом 2^{56}. После каждого созданного ключа значение счетчика увеличивается на 1. Таким образом, псевдослучайная последовательность, полученная по данной схеме, имеет полный период: каждое выходное значение $X_0, X_1, \ldots X_{N-1}$ основано на различных значениях счетчика и, следовательно, $X_0 \neq X_1 \neq X_{N-1}$. Так как мастер-ключ защищен, легко показать, что любой секретный ключ не зависит от знания одного или более предыдущих секретных ключей.

Для дальнейшего усиления алгоритма вход должен быть выходом полнопериодического генератора *псевдослучайных чисел*, а не простой последовательностью.

Режим Output Feedback DES

Режим *OFB* DES может применяться для генерации ключа, аналогично тому, как он используется для потокового шифрования. Заметим, что выходом каждой стадии шифрования является 64-битное значение, из которого только левые j битов подаются обратно для шифрования. 64-битные выходы составляют последовательность *псевдослучайных чисел* с хорошими статистическими свойствами.

Генератор псевдослучайных чисел ANSI X9.17

Один из наиболее сильных генераторов *псевдослучайных чисел* описан в ANSI X9.17. В число приложений, использующих эту технологию, входят приложения финансовой безопасности и PGP.

Алгоритмом шифрования является тройной DES. Генератор ANSI X9.17 состоит из следующих частей:

1. **Вход:** генератором управляют два псевдослучайных входа. Один является 64-битным представлением текущих даты и времени, которые изменяются каждый раз при создании числа. Другой является 64-битным начальным значением; оно инициализируется некоторым произвольным значением и изменяется в ходе генерации последовательности *псевдослучайных чисел*.

2. **Ключи:** генератор использует три модуля тройного DES. Все три используют одну и ту же пару 56-битных ключей, которая должна держаться в секрете и применяться только для генерации *псевдослучайного числа*.
3. **Выход:** выход состоит из 64-битного *псевдослучайного числа* и 64-битного значения, которое будет использоваться в качестве начального значения при создании следующего числа.

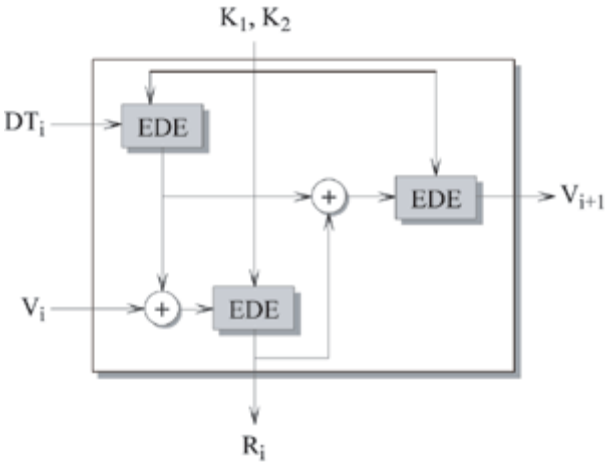

Рис. 3.15. Генератор псевдослучайных чисел ANSI X9.17

DT_i - значение даты и времени на начало i-ой стадии генерации.

V_i - начальное значение для i-ой стадии генерации.

R_i - *псевдослучайное число*, созданное на i-ой стадии генерации.

K_1, K_2 - ключи, используемые на каждой стадии.

Тогда:

$$R_i = EDE_{K1,K2} [EDE_{K1,K2} [DT_i] \oplus V_i]$$
$$V_{i+1} = EDE_{K1,K2} [EDE_{K1,K2} [DT_i] R_i]$$

Схема включает использование 112-битного ключа и трех EDE-шифрований. На вход подаются два *псевдослучайных значения*: значение даты и времени и начальное значение очередной итерации, на выходе создаются начальное значение для следующей итерации и очередное *псевдослучайное значение*. Даже если *псевдослучайное число* R_i будет скомпрометировано, вычислить V_{i+1} из R_i невозможно, и, следовательно, следующее *псевдослучайное значение* R_{i+1}, так как для получения V_{i+1} дополнительно выполняются три операции EDE.

4. Лекция: Алгоритмы симметричного шифрования. Часть 3. Разработка Advanced Encryption Standard (AES)

Разработка Advanced Encryption Standard (AES)

Обзор процесса разработки AES

Инициатива в разработке *AES* принадлежит NIST. Основная цель состояла в создании федерального стандарта (FIPS), который бы описывал алгоритм шифрования, используемый для защиты информации как в государственном, так и в частном секторе.

Конкуренция среди финалистов была достаточно серьезной, и в результате длительного процесса оценки NIST выбрал *Rijndael* в качестве алгоритма *AES*. Мы кратко рассмотрим этот процесс и суммируем различные характеристики алгоритмов,

которые были описаны на данном этапе. Следующий раздел представляет собой обзор разработки *AES* и обсуждение деталей алгоритмов.

Историческая справка

2 января 1997 года NIST объявил о начале разработки *AES*, и 12 сентября 1997 года были представлены официальные требования к алгоритмам. В этих требованиях указывалось, что целью NIST является разработка неклассифицированного, хорошо проанализированного алгоритма шифрования, доступного для широкого применения. Как минимум алгоритм должен быть симметричным и блочным и поддерживать *длину блока* 128 бит и *длину ключа* 128, 192 и 256 бит.

20 августа 1998 года NIST анонсировал пятнадцать кандидатов на алгоритм *AES* на первой конференции кандидатов *AES* (AES1), и было предложено прокомментировать их характеристики. Данные алгоритмы были разработаны промышленными и академическими кругами двенадцати стран. Вторая конференция кандидатов *AES* (AES2) была проведена в марте 1999 года с целью обсуждения результатов анализа предложенных алгоритмов. В августе 1999 года были представлены выбранные NIST пять финалистов. Ими стали *MARS*, *RC6*™, *Rijndael*, *Serpent* и *Twofish*.

Обзор финалистов

Все пять финалистов являются итерационными блочными алгоритмами шифрования: они определяют преобразование, которое повторяется определенное число раз над блоком шифруемых или дешифруемых данных. Шифруемый блок данных называется plaintext; зашифрованный plaintext называется ciphertext. Для дешифрования в качестве обрабатываемого блока данных используется ciphertext. Каждый финалист также определяет метод создания серии ключей из исходного ключа, называемый управлением ключом. Полученные ключи называются подключами. Функции раунда используют в качестве входа различные подключи для конкретного блока данных.

У каждого финалиста первая и последняя криптографические операции являются некоторой формой перемешивания подключей и блока данных. Такие операции, используемые на начальном шаге первого раунда и заключительном шаге последнего раунда, называются pre- и post-забеливанием (whitening) и могут быть определены отдельно.

Существуют также некоторые другие технические особенности финалистов. Четыре финалиста определяют таблицы подстановки, называемые S-box: AxB-битный S-box заменяет A входных битов на B выходных битов. Три финалиста определяют функции раунда, являющиеся сетью Фейштеля. В классической сети Фейштеля одна половина блока данных используется для модификации другой половины блока данных, затем половины меняются местами. Два финалиста не используют сеть Фейштеля, в каждом раунде обрабатывают параллельно весь блок данных, применяя подстановки и линейные преобразования; таким образом, эти два финалиста являются примерами алгоритмов, использующих линейно-подстановочное преобразование.

Далее рассмотрим каждый из алгоритмов в алфавитном порядке; профили и оценки будут представлены в следующих разделах.

MARS выполняет последовательность преобразований в следующем порядке: сложение с ключом в качестве pre-whitening, 8 раундов прямого перемешивания без использования ключа, 8 раундов прямого преобразования с использованием ключа, 8 раундов обратного преобразования с использованием ключа, 8 раундов обратного перемешивания без использования ключа и вычитание ключа в качестве post-whitening. 16 раундов с использованием ключа называются криптографическим ядром. Раунды без ключа используют два 8x16- битных S-boxes и операции сложения и XOR. В

дополнение к этим элементам раунды с ключом используют 32-битное умножение ключа, зависимые от данных циклические сдвиги и добавление ключа. Как раунды перемешивания, так и раунды ядра являются раундами модифицированной сети Фейштеля, в которых четверть блока данных используется для изменения остальных трех четвертей блока данных. *MARS* предложен корпорацией IBM.

RC6 является параметризуемым семейством алгоритмов шифрования, основанных на сети Фейштеля; для *AES* было предложено использовать 20 раундов. Функция раунда в *RC6* задействует переменные циклические сдвиги, которые определяются квадратичной функцией от данных. Каждый раунд также включает умножение по модулю 32, сложение, XOR и сложение с ключом. Сложение с ключом также используется для pre- и pos-whitening. *RC6* был предложен лабораторией RSA.

Rijndael представляет собой алгоритм, использующий линейно-подстановочные преобразования и состоящий из 10, 12 или 14 раундов в зависимости от *длины ключа*. Блок данных, обрабатываемый с использованием *Rijndael*, делится на массивы байтов, и каждая операция шифрования является байт-ориентированной. Функция раунда *Rijndael* состоит из четырех слоев. В первом слое для каждого байта применяется S-box размерностью 8x8 бит. Второй и третий слои являются линейными перемешиваниями, в которых строки рассматриваются в качестве сдвигаемых массивов и столбцы перемешиваются. В четвертом слое выполняется операция XOR байтов подключа и каждого байта массива. В последнем раунде перемешивание столбцов опущено. *Rijndael* предложен Joan Daemen (Proton World International) и Vincent Rijmen (Katholieke Universiteit Leuven).

Serpent является алгоритмом, использующим линейно-подстановочные преобразования и состоящим из 32 раундов. Serpent также определяет некриптографические начальную и заключительную перестановки, которые облегчают альтернативный режим реализации, называемый bitslice. Функция раунда состоит из трех слоев: операция XOR с ключом, 32-х параллельное применение одного из восьми фиксированных S-boxes и линейное преобразование. В последнем раунде слой XOR с ключом заменен на линейное преобразование. *Serpent* предложен Ross Anderson (University of Cambridge), Eli Biham (Technion) и LarsKnudsen (University of California San Diego).

Twofish является сетью Фейштеля с 16 раундами. Сеть Фейштеля незначительно модифицирована с использованием однобитных ротаций. Функция раунда влияет на 32-битные слова, используя четыре зависящих от ключа S-boxes, за которыми следуют фиксированные максимально удаленные отдельные матрицы в GF(2^8), преобразование псевдо-Адамара и добавление ключа. *Twofish* был предложен Bruce Schneier, John Kelsey и Niels Ferguson (Counterpane Internet Security, Inc.), Doug Whiting (Hi/fn, Inc.), David Wagner (University of California Berkley) и Chris Hall (Princeton University).

При объявлении финалистов представители NIST предложили обсудить и прокомментировать алгоритмы. На третьей конференции кандидатов *AES* (AES3), состоявшейся в апреле 2000 года, представленные комментарии были рассмотрены. Период открытого обсуждения был завершен 15 мая 2000 года.

Критерий оценки

В сентябре 1997 года, объявив об алгоритмах кандидатов, специалисты NIST определили общий критерий, который должен использоваться при сравнении алгоритмов.

Критерий оценки был разделен на три основных категории:

1. Безопасность.
2. Стоимость.

3. Характеристики алгоритма и его реализации.

Безопасность является важнейшим фактором при оценке и сравнении таких возможностей как стойкость алгоритма к криптоанализу, исследование его математической основы, случайность выходных значений алгоритма и относительная безопасность по сравнению с другими кандидатами.

Стоимость является второй важной областью оценки, которая характеризует лицензионные требования, вычислительную эффективность (скорость) на различных платформах и требования к памяти. Так как одной из целей NIST была возможность широкой доступности алгоритма *AES* без лицензионных ограничений, обсуждались в основном требования защиты интеллектуальной собственности и потенциальные конфликты. Рассматривалась также скорость работы алгоритма на различных платформах. При первом обсуждении основное внимание уделялось скорости, связанной со 128-битными ключами. При втором обсуждении рассматривались аппаратные реализации и скорости, связанные со 192- и 256-битными ключами. Также важно рассматривать требования памяти и ограничения программной реализации.

Третьей областью оценки являлись характеристики алгоритма и реализации, такие как гибкость, аппаратное и программное соответствие и простота алгоритма. Гибкость включает возможность алгоритма:

- управлять размером ключа и блока сверх того, который минимально должен поддерживаться;
- безопасно и эффективно реализовываться в различных типах окружений;
- реализовываться в качестве поточного алгоритма шифрования, хэш-функции и обеспечивать дополнительные криптографические сервисы.

Должна быть возможность реализовать алгоритм как аппаратно, так и программно, эффективность смешанных (firmware) реализаций также считается преимуществом. Относительная простота разработки алгоритма также является фактором оценки.

На первом и втором этапах обсуждения стало очевидно, что различные выводы, полученные при анализе, часто переходят из одного рассмотренного выше критерия в другой. Таким образом, критерии стоимости и характеристик алгоритма рассматривались вместе в качестве второго критерия после безопасности.

Результаты второго этапа обсуждения

Считается, что второй этап обсуждения начинается с официального объявления пяти финалистов *AES* 20 августа 1999 года и заканчивается официальным завершением обсуждений 15 мая 2000 года.

NIST выполнил анализ оптимизированных реализаций алгоритмов на ANSI C и Java, которые были предоставлены после первого этапа обсуждений. При тестировании реализаций на ANSI C основное внимание уделялось скорости выполнения на компьютерах, использующих различные комбинации процессоров, операционных систем и компиляторов. Код Java был протестирован на скорость и используемую память на различных системах. Дополнительно было выполнено статистическое тестирование.

Процесс выбора

Была проведена серия встреч, чтобы выработать стратегию выбора алгоритма *AES*. В результате этих встреч все пришли к выводу, что выбранный алгоритм обеспечивает достаточную безопасность на обозримое будущее, эффективен на различных

платформах и в различных окружениях, обеспечивает приемлемую гибкость с учетом возможных требований в будущем.

Методология и результаты выбора

Принципы выбора алгоритма

Команда NIST до выбора алгоритма должна была принять несколько фундаментальных решений:

- Принять количественный или качественный критерий при выборе алгоритма.
- Выбрать один или несколько алгоритмов в качестве *AES*.
- Выбрать запасной алгоритм(ы).
- Рассмотреть предложения по модификации алгоритмов.

Кратко рассмотрим полученные результаты.

Качественный или количественный критерий

На одной из первых встреч по планированию второго этапа обсуждений была рассмотрена возможность количественного подхода, используя который каждый алгоритм или комбинация алгоритмов будет получать определенное количество очков, основываясь на критерии оценки. Как при осуществлении такого подхода определить конкретные значения и обеспечить сравнение алгоритмов? Количественный подход также обеспечивает явные веса для каждого выбранного фактора *AES*. Тем не менее, было достигнуто соглашение, что степень субъективности в критерии приведет к большому количеству дискуссий. Было также высказано предположение, что определение количественной системы счетчиков невозможно без широкого обсуждения того, что такая система является объективной. Поэтому был сделан вывод, что количественный подход неприменим, и было решено рассмотреть его после первого этапа обсуждений. То есть было решено рассмотреть безопасность алгоритма, выполнение, реализацию и остальные характеристики и принять решение, основываясь на качественной оценке каждого алгоритма - помня, что обсуждение безопасности является самым главным.

Количество алгоритмов AES

В течение первого и второго этапов обсуждений было высказано несколько аргументов относительно количества алгоритмов, которые должны быть выбраны для включения в *AES*. Дополнительно был сделан вывод относительно "запасного" алгоритма для случая, если будет выбран единственный AES-алгоритм, и последующие изменения окажутся невозможными. Это может произойти в случае, например, фактической атаки на алгоритм или простого обсуждения свойств. Было решено, что это необходимо считать частью параметров, которые в дальнейшем будут рассматриваться как часть процесса выбора.

Некоторые аргументы, выдвинутые в защиту нескольких алгоритмов (против единственного алгоритма), включают:

- В интересах безопасности; на случай, если один алгоритм *AES* будет взломан, в продуктах должно быть реализовано более одного AES-алгоритма. Некоторые считают, что широкое применение единственного алгоритма является рискованным, если данный алгоритм проявит себя как небезопасный.
- Понятия интеллектуальной собственности могут быть рассмотрены позднее в вопросе отсутствия лицензионных ограничений для конкретного алгоритма. Альтернативный алгоритм может предоставлять промежуточный вариант, который не влияет на рассмотренное понятие интеллектуальной собственности.

- Множество алгоритмов *AES* может охватывать более широкий диапазон характеристик, чем единственный алгоритм. В частности, можно будет предложить как большую степень безопасности, так и большую степень эффективности, что не всегда возможно при использовании единственного алгоритма.

Были также высказаны мнения в пользу единственного алгоритма (и/или против нескольких алгоритмов). Некоторые из этих аргументов таковы:

- Использование нескольких AES-алгоритмов уменьшает интероперабельность и увеличивает стоимость продукта.
- Несколько алгоритмов могут представлять собой определенное число "атак на интеллектуальную собственность", направленных против реализаций.
- Описание нескольких алгоритмов может вызвать обсуждение вопросов безопасности любого из алгоритмов.
- Аппаратные реализации могут лучше использовать доступные ресурсы, оптимизируя выполнение единственного алгоритма, а не нескольких алгоритмов.

Во время второго этапа обсуждались эти и другие проблемы, касающиеся единственного или нескольких алгоритмов *AES*. Как выяснилось, существует вероятность, доказываемая коммерческими продуктами сегодня, что скоро продукты будут реализовывать несколько алгоритмов, которые определяются нуждами потребителей, требованиями интероперабельности с наследуемыми или собственными системами и так далее. Тройной DES, который NIST предлагает сделать в обозримом будущем соответствующим FIPS-алгоритмом, доступен во многих коммерческих продуктах, как и другие FIPS и не-FIPS алгоритмы. Следовательно, считается, что наличие этих нескольких алгоритмов в конкретном продукте обеспечивает большую степень надежности, как и наличие нескольких *длин ключей* в *AES*. В случае атаки на выбранный алгоритм NIST предлагает задействовать соответствующие исследованные опции, которые включают использование других финалистов *AES*, у которых отсутствуют подобные атаки, либо в случае необходимости определить полностью новые подходы.

В соответствии с выводами об интеллектуальной собственности отмечалось, что если будет выбрано несколько алгоритмов *AES*, то возникнет необходимость реализовывать все AES-алгоритмы, таким образом создавая дополнительные риски в отношении интеллектуальной собственности.

На конференции AES3 состоялась дискуссия по поводу количества алгоритмов, которые должны быть включены в *AES*. Большинство присутствующих выразили поддержку выбора единственного алгоритма. Некоторые поддержали и выбор запасного алгоритма, но не было согласовано, как это должно быть выполнено. Указанное выше мнение было отражено в письменных комментариях, представленных в NIST многими из присутствующих на конференции.

Перед тем как принять решение в пользу единственного алгоритма *AES*, были рассмотрены все комментарии и аргументы. Остальные соответствующие FIPS алгоритмы будут обеспечивать определенную степень надежности, единственный же AES-алгоритм обеспечивает интероперабельность, способствует решению проблем интеллектуальной собственности.

Запасной алгоритм

Как уже отмечалось, существует взаимосвязь между обсуждениями проблемы нескольких алгоритмов *AES* и выбором запасного алгоритма, особенно в случае единственного алгоритма *AES*. Backup может иметь несколько форм, от алгоритма, который требуется реализовывать в продуктах *AES* ("cold backup"), до определяемого в *AES* запасного алгоритма ("hot backup"). Было доказано, что запасной алгоритм во

многом эквивалентен второму AES-алгоритму, так как многие пользователи пожелают, чтобы даже "cold backup" был реализован в продуктах.

Итак, имея

- представления о том, что запасной алгоритм должен de facto требоваться в продуктах;
- сомнения относительно потенциальной применимости в связи с возможными достижениями криптоанализа;
- заинтересованность NIST в обеспечении интероперабельности;
- доступность в коммерческих продуктах других алгоритмов (как FIPS, так и не-FIPS);

было принято решение не выбирать запасной алгоритм.

Как и в случае с другими стандартами на криптографические алгоритмы, NIST продолжит исследования в области криптоанализа AES-алгоритма, и стандарт будет пересматриваться каждые пять лет. Если полученные результаты потребуют более быстрой реакции, NIST будет действовать соответствующим образом и рассмотрит все возможные альтернативы.

Модификация алгоритмов

На первом и втором этапах обсуждения был отмечен интерес к увеличению *числа раундов* в некоторых алгоритмах. Во многих случаях увеличение количества раундов объяснялось. Так, про *Rijndael* было сказано, что *число раундов Rijndael* обеспечивает достаточный запас безопасности по отношению к атакам криптоанализа.

При обсуждении были отмечены следующие результаты и введены понятия:

- Для некоторых алгоритмов неясно, каково полное определение алгоритма (например, управление ключом) при различном количестве раундов и как такое изменение влияет на анализ безопасности.
- Изменение количества раундов влияет на анализ выполнения. Все полученные данные для модифицированного алгоритма необходимо либо оценить, либо получить заново. В некоторых случаях, особенно при аппаратной реализации и в окружениях с ограниченной памятью, оценка алгоритма с новым количеством раундов должна производиться особенно тщательно.
- Должно быть достаточное количество аргументов для добавления *числа раундов* и соответствующего изменения алгоритма.

После многочисленных обсуждений приведенных выше аргументов было решено рекомендовать не изменять количество раундов AES-алгоритма.

Технические детали второго этапа обсуждений

Общая безопасность

Представленный здесь анализ был выполнен с использованием исходных спецификаций финалистов, полученных до начала второго этапа.

Безопасность являлась самым важным фактором при оценке финалистов. В отношении какого-либо из алгоритмов никаких атак не зафиксировано.

Были зафиксированы только атаки против простейших вариантов алгоритмов, когда *число раундов* было уменьшено или были сделаны упрощения другими способами.

Ниже дается краткое описание этих атак против вариантов с уменьшенным *числом раундов*, а также перечислены необходимые вычислительные ресурсы и ресурсы памяти.

Трудно оценить важность атак на варианты с уменьшенным *числом раундов*. С одной стороны, варианты с уменьшенным *числом раундов* на самом деле являются другими алгоритмами, и таким образом атаки на них никак не характеризуют безопасность исходных алгоритмов. Алгоритм может быть безопасен при n раундах, даже если он уязвим при n-1 раунде. С другой стороны, обычной практикой в современном криптоанализе являются попытки сконструировать атаки на варианты с уменьшенным *числом раундов*. С этой точки зрения вполне понятны попытки оценить "*резерв безопасности*" рассматриваемых кандидатов, основываясь на атаках на варианты с уменьшенным *числом раундов*.

Одним из возможных критериев ***резерва безопасности*** является число, на которое полное *число раундов* алгоритма превышает наибольшее *число раундов*, при котором возможна атака. Существует ряд причин, объясняющих, почему не следует полагаться исключительно на подобную метрику для определения силы алгоритма; тем не менее, данная метрика *резерва безопасности* может быть полезна.

NIST рассмотрел и другие характеристики финалистов, которые могут повлиять на их безопасность. Уверенность в анализе безопасности, выполненном при разработки *AES*, зависит от происхождения алгоритмов и принципов их разработки, а также от трудности анализа конкретных комбинаций операций, используемых в каждом алгоритме.

Атаки на варианты с уменьшенным числом раундов

Ниже в таблице приведены атаки на варианты с уменьшенным *числом раундов*. Для каждой атаки в таблице указано *число раундов*, при котором может осуществляться атака, *длина ключа*, тип атаки и необходимые ресурсы. Для атаки может требоваться три категории ресурсов: вычислительные, память, информация.

В столбце "Текст" указана информация, необходимая для осуществления атаки, в частности, количество блоков незашифрованного текста и соответствующих им блоков зашифрованного данным ключом текста. Для большинства атак противнику недостаточно перехватить произвольные тексты; незашифрованный текст должен иметь конкретную форму, выбранную противником. Такие незашифрованные тексты называются выбранными незашифрованными текстами. Следует заметить, что существуют атаки, которые могут использовать любой известный незашифрованный текст в противоположность выбранному незашифрованному тексту.

В столбце "Байты памяти" указано наибольшее число байтов памяти, которые требуются в любой точке осуществления атаки; это необязательно эквивалентно хранению всей необходимой информации.

Столбец "Операции" содержит ожидаемое число операций, которое необходимо для осуществления атаки. Трудно преобразовать данное число в оценку времени, так как время зависит от вычислительных возможностей, а также от возможности параллельного выполнения процедур. Природа операций также является определенным фактором; обычно рассматриваются операции полного шифрования, но операцией может быть также частичное шифрование или другая операция. Даже в случае полного шифрования для выполнения может требоваться различное время. Следовательно, число операций, необходимых для атаки, должно рассматриваться только как приблизительная основа для сравнения различных атак.

Могут использоваться различные наборы тестов, обеспечивающие полный перебор ключей. В принципе, таким способом может быть атакован любой блочный алгоритм

шифрования. Для трех вариантов размера ключа *AES* полный перебор ключей требует в среднем 2^{127}, 2^{191} или 2^{255} операций. Даже наименьшее из этих чисел говорит о том, что на сегодняшний день атака посредством перебора всех ключей не имеет практического значения.

Таблица 4.1. Атаки на варианты с уменьшенным числом раундов

Алгоритм, раунды	Раунды (длина ключа)	Тип атаки	Текст	Байты памяти	Операции
MARS 16 Core (C)	11C	Amp. Boomerang	2^{65}	2^{70}	2^{229}
16 Mixing (M)	16M, 5C	Meet-in-Middle	8	2^{236}	2^{232}
	16M, 5C	Diff. M-i-M	2^{50}	2^{197}	2^{247}
	6M, 6C	Amp. Boomerang	2^{69}	2^{73}	2^{197}
RC6 20	14	Stat. Disting.	2^{118}	2^{112}	2^{122}
	12	Stat. Disting.	2^{94}	2^{42}	2^{119}
	14 (192,256)	Stat. Disting.	2^{110}	2^{42}	2^{135}
	14 (192,256)	Stat. Disting.	2^{108}	2^{74}	2^{160}
	15 (256)	Stat. Disting.	2^{119}	2^{138}	2^{215}
Rijndael 10 (128)	4	Truncated Diff.	2^{9}	Small	2^{9}
	5	Truncated Diff.	2^{11}	Small	2^{40}
	6	Truncated Diff.	2^{32}	$7*2^{32}$	2^{72}
	6	Truncated Diff.	$6*2^{32}$	$7*2^{32}$	2^{44}
	7 (192)	Truncated Diff.	$19*2^{32}$	$7*2^{32}$	2^{155}
	7 (256)	Truncated Diff.	$21*2^{32}$	$7*2^{32}$	2^{172}
	7	Truncated Diff.	2^{128} - 2^{199}	2^{61}	2^{120}
	8 (256)	Truncated Diff.	2^{128} - 2^{199}	2^{101}	2^{204}
	9 (256)	Related Key	2^{77}	NA	2^{224}
Rijndael 12 (192)	7 (192)	Truncated Diff.	2^{32}	$7*2^{32}$	2^{184}
	7 (256)	Truncated Diff.	2^{32}	$7*2^{32}$	2^{200}
Rijndael 14 (256)	7 (192, 256)	Truncated Diff.	2^{32}	$7*2^{32}$	2^{140}
Serpent 32	8 (192, 256)	Amp. Boomerang	2^{113}	2^{119}	2^{179}
	6 (256)	Meet-in-Middle	512	2^{246}	2^{247}
	6	Differential	2^{83}	2^{40}	2^{90}
	6	Differential	2^{71}	2^{75}	2^{103}
	6 (192, 256)	Differential	2^{41}	2^{45}	2^{163}
	7 (256)	Differential	2^{122}	2^{126}	2^{248}
	8 (192, 256)	Amp. Boomerang	2^{128}	2^{133}	2^{163}
	8 (192, 256)	Amp. Boomerang	2^{110}	2^{115}	2^{175}
	9 (256)	Amp. Boomerang	2^{110}	2^{212}	2^{252}

Twofish 16	6 (256)	Impossible Diff.	NA	NA	2^{256}
	6	Related Key	NA	NA	NA

NA - информация недоступна.

Полный перебор ключа требует меньше памяти и информации и может легко выполняться параллельно с использованием нескольких процессоров. Таким образом, любую атаку, требующую операций больше, чем необходимо при полном переборе ключей, осуществить сложнее. По этой причине многие из атак на варианты с уменьшенным *числом раундов* относятся только к большей *длине ключа AES*, хотя и в данном случае требования выполнения не представляют сегодня практического интереса. Аналогично требования памяти важны для многих атак на варианты с уменьшенным *числом раундов*.

То же относится к информации, необходимой для выполнения атак на варианты с уменьшенным *числом раундов*. Почти все такие атаки требуют более 2^{30} шифрований выбранного текста; другими словами, более биллиона шифрований, а иногда даже больше. Даже если один и тот же ключ используется такое количество раз, не представляется реальным для противника собрать такое количество информации. Например, в случае линейной или дифференциальной атаки на DES требуется 2^{43} известного незашифрованного текста и 2^{47} шифрований выбранного незашифрованного текста. Тем не менее, случаи подобных атак против DES известны.

Одна из моделей для сбора такого большого количества информации требует от противника физического доступа к одному или нескольким устройствам шифрования, которые используют тот же самый секретный ключ. В этом случае другим полезным набором тестов является набор, который не требует хранить всю "codebook", т.е. таблица содержит только блоки зашифрованного текста, соответствующие каждому возможному блоку незашифрованного текста. Такая таблица требует для хранения 2^{132} байт памяти.

MARS

Существует много способов упростить *MARS* для анализа, так как он имеет гетерогенную структуру, состоящую из четырех различных типов раундов. 16 раундов с ключом криптографического ядра разделены 16 раундами перемешивания без использования ключа, а также pre- и post-whitening.

Известны четыре атаки на три упрощенных варианта *MARS*. Первый вариант включает 11 раундов ядра без раундов перемешивания и забеливания. Предложен новый тип урезанной дифференциальной атаки, названной boomerang-amplifier. Второй вариант включает забеливание и все 16 раундов перемешивания, но снижает количество раундов ядра с 16 до 5. В данном варианте предложены две различные атаки meet-in-the-middle; для такой атаки противнику не требуется выбирать незашифрованные тексты. Третий вариант включает забеливание, но снижает как *число раундов* перемешивания, так и *число раундов* ядра с 16 до 6.

RC6

Известны две атаки на варианты *RC6*, представляющие небольшие, но итерационные статистические предположения относительно функции раунда. Полученные статистические взаимосвязи между определенной формой входов и соответствующими им выходами могут быть использованы для нахождения отличия определенного *числа раундов RC6* от случайной перестановки. Предполагается, что различие подключей является равномерно случайным. Атака представляет собой метод восстановления секретного ключа для 12-, 14- и 15-раундовых вариантов.

Rijndael

Спецификация *Rijndael* описывает урезанную дифференциальную атаку для 4-, 5- и 6-раундовых вариантов *Rijndael* на основе 3-раундовых отличий *Rijndael*. Данная атака названа Square, по имени алгоритма шифрования, к которому она впервые была применена.

Возможно также создание атак на варианты *Rijndael*, непосредственно полученных из Square-атаки. Square-атака может быть расширена на 7-раундовый вариант *Rijndael* с помощью дополнительного раунда для подключей. В таблице были приведены результаты для 192- и 256-битных ключей, когда общее число операций остается меньше, чем требуется для экстенсивного перебора.

Serpent

Используется amplified boomerang технология для создания 7 раундов отличий в *Serpent*, что приводит к атаке на вариант *Serpent* с 8 раундами для 192- и 256-битных ключей. Уточнение основано на экспериментальном наблюдении уменьшения текстов, памяти и обработки, требуемых для атаки; также предлагается расширение атаки на 9-раундовый вариант. Кроме того, возможна стандартная атака meet-in-the-middle и дифференциальные атаки на 8-раундовый вариант *Serpent*, что требует полного codebook.

Twofish

Обнаружены две атаки на варианты *Twofish*. Пять раундов дифференциала используются для атаки 6-раундового варианта *Twofish* при 256-битном ключе, что требует количества выполняемых операций, эквивалентного тому, которое необходимо для экстенсивного поиска. Если pre- и post-whitening из варианта удалить, то атака может быть расширена на 7 раундов; в качестве альтернативы вариант из 6 раундов может быть атакован со сложностью, меньшей чем экстенсивный перебор для каждой *длины ключа*. Также возможны атаки на *Twofish* с уменьшенным *числом раундов*, для которого упрощения сделаны другими способами, например с помощью фиксированных S-boxes, посредством удаления забеливания или подключей или допуская частичное угадывание ключа.

Пока не известны атаки на *Twofish* с помощью простого уменьшения *числа раундов*. Известны дифференциальные характеристики 6 раундов, которые применимы только к определенным зависящим от ключа S-boxes, таким образом атака оказывается возможной только на определенное подмножество ключей. Данное конкретное подмножество ключей может рассматриваться как класс слабых ключей, так как авторы утверждают, что характеристики, подобные этим, непосредственно приводят к атакам на 7- и 8-раундовые варианты *Twofish*.

Резерв безопасности

NIST планирует оценить вероятность того, что будут найдены короткие аналитические атаки на рассматриваемые алгоритмы со всеми указанными для них раундами в ближайшие несколько десятилетий или атаки на ключ с помощью простого перебора станут практически возможными. Тем не менее, достаточно трудно экстраполировать данные для вариантов с уменьшенным *числом раундов* для реальных алгоритмов. Атаки на варианты с уменьшенным *числом раундов* не представляют практического интереса для атакующего, так как требуют большого количества ресурсов и поэтому более трудны для выполнения, чем экстенсивный перебор всех ключей. Более того, даже если короткая атака на упрощенный вариант имеет успех, исходный алгоритм может оставаться в безопасности.

Тем не менее, техника атак будет совершенствоваться, и ресурсов, доступных для их выполнения, будет больше, поэтому следует проявлять большую осмотрительность при выборе алгоритмов, предпочитая больший *резерв безопасности*. Если уже незначительное упрощение допускает атаку на некоторый алгоритм, а другой алгоритм может быть атакован при большем упрощении, то это говорит о том, что второй алгоритм имеет больший *резерв безопасности*. Упрощение, состоящее в уменьшении *числа раундов*, вполне целесообразно, потому что большинство атак, дифференциальный и линейный криптоанализ могут быть эффективно остановлены, если *число раундов* будет достаточно большим. Следовательно, полное *число раундов*, указанное для алгоритма, сравнивается с наибольшим *числом раундов*, для которого в настоящий момент существует атака. Отношение этих чисел определяется как "фактор безопасности" и вычисляется для каждого кандидата.

Существует несколько проблем, касающихся данной метрики. В общем случае результаты оказывают влияние на алгоритмы, для которых была проведена большая проверка в ограниченный период анализа. Это, вероятно, может произойти, если конкретный алгоритм является или, по крайней мере, кажется более простым при проверке на определенные атаки. Другим фактором может быть происхождение алгоритма и выбранные им технологии, а также существование ранее разработанных атак. Предложенная метрика может быть недостаточно хорошим критерием относительно сопротивляемости алгоритмов новым и неизвестным технологиям атак.

Разработка метрики основана на небольшом *числе раундов*, атаки на которые в настоящий момент технически возможны. Не существует исходного определения количества анализируемых раундов или даже общего *числа раундов*, определенного для каждого алгоритма. Например, должно ли забеливание в *MARS*, *Twofish*, *RC6* и *Rijndael* считаться раундом или частью раунда? *MARS* имеет 16 раундов перемешивания без использования ключа и 16 раундов ядра с использованием ключа: является *MARS* 16-раундовым алгоритмом, 32-раундовым или чем-то промежуточным? Должны ли атаки игнорировать раунды перемешивания? Должны ли варианты с уменьшенным *числом раундов Serpent* или *Rijndael* требовать какой-либо модификации заключительного раунда? Другим неопределенным фактором является *длина ключа*, особенно для *Rijndael*, в котором *число раундов* зависит от *длины ключа*.

Какие типы атак должны анализироваться? Одни атаки будут успешными только против небольшой доли ключей; для других требуются операции шифрования для неизвестных ключей; третьи определяют различия для выходов случайных перестановок без явного метода восстановления ключа; некоторые атаки основаны на экспериментальных предположениях. Более того, атаки требуют различных ресурсов; многие даже предполагают, что атакующему доступен весь codebook.

Таким образом, NIST не пытался сократить свои исследования ресурсов безопасности финалистов до единственной метрики. Были рассмотрены все предложенные данные и использовано исходное число анализируемых раундов относительно общего *числа раундов*, указанного для алгоритма. Результат приведен ниже для каждого финалиста.

Заметим, что раунды, определенные для кандидатов, не обязательно сопоставимы друг с другом. Например, алгоритмы, основанные на сети Фейштеля, т.е. *MARS*, *RC6* и *Twofish*, требуют двух раундов для изменения полного слова данных, в то время как для *Rijndael* и *Serpent* это выполняется в единственном раунде.

MARS: результаты для *MARS* зависят от обработки "wrapper", например, pre- и post-whitening и 16 раундов перемешивания без использования ключа, которые окружают 16 раундов ядра с использованием ключа. Без подобного wrapper 11 из 16 раундов ключа могут быть атакованы. С wrapper *MARS* имеет намного больше раундов, чем то количество, которое может быть атаковано: только 5 из 16 раундов ядра или 21 из 32 общего *числа раундов* может быть атаковано. Либо, если wrapper считать как часть раунда с ключом, то 7 из 18 раундов может быть атаковано. Для любого из этих случаев *MARS* демонстрирует достаточно большой *резерв безопасности*.

RC6: атаки созданы для 12, 14 и 15 из 20 раундов *RC6*, в зависимости от *длины ключа*. Тем самым *RC6* показывает адекватный *резерв безопасности*.

Rijndael: для 128-битных ключей 6 или 7 из 10 раундов *Rijndael* могут быть атакованы, атака на 7 раундов требует полного codebook. Для 192-битных ключей 7 из 12 раундов могут быть атакованы. Для 256-битных ключей 7, 8 или 9 из 14 раундов могут быть атакованы. Атака на 8 раундов требует полного codebook и атака на 9 раундов требует шифрования для неизвестных ключей. *Rijndael* демонстрирует адекватный *резерв безопасности*.

Serpent: атаки созданы для 6, 8 или 9 из 32 раундов *Serpent*, в зависимости от *длины ключа*. *Serpent* показывает адекватный *резерв безопасности*.

Twofish: предпринята атака на 6 из 16 раундов *Twofish*, которая требует операций шифрования для неизвестных ключей. Другая атака, предложенная для 6 раундов для 256-битного ключа, является более эффективной, чем экстенсивный поиск ключа. *Twofish* демонстрирует адекватный *резерв безопасности*.

Принципы разработки и происхождение алгоритмов

Исторические корни, лежащие в основе принципов разработки, влияют на доверие к алгоритму и на анализ его безопасности. Это также применимо к составным элементам алгоритма, таким как S-boxes. Требуется много времени для разработки атак на неизвестные технологии, традиционные технологии обычно анализируются больше, особенно если уже существует атака на них. Например, сеть Фейштеля хорошо изучена, т.к. она применяется в DES, и три финалиста используют варианты этой конструкции. Другим элементом, способным повлиять на доверие к алгоритму, является разработка S-boxes, которые могут содержать различные скрытые "люки". Это обсуждается далее для каждого финалиста.

MARS: разнородная структура раунда *MARS* не исследована. Как раунд перемешивания, так и раунд ядра основаны на сети Фейштеля со значительными изменениями. *MARS* использует много различных операций, большинство из которых традиционны. Материал ключа и данные применяются для регулирования различных операций ротации. S-box создается детерминировано с учетом требуемых свойств; таким образом, спецификация *MARS* утверждает, хотя это маловероятно, что *MARS* содержит какую-либо структуру, которая может использоваться в качестве люка для атаки. Спецификация *MARS* не рассматривает какой-либо алгоритм в качестве своего предшественника.

RC6: разработка *RC6* развивалась из разработки RC5, который анализировался несколько лет. Безопасность обоих алгоритмов основана на переменных ротациях как основной источник нелинейности; S-boxes в алгоритме не существует. Операция переменной ротации в *RC6*, в отличие от RC5, регулируется квадратичной функцией от данных. Управление ключом в *RC6* и RC5 не отличается. Структура раунда *RC6* является вариантом сети Фейштеля. Спецификация *RC6* утверждает, что в *RC6* не существует люков, потому что при вычислении ключа используется только часть *RC6*, которая математически хорошо изучена.

Rijndael: *Rijndael* является байт-ориентированным алгоритмом шифрования, основанным на "Квадрате (Square)". Представляется, что Square-атака является отправной точкой для дальнейшего анализа. Типы операций подстановки и перестановки, используемые в *Rijndael*, являются стандартными. S-box имеет математическую структуру, основанную на комбинации инверсии в поле Галуа и афинного преобразования. Хотя данная математическая структура может помочь в осуществлении атаки, структура не является скрытой. Спецификация *Rijndael* утверждает, что если есть подозрение, что S-box содержит люк, то S-box может быть заменен.

Serpent: Serpent является байт-ориентированным алгоритмом. Типы операций подстановки и перестановки являются стандартными. S-boxes создаются детерминировано, как и в случае DES, и обладают теми же свойствами; спецификация *Serpent* устанавливает, что при такой конструкции нет опасения, что существуют люки. Спецификация *Serpent* не рассматривает какой-либо алгоритм в качестве своего предшественника.

Twofish: *Twofish* использует незначительную модификацию сети Фейштеля. Спецификация не рассматривает какой-либо алгоритм в качестве своего предшественника, но существует несколько предшествующих алгоритмов, которые разделяют важные свойства *Twofish*, такие как зависящие от ключа S-boxes и подходы к их разработке. Спецификация *Twofish* предполагает, что *Twofish* не имеет люков и доказывает это утверждение несколькими аргументами, включая переменные S-boxes.

Простота

Простота является свойством, влияние которого на безопасность трудно оценивать. С одной стороны, сложные алгоритмы могут считаться менее удобными для атаки. С другой стороны, результаты легче получить для простого алгоритма, и простой алгоритм проще проанализировать. Следовательно, при анализе *AES* легче выполнить анализ более простого алгоритма.

Не существует единого мнения по поводу того, что следует понимать под простотой. *MARS* часто характеризуется как сложный алгоритм, но разработчики утверждают, что для *MARS* требуется всего несколько строчек кода на C. *RC6*, в противоположность этому, считается самым простым из финалистов, однако операция модульного умножения, которую он содержит, является куда более сложной, чем обычные операции шифрования. Наиболее простой алгоритм шифрования не обязательно является самым легким для анализа.

Для стандартного дифференциального криптоанализа реально используемые типы операций влияют на строгость анализа безопасности. Если материал ключа перемешивается с данными только операцией XOR, как в *Serpent* и *Rijndael*, то пары незашифрованного текста с данной XOR разницей являются естественными входами, и анализ безопасности относительно очевиден. Если материал ключа перемешивается с данными более чем в одной операции, как в случае остальных финалистов, то исходного описания разницы не существует, и анализ безопасности требует больших усилий. Аналогично, использование переменных ротаций в *MARS* и *RC6* скрывает возможность получения ясных результатов относительно безопасности при линейных и дифференциальных атаках.

Другим аспектом простоты, который относится к анализу безопасности, является масштабируемость. Если упрощенный вариант может, например, создаваться с меньшей *длиной блока*, то проводить эксперименты с вариантами становится легче, и при этом изучаются свойства исходного алгоритма. Считается, что отсутствие уменьшенных версий *MARS* существенно затрудняет анализ и экспериментирование. Аналогично, "реальные" масштабируемые в сторону упрощения варианты *Twofish* создавать трудно. Оба требования являются правдоподобными, хотя следует заметить, что спецификации *MARS* и *Twofish* содержат достаточный анализ своих элементов разработки. В спецификации *Serpent* утверждается, что создать масштабируемый в сторону упрощения вариант *Serpent* нетрудно. *RC6* и *Rijndael* являются масштабируемыми в силу способа разработки.

Статистическое тестирование

Специалисты NIST разработали статистические тесты для финалистов *AES*, чтобы можно было оценить, имеют ли выходы алгоритмов при определенных условиях тестов свойства, которые ожидаются от случайно созданных выходов. Такие тесты созданы

для каждой из трех *длин ключа*. Дополнительно было разработано подмножество тестов для версий с уменьшенным *числом раундов* для каждого алгоритма.

При полном тестировании каждый из алгоритмов создает выглядящие случайными выходы для каждой из *длин ключа*. Для тестирования уменьшенного *числа раундов* выходы предыдущего раунда должны быть случайными, как и выходы каждого последующего раунда. Оказалось, что выходы *MARS* проявляют случайность на 4 и более раундах ядра, *RC6* и *Serpent* - на 4 и более раундах, *Rijndael* - на 3 и более раундах, *Twofish* - на 2 и более раундах.

Количественная мера, включающая степень полноты, лавинный эффект и критерий ограничения лавинного эффекта, является "неясной при случайных перестановках после очень небольшого *числа раундов*" для всех финалистов.

В заключении следует отметить, что ни один из финалистов не является статистически отличимым от случайной функции.

Другие замечания по безопасности

Существует много обзоров, касающихся различных свойств, которые могут влиять на безопасность финалистов.

MARS: разнородная структура *MARS* и разнообразие операций обеспечивают защиту от неизвестных атак. Управление ключом в *MARS* требует нескольких стадий перемешивания.

RC6: перемешивание, осуществляемое в *RC6* при управлении ключом, считается преимуществом с точки зрения безопасности.

Rijndael: обсуждаются три понятия, касающиеся математической структуры *Rijndael* и потенциальных уязвимостей. Во-первых, все операции алгоритма выполняются над целым байтом, а не над отдельными битами; это свойство позволяет осуществлять Square-атаку на варианты с уменьшенным *числом раундов*. Более того, при симметричном перемещении байтов уязвимость повышается. Для нарушения симметрии при управлении ключом используются различные константы раунда, причем эти константы имеют только один бит. Если *Rijndael* упростить таким образом, чтобы опустить эти константы раунда, шифрование будет сравнимо с ротацией каждого слова данных и подключей на байт.

Во-вторых, *Rijndael* является наиболее линейным. Можно показать, как применить линейное отображение на биты в каждом байте без изменения всего алгоритма, компенсируя линейное отображение в остальных элементах алгоритма, включая управление ключом. Аналогично, поле Галуа, лежащее в основе S-box, может быть представлено различными базисными векторами или преобразовано в другие поля Галуа с другими определяющими полиномами. Другими словами, математическая структура *Rijndael* допускает много эквивалентных формулировок. Существует предположение, что выполняя серию преобразований над S-box, атакующий может найти формулировку *Rijndael*, имеющую уязвимости.

Третье утверждение относится к относительно простой алгебраической формуле для S-box, которая приведена в спецификации *Rijndael*. Формула является полиномом степени 254 над данным полем Галуа, но в полиноме существует только девять термов, что намного меньше, чем ожидается для обычно случайно созданного S-box того же размера. Математическое выражение для повторения нескольких раундов *Rijndael* будет намного более сложным, но увеличение размера выражения как функции от *числа раундов* должно быть проанализировано в деталях.

Также отмечается, что атакующий, который восстанавливает или угадывает соответствующие биты подключей *Rijndael*, имеет возможность вычислить дополнительные биты подключей. (В случае DES это свойство помогает сконструировать линейные и дифференциальные атаки).

Serpent: *Serpent* имеет небольшой размер S-boxes, хотя следует отметить, что они хорошо разработаны с учетом линейного и дифференциального криптоанализа. Атакующий, который восстанавливает или угадывает соответствующие биты подключей, имеет возможность вычислить дополнительные биты подключей.

Twofish: *Twofish* использует новое понятие, которое состоит в формировании зависящих от ключа S-boxes. Это обеспечивает нестандартную зависимость между безопасностью алгоритма и структурой управления ключом и S-boxes. В случае 128-битного ключа (когда существует 128-битная энтропия) *Twofish* можно считать набором из 2^{64} различных криптосистем. 64-битное количество (представляющее собой 64 бита из исходных 128 бит энтропии), которое получено из исходного ключа, управляет выбором криптосистемы. Для любой конкретной криптосистемы 64 оставшиеся бита используются для ключа. Как результат такого разделения 128 битов высказываются утверждения, что *Twofish* может быть подвержен атаке divide-and-conquer. При такой атаке взломщик выясняет, какая из 2^{64} криптосистем используется, а затем определяет ее ключ. Однако не существует общей атаки, соответствующей указанной стратегии. Это означает, что если атакующий столкнулся с задачей дешифрования зашифрованного 128-битным ключом текста, то неясно, как разделить 128-битный ключ энтропии. С другой стороны, если 128-битный ключ используется повторно, то каждое применение может дать некоторую информацию о выбранной криптосистеме. Если атакующий имеет возможность выполнять повторяющееся исследование активной криптосистемы, то он имеет возможность определить, какая из 2^{64} криптосистем используется. То же можно сказать и о большей *длине ключа* (в общем случае для k-битных ключей криптосистема определяется $k/2$ битами энтропии).

Эта особенность *Twofish* называется свойством разделения ключа. Зависимость S-boxes в *Twofish* только от 64 бит энтропии в случае 128-битного ключа была специально предусмотрена разработчиками. Это решение является некоторым аналогом безопасности/эффективности, включающих определение *числа раундов* в системах с фиксированной функцией раунда. Утверждается, что если S-boxes зависят от 128 бит энтропии, то *число раундов Twofish* можно уменьшить, чтобы избежать отрицательного влияния на свойства ключа и/или количество исходного материала.

Краткая характеристика финалистов

Как уже отмечалось, ни против одного из финалистов не существует общих атак. Однако определение уровня безопасности, обеспечиваемого финалистами, является достаточно приблизительным. Суммируем некоторые известные характеристики безопасности финалистов.

MARS показывает высокую степень *резерва безопасности*. Кратко охарактеризовать *MARS* трудно, потому что фактически *MARS* реализует два различных типа раунда. *MARS* даже критиковали за сложность, которая может препятствовать анализу его безопасности.

RC6 показал адекватный *резерв безопасности*. Однако *RC6* критиковали за небольшой *резерв безопасности* по сравнению с другими финалистами. С другой стороны, все высоко оценили простоту *RC6*, облегчающую анализ безопасности. *RC6* произошел от RC5, который уже достаточно хорошо проанализирован.

Rijndael показал адекватный *резерв безопасности*. *Резерв безопасности* довольно трудно измерить, потому что *число раундов* изменяется в зависимости от *длины ключа*. *Rijndael* критиковался по двум направлениям: что его *резерв безопасности* меньше, чем

у других финалистов, и что его математическая структура может привести к атакам. Тем не менее, его структура достаточно проста и обеспечивает возможность анализа безопасности.

Serpent показал значительный *резерв безопасности*. *Serpent* также имеет простую структуру, безопасность которой легко проанализировать.

Twofish показал высокий *резерв безопасности*. Поскольку *Twofish* использует зависящую от ключа функцию раунда, для него замечания о *резерве безопасности* могут иметь меньшее значение, чем для других финалистов. Зависимость S-boxes *Twofish* только от `k/2` битов энтропии в случае k-битного ключа позволяет сделать вывод, что *Twofish* может быть подвегнут divide-and-conquer-атаке, хотя такая атака не найдена. *Twofish* был подвергнут критике за сложность, которая затрудняет его анализ.

5. Лекция: Алгоритмы симметричного шифрования. Часть 3. Разработка Advanced Encryption Standard (AES) (продолжение)

Программные реализации

Программные реализации охватывают широкий диапазон. В некоторых случаях память никак не ограничена; в других случаях RAM и/или ROM могут быть существенно ограничены. Иногда большое количество данных шифруется и дешифруется единственным ключом. В остальных случаях ключ изменяется часто, возможно, для каждого блока данных.

Скорость шифрования и/или дешифрования является прямой или косвенной противоположностью безопасности. Это означает, что число раундов, указанное для алгоритма, является фактором безопасности; скорость шифрования или дешифрования приблизительно пропорциональна числу раундов. Таким образом, скорость не может исследоваться независимо от безопасности.

Существует много других аспектов *программных реализаций*. Некоторые из них будут перечислены ниже, включая скорость и стоимость.

Размер машинного слова

Одной из проблем, которая возникает в *программных реализациях*, является лежащая в их основе архитектура. Платформы, на которых специалисты NIST выполняли тестирование, ориентированы на 32-битные архитектуры. Однако выполнение на 8-битных и 64-битных машинах также важно. Трудно прогнозировать, как различные архитектуры будут отличаться через 30 лет. Но также трудно назначить весовые коэффициенты для различных типов выполнения для текущего отрезка времени. Тем не менее, ожидается следующая картина:

Считается, что в ближайшие 30 лет 8-битные, 32-битные и 64-битные архитектуры будут играть важнейшую роль (в какой-то момент будут добавлены 128-битные архитектуры). Хотя 8-битные архитектуры используются приложениями, которые имеют и 32-битные версии, 8-битные архитектуры не исчезнут окончательно. Между тем некоторые 32-битные архитектуры будут вытеснены 64-битными версиями, но 32-битные архитектуры будут использоваться приложениями с более низкими требованиями, т.е. важность 32-битных архитектур также останется высокой. Важность 64-битных архитектур будет возрастать. Таким образом, *AES* должен хорошо выполняться на различных архитектурах.

Следует заметить, что выполнение не может быть классифицировано только на основе длины слова. Учитывается также еще один дополнительный фактор, предоставляемый ПО, который обсуждается в следующем разделе.

Другие проблемы архитектуры

Как *MARS*, так и *RC6* используют 32-битное умножение и 32-битную переменную ротацию. Эти операции, особенно ротация, не поддерживаются на некоторых 32-битных процессорах. Операции 32-битного умножения и ротации затруднены для реализации на процессорах с другой длиной слова. Более того, некоторые компиляторы на самом деле не используют операции ротации, даже если они существуют в наборе команд процессора. Следовательно, выполнение *MARS* и *RC6* может варьироваться от платформы (процессор и компилятор) к платформе больше, чем три остальных финалиста.

Языки реализации ПО

Выполнение также зависит от использования конкретного языка (например, ассемблер, компилируемый или интерпретируемый язык высокого уровня). В некоторых случаях играет роль конкретное ПО. Существует целый спектр возможностей. Как одна из крайностей, вручную созданный ассемблерный код обеспечивает выполнение лучшее, чем даже оптимизирующий компилятор. Другая крайность - это интерпретируемые языки, которые в общем случае плохо решают задачи оптимизации. Оптимизация, выполняемая компиляторами, находится где-то посередине. Также следует заметить, что некоторые компиляторы работают лучше других, обеспечивая поддержку предоставляемых архитектурой операций, например, 32-битную ротацию. Это увеличивает трудность оценки выполнения на различных платформах.

Относительно важности различных языков единого мнения нет. Многие считают, что ассемблерное кодирование больше всего подходит для оценки выполнения на данной архитектуре. Причина этого в том, что ручное кодирование выполняется тогда, когда важна скорость и недоступна аппаратная реализация. С другой стороны использование ассемблера или других способов увеличения скорости может увеличить стоимость. Стоимость разработки кода может быть существенной, особенно если целью является максимальная скорость. Например, оптимизация может быть эффективной, если используется ручное кодирование для языков высокого уровня, таких как C, или используется ассемблерный код. В некоторых окружениях скорость, с которой выполняется код, считается первостепенной при оценке эффективности по сравнению со стоимостью. В других случаях скорость установки ключа является более важной, чем скорость шифрования или дешифрования. Это затрудняет разработку универсальной метрики для оценки выполнения финалистов.

Стоимость разработки кода может быть противопоставлена скорости. Это означает, что использование стандартного кода может минимизировать стоимость, но при этом может отсутствовать оптимизация для данного окружения. С другой стороны использование нестандартного кода, такого как код на ассемблере, может оптимизировать скорость за счет высокой стоимости разработки.

Оптимизация имеет широкий диапазон. Иногда ее можно осуществить без особых усилий. Более того, некоторые оптимизации могут быть портированы на другие платформы. В качестве противоположного примера можно сказать, что ряд оптимизаций требует больших усилий и/или ограничений для конкретных платформ.

Изменение скорости выполнения в зависимости от длины ключа

Программное выполнение *MARS*, *RC6* и *Serpent* не очень сильно изменяется в зависимости от длины ключа. Однако для *Rijndael* и *Twofish* установление ключа или шифрование/дешифрование заметно медленнее для 192-битных ключей, чем для 128-битных ключей и еще медленнее для 256-битных ключей.

Rijndael определяет больше раундов для большей длины ключа, что, естественно, затрагивает как скорость шифрования/дешифрования, так и время установления

ключа. Тем не менее, время установления ключа остается более коротким среди финалистов для больших размеров ключа.

Для большего размера ключа *Twofish* определяет дополнительные уровни генерации подключей и создания зависящих от ключа S-boxes. Вычисление подключа влияет только на скорость установки ключа. Однако создание S-boxes может повлиять на скорость как установки ключа, так и шифрования/дешифрования в зависимости от установленных опций, при которых вычисляются S-boxes.

Краткий вывод о скорости выполнения на основных программных платформах

Огромное количество информации собрано относительно скорости финалистов на различных программных платформах. Эти платформы включают 32-битные процессоры (реализации на C и Java), 64-битные процессоров (C и ассемблер), 8-битные процессоры (C и ассемблер), 32-битные смарт-карты (ARM) и цифровые сигнальные процессоры. В таблицах описано сравнение выполнения финалистов на различных платформах при использовании 128-битных ключей. Скорости финалистов сгруппированы следующим образом. Первая группа (I) обладает самой высокой скоростью выполнения, далее следуют вторая (II) и третья (III) группы.

Таблица 5.1. Выполнение шифрования и дешифрования на различных платформах

	32 бита(C)	32 бита (Java)	64 бита (C и ассемблер)	8 бит (C и ассемблер)	32 бита смарт-карты (ARM)	Цифровые сигнальные процессоры
MARS	II	II	II	II	II	II
RC6	I	I	II	II	I	II
Rijndael	II	II	I	I	I	I
Serpent	III	III	III	III	III	III
Twofish	II	III	I	II	III	I

Таблица 5.2. Выполнение управления ключом в зависимости от платформы

	32 бита(C)	32 бита (Java)	64 бита (C и ассемблер)	8 бит (C и ассемблер)	Цифровые сигнальные процессоры
MARS	II	II	III	II	II
RC6	II	II	II	III	II
Rijndael	I	I	I	I	I
Serpent	III	II	II	III	I
Twofish	III	III	III	II	III

Таблица 5.3. Полная оценка выполнения

	Шифрование/ дешифрование	Установление ключа
MARS	II	II
RC6	I	II
Rijndael	I	I
Serpent	III	II
Twofish	II	III

В следующих оценках "низкое", "среднее" и "высокое" являются терминами, используемыми только в контексте этих пяти финалистов.

MARS обеспечивает среднее выполнение для шифрования, дешифрования и установления ключа.

RC6 обеспечивает от среднего до высокого выполнение для шифрования и дешифрования и среднее выполнение для установления ключа.

Rijndael обеспечивает последовательно высокое выполнение для шифрования, дешифрования и установления ключа, хотя выполнение и понижается для 192- и 256-битных ключей.

Serpent обеспечивает последовательно низкое выполнение для шифрования и дешифрования и платформно-зависимое выполнение для установления ключа.

Twofish обеспечивает платформно-зависимое выполнение для шифрования и дешифрования и последовательно низкое выполнение для установления ключа. В реализациях используется опция "Full Keying". Данная опция обеспечивает максимально быстрое шифрование с помощью выполнения большинства вычислений при установлении ключа. Выполнение шифрования/дешифрования или установления ключа понижается при увеличении размера ключа в зависимости от используемой опции ключа.

Изменение скорости в зависимости от режима

Другим фактором, который может влиять на выполнение алгоритма, является используемый режим операции. Алгоритм, выполняемый в режиме non-feedback (например, ECB), может быть реализован для независимой и, следовательно, одновременной, обработки блоков данных. Результаты одновременного выполнения затем собираются для создания потока информации, который должен быть идентичен потоку, создаваемому при последовательной обработке. Считается, что реализации, использующие данный подход, применяют "interleaved режим". Это противоречит режимам с обратной связью (например, Cipher Feedback, Cipher Block Chaining и т.д.), которые должны обрабатывать блоки данных последовательно. Режимы interleaved имеют преимущества на процессорах с возможностью параллельного выполнения.

Окружения с ограничениями пространства

В некоторых *окружениях*, обладающих небольшими объемами RAM и/или ROM для таких целей, как хранение кода (обычно в ROM), представление объектов данных, таких как S-boxes (которые могут храниться в RAM или ROM в зависимости от того, известны ли они заранее или нет) и хранение подключей (в RAM), существуют значительные ограничения. Теоретически могут использоваться промежуточные формы хранения, например EEPROM, для таких значений как подключи. Тем не менее, можно предполагать, что подключи также хранятся в RAM.

Кроме того, следует учесть, что доступная RAM должна использоваться для различных целей, таких как хранение промежуточных переменных при выполнении программы. Таким образом, не следует предполагать, что большие порции данного пространства доступны для хранения подключей.

В *окружениях с ограниченной памятью* объем ROM и RAM, необходимый для реализации финалистов, может быть основным фактором, определяющим их пригодность для данного окружения. Важным преимуществом является возможность вычисления подключей на лету.

Замечания по финалистам

MARS имеет определенные сложности в силу гетерогенной структуры раунда (четыре различных типа раунда). Для S-boxes необходимо 2 Kbytes ROM, что не является проблемой, т.к. всегда доступен достаточный объем ROM.

Обработка слабых ключей имеет некоторые проблемы в *окружениях с ограниченными ресурсами*. Необходимо использовать определенную форму образца для оценки полученных ключей. Необходимость проверок увеличивает время выполнения и объем ROM. Если подключи требуется создавать повторно, то это также влияет на время выполнения. При необходимости повторного создания подключей открывается возможность для атак, связанных со временем.

Переменные ротации также могут вызвать проблемы на ограниченном оборудовании. Здесь может помочь сопроцессор, эмулирующий переменные ротации использованием умножений по модулю.

Таким образом, следует отметить, что *MARS* имеет проблемы в *окружениях с ограниченными ресурсами*.

RC6: шифрование в *RC6* хорошо подходит для использования в смарт-картах. Как и в случае *MARS* переменные ротации могут эмулироваться умножениями по модулю.

Управление ключом является простым, но вычисление подключей на лету невозможно. Это может вызывать проблемы в некоторых смарт-картах. Хранение подключей может быть самым разным. С другой стороны установление ключа требует долгих вычислений по сравнению с циклами шифрования.

Rijndael является наиболее эффективным из всех финалистов. Операция AddRoundKey выполняется на сопроцессоре. Установление ключа очень эффективно. Однако преимущества *Rijndael* по сравнению с другими финалистами уменьшаются, если шифрование и дешифрование реализуются одновременно, что обусловлено относительным отсутствием ресурсов, разделяемых между шифрованием и дешифрованием.

Serpent: возможны два различных режима реализации *Serpent*: обычный и bit-sliced. Рассмотрим только обычную реализацию. Для таблиц требуется 2 Kbytes ROM, что не является проблемой.

Можно реализовать *Serpent*, используя 80 байт RAM и вычисляя подключи на лету.

Twofish: существует несколько возможных режимов реализации *Twofish*; все они соответствуют *окружениям с ограниченной памятью*.

Профили финалистов

MARS

Общая безопасность

MARS не имеет известных атак безопасности.

В отличие от других финалистов, *MARS* использует в качестве нелинейных компонентов зависящие от данных ротации и S-boxes. В силу нестандартной, разнородной структуры раунда (16 раундов перемешивания и 16 раундов ядра) трудно оценить предоставляемый *MARS* резерв безопасности. Данного финалиста критикуют за сложность, которая затрудняет анализ его безопасности.

Программные реализации

Эффективность *программных реализаций* зависит от того, насколько хорошо комбинация процессор/язык управляет операциями 32-битного умножения и переменной ротации. Это может привести к различным вариациям между процессорами и между компиляторами для данного процессора. По эффективности выполнения *MARS* принадлежит к среднему диапазону для шифрования/дешифрования и для установления ключа.

Окружения с ограничениями пространства

MARS недостаточно соответствует *окружениям с ограничениями пространства*, т.к. требует большого объема ROM. Причем следует отметить, что требования ROM имеют тенденцию роста.

Аппаратные реализации

MARS имеет средние потребности. Его эффективность ниже средней. Скорость реализации *MARS* зависит от используемой длины ключа.

Шифрование vs. дешифрование

Шифрование и дешифрование в *MARS* имеют аналогичные функции. Таким образом, скорость *MARS* при шифровании и дешифровании существенно не изменяется.

Свойства ключа

MARS требует вычисления 10 из 40 подключей за один раз, требуя дополнительных ресурсов для хранения этих 10 подключей. Это неудобно для *окружений с ограниченной памятью*. *MARS* также требует однократного выполнения управления ключом для создания всех подключей до первого дешифрования с конкретным ключом. Вычисление нескольких подключей за один раз требует больше ресурсов памяти, чем при вычислении подключей на лету.

Другие возможности настройки

MARS поддерживает длину ключа от 128 до 448 бит.

RC6

Общая безопасность

RC6 не имеет известных атак безопасности.

RC6 использует зависящие от данных ротации в качестве нелинейных компонентов. Его резерв безопасности является адекватным. Аналитики высоко оценивают простоту *RC6*, которая может способствовать анализу его безопасности. Предшественником *RC6* является RC5, который также стал предметом тщательного анализа.

Программные реализации

Преобладающие операции в *RC6* - умножение и переменные ротации. Программное выполнение зависит от того, насколько хорошо комбинация процессор/язык обрабатывает эти операции. *RC6* является наиболее быстрым из финалистов на 32-битных платформах. Однако его выполнение замедляется на 64-битных процессорах. Выполнение *RC6* улучшается, если он используется в режиме, допускающем interleaving. Время установления ключа признано средним.

Окружения с ограничениями пространства

RC6 имеет небольшие требования к ROM, что является преимуществом в *окружениях с ограничениями пространства*. У *RC6* при дешифровании отсутствует возможность вычисления подключа на лету, что создает повышенные требования к RAM. Это не вполне соответствует реализации в устройстве с существенными ограничениями на RAM, если требуется дешифрование.

Аппаратные реализации

RC6 может быть эффективно реализован.

Шифрование vs. дешифрование

Шифрование и дешифрование в *RC6* имеют аналогичные функции. Таким образом, эффективность *RC6* при шифровании и дешифровании существенно не отличается.

Свойства ключа

RC6 поддерживает возможность вычисления подключей на лету только для шифрования, используя порядка 100 байтов для промежуточных значений. Подключи дешифрования должны быть вычислены заранее.

Другие возможности настройки

Размеры блока, ключа и число раундов параметризуемы. *RC6* поддерживает размер ключа намного больше 256 бит.

Rijndael

Общая безопасность

Rijndael не имеет известных атак безопасности.

Rijndael использует S-boxes в качестве нелинейной компоненты. *Rijndael* демонстрирует адекватный резерв безопасности, но подвергается критике из-за математической структуры, которая может привести к атакам. С другой стороны, простая структура облегчает анализ безопасности.

Программные реализации

Rijndael очень хорошо выполняет шифрование и дешифрование на различных платформах, включая 8-битные и 64-битные платформы. Однако выполнение замедляется с увеличением длины ключа, т.к. возрастает число раундов. *Rijndael* имеет возможность параллельного выполнения, что позволяет эффективно использовать ресурсы процессора. Время установления ключа в *Rijndael* небольшое.

Окружения с ограничениями пространства

В целом, *Rijndael* хорошо соответствует *окружениям с ограничениями пространства*, в которых реализовано либо шифрование, либо дешифрование (но не оба одновременно). Он имеет очень небольшие требования к RAM и ROM. Недостатком является то, что требования ROM возрастают, если шифрование и дешифрование реализованы одновременно, хотя он все равно остается подходящим для такого окружения. Управление ключом для шифрования и дешифрования различается.

Аппаратные реализации

Rijndael имеет самую высокую производительность среди финалистов для feedback режимов и является вторым по производительности для не-feedback режимов. Для 192- и 256-битных ключей производительность падает из-за дополнительного числа раундов.

Шифрование vs. дешифрование

В *Rijndael* функции шифрования и дешифрования различны. При этом скорости шифрования и дешифрования существенно не отличаются, но установление ключа выполняется медленнее для дешифрования, чем для шифрования.

Свойства ключа

Rijndael поддерживает для шифрования вычисление подключей на лету. *Rijndael* требует однократного выполнения управления ключом для создания всех подключей до первого дешифрования с конкретным ключом.

Другие возможности настройки

Rijndael полностью поддерживает размеры блока и размеры ключа в 128, 192 и 256 бит в любой комбинации. В принципе, структуру *Rijndael* можно приспособить к любым размерам блока и ключа, кратным 32, а также изменить число раундов.

Serpent

Общая безопасность

Serpent не имеет известных атак безопасности.

Serpent использует S-boxes в качестве нелинейных компонент. *Serpent* демонстрирует значительный резерв безопасности и имеет простую структуру, которую легко анализировать.

Программные реализации

Serpent является наиболее медленным из финалистов при *программной реализации* шифрования и дешифрования.

Окружения с ограничениями пространства

Serpent хорошо соответствует *окружениям с ограничениями пространства*, включая низкие требования к RAM и ROM. Недостатком является то, что требования к ROM возрастают, если шифрование и дешифрование реализованы одновременно, но и при этом *Serpent* по-прежнему соответствует *окружениям с ограничениями пространства*.

Аппаратные реализации

Конвейерные реализации *Serpent* показывают самую высокую среди финалистов производительность для не-feedback режимов. *Serpent* занимает второе место по производительности в feedback режиме для основной архитектуры. Скорость выполнения *Serpent* не зависит от длины ключа.

Шифрование vs. дешифрование

Шифрование и дешифрование в *Serpent* имеют различные функции, которые разделяют ограниченные ресурсы. Одновременная реализация как шифрования, так и дешифрования требует пространства приблизительно в два раза больше, чем

необходимо отдельно для шифрования. Это является недостатком, когда в аппаратуре нужно одновременно реализовать обе функции.

Скорость выполнения *Serpent* при шифровании и дешифровании отличается несущественно.

Свойства ключа

Serpent поддерживает для шифрования и дешифрования возможность вычисления подключей на лету. Для дешифрования необходимо единственное вычисление для получения первого подключа дешифрования из исходного ключа. Это вычисление отличается от преобразования, которое используется для остальных подключей.

Другие возможности настройки

Serpent может обрабатывать любую длину ключа более 256 бит. Кроме того, для улучшения выполнения может использоваться технология bitslice на 32-битных процессорах.

Twofish

Общая безопасность

Twofish не имеет известных атак безопасности.

Twofish использует S-boxes в качестве нелинейных компонент. Демонстрирует высокий резерв безопасности, но вызвала критические замечания за свойство разделения ключа и за сложность, которая затрудняет анализ безопасности.

Программные реализации

Twofish показывает смешанные результаты при выполнении шифрования и дешифрования. Установление ключа происходит медленно. Время выполнения шифрования/дешифрования или установления ключа уменьшаются для большей длины ключа, в зависимости от используемой опции ключа.

Окружения с ограничениями пространства

Требования RAM и ROM соответствуют *окружениям с ограничениями пространства*.

Аппаратные реализации

Производительность и эффективность на основной аппаратуре являются средними. Возможны компактные реализации.

Шифрование vs. дешифрование

Шифрование и дешифрование в *Twofish* имеют идентичные функции. Таким образом, эффективность *Twofish* при шифровании и дешифровании не отличается. Одновременная реализация шифрования и дешифрования всего на 10 % увеличивает объем пространства по сравнению с реализацией одного только шифрования.

Свойства ключа

Twofish поддерживает вычисление подключей на лету как для шифрования, так и для дешифрования.

Другие возможности настройки

Спецификация *Twofish* описывает четыре опции для реализации зависимых от ключа S-boxes, позволяя варьировать выполнение. *Twofish* поддерживает произвольную длину ключа более 256 бит.

Заключительные оценки финалистов

Как уже не раз отмечалось, безопасность считается самым важным свойством при оценке финалистов. Так как многие из оставшихся результатов анализа часто охватывают как стоимость, так и характеристики алгоритма, было принято решение рассматривать эти факторы вместе в качестве второго после безопасности свойства.

Общая безопасность

На основании выполненного анализа безопасности можно утверждать, что не существует известных атак безопасности ни на одного из финалистов, и все пять финалистов проявили необходимый для *AES* уровень безопасности. В терминах резерва безопасности *MARS*, *Serpent* и *Twofish* показали самый высокий резерв безопасности, хотя резерв *RC6* и *Rijndael* также можно назвать адекватным. Ряд критических замечаний вызвала математическая структура *Rijndael* и свойство разделения ключа *Twofish*; однако это не приводит к атакам.

Программные реализации

RC6 и *Rijndael* демонстрируют скорость шифрования и дешифрования выше средней для 128-битных ключей, при этом *RC6* имеет особенно хорошие показатели на 32-битных платформах, а *Rijndael* выполняется более ровно на всех платформах. *MARS* имеет среднее выполнение шифрования и дешифрования на всех платформах, в зависимости от того, выполняет ли процессор 32-битные умножения и переменные ротации. *Twofish* демонстрирует смешанные результаты среди различных платформ для шифрования и дешифрования, но в целом имеет средние показатели по сравнению с другими финалистами. *Serpent* при шифровании и дешифровании несколько медленнее остальных финалистов.

Установление ключа *Rijndael* выполняет быстрее всех. Для *MARS*, *RC6* и *Serpent* этот показатель является средним, а *Twofish* здесь на последнем месте.

MARS, *RC6* и *Serpent* демонстрируют одинаковое выполнение шифрования и дешифрования для всех трех длин ключа. Выполнение шифрования и дешифрования *Rijndael* возрастает с увеличением длины ключа из-за увеличения числа раундов. Выполнение шифрования/дешифрования или установления ключа *Twofish* уменьшается при большей длине ключа в зависимости от используемых опций ключа.

RC6 является лучшим среди финалистов, если использует режим, обеспечивающий interleaving.

Окружения с ограничениями пространства

Rijndael предъявляет очень низкие требования к RAM и ROM и отлично соответствует *окружениям с ограничениями пространства*, если реализовано либо шифрование, либо дешифрование. Недостаток его в том, что требования к ROM возрастают, если как шифрование, так и дешифрование реализованы одновременно, хотя и в этом случае *Rijndael* остается соответствующим *окружению с ограничениями пространства*.

Serpent имеет низкие требования к RAM и ROM и очень хорошо соответствует *окружениям с ограничениями пространства*, когда реализовано либо шифрование, либо дешифрование. Как и у *Rijndael*, у *Serpent* требования к ROM возрастают, если шифрование и дешифрование реализованы одновременно, но, тем не менее, алгоритм остается соответствующим *окружениям с ограничениями пространства*.

Требования RAM и ROM *Twofish* соответствуют *окружениям с ограничениями пространства*.

RC6 имеет небольшие требования к ROM, что является преимуществом в *окружениях с ограничениями пространства*. Однако алгоритм не допускает возможности вычисления подключей на лету при дешифровании, увеличивая тем самым требования к RAM относительно других финалистов. Следовательно, *RC6* недостаточно хорошо соответствует реализации в устройствах с существенными ограничениями доступной RAM, если требуется шифрование.

MARS недостаточно соответствует *окружению с ограничениями пространства* вследствие требований к ROM, самых высоких среди финалистов. Кроме того, управление ключом *MARS* включает операции сравнения с образцом, которые требуют дополнительных ресурсов.

Аппаратные реализации

Serpent и *Rijndael* имеют лучшую аппаратную производительность среди финалистов как для режимов feedback, так и для других. *Serpent* демонстрирует самую высокую производительность среди финалистов в не-feedback режимах, и его эффективность (производительность/пространство) обычно очень высокая. *Rijndael* показал лучшую производительность среди финалистов в feedback-режимах. При большей длине ключа производительность *Rijndael* падает. Эффективность *Rijndael* также обычно заслуживает самой высокой оценки.

RC6 и *Twofish* обычно проявляют среднюю производительность. Производительность *RC6* возрастает в не-feedback режимах. Производительность *Twofish* иногда понижается при больших размерах ключа. *MARS* имеет средние требования к памяти, его производительность выше средней, и эффективность также.

Шифрование vs. дешифрование

Twofish, *MARS* и *RC6* нуждаются в небольшом дополнительном пространстве для одновременной реализации в аппаратуре как шифрования, так и дешифрования в противоположность реализации одного только шифрования. Функции шифрования и дешифрования для *Twofish* почти не отличаются , а для *MARS* и *RC6* являются аналогичными.

Шифрование и дешифрование *Rijndael* отличаются больше, чем у *Twofish*, *MARS* и *RC6*, хотя *Rijndael* может быть реализован таким способом, чтобы разделять некоторые аппаратные ресурсы.

Для *Serpent* функции шифрования и дешифрования отличаются, что позволяет разделять только очень ограниченные аппаратные ресурсы.

Все финалисты показывают очень небольшие различия в скорости между шифрованием и дешифрованием для данной длины ключа. Установление ключа *Rijndael* для дешифрования выполняется медленнее, чем для шифрования.

Свойства ключа

Twofish поддерживает вычисление подключей на лету как для шифрования, так и для дешифрования. О *Serpent* можно сказать то же самое, однако процесс дешифрования требует одного дополнительного вычисления. *Rijndael* поддерживает вычисление подключей на лету для шифрования, но требует вычисления за один раз всего управления ключом до выполнения первого дешифрования. *MARS* имеет характеристики, аналогичные *Rijndael*, за исключением того, что 10 ключей должны вычисляться и храниться одновременно. Их размещение требует дополнительных ресурсов при реализации *MARS*. *RC6* поддерживает вычисление ключей на лету только для шифрования, создавая при этом промежуточные значения.

Другие возможности настройки

MARS поддерживает длину ключа в диапазоне от 128 до 448 бит.

RC6 имеет параметризуемые размеры блока и ключа и параметризуемое число раундов, включая поддержку размеров ключа более 256 бит.

Rijndael поддерживает дополнительные размеры блока и ключа с приращением в 32 бита, число раундов также может быть изменено.

Serpent поддерживает любую длину ключа более 256 бит, и bitslice-реализация может обеспечить его выполнение на многих процессорах.

Twofish поддерживает произвольный размер ключа более 256 бит, и спецификация алгоритма предлагает дополнительно четыре опции.

Заключение

Каждый из финалистов показал адекватную безопасность, имеет определенные преимущества, и каждый может использоваться в качестве *AES*. Но в то же время каждый из алгоритмов в одной или нескольких областях уступает другим; ни один из финалистов не является абсолютным лидером.

В качестве предлагаемого *AES* алгоритма в результате длительного и сложного процесса оценки специалисты NIST выбрали *Rijndael*.

Rijndael очень хорошо выполняется как в программной, так и в аппаратной реализации в широком диапазоне окружений независимо от использования feedback или не-feedback режимов. Он показал отличное время установления ключа и хорошие свойства ключа. *Rijndael* имеет небольшие требования к памяти, что делает его пригодным для *окружений с ограниченными ресурсами*. В этом случае он также демонстрирует отличное выполнение.

6. Лекция: Алгоритмы симметричного шифрования. Часть 3. Алгоритмы Rijndael и RC6

Алгоритм Rijndael

Предварительные математические понятия

Практически все операции *Rijndael* определяются на уровне байта. Байты можно рассматривать как элементы конечного поля GF (2^8). Некоторые операции определены в терминах четырехбайтных слов. Введем основные математические понятия, необходимые для обсуждения алгоритма.

Поле GF(2^8)

Элементы конечного поля могут быть представлены несколькими различными способами. Для любой степени простого числа существует единственное конечное поле, поэтому все представления GF (2^8) являются изоморфными. Несмотря на подобную эквивалентность, представление влияет на сложность реализации. Выберем классическое полиномиальное представление.

Байт b, состоящий из битов $b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0$, представляется в виде полинома с коэффициентами из $\{0, 1\}$:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0$$

Пример: байт с шестнадцатеричным значением '57' (двоичное 01010111) соответствует полиному

$$x^6 + x^4 + x^2 + x + 1$$

Сложение

В полиномиальном представлении сумма двух элементов является полиномом с коэффициентами, которые равны сумме по модулю 2 (т.е. $1 + 1 = 0$) коэффициентов слагаемых.

Пример: '57' + '83' = 'DA' или в полиномиальной нотации:

$$(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2$$

В бинарной нотации мы имеем: $01010111 + 10000011 = 11011010$. Очевидно, что сложение соответствует простому XOR (обозначается как $\oplus$) на уровне байта.

Выполнены все необходимые условия Абелевой группы: операция сложения (каждой паре элементов сопоставляется третий элемент группы, называемый их суммой), ассоциативность, нулевой элемент ('00'), обратный элемент (относительно операции сложения) и коммутативность.

Умножение

В полиномиальном представлении умножение в GF (2^8) соответствует умножению полиномов по модулю неприводимого двоичного полинома степени 8. Полином является неприводимым, если он не имеет делителей, кроме 1 и самого себя. Для *Rijndael* такой полином называется $m(x)$ и определяется следующим образом:

$$m(x) = x^8 + x^4 + x^3 + x + 1$$

или '11B' в шестнадцатеричном представлении.

Пример: '57' • '83' = 'C1'

Или

$$(x^6 + x^4 + x^2 + x + 1)\ (x^7 + x + 1) =$$
$$= x^{13} + x^{11} + x^9 + x^8 + x^7 + x^7 + x^5 + x^3 + x^2 + x + x^6 + x^4 + x^2 + x + 1 =$$
$$= x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1$$

$(x^8 + x^4 + x^3 + x + 1)\ (x^5 + x^3) + x^7 + x^6 + 1 = x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1$

Следовательно,

$x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^8 + 1 \mod (x^8 + x^4 + x^3 + x + 1) = x^7 + x^6 + 1$

Ясно, что результат является двоичным полиномом не выше 8 степени. В отличие от сложения, простой операции умножения на уровне байтов не существует.

Умножение, определенное выше, является ассоциативным, и существует единичный элемент ('01'). Для любого двоичного полинома $b(x)$ не выше 8-й степени можно использовать расширенный алгоритм Евклида для вычисления полиномов $a(x)$ и $c(x)$ таких, что

$$b(x)\ a(x) + m(x)\ c(x) = 1$$

Следовательно,

$$a(x) \bullet b(x) \mod m(x) = 1$$

или

$$b^{-1}(x) = a(x) \mod m(x)$$

Более того, можно показать, что

$$a(x) \bullet (b(x) + c(x)) = a(x) \bullet b(x) + a(x) \bullet c(x)$$

Из всего этого следует, что множество из 256 возможных значений байта образует конечное поле GF (2^8) с XOR в качестве сложения и умножением, определенным выше.

Умножение на x

Если умножить $b(x)$ на полином x, мы будем иметь:

$$b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x$$

$x \bullet b(x)$ получается понижением предыдущего результата по модулю $m(x)$. Если $b_7 = 0$, то данное понижение является тождественной операцией. Если $b_7 = 1$, $m(x)$ следует вычесть (т.е. XORed). Из этого следует, что умножение на x может быть реализовано на уровне байта как левый сдвиг и последующий побитовый XOR с '1B'. Данная операция обозначается как $b = xtime\ (a)$.

Полиномы с коэффициентами из GF

Полиномы могут быть определены с коэффициентами из GF(2^8). В этом случае четырехбайтный вектор соответствует полиному степени 4.

Полиномы могут быть сложены простым сложением соответствующих коэффициентов. Как сложение в GF(2^8) является побитовым XOR, так и сложение двух векторов является простым побитовым XOR.

Умножение представляет собой более сложное действие. Предположим, что мы имеем два полинома в GF(2^8).

$$a(x) = a_3x^3 + a_2x^2 + a_1x + a_0$$
$$b(x) = b_3x^3 + b_2x^2 + b_1x + b_0$$

$c(x) = a(x)\ b(x)$ определяется следующим образом:

$$c(x) = c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0$$

$$c_0 = a_0 \bullet b_0$$
$$c_1 = a_1 \bullet b_0 \oplus a_0 \bullet b_1$$
$$c_2 = a_2 \bullet b_0 \oplus a_1 \bullet b_1 \oplus a_0 \bullet b_2$$
$$c_3 = a_3 \bullet b_0 \oplus a_2 \bullet b_1 \oplus a_1 \bullet b_2 \oplus a_0 \bullet b_3$$
$$c_4 = a_3 \bullet b_1 \oplus a_2 \bullet b_2 \oplus a_1 \bullet b_3$$
$$c_5 = a_3 \bullet b_2 \oplus a_2 \bullet b_3$$
$$c_6 = a_3 \bullet b_3$$

Ясно, что в таком виде $c(x)$ не может быть представлен четырехбайтным вектором. Понижая $c(x)$ по модулю полинома 4-й степени, результат может быть полиномом степени ниже 4. В *Rijndael* это сделано с помощью полинома

$$M(x) = x^4 + 1$$

так как

$$x^j \bmod (x^4 + 1) = x^{j \bmod 4}$$

Модуль, получаемый из $a(x)$ и $b(x)$, обозначаемый $d(x) = a(x) \otimes b(x)$, получается следующим образом:

$$d_0 = a_0 \bullet b_0 \oplus a_3 \bullet b_1 \oplus a_2 \bullet b_2 \oplus a_1 \bullet b_3$$
$$d_1 = a_1 \bullet b_0 \oplus a_0 \bullet b_1 \oplus a_3 \bullet b_2 \oplus a_2 \bullet b_3$$
$$d_2 = a_2 \bullet b_0 \oplus a_1 \bullet b_1 \oplus a_0 \bullet b_2 \oplus a_3 \bullet b_3$$
$$d_3 = a_3 \bullet b_0 \oplus a_2 \bullet b_1 \oplus a_1 \bullet b_2 \oplus a_0 \bullet b_3$$

Операция, состоящая из умножения фиксированного полинома $a(x)$, может быть записана как умножение матрицы, где матрица является циклической. Мы имеем

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Замечание: $x^4 + 1$ не является несократимым полиномом в GF (2^8), следовательно, умножение на фиксированный полином необязательно обратимо. В алгоритме *Rijndael* выбран фиксированный полином, который имеет обратный.

Умножение на x

При умножении $b(x)$ на полином x будем иметь:

$$b_3x^4 + b_2x^3 + b_1x^2 + b_0x$$

$x \otimes b(x)$ получается понижением предыдущего результата по модулю $1 + x^4$.

Это дает

$$b_2x^3 + b_1x^2 + b_0x + b_3$$

Умножение на x эквивалентно умножению на матрицу, как описано выше со всеми $a_i = '00'$ за исключением $a_1 = '01'$. Имеем:

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 00 & 00 & 00 & 01 \\ 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Следовательно, умножение на x соответствует циклическому сдвигу байтов внутри вектора.

Обоснование разработки

При разработке алгоритма учитывались следующие три критерия:

- противодействие всем известным атакам;
- скорость и компактность кода для широкого круга платформ;
- простота разработки.

В большинстве алгоритмов шифрования преобразование каждого раунда имеет структуру *сети Фейштеля* . В этом случае обычно часть битов в каждом промежуточном состоянии просто перемещается без изменения в другую половину. Преобразование *раунда* алгоритма *Rijndael* не имеет структуру *сети Фейштеля*. Вместо этого преобразование каждого *раунда* состоит из четырех различных преобразований, называемых слоями.

Каждый слой разрабатывался с учетом противодействия линейному и дифференциальному криптоанализу. В основу каждого слоя положена своя собственная функция:

1. Нелинейный слой состоит из параллельного применения *S-boxes* для оптимизации нелинейных свойств в наихудшем случае.
2. Слой линейного перемешивания строк гарантирует высокую степень диффузии для нескольких *раундов*.
3. Слой линейного перемешивания столбцов также гарантирует высокую степень диффузии для нескольких *раундов*.
4. Дополнительный слой ключа состоит из простого XOR промежуточного состояния с ключом *раунда*.

Перед первым *раундом* применяется дополнительное *забеливание* с использованием ключа. Причина этого состоит в следующем. Любой слой после последнего или до первого добавления ключа может быть просто снят без знания ключа и тем самым не добавляет безопасности в алгоритм (например, начальная и конечная перестановки в DES). Начальное или конечное добавление ключа применяется также в некоторых других алгоритмах, например IDEA, SAFER и Blowfish.

Для того чтобы сделать структуру алгоритма более простой, слой линейного перемешивания последнего *раунда* отличается от слоя перемешивания других *раундов*. Можно показать, что это в любом случае не повышает и не понижает безопасность. Это аналогично отсутствию операции swap в последнем *раунде* DES.

Спецификация алгоритма

Rijndael является блочным алгоритмом шифрования с переменной длиной блока и переменной длиной ключа. Длина блока и длина ключа могут быть независимо установлены в 128, 192 или 256 бит.

Состояние, ключ шифрования и число раундов

Различные преобразования выполняются над промежуточным результатом, называемым состоянием.

Состояние можно рассматривать как двумерный массив байтов. Этот массив имеет четыре строки и различное число столбцов, обозначаемое как `Nb`, равное длине блока, деленной на `32`. Ключ также можно рассматривать как двумерный массив с четырьмя строками. Число столбцов *ключа шифрования*, обозначаемое как `Nk`, равно длине ключа, деленной на `32`.

В некоторых случаях эти блоки также рассматриваются как одномерные массивы четырехбайтных векторов, где каждый вектор состоит из соответствующего столбца. Такие массивы имеют длину 4, 6 или 8 соответственно, и индексы в диапазонах 0 … 3, 0 … 5 или 0 … 7. Четырехбайтные вектора иногда мы будем называть словами.

Если необходимо указать четыре отдельных байта в четырехбайтном векторе, будет использоваться нотация `(a, b, c, d)`, где `a`, `b`, `c` и `d` являются байтами в позициях 0, 1, 2 и 3, соответственно, в рассматриваемом столбце, векторе или слове.

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{0,4}$	$A_{0,5}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$	$A_{1,5}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$	$A_{2,5}$
$A_{3,0}$	$A_{3,1}$	$A_{3,1}$	$A_{3,3}$	$A_{3,4}$	$A_{3,5}$

$K_{0,0}$	$K_{0,1}$	$K_{0,2}$	$K_{0,3}$
$K_{1,0}$	$K_{1,1}$	$K_{1,2}$	$K_{1,3}$
$K_{2,0}$	$K_{2,1}$	$K_{2,2}$	$K_{2,3}$
$K_{3,0}$	$K_{3,1}$	$K_{3,2}$	$K_{3,3}$

Рис. 6.1. Пример состояния (с Nb = 6) и ключа шифрования (с Nk = 4)

Входы и выходы *Rijndael* считаются одномерными массивами из 8 байтов, пронумерованными от `0` до `4* Nb - 1`. Следовательно, эти блоки имеют длину 16, 24 или 32 байта, и массив индексируется в диапазонах `0 … 15`, `0 … 23` или `0 … 31`. Ключ считается одномерным массивом 8-битных байтов, пронумерованных от `0` до `4* Nk - 1`. Следовательно, эти блоки имеют длину 16, 24 или 32 байта, и массив индексируется в диапазонах `0 … 15`, `0 … 23` или `0 … 31`.

Входные байты алгоритма отображаются в байты состояния в следующем порядке: $A_{0,0}$, $A_{1,0}$, $A_{2,0}$, $A_{3,0}$, $A_{0,1}$, $A_{1,1}$, $A_{2,1}$, $A_{3,1}$, … Байты *ключа шифрования* отображаются в массив в следующем порядке: $K_{0,0}$, $K_{1,0}$, $K_{2,0}$, $K_{3,0}$, $K_{0,1}$, $K_{1,1}$, $K_{2,1}$, $K_{3,1}$, … После выполнения операции шифрования выход алгоритма получается из байтов состояния аналогичным образом.

Следовательно, если одноразмерный индекс байта в блоке есть `n`, и двухмерный индекс есть `(i,j)`, то мы имеем:

```
I = n mod 4
J = ⌊ n / 4 ⌋
N = i + 4*j
```

Более того, индекс `i` является также номером байта в четырехбайтном векторе или слове, `j` является индексом вектора или слова во вложенном блоке.

Число *раундов* обозначается `Nr` и зависит от значений `Nb` и `Nk`, что показано в следующей таблице.

Таблица 6.1. Число раундов как функция от длины блока и длины ключа

Nr	Nb = 4	Nb = 6	Nb = 8
Nk = 4	10	12	14
Nk = 6	12	12	14
Nk = 8	14	14	14

Преобразование раунда

Преобразование *раунда* состоит из четырех различных преобразований. В нотации на псевдо C это можно записать следующим образом:

```
Round (State, RoundKey)
{
   ByteSub (State);
   ShiftRow (State);
   MixColumn (State);
   AddRoundKey (State, RoundKey);
}
```

Заключительный *раунд* алгоритма немного отличается и выглядит следующим образом:

```
FinalRound (State, RoundKey)
{
   ByteSub (State);
   ShiftRow (State);
   AddRoundKey (State, RoundKey);
}
```

Как мы видим, заключительный *раунд* эквивалентен остальным, за исключением того, что отсутствует слой `MixColumn`.

Преобразование ByteSub

Преобразование `ByteSub` является нелинейной байтовой подстановкой, выполняющейся для каждого байта состояния независимо. Таблица подстановки является обратимой и сконструирована в виде композиции двух преобразований:

1. Во-первых, берется мультипликативная инверсия в GF (2^8) с определенным выше представлением. '00' отображается сам в себя.
2. Затем применяется афинное (в GF (2)) преобразование, определяемое следующим образом:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Применение описанного *S-box* ко всем байтам состояния обозначается как

`ByteSub (State)`

На рисунке 6.2 показан результат применения преобразования `ByteSub` к `State`.

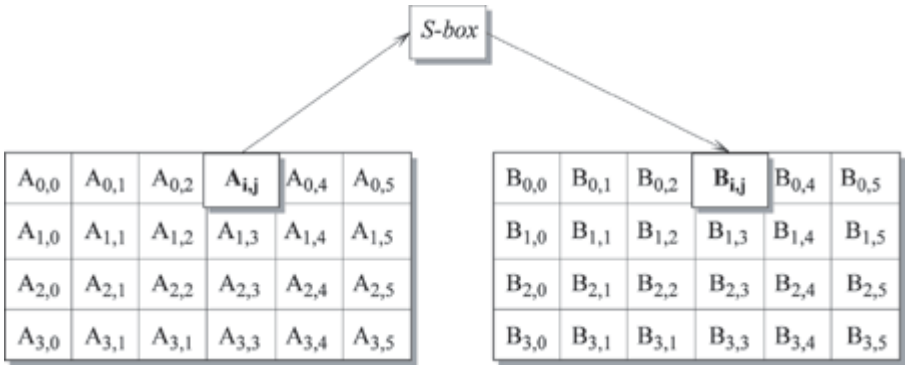

Рис. 6.2. Применение ByteSub для каждого байта в State

Инверсия `ByteSub` есть применение байтовой подстановки в соответствии с инверсной таблицей. Это получается инверсией афинного отображения и мультипликативной инверсией в GF (2^8).

Преобразование ShiftRow

В `ShiftRow` строки состояния циклически сдвигаются на различные значения. Нулевая строка не сдвигается. Строка 1 сдвигается на C_1 байтов, строка 2 на C_2 байтов, строка 3 на C_3 байтов. Величины C_1, C_2 и C_3 зависят от `Nb`. Значения приведены в следующей таблице.

Таблица 6.2. Величина сдвига в зависимости от длины блока

Nb	C_1	C_2	C_3

4	1	2	3
6	1	2	3
8	1	3	4

Операция сдвига строк на указанные значения обозначается как

`ShiftRow (State)`

Инверсией для `ShiftRow` является циклический сдвиг трех нижних строк соответственно на $Nb - C_1$, $Nb - C_2$ и $Nb - C_3$ байт, чтобы байт в позиции `j` в строке `i` перемещался в позицию $(j + Nb - C_i) \bmod Nb$.

Преобразование MixColumn

В `MixColumn` столбцы состояния рассматриваются как полиномы в GF (2^8) и умножаются по модулю $x^4 + 1$ на фиксированный полином:

$c(x) = '03'\ x^3 + '01'\ x^2 + '01'\ x + '02'$

Данный полином является взаимнопростым с $x^4 + 1$ и, следовательно, инвертируем. Как было описано выше, это может быть записано в виде умножения матрицы. Пусть $b(x) = c(x) \otimes a(x)$

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Применение данной операции ко всем столбцам состояния обозначается как

`MixColumn (State)`

Инверсия `MixColumn` является аналогичным `MixColumn`. Каждый столбец преобразуется умножением его на полином `d(x)`, определяемый следующим образом:

$('03'\ x^3 + '01'\ x^2 + '01'\ x + '02') \otimes d(x) = '01'$

В результате получаем

$d(x) = '0B'\ x^3 + '0D'\ x^2 + '09'\ x + '0E'$

Сложение с ключом раунда

Выполняется операция побитового `XOR` ключа *раунда* с текущим состоянием. Длина ключа *раунда* равна длине блока `Nb`. Данное преобразование обозначается как

`AddRoundKey (State, RoundKey)`

`AddRoundKey` является инверсией самого себя.

Создание ключей раунда

Ключи *раунда* получаются из *ключа шифрования* с помощью преобразования, состоящего из двух компонентов: расширение ключа и выбор ключа *раунда*. Основной принцип состоит в следующем:

- Общее число битов ключа *раунда* равно длине блока, умноженной на количество *раундов* плюс 1. Например, для длины блока 128 бит и 10 *раундов* необходимо 1408 битов ключа *раунда*.
- *Ключ шифрования* расширяется в `ExpandedKey`.
- Ключи *раунда* получаются из этого `ExpandedKey` следующим способом: первый ключ *раунда* состоит из первых `Nb` слов, второй состоит из следующих `Nb` слов и т.д.

Расширение ключа

`Expanded Key` является линейным массивом четырехбайтных слов и обозначается как `W [Nb * (Nr + 1)]`. Первые `Nk` слов состоят из *ключа шифрования*. Остальные слова определяются рекурсивно. Функция расширения ключа зависит от значения `Nk`: существует версия функции для `Nk`, равным или меньшим 6, и версия для `Nk` больше 6.

Для `Nk` ≤6 мы имеем:

```
KeyExpansion (byte Key [4*Nk]
              word W[Nb * (Nr + 1)])
{
  for (i = 0; i < Nk; i++)
    W[i] =(Key [4*i], Key [4*i+1],
           Key [4*i+2], Key [4*i+3]);
  for (i = Nk; i < Nb * (Nr + 1); i++) {
    temp = W [i - 1];
    if (i % Nk == 0)
      temp = SubByte (RotByte (temp)) ^
             Rcon [i / Nk];
    W [i] = W [i- Nk] ^ temp;
  }
}
```

В данном случае `SubByte (W)` является функцией, которая возвращает четырехбайтное слово, в котором каждый байт является результатом применения *S-box Rijndael* к байту в соответствующей позиции во входном слове. Функция `RotByte (W)` возвращает слово, в котором байты циклически переставлены таким образом, что для входного слова `(a, b, c, d)` создается выходное слово `(b, c, d, a)`.

Можно заметить, что первые `Nk` слов заполняются *ключом шифрования*. Каждое следующее слово `W[i]` равно `XOR` предыдущего слова `W[i-1]` и позиций слова `Nk` до `W[i - Nk]`. Для слов в позициях, которые кратны `Nk`, сначала применяется преобразование `XOR` к `W[i-1]` и константой *раунда*. Данное преобразование состоит из циклического сдвига байтов в слове `RotByte`, за которым следуют применение табличной подстановки для всех четырех байтов в слове `(SubByte)`.

Для `Nk > 6` мы имеем:

```
KeyExpansion (byte Key [4*Nk]
              word W [Nb* (Nr+1)])
{
```

```
  for (i=0; i < Nk; i++)
    W[i]= (key [4*i], key [4*i+1],
           key [4*i+2], key [4*i+3]);
  for (i = Nk; i < Nb * (Nr + 1); i++) {
    temp = W [i-1];
    if (i % Nk == 0)
      temp = SubByte (RotByte (temp)) ^
             Rcon [i / Nk];
    else if (i % Nk == 4)
      temp = SubByte (temp);
    W[i] = W[i - Nk] ^ temp;
  }
}
```

Отличие в схеме для `Nk` ≤6 состоит в том, что для `i-4` кратных `Nk`, `SubByte` применяется для `W[i-1]` перед `XOR`.

Константы *раунда* не зависят от `Nk` и определяются следующим образом:

```
Rcon [i] = (RC [i], '00', '00', '00')
```

`RC [i]` являются элементами в GF (2^8) со значением $x^{(i-1)}$ таким, что:

```
RC [1] = 1 (т.е. '01')
RC [i] = x (т.е. '02') • (RC [i-1]) = x(i-1)
```

Выбор ключа раунда

Ключ *раунда* `i` получается из слов буфера ключа *раунда* `W [Nb * i] до W [Nb * (i+1)]`.

Алгоритм шифрования

Алгоритм шифрования `Rijndael` состоит из

- начального сложения с ключом;
- `Nr - 1` *раундов*;
- заключительного *раунда*.

В С-подобном представлении это выглядит так:

```
Rijndael (State, CipherKey)
{
   KeyExpansion (CipherKey, ExpandedKey);
   AddRoundKey (State, ExpandedKey);
   for (i=1; i < Nr; i++)
      Round (State, ExpandedKey + Nb*i);
   FinalRound (State, ExpandedKey + Nb*Nr)
}
```

Расширение ключа может быть выполнено заранее, и `Rijndael` может быть специфицирован в терминах расширенного ключа.

```
Rijndael (State, ExpandedKey)
{
   AddRoundKey (State, ExpandedKey);
   for (i=1; i < Nr; i++)
      Round (State, ExpandedKey + Nb*i);
   FinalRound (State, ExpandedKey + Nb*Nr)
```

```
}
```

Замечание: расширенный ключ всегда получается из *ключа шифрования* и никогда не специфицируется непосредственно. Тем не менее, на выбор самого *ключа шифрования* ограничений не существует.

Преимущества алгоритма

Преимущества, относящиеся к аспектам реализации:

- *Rijndael* может выполняться быстрее, чем обычный блочный алгоритм шифрования. Выполнена оптимизация между размером таблицы и скоростью выполнения.
- *Rijndael* можно реализовать в смарт-карте в виде кода, используя небольшой RAM и имея небольшое число циклов. Выполнена оптимизация размера ROM и скорости выполнения.
- Преобразование *раунда* допускает параллельное выполнение, что является важным преимуществом для будущих процессоров и специализированной аппаратуры.
- Алгоритм шифрования не использует арифметические операции, поэтому тип архитектуры процессора не имеет значения.

Простота разработки:

- Алгоритм шифрования полностью "самоподдерживаемый". Он не использует других криптографических компонентов, *S-box*'ов, взятых из хорошо известных алгоритмов, битов, полученных из специальных таблиц, чисел типа p и тому подобных уловок.
- Алгоритм не основывает свою безопасность или часть ее на неясностях или плохо понимаемых итерациях арифметических операций.
- Компактная разработка алгоритма не дает возможности спрятать люки.

Переменная длина блока:

- Длины блоков от 192 до 256 бит позволяют создавать хэш-функции без коллизий, использующие *Rijndael* в качестве функции сжатия. Длина блока 128 бит сегодня считается для этой цели недостаточной.

Расширения:

- Разработка позволяет специфицировать варианты длины блока и длины ключа в диапазоне от 128 до 256 бит с шагом в 32 бита.
- Хотя число *раундов Rijndael* зафиксировано в данной спецификации, в случае возникновения проблем с безопасностью он может модифицироваться и иметь число *раундов* в качестве параметра.

Расширения

Различная длина блока и ключа шифрования

Обработка ключа поддерживает длину ключа, которая была бы кратна 4 байтам. Единственным параметром, который необходим для определения другой длины ключа, отличной от 128, 192 или 256 бит, является число *раундов* алгоритма.

Структура алгоритма допускает произвольную длину блока, кратную 4 байтам, с минимумом в 16 байтов. Добавление ключа и `ByteSub` и `MixColumn` преобразования не

зависят от длины блока. Единственным преобразованием, которое зависит от длины блока, является `ShiftRow`. Для каждой длины блока должен быть определен специальный массив C_1, C_2, C_3.

Можно определить расширение *Rijndael*, которое также поддерживает длины блока и ключа между 128 и 256 битами с приращением в 32 бита. Число *раундов* определяется так:

```
Nr = max (Nk, Nb) + 6
```

Это расширяет правило для количества *раундов* для альтернативных длин блока и ключа.

Дополнительные значения C_1, C_2 и C_3 определены в следующей таблице.

Таблица 6.3. Величина сдвига в зависимости от длины блока

Nb	C1	C2	C3
5	1	2	3
7	1	2	4

Другие возможности

Обсудим функции, отличные от шифрования, которые могут выполняться алгоритмом *Rijndael*.

MAC

Rijndael может применяться в качестве алгоритма MAC. Для этого следует использовать блочный алгоритм в режиме CBC-MAC.

Хэш-функция

Rijndael может использоваться в качестве итерационной хэш-функции. При этом *Rijndael* применяется в качестве функции *раунда*. Существует одна возможная реализация. Рекомендуется использовать длину блока и ключа, равной 256 битам. Блок сообщения подается на вход в качестве *ключа шифрования*. Другим входом является переменная, выход алгоритма XORed с данной переменной, и полученное значение является новым значением этой переменной.

Генератор псевдослучайных чисел

Существует много способов, с помощью которых *Rijndael* можно использовать в качестве генератора псевдослучайных чисел. Рассмотрим один из них, в котором применяются длина блока и длина ключа 256 бит.

Существует три операции:

Reset:

- Ключ алгоритма шифрования и состояние устанавливаются в ноль.

Seeding (и reseeding):

- "Seed биты" выбираются таким образом, чтобы обеспечивать минимальную энтропию. Они дополняются нулями до тех пор, пока результирующая строка не будет иметь длину, кратную 256 битам.
- Вычисляется новый *ключ шифрования* шифрованием с помощью *Rijndael* блока битов seed, используя текущий *ключ шифрования*. Это выполняется рекурсивно до тех пор, пока все блоки seed не будут обработаны.
- Состояние изменяется путем применения *Rijndael* с новым *ключом шифрования*.

Генератор псевдослучайного числа:

- Состояние изменяется путем применения *Rijndael* с новым *ключом шифрования*. Первые 128 бит состояния рассматриваются как псевдослучайное число. Данный шаг может быть повторен много раз.

Алгоритм RC6

RC6 является полностью параметризуемым семейством алгоритмов шифрования. RC6 правильнее указывать как RC6-w/r/b, где `w` - длина слова в битах, `r` - число *раундов*, `b` - длина ключа. Обычно используются значения `w = 32` и `r = 20`.

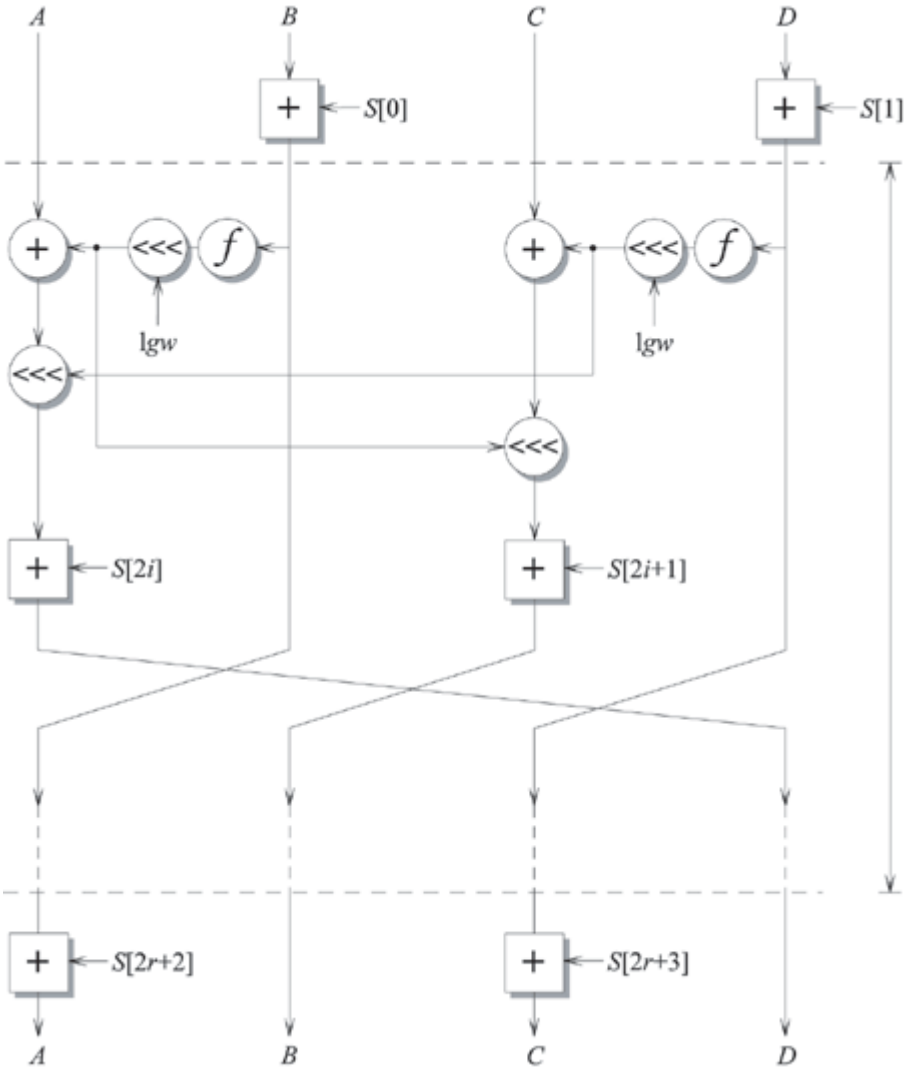

Рис. 6.3. Алгоритм RC6

Алгоритм является *сетью Фейштеля* с 4 ветвями смешанного типа: два четных подблока используются для одновременного изменения содержимого двух нечетных подблоков.

Затем производится обычный для *сети Фейштеля* сдвиг на одно слово, что меняет четные и нечетные подблоки местами.

$f(x) = x(2x + 1)$

$a + b$ - сложение целых по модулю 2^w

$a - b$ - вычитание целых по модулю 2^w

$a \oplus b$ - XOR w-битных слов

$a \times b$ - умножение целых по модулю 2^w

$a <<< b$ - ротация влево на b бит w-битного слова a

$a >>> b$ - ротация вправо на b бит w-битного слова a

$S[0, \ldots, 2r + 3]$ - w-битные подключи *раунда*

7. Лекция: Криптография с открытым ключом

Основные требования к алгоритмам асимметричного шифрования

Создание *алгоритмов асимметричного шифрования* является величайшим и, возможно, единственным революционным достижением в истории криптографии.

Алгоритмы шифрования с *открытым ключом* разрабатывались для того, чтобы решить две наиболее трудные задачи, возникшие при использовании симметричного шифрования.

Первой задачей является распределение ключа. При симметричном шифровании требуется, чтобы обе стороны уже имели общий ключ, который каким-то образом должен быть им заранее передан. Диффи, один из основоположников шифрования с *открытым ключом*, заметил, что это требование отрицает всю суть криптографии, а именно возможность поддерживать всеобщую секретность при коммуникациях.

Второй задачей является необходимость создания таких механизмов, при использовании которых невозможно было бы подменить кого-либо из участников, т.е. нужна *цифровая подпись*. При использовании коммуникаций для решения широкого круга задач, например в коммерческих и частных целях, электронные сообщения и документы должны иметь эквивалент подписи, содержащейся в бумажных документах. Необходимо создать метод, при использовании которого все участники будут убеждены, что электронное сообщение было послано конкретным участником. Это более сильное требование, чем аутентификация.

Диффи и Хеллман достигли значительных результатов, предложив способ решения обеих задач, который радикально отличается от всех предыдущих подходов к шифрованию.

Сначала рассмотрим общие черты алгоритмов шифрования с *открытым ключом* и требования к этим алгоритмам. Определим требования, которым должен соответствовать алгоритм, использующий один ключ для шифрования, другой ключ - для дешифрования, и при этом вычислительно невозможно определить дешифрующий ключ, зная только алгоритм шифрования и шифрующий ключ.

Кроме того, некоторые алгоритмы, например RSA, имеют следующую характеристику: каждый из двух ключей может использоваться как для шифрования, так и для дешифрования.

Сначала рассмотрим алгоритмы, обладающие обеими характеристиками, а затем перейдем к алгоритмам *открытого ключа*, которые не обладают вторым свойством.

При описании симметричного шифрования и шифрования с *открытым ключом* будем использовать следующую терминологию. Ключ, используемый в симметричном шифровании, будем называть секретным ключом. Два ключа, используемые при шифровании с *открытым ключом*, будем называть ***открытым ключом*** и ***закрытым ключом***. *Закрытый ключ* держится в секрете, но называть его будем *закрытым ключом*, а не секретным, чтобы избежать путаницы с ключом, используемым в симметричном шифровании. *Закрытый ключ* будем обозначать KR, *открытый ключ* - KU.

Будем предполагать, что все участники имеют доступ к *открытым ключам* друг друга, а *закрытые ключи* создаются локально каждым участником и, следовательно, распределяться не должны.

В любое время участник может изменить свой *закрытый ключ* и опубликовать составляющий пару *открытый ключ*, заменив им старый *открытый ключ*.

Диффи и Хеллман описывают требования, которым должен удовлетворять алгоритм шифрования с *открытым ключом*.

1. Вычислительно легко создавать пару (*открытый ключ* KU , *закрытый ключ* KR).
2. Вычислительно легко, имея *открытый ключ* и незашифрованное сообщение M, создать соответствующее зашифрованное сообщение:

   $$C = E_{KU}[M]$$

3. Вычислительно легко дешифровать сообщение, используя *закрытый ключ*:

   $$M = D_{KR}[C] = D_{KR}[E_{KU}[M]]$$

4. Вычислительно невозможно, зная *открытый ключ* KU, определить *закрытый ключ* KR.
5. Вычислительно невозможно, зная *открытый ключ* KU и зашифрованное сообщение C, восстановить исходное сообщение M.

   Можно добавить шестое требование, хотя оно не выполняется для всех алгоритмов с *открытым ключом*:

6. Шифрующие и дешифрующие функции могут применяться в любом порядке:

   $$M = E_{KU}[D_{KR}[M]]$$

Это достаточно сильные требования, которые вводят понятие *односторонней функции с люком*. ***Односторонней функцией*** называется такая функция, у которой каждый аргумент имеет единственное обратное значение, при этом вычислить саму функцию легко, а вычислить обратную функцию трудно.

$Y = f(X)$ - легко

$X = f^{-1}(Y)$ - трудно

Обычно "легко" означает, что проблема может быть решена за полиномиальное время от длины входа. Таким образом, если длина входа имеет n битов, то время вычисления функции пропорционально n^a, где a - фиксированная константа. Таким образом, говорят, что алгоритм принадлежит классу полиномиальных алгоритмов P. Термин "трудно" означает более сложное понятие. В общем случае будем считать, что

проблему решить невозможно, если усилия для ее решения больше полиномиального времени от величины входа. Например, если длина входа n битов, и время вычисления функции пропорционально 2^n, то это считается вычислительно невозможной задачей. К сожалению, тяжело определить, проявляет ли конкретный алгоритм такую сложность. Более того, традиционные представления о вычислительной сложности фокусируются на худшем случае или на среднем случае сложности алгоритма. Это неприемлемо для криптографии, где требуется невозможность инвертировать функцию для всех или почти всех значений входов.

Вернемся к определению ***односторонней функции с люком***, которую, подобно *односторонней функции*, легко вычислить в одном направлении и трудно вычислить в обратном направлении до тех пор, пока недоступна некоторая дополнительная информация. При наличии этой дополнительной информации инверсию можно вычислить за полиномиальное время. Таким образом, *односторонняя функция* с люком принадлежит семейству *односторонних функций* f_k таких, что

$Y = f_k(X)$ - легко, если k и X известны

$X = f_k^{-1}(Y)$ - легко, если k и Y известны

$X = f_k^{-1}(Y)$ - трудно, если Y известно, но k неизвестно

Мы видим, что разработка конкретного алгоритма с *открытым ключом* зависит от открытия соответствующей односторонней функции с люком.

Криптоанализ алгоритмов с открытым ключом

Как и в случае симметричного шифрования, алгоритм шифрования с *открытым ключом* уязвим для лобовой атаки. Контрмера стандартная: использовать большие ключи.

Криптосистема с *открытым ключом* применяет определенные неинвертируемые математические функции. Сложность вычислений таких функций не является линейной от количества битов ключа, а возрастает быстрее, чем ключ. Таким образом, размер ключа должен быть достаточно большим, чтобы сделать лобовую атаку непрактичной, и достаточно маленьким для возможности практического шифрования. На практике размер ключа делают таким, чтобы лобовая атака была непрактичной, но в результате скорость шифрования оказывается достаточно медленной для использования алгоритма в общих целях. Поэтому шифрование с *открытым ключом* в настоящее время в основном ограничивается приложениями управления ключом и подписи, в которых требуется шифрование небольшого блока данных.

Другая форма атаки состоит в том, чтобы найти способ вычисления *закрытого ключа*, зная *открытый ключ*. Невозможно математически доказать, что данная форма атаки исключена для конкретного алгоритма *открытого ключа*. Таким образом, любой алгоритм, включая широко используемый *алгоритм RSA*, является подозрительным.

Наконец, существует форма атаки, специфичная для способов использования систем с *открытым ключом*. Это атака вероятного сообщения. Предположим, например, что посылаемое сообщение состоит исключительно из 56-битного ключа сессии для алгоритма симметричного шифрования. Противник может зашифровать все возможные ключи, используя *открытый ключ*, и может дешифровать любое сообщение, соответствующее передаваемому зашифрованному тексту. Таким образом, независимо от размера ключа схемы *открытого ключа*, атака сводится к лобовой атаке на 56-битный симметричный ключ. Защита от подобной атаки состоит в добавлении определенного количества случайных битов в простые сообщения.

Основные способы использования алгоритмов с открытым ключом

Основными способами использования алгоритмов с *открытым ключом* являются шифрование/дешифрование, создание и проверка подписи и обмен ключа.

Шифрование с *открытым ключом* состоит из следующих шагов:

Рис. 7.1. Шифрование с открытым ключом

1. Пользователь B создает пару ключей KU_b и KR_b, используемых для шифрования и дешифрования передаваемых сообщений.
2. Пользователь B делает доступным некоторым надежным способом свой ключ шифрования, т.е. *открытый ключ* KU_b. Составляющий пару *закрытый ключ* KR_b держится в секрете.
3. Если A хочет послать сообщение B, он шифрует сообщение, используя *открытый ключ* B KU_b .
4. Когда B получает сообщение, он дешифрует его, используя свой *закрытый ключ* KR_b. Никто другой не сможет дешифровать сообщение, так как этот *закрытый ключ* знает только B.

Если пользователь (конечная система) надежно хранит свой *закрытый ключ*, никто не сможет подсмотреть передаваемые сообщения.

Создание и проверка подписи состоит из следующих шагов:

Рис. 7.2. Создание и проверка подписи

1. Пользователь A создает пару ключей KR_A и KU_A, используемых для создания и проверки подписи передаваемых сообщений.

2. Пользователь A делает доступным некоторым надежным способом свой ключ проверки, т.е. *открытый ключ* KU_A. Составляющий пару *закрытый ключ* KR_A держится в секрете.
3. Если A хочет послать подписанное сообщение B, он создает подпись $E_{KRa}[M]$ для этого сообщения, используя свой *закрытый ключ* KR_A.
4. Когда B получает подписанное сообщение, он проверяет подпись $D_{KUa}[M]$, используя *открытый ключ* A KU_A. Никто другой не может подписать сообщение, так как этот *закрытый ключ* знает только A.

До тех пор, пока пользователь или прикладная система надежно хранит свой *закрытый ключ*, их подписи достоверны.

Кроме того, невозможно изменить сообщение, не имея доступа к *закрытому ключу* A; тем самым обеспечивается аутентификация и целостность данных.

В этой схеме все сообщение подписывается, причем для подтверждения целостности сообщения требуется много памяти. Каждое сообщение должно храниться в незашифрованном виде для использования в практических целях. Кроме того, копия сообщения также должна храниться в зашифрованном виде, чтобы можно было проверить в случае необходимости подпись. Более эффективным способом является шифрование небольшого блока битов, который является функцией от сообщения. Такой блок, называемый аутентификатором, должен обладать свойством невозможности изменения сообщения без изменения аутентификатора. Если аутентификатор зашифрован *закрытым ключом* отправителя, он является ***цифровой подписью***, с помощью которой можно проверить исходное сообщение. Далее эта технология будет рассматриваться в деталях.

Важно подчеркнуть, что описанный процесс создания подписи не обеспечивает конфиденциальность. Это означает, что сообщение, посланное таким способом, невозможно изменить, но можно подсмотреть. Это очевидно в том случае, если подпись основана на аутентификаторе, так как само сообщение передается в явном виде. Но даже если осуществляется шифрование всего сообщения, конфиденциальность не обеспечивается, так как любой может расшифровать сообщение, используя *открытый ключ* отправителя.

Обмен ключей: две стороны взаимодействуют для обмена ключом сессии, который в дальнейшем можно использовать в алгоритме симметричного шифрования.

Некоторые алгоритмы можно задействовать тремя способами, в то время как другие могут использоваться одним или двумя способами.

Перечислим наиболее популярные алгоритмы с *открытым ключом* и возможные способы их применения.

Алгоритм	Шифрование / дешифрование	***Цифровая подпись***	Обмен ключей
RSA	Да; непригоден для больших блоков	Да	Да
DSS	Нет	Да	Нет
Диффи-Хеллман	Нет	Нет	Да

Алгоритм RSA

Диффи и Хеллман определили новый подход к шифрованию, что вызвало к жизни разработку алгоритмов шифрования, удовлетворяющих требованиям систем с *открытым ключом*. Одним из первых результатов был алгоритм, разработанный в 1977 году Роном Ривестом, Ади Шамиром и Леном Адлеманом и опубликованный в 1978 году. С тех пор алгоритм Rivest-Shamir-Adleman (*RSA*) широко применяется практически во всех приложениях, использующих криптографию с *открытым ключом*.

Алгоритм основан на использовании того факта, что задача ***факторизации*** является трудной, т.е. легко перемножить два числа, в то время как не существует полиномиального алгоритма нахождения простых сомножителей большого числа.

Алгоритм RSA представляет собой блочный алгоритм шифрования, где зашифрованные и незашифрованные данные являются целыми между 0 и $n-1$ для некоторого n.

Описание алгоритма

Алгоритм, разработанный Ривестом, Шамиром и Адлеманом, использует выражения с экспонентами. Данные шифруются блоками, каждый блок рассматривается как число, меньшее некоторого числа n. Шифрование и дешифрование имеют следующий вид для некоторого незашифрованного блока M и зашифрованного блока C.

$$C = M^e \pmod n$$
$$M = C^d \pmod n = (M^e)^d \pmod n = M^{ed} \pmod n$$

Как отправитель, так и получатель должны знать значение n. Отправитель знает значение e, получатель знает значение d. Таким образом, *открытый ключ* есть $KU = \{e, n\}$ и *закрытый ключ* есть $KR = \{d, n\}$. При этом должны выполняться следующие условия:

1. Возможность найти значения e, d и n такие, что $M^{ed} = M \bmod n$ для всех $M < n$.
2. Относительная легкость вычисления M^e и C^d для всех значений $M < n$.
3. Невозможность определить d, зная e и n.

Сначала рассмотрим первое условие. Нам необходимо выполнение равенства:

$$M^{ed} = M \pmod n$$

Рассмотрим некоторые математические понятия, свойства и теоремы, которые позволят нам определить e, d и n.

1. Если $(a \cdot b) \equiv (a \cdot c) \bmod n$, то $b \equiv c \bmod n$, если a и n взаимнопростые, т.е $\gcd(a, n) = 1$.
2. Обозначим Z_p - все числа, взаимнопростые с p и меньшие p. Если p - простое, то Z_p - это все остатки. Обозначим w^{-1} такое число, что $w \cdot w^{-1} \equiv 1 \bmod p$.

   Тогда $\forall w \in Z_p \ \exists z : w \cdot z \equiv 1 \bmod p$

   Доказательство этого следует из того, что т.к. w и p взаимнопростые, то при умножении всех элементов Z_p на w остатками будут все элементы Z_p, возможно, переставленные. Таким образом, хотя бы один остаток будет равен 1.

3. Определим функцию Эйлера следующим образом: $\Phi(n)$ - число положительных чисел, меньших n и взаимнопростых с n. Если p - простое, то $\Phi(p) = p-1$.

Если p и q - простые, то $\Phi(p \cdot q) = (p-1) \cdot (q-1)$.

В этом случае $Z_{p \cdot q} = \{0, 1, j, (p \cdot q - 1)\}$.

Перечислим остатки, которые не являются взаимнопростыми с $p \cdot q$:

$\{p, 2 \cdot p, j, (q-1) \cdot p\}$
$\{q, 2 \cdot q, j, (p-1) \cdot q\}$
0

Таким образом $\Phi(p \cdot q) = p \cdot q - [(q-1) + (p-1) + 1] = p \cdot q - (p+q) + 1 = (p-1) \cdot (q-1)$.

4. ***Теорема Ферма***.

$a^{n-1} \equiv 1 \mod n$, если n - простое.

Если все элементы Z_n умножить на a по модулю n, то в результате получим все элементы Z_n, быть может, в другом порядке. Рассмотрим следующие числа:

$\{a \mod n, 2 \cdot a \mod n, j, (n-1) \cdot a \mod n\}$ являются числами $\{1, 2, j, (n-1)\}$, быть может, в некотором другом порядке. Теперь перемножим по модулю n числа из этих двух множеств.

$[(a \mod n) \cdot (2a \mod n) \cdot \ldots \cdot (n-1)a \mod n]$
$\mod n \equiv (n-1)! \mod n (n-1)! \; a^{n-1} \equiv (n-1)! \mod n$

n и $(n-1)!$ являются взаимнопростыми, если n - простое, следовательно, $a^{n-1} \equiv 1 \mod n$.

5. ***Теорема Эйлера***.

$a^{\Phi(n)} \equiv 1 \mod n$ для всех взаимнопростых a и n.

Это верно, если n - простое, т.к. в этом случае $\Phi(n) = n-1$. Рассмотрим множество $R = \{x_1, x_2, j, x\Phi(n)\}$. Теперь умножим по модулю n каждый элемент этого множества на a. Получим множество $S = \{a \cdot x_1 \mod n, a \cdot x_2 \mod n, j, a \cdot x\Phi(n) \mod n\}$. Это множество является перестановкой множества R по следующим причинам.

Так как a является взаимнопростым с n и x_i являются взаимнопростыми с n, то $a \cdot x_i$ также являются взаимнопростыми с n. Таким образом, S - это множество целых, меньших n и взаимнопростых с n.

В S нет дублей, т.к. если $a \cdot x_i \mod n = a \cdot x_j \mod n \Rightarrow x_i = x_j$.

Следовательно, перемножив элементы множеств S и R, получим:

$$\prod_{i=1}^{\Phi(n)} (a \cdot x_i \mod n) \equiv \prod_{i=1}^{\Phi(n)} x_i \mod n$$

$$\left(\prod_{i=1}^{\Phi(n)} a \cdot x_i \equiv \prod_{i=1}^{\Phi(n)} x_i \right) \mod n$$

$$\left(a^{\Phi(n)} \cdot \prod_{i=1}^{\Phi(n)} x_i \equiv \prod_{i=1}^{\Phi(n)} x_i \right) \bmod n$$

$$\left(a^{\Phi(n)} \equiv 1 \right) \bmod n$$

Теперь рассмотрим сам *алгоритм RSA*. Пусть p и q - простые.

$n = p \cdot q$.

Надо доказать, что $\forall M < n: M^{\Phi(n)} = M^{(p-1) \cdot (q-1)} \equiv 1 \bmod n$

Если $\gcd(M, n) = 1$, то соотношение выполняется. Теперь предположим, что $\gcd(M, n) \neq 1$, т.е. $\gcd(M, p \cdot q) \neq 1$. Пусть $\gcd(M, p) \neq 1$, т.е. $M = c \cdot p \Rightarrow \gcd(M, q) = 1$, так как в противном случае $M = c \cdot p$ и $M = l \cdot q$, но по условию $M < p \cdot q$.

Следовательно,

$$M^{\Phi(q)} \equiv 1 \bmod q$$

$$(M^{\Phi(q)})(p) \equiv 1 \bmod q$$

$$M^{\Phi(n)} \equiv 1 \bmod q$$

По определению модуля это означает, что $M^{\Phi(n)} = 1 + k \cdot q$. Умножим обе части равенства на $M = c \cdot p$. Получим

$$M^{\Phi(n)+1} = c \cdot p + k \cdot q \cdot c \cdot p.$$

$$M^{\Phi(n)} \equiv 1 \bmod n$$

Или

$$M^{\Phi(n)+1} \equiv M \bmod n$$

Таким образом, следует выбрать e и d такие, что $e \cdot d \equiv 1 \bmod (n)$

Или $e \equiv d^{-1} \bmod \Phi(n)$

e и d являются взаимнообратными по умножению по модулю $\Phi(n)$. Заметим, что в соответствии с правилами модульной арифметики, это верно только в том случае, если d (и следовательно, e) являются взаимнопростыми с $\Phi(n)$. Таким образом, $\gcd(\Phi(n), d) = 1$.

Теперь рассмотрим все элементы *алгоритма RSA*.

p, q - два простых целых числа	- открыто, вычисляемо.
$n = p \cdot q$	- закрыто, вычисляемо.
$d, \gcd(\Phi(n), d) = 1;$ $1 < d < \Phi(n)$	- открыто, выбираемо.
$e \equiv d^{-1} \bmod \Phi(n)$	- закрыты, выбираемы.

Закрытый ключ состоит из $\{d, n\}$, *открытый ключ* состоит из $\{e, n\}$. Предположим, что пользователь A опубликовал свой *открытый ключ*, и что пользователь B хочет послать пользователю A сообщение M. Тогда B вычисляет $C = M^e \pmod n$ и передает C. При

получении этого зашифрованного текста пользователь A дешифрует вычислением $M = C^d \pmod n$.

Суммируем *алгоритм RSA*:

Создание ключей

Выбрать простые p и q

Вычислить $n = p \cdot q$

Выбрать d $\quad \gcd(\Phi(n), d) = 1;\ 1 < d < \Phi(n)$

Вычислить e $\quad e = d^{-1} \bmod \Phi(n)$

Открытый ключ $KU = \{e, n\}$

Закрытый ключ $KR = \{d, n\}$

Шифрование

Незашифрованный текст: $M < n$

Зашифрованный текст: $C = M^e \pmod n$

Дешифрование

Зашифрованный текст: C

Незашифрованный текст: $M = C^d \pmod n$

Рассмотрим конкретный пример:

Выбрать два простых числа: $p = 7,\ q = 17$.

Вычислить $n = p \cdot q = 7 \cdot 17 = 119$.

Вычислить $\Phi(n) = (p - 1) \cdot (q - 1) = 96$.

Выбрать e так, чтобы e было взаимнопростым с $\Phi(n) = 96$ и меньше, чем $\Phi(n)$: $e = 5$.

Определить d так, чтобы $d \cdot e \equiv 1 \bmod 96$ и $d < 96$.

$d = 77$, так как $77 \cdot 5 = 385 = 4 \cdot 96 + 1$.

Результирующие ключи *открытый* $KU = \{5, 119\}$ и *закрытый* $KR = \{77, 119\}$.

Например, требуется зашифровать сообщение $M = 19$.

$19^5 = 66 \pmod{119};\ C = 66$.

Для дешифрования вычисляется $66^{77} \pmod{119} = 19$.

Вычислительные аспекты

Рассмотрим сложность вычислений в *алгоритме RSA* при создании ключей и при шифровании/дешифровании.

Шифрование/дешифрование

Как шифрование, так и дешифрование включают возведение целого числа в целую степень по модулю n. При этом промежуточные значения будут громадными. Для того, чтобы частично этого избежать, используется следующее свойство модульной арифметики:

$$[(a \bmod n) \cdot (b \bmod n)] \bmod n = (a \cdot b) \bmod n$$

Другая оптимизация состоит в эффективном использовании показателя степени, так как в случае *RSA* показатели степени очень большие. Предположим, что необходимо вычислить x^{16}. Прямой подход требует 15 умножений. Однако можно добиться того же

конечного результата с помощью только четырех умножений, если использовать квадрат каждого промежуточного результата: x^2, x^4, x^8, x^{16}.

Создание ключей

Создание ключей включает следующие задачи:

1. Определить два простых числа p и q.
2. Выбрать e и вычислить d.

Прежде всего, рассмотрим проблемы, связанные с выбором p и q. Так как значение $n = p \cdot q$ будет известно любому потенциальному противнику, для предотвращения раскрытия p и q эти простые числа должны быть выбраны из достаточно большого множества, т.е. p и q должны быть большими числами. С другой стороны, метод, используемый для поиска большого простого числа, должен быть достаточно эффективным.

В настоящее время неизвестны алгоритмы, которые создают произвольно большие простые числа. Процедура, которая используется для этого, выбирает случайное нечетное число из требуемого диапазона и проверяет, является ли оно простым. Если число не является простым, то опять выбирается случайное число до тех пор, пока не будет найдено простое.

Были разработаны различные тесты для определения того, является ли число простым. Это тесты вероятностные, то есть тест показывает, что данное число вероятно является простым. Несмотря на это они могут выполняться таким образом, что сделают вероятность близкой к 1. Если n "проваливает" тест, то оно не является простым. Если n "пропускает" тест, то n может как быть, так и не быть простым. Если n пропускает много таких тестов, то можно с высокой степенью достоверности сказать, что n является простым. Это достаточно долгая процедура, но она выполняется относительно редко: только при создании новой пары (KU, KR).

На сложность вычислений также влияет то, какое количество чисел будет отвергнуто перед тем, как будет найдено простое число. Результат из теории чисел, известный как теорема простого числа, говорит, что простых чисел, расположенных около n в среднем одно на каждые $\ln(n)$ чисел. Таким образом, в среднем требуется проверить последовательность из $\ln(n)$ целых, прежде чем будет найдено простое число. Так как все четные числа могут быть отвергнуты без проверки, то требуется выполнить приблизительно $\ln(n)/2$ проверок. Например, если простое число ищется в диапазоне величин 2^{200}, то необходимо выполнить около $\ln(2^{200}) / 2 = 70$ проверок.

Выбрав простые числа p и q, далее следует выбрать значение e так, чтобы $\gcd(\Phi(n), e) = 1$ и вычислить значение d, $d = e^{-1} \bmod \Phi(n)$. Существует единственный алгоритм, называемый расширенным алгоритмом Евклида, который за фиксированное время вычисляет наибольший общий делитель двух целых и если этот общий делитель равен единице, определяет инверсное значение одного по модулю другого. Таким образом, процедура состоит в генерации серии случайных чисел и проверке каждого относительно $\Phi(n)$ до тех пор, пока не будет найдено число, взаимнопростое с $\Phi(n)$. Возникает вопрос, как много случайных чисел придется проверить до тех пор, пока не найдется нужное число, которое будет взаимнопростым с $\Phi(n)$. Результаты показывают, что вероятность того, что два случайных числа являются взаимнопростыми, равна 0.6.

Обсуждение криптоанализа

Можно определить четыре возможных подхода для криптоанализа *алгоритма RSA*:

1. Лобовая атака: перебрать все возможные *закрытые ключи*.
2. Разложить n на два простых сомножителя. Это даст возможность вычислить $\Phi(n) = (p-1) \cdot (q-1)$ и $d = e^{-1} \pmod{\Phi(n)}$.
3. Определить $\Phi(n)$ непосредственно, без начального определения p и q. Это также даст возможность определить $d = e^{-1} \pmod{\Phi(n)}$.
4. Определить d непосредственно, без начального определения $\Phi(n)$.

Защита от лобовой атаки для *RSA* и ему подобных алгоритмов состоит в использовании большой длины ключа. Таким образом, чем больше битов в e и d, тем лучше. Однако, так как вычисления необходимы как при создании ключей, так и при шифровании/дешифровании, чем больше размер ключа, тем медленнее работает система.

Большинство дискуссий о криптоанализе *RSA* фокусируется на задаче разложения n на два простых сомножителя. В настоящее время неизвестны алгоритмы, с помощью которых можно было бы разложить число на два простых множителя для очень больших чисел (т.е. несколько сотен десятичных цифр). Лучший из известных алгоритмов дает результат, пропорциональный:

$$L(n) = e^{\sqrt{\ln n * \ln(\ln n)}}$$

Пока не разработаны лучшие алгоритмы разложения числа на простые множители, можно считать, что величина n от 100 до 200 цифр в настоящее время является достаточно безопасной. На современном этапе считается, что число из 100 цифр может быть разложено на множители за время порядка двух недель. Для дорогих конфигураций (т.е. порядка $10 млн) число из 150 цифр может быть разложено приблизительно за год. Разложение числа из 200 цифр находится за пределами вычислительных возможностей. Например, даже если вычислительный уровень в 10^{12} операций в секунду достижим, что выше возможностей современных технологий, то потребуется свыше 10 лет для разложения на множители числа из 200 цифр с использованием существующих алгоритмов.

Для известных в настоящее время алгоритмов задача определения (n) по данным e и n, по крайней мере, сопоставима по времени с задачей разложения числа на множители.

Для того чтобы избежать выбора значения n, которое могло бы легко раскладываться на сомножители, на p и q должно быть наложено много дополнительных ограничений: p и q должны друг от друга отличаться по длине только несколькими цифрами. Таким образом, оба значения p и q должны быть от 10^{75} до 10^{100}.

Оба числа $(p - 1)$ и $(q - 1)$ должны содержать большой простой сомножитель.

$\gcd(p - 1, q - 1)$ должен быть маленьким.

Алгоритм обмена ключа Диффи-Хеллмана

Первая публикация данного алгоритма *открытого ключа* появилась в статье Диффи и Хеллмана, в которой вводились основные понятия криптографии с *открытым ключом* и в общих чертах упоминался алгоритм обмена ключа *Диффи-Хеллмана*.

Цель алгоритма состоит в том, чтобы два участника могли безопасно обменяться ключом, который в дальнейшем может использоваться в каком-либо алгоритме симметричного шифрования. Сам ***алгоритм Диффи-Хеллмана*** может применяться только для обмена ключами.

Алгоритм основан на трудности вычислений *дискретных логарифмов*. ***Дискретный логарифм*** определяется следующим образом. Вводится понятие ***примитивного корня простого числа*** Q как числа, чьи степени создают все целые от 1 до $Q - 1$. Это означает, что если A является *примитивным корнем простого числа* Q, тогда числа

$$A \bmod Q,\ A^2 \bmod Q,\ \ldots,\ A^{Q-1} \bmod Q$$

являются различными и состоят из целых от 1 до $Q - 1$ с некоторыми перестановками. В этом случае для любого целого $Y < Q$ и *примитивного корня* A простого числа Q можно найти единственную экспоненту X, такую, что

$$Y = A^X \bmod Q, \text{ где } 0 \leq X \leq (Q - 1)$$

Экспонента X называется ***дискретным логарифмом***, или индексом Y, по основанию $A \bmod Q$. Это обозначается как

$$\text{ind}_{A,\,Q}(Y).$$

Теперь опишем алгоритм обмена ключей *Диффи-Хеллмана*.

Общеизвестные элементы	
Q	простое число
A	$A < Q$ и A является *примитивным корнем* Q
Создание пары ключей пользователем I	
Выбор случайного числа X_i (*закрытый ключ*)	$X_i < Q$
Вычисление числа Y_i (*открытый ключ*)	$Yi = A^{Xi} \bmod Q$
Создание *открытого ключа* пользователем J	
Выбор случайного числа X_j (*закрытый ключ*)	$X_j < Q$
Вычисление случайного числа Y_j (*открытый ключ*)	$Yj = A^{Xj} \bmod Q$
Создание общего секретного ключа пользователем I	
$K = (Y_j)^{Xi} \bmod Q$	
Создание общего секретного ключа пользователем J	
$K = (Y_i)^{Xj} \bmod Q$	

Предполагается, что существуют два известных всем числа: простое число Q и целое A, которое является *примитивным корнем* Q. Теперь предположим, что пользователи I и J хотят обменяться ключом для алгоритма симметричного шифрования. Пользователь I выбирает случайное число $X_i < Q$ и вычисляет $Y_i = A^{Xi} \bmod Q$. Аналогично пользователь J независимо выбирает случайное целое число $X_j < Q$ и вычисляет $Y_j = A^{Xj} \bmod Q$. Каждая сторона держит значение X в секрете и делает значение Y доступным для другой стороны. Теперь пользователь I вычисляет ключ как $K = (Y_j)^{Xi} \bmod Q$, и пользователь J вычисляет ключ как $K = (Y_i)^{Xj} \bmod Q$. В результате оба получат одно и то же значение:

$$\begin{aligned}
K &= (Y_j)^{Xi} \bmod Q \\
&= (A^{Xj} \bmod Q)^{Xi} \bmod Q \\
&= (A^{Xj})^{Xi} \bmod Q \\
&\text{по правилам модульной арифметики} \\
&= A^{Xj\,Xi} \bmod Q \\
&= (A^{Xj})Xj \bmod Q \\
&= (A^{Xi} \bmod Q)^{Xj} \bmod Q \\
&= (Y_i)^{Xj} \bmod Q
\end{aligned}$$

Таким образом, две стороны обменялись секретным ключом. Так как X_i и X_j являются закрытыми, противник может получить только следующие значения: Q, A, Y_i и Y_j. Для вычисления ключа атакующий должен взломать *дискретный логарифм*, т.е. вычислить

$$X_j = ind_{a,\ q}\ (Y_j)$$

Безопасность обмена ключа в *алгоритме Диффи-Хеллмана* вытекает из того факта, что, хотя относительно легко вычислить экспоненты по модулю простого числа, очень трудно вычислить *дискретные логарифмы*. Для больших простых чисел задача считается неразрешимой.

Следует заметить, что данный алгоритм уязвим для атак типа "man-in-the-middle". Если противник может осуществить активную атаку, т.е. имеет возможность не только перехватывать сообщения, но и заменять их другими, он может перехватить *открытые ключи* участников Y_i и Y_j , создать свою пару *открытого* и *закрытого ключа* $(X_{оп},\ Y_{оп})$ и послать каждому из участников свой *открытый ключ*. После этого каждый участник вычислит ключ, который будет общим с противником, а не с другим участником. Если нет контроля целостности, то участники не смогут обнаружить подобную подмену.

8. Лекция: Хэш-функции и аутентификация сообщений. Часть 1

Хэш-функции

Требования к хэш-функциям

Хэш-функцией называется односторонняя функция, предназначенная для получения *дайджеста* или "отпечатков пальцев" файла, сообщения или некоторого блока данных.

Хэш-код создается функцией H:

$$h = H\ (M)$$

Где M является сообщением произвольной длины и h является *хэш-кодом* фиксированной длины.

Рассмотрим требования, которым должна соответствовать *хэш-функция* для того, чтобы она могла использоваться в качестве аутентификатора сообщения. Рассмотрим очень простой пример *хэш-функции*. Затем проанализируем несколько подходов к построению *хэш-функции*.

Хэш-функция H, которая используется для аутентификации сообщений, должна обладать следующими свойствами:

1. *Хэш-функция* H должна применяться к блоку данных любой длины.
2. *Хэш-функция* H создает выход фиксированной длины.
3. $H\ (M)$ относительно легко (за полиномиальное время) вычисляется для любого значения M.
4. Для любого данного значения *хэш-кода* h вычислительно невозможно найти M такое, что $H\ (M) = h$.
5. Для любого данного x вычислительно невозможно найти $y \neq x$, что $H\ (y) = H\ (x)$.
6. Вычислительно невозможно найти произвольную пару $(x,\ y)$ такую, что $H\ (y) = H\ (x)$.

Первые три свойства требуют, чтобы *хэш-функция* создавала *хэш-код* для любого сообщения.

Четвертое свойство определяет требование односторонности *хэш-функции*: легко создать *хэш-код* по данному сообщению, но невозможно восстановить сообщение по данному *хэш-коду*. Это свойство важно, если аутентификация с использованием *хэш-функции* включает секретное значение. Само секретное значение может не посылаться, тем не менее, если *хэш-функция* не является односторонней, противник может легко раскрыть секретное значение следующим образом. При перехвате передачи атакующий получает сообщение M и *хэш-код* $C = H\ (S_{AB}\ ||\ M)$. Если атакующий может инвертировать *хэш-функцию*, то, следовательно, он может получить $S_{AB}\ ||\ M = H^{-1}\ (C)$. Так как атакующий теперь знает и M и $S_{AB}\ ||\ M$, получить S_{AB} совсем просто.

Пятое свойство гарантирует, что невозможно найти другое сообщение, чье значение *хэш-функции* совпадало бы со значением *хэш-функции* данного сообщения. Это предотвращает подделку аутентификатора при использовании зашифрованного *хэш-кода*. В данном случае противник может читать сообщение и, следовательно, создать его *хэш-код*. Но так как противник не владеет секретным ключом, он не имеет возможности изменить сообщение так, чтобы получатель этого не обнаружил . Если данное свойство не выполняется, атакующий имеет возможность выполнить следующую последовательность действий: перехватить сообщение и его зашифрованный *хэш-код*, вычислить *хэш-код* сообщения, создать альтернативное сообщение с тем же самым *хэш-кодом*, заменить исходное сообщение на поддельное. Поскольку *хэш-коды* этих сообщений совпадают, получатель не обнаружит подмены.

Хэш-функция, которая удовлетворяет первым пяти свойствам, называется ***простой или слабой хэш-функцией***. Если кроме того выполняется шестое свойство, то такая функция называется ***сильной хэш-функцией***. Шестое свойство защищает против класса атак, известных как атака "*день рождения*".

Простые хэш-функции

Все *хэш-функции* выполняются следующим образом. Входное значение (сообщение, файл и т.п.) рассматривается как последовательность n-битных блоков. Входное значение обрабатывается последовательно блок за блоком, и создается m-битное значение *хэш-кода*.

Одним из простейших примеров *хэш-функции* является побитный XOR каждого блока:

$$C_i = b_{i1} \oplus b_{i2} \oplus \ldots \oplus b_{ik}$$

Где

C_i - i-ый бит *хэш-кода*, $1 \leq i \leq n$.

k - число n-битных блоков входа.

b_{ij} - i-ый бит в j-ом блоке.

$\oplus$- операция XOR.

В результате получается *хэш-код* длины n, известный как продольный избыточный контроль. Это эффективно при случайных сбоях для проверки целостности данных.

Часто при использовании подобного продольного избыточного контроля для каждого блока выполняется однобитный циклический сдвиг после вычисления *хэш-кода*. Это можно описать следующим образом.

- Установить n-битный *хэш-код* в ноль.
- Для каждого n-битного блока данных выполнить следующие операции:
    - сдвинуть циклически текущий *хэш-код* влево на один бит;
    - выполнить операцию XOR для очередного блока и *хэш-кода*.

Это даст эффект "случайности" входа и уничтожит любую регулярность, которая присутствует во входных значениях.

Хотя второй вариант считается более предпочтительным для обеспечения целостности данных и предохранения от случайных сбоев, он не может использоваться для обнаружения преднамеренных модификаций передаваемых сообщений. Зная сообщение, атакующий легко может создать новое сообщение, которое имеет тот же самый *хэш-код*. Для этого следует подготовить альтернативное сообщение и затем присоединить n-битный блок, который является *хэш-кодом* нового сообщения, и блок, который является *хэш-кодом* старого сообщения.

Хотя простого XOR или ротационного XOR (RXOR) недостаточно, если целостность обеспечивается только зашифрованным *хэш-кодом*, а само сообщение не шифруется, подобная простая функция может использоваться, когда все сообщение и присоединенный к нему *хэш-код* шифруются. Но и в этом случае следует помнить о том, что подобная *хэш-функция* не может проследить за тем, чтобы при передаче последовательность блоков не изменилась. Это происходит в силу того, что данная *хэш-функция* определяется следующим образом: для сообщения, состоящего из последовательности 64-битных блоков X_1, X_2,..., X_N, определяется *хэш-код* C как поблочный XOR всех блоков, который присоединяется в качестве последнего блока:

$$C = X_{N+1} = X_1 \oplus X_2 \oplus \ldots \oplus X_N$$

Затем все сообщение шифруется, включая *хэш-код*, в режиме CBC для создания зашифрованных блоков Y_1, Y_2, ..., Y_{N+1}. По определению CBC имеем:

$$X_1 = IV \oplus D_K [Y_1]$$
$$X_i = Y_{i-1} \oplus D_K [Y_i]$$
$$X_{N+1} = Y_N \oplus D_K [Y_{N+1}]$$

Но X_{N+1} является *хэш-кодом*:

$$X_{N+1} = X_1 \oplus X_2 \oplus \ldots \oplus X_N =$$
$$(IV \oplus D_K [Y_1]) \oplus (Y_1 \oplus D_K [Y_2]) \oplus \ldots \oplus$$
$$(Y_{N-1} \oplus D_K [Y_N])$$

Так как сомножители в предыдущем равенстве могут вычисляться в любом порядке, следовательно, *хэш-код* не будет изменен, если зашифрованные блоки будут переставлены.

Первоначальный стандарт, предложенный NIST, использовал простой XOR, который применялся к 64-битным блокам сообщения, затем все сообщение шифровалось, используя режим CBC.

"Парадокс дня рождения"

Прежде чем рассматривать более сложные *хэш-функции*, необходимо проанализировать одну конкретную атаку на простые *хэш-функции*.

Так называемый "***парадокс дня рождения***" состоит в следующем. Предположим, количество выходных значений *хэш-функции* H равно n. Каким должно быть число k, чтобы для конкретного значения X и значений Y_1, …, Y_k вероятность того, что хотя бы для одного Y_i выполнялось равенство

$$H(X) = H(Y)$$

была бы больше $0,5$.

Для одного Y вероятность того, что $H(X) = H(Y)$, равна $1/n$.

Соответственно, вероятность того, что $H(X) \neq H(Y)$, равна $1 - 1/n$.

Если создать k значений, то вероятность того, что ни для одного из них не будет совпадений, равна произведению вероятностей, соответствующих одному значению, т.е. $(1 - 1/n)^k$.

Следовательно, вероятность, по крайней мере, одного совпадения равна

$$1 - (1 - 1/n)^k$$

По формуле бинома Ньютона

$$(1 - a)^k =$$
$$1 - ka + (k(k-1)/2!)a^2 - \ldots \approx 1 - ka$$
$$1 - (1 - k/n) = k/n = 0,5$$
$$k = n/2$$

Таким образом, мы выяснили, что для m-битового *хэш-кода* достаточно выбрать 2^{m-1} сообщений, чтобы вероятность совпадения *хэш-кодов* была больше 0,5.

Теперь рассмотрим следующую задачу: обозначим $P(n, k)$ вероятность того, что в множестве из k элементов, каждый из которых может принимать n значений, есть хотя бы два с одинаковыми значениями. Чему должно быть равно k, чтобы $P(n, k)$ была бы больше $0,5$?

Число различных способов выбора элементов таким образом, чтобы при этом не было дублей, равно

$$n(n-1) \ldots (n-k+1)n!/(n-k)!$$

Всего возможных способов выбора элементов равно

$$n^k$$

Вероятность того, что дублей нет, равна

$$n!/(n-k)!n^k$$

Вероятность того, что есть дубли, соответственно равна

$$1 - n!/(n-k)!n^k$$
$$P(n, k) = 1 - n! / ((n-k)! \times n^k) =$$
$$1 - (n \times (n-1) \times \ldots \times (n-k-1)) / n^k =$$
$$1 - [(n-1)/n \times (n-2)/n \times \ldots \times (n-k+1)/n] =$$
$$1 - [(1- 1/n) \times (1 - 2/n) \times \ldots \times (1 - (k-1)/n)]$$

Известно, что

$$1 - x \leqslant e^{-x}$$
$$P(n, k) > 1 - [e^{-1/n} \times e^{-2}/n \times \ldots \times e^{-k}/n]$$
$$P(n, k) > 1 - e^{-k(k-1)/n}$$
$$1/2 = 1 - e^{-k(k-1)/n}$$
$$2 = e^{k(k-1)/n}$$
$$\ln 2 = k(k-1) / 2n$$
$$k(k-1) \approx k^2$$
$$k = (2n \times \ln 2)^{1/2} = 1{,}17\, n^{1/2} \approx n^{1/2}$$

Если *хэш-код* имеет длину m бит, т.е. принимает 2^m значений, то

$$k = \sqrt{2m} = 2^{m/2}$$

Подобный результат называется "парадоксом дня рождения", потому что в соответствии с приведенными выше рассуждениями для того, чтобы вероятность совпадения дней рождения у двух человек была больше $0{,}5$, в группе должно быть всего 23 человека. Этот результат кажется удивительным, возможно, потому, что для каждого отдельного человека в группе вероятность того, что с его днем рождения совпадет день рождения кого-то другого в группе, достаточно мала.

Вернемся к рассмотрению свойств *хэш-функций*. Предположим, что используется 64-битный *хэш-код*. Можно считать, что это вполне достаточная и, следовательно, безопасная длина для *хэш-кода*. Например, если зашифрованный *хэш-код* C передается с соответствующим незашифрованным сообщением M, то противнику необходимо будет найти M' такое, что

$$H(M') = H(M)$$

для того, чтобы подменить сообщение и обмануть получателя. В среднем противник должен перебрать 2^{63} сообщений для того, чтобы найти такое, у которого *хэш-код* равен перехваченному сообщению.

Тем не менее, возможны различного рода атаки, основанные на "парадоксе дня рождения". Возможна следующая стратегия:

1. Противник создает $2^{m/2}$ вариантов сообщения, каждое из которых имеет некоторый определенный смысл. Противник подготавливает такое же количество сообщений, каждое из которых является поддельным и предназначено для замены настоящего сообщения.
2. Два набора сообщений сравниваются в поисках пары сообщений, имеющих одинаковый *хэш-код*. Вероятность успеха в соответствии с "парадоксом дня рождения" больше, чем $0{,}5$. Если соответствующая пара не найдена, то создаются дополнительные исходные и поддельные сообщения до тех пор, пока не будет найдена пара.
3. Атакующий предлагает отправителю исходный вариант сообщения для подписи. Эта подпись может быть затем присоединена к поддельному варианту для передачи получателю. Так как оба варианта имеют один и тот же *хэш-код*, будет создана одинаковая подпись. Противник будет уверен в успехе, даже не зная ключа шифрования.

Таким образом, если используется 64-битный *хэш-код*, то необходимая сложность вычислений составляет порядка 2^{32}.

В заключение отметим, что длина *хэш-кода* должна быть достаточно большой. Длина, равная 64 битам, в настоящее время не считается безопасной. Предпочтительнее, чтобы длина составляла порядка 100 битов.

Использование цепочки зашифрованных блоков

Существуют различные *хэш-функции*, основанные на создании цепочки зашифрованных блоков, но без использования секретного ключа. Одна из таких *хэш-функций* была предложена Рабином. Сообщение M разбивается на блоки фиксированной длины M_1, M_2, . . . , M_N и используется алгоритм симметричного шифрования, например DES, для вычисления *хэш-кода* G следующим образом:

$$H_0 = \text{начальное значение}$$
$$H_i = E_{Mi}\ [H_{i-1}]$$
$$G = H_N$$

Это аналогично использованию шифрования в режиме CBC, но в данном случае секретного ключа нет. Как и в случае любой *простой хэш-функции*, этот алгоритм подвержен "атаке дня рождения", и если шифрующим алгоритмом является DES и создается только 64-битный *хэш-код*, то система считается достаточно уязвимой.

Могут осуществляться другие атаки типа "дня рождения", которые возможны даже в том случае, если противник имеет доступ только к одному сообщению и соответствующему ему зашифрованному *хэш-коду* и не может получить несколько пар сообщений и зашифрованных *хэш-кодов*. Возможен следующий сценарий: предположим, что противник перехватил сообщение с аутентификатором в виде зашифрованного *хэш-кода*, и известно, что незашифрованный *хэш-код* имеет длину m битов. Далее противник должен выполнить следующие действия:

- Используя описанный выше алгоритм, вычислить незашифрованный *хэш-код* G.
- Создать поддельное сообщение в виде Q_1, Q_2, . . . , Q_{N-2}.
- Вычислить $H_i = E_{Qi}[H_{i-1}]$ для $1 \leq i \leq N-2$.
- Создать $2^{m/2}$ случайных блока X и для каждого такого блока X вычислить $E_X[H_{N-2}]$. Создать дополнительно $2^{m/2}$ случайных блока Y и для каждого блока Y вычислить $D_Y[G]$, где D - дешифрующая функция, соответствующая E. Основываясь на "парадоксе дня рождения" можно сказать, что с высокой степенью вероятности эта последовательность будет содержать блоки X и Y такие, что $E_X[H_{N-2}] = D_Y[Y]$.
- Создать сообщение Q_1, Q_2, . . . ,Q_{N-2}, X, Y. Это сообщение имеет *хэш-код* G и, следовательно, может быть использовано вместе с зашифрованным аутентификатором.

Эта форма атаки известна как атака "встреча посередине". В различных исследованиях предлагаются более тонкие методы для усиления подхода, основанного на цепочке блоков. Например, Девис и Прайс описали следующий вариант:

$$H_i = E_{Mi}\ [H_{i-1}] \oplus H_{i-1}$$

Возможен другой вариант:

$$H_i = E_{Hi-1}\ [M_i] \oplus M_i$$

Однако обе эти схемы также имеют уязвимости при различных атаках. В более общем случае, можно показать, что некоторая форма "атаки дня рождения" имеет успех при любом хэш-алгоритме, включающем использование цепочки шифрованных блоков без применения секретного ключа.

Дальнейшие исследования были направлены на поиск других подходов к созданию функций хэширования.

Хэш-функция MD5

Рассмотрим алгоритм получения *дайджеста сообщения MD5* (RFC 1321), разработанный Роном Ривестом из MIT.

Логика выполнения MD5

Алгоритм получает на входе сообщение произвольной длины и создает в качестве выхода *дайджест сообщения* длиной 128 бит. Алгоритм состоит из следующих шагов:

Рис. 8.1. Логика выполнения MD5

Шаг 1: добавление недостающих битов

Сообщение дополняется таким образом, чтобы его длина стала равна `448` по модулю `512` (`длина ≡448 mod 512`). Это означает, что длина добавленного сообщения на `64` бита меньше, чем число, кратное `512`. Добавление производится всегда, даже если сообщение имеет нужную длину. Например, если длина сообщения `448` битов, оно дополняется `512` битами до `960` битов. Таким образом, число добавляемых битов находится в диапазоне от `1` до `512`.

Добавление состоит из единицы, за которой следует необходимое количество нулей.

Шаг 2: добавление длины

64-битное представление длины исходного (до добавления) сообщения в битах присоединяется к результату первого шага. Если первоначальная длина больше, чем 2^{64}, то используются только последние 64 бита. Таким образом, поле содержит длину исходного сообщения по модулю 2^{64}.

В результате первых двух шагов создается сообщение, длина которого кратна `512` битам. Это расширенное сообщение представляется как последовательность 512-битных блоков $Y_0, Y_1, \ldots, Y_{L-1}$, при этом общая длина расширенного сообщения равна `L * 512` битам. Таким образом, длина полученного расширенного сообщения кратна шестнадцати 32-битным словам.

Рис. 8.2. Структура расширенного сообщения

Шаг 3: инициализация MD-буфера

Используется 128-битный буфер для хранения промежуточных и окончательных результатов *хэш-функции*. Буфер может быть представлен как четыре 32-битных регистра (`A`, `B`, `C`, `D`). Эти регистры инициализируются следующими шестнадцатеричными числами:

```
A = 01234567
B = 89ABCDEF
C = FEDCBA98
D = 76543210
```

Шаг 4: обработка последовательности 512-битных (16-словных) блоков

Основой алгоритма является модуль, состоящий из четырех циклических обработок, обозначенный как HMD5. Четыре цикла имеют похожую структуру, но каждый цикл использует свою элементарную логическую функцию, обозначаемую f_F, f_G, f_H и f_I соответственно.

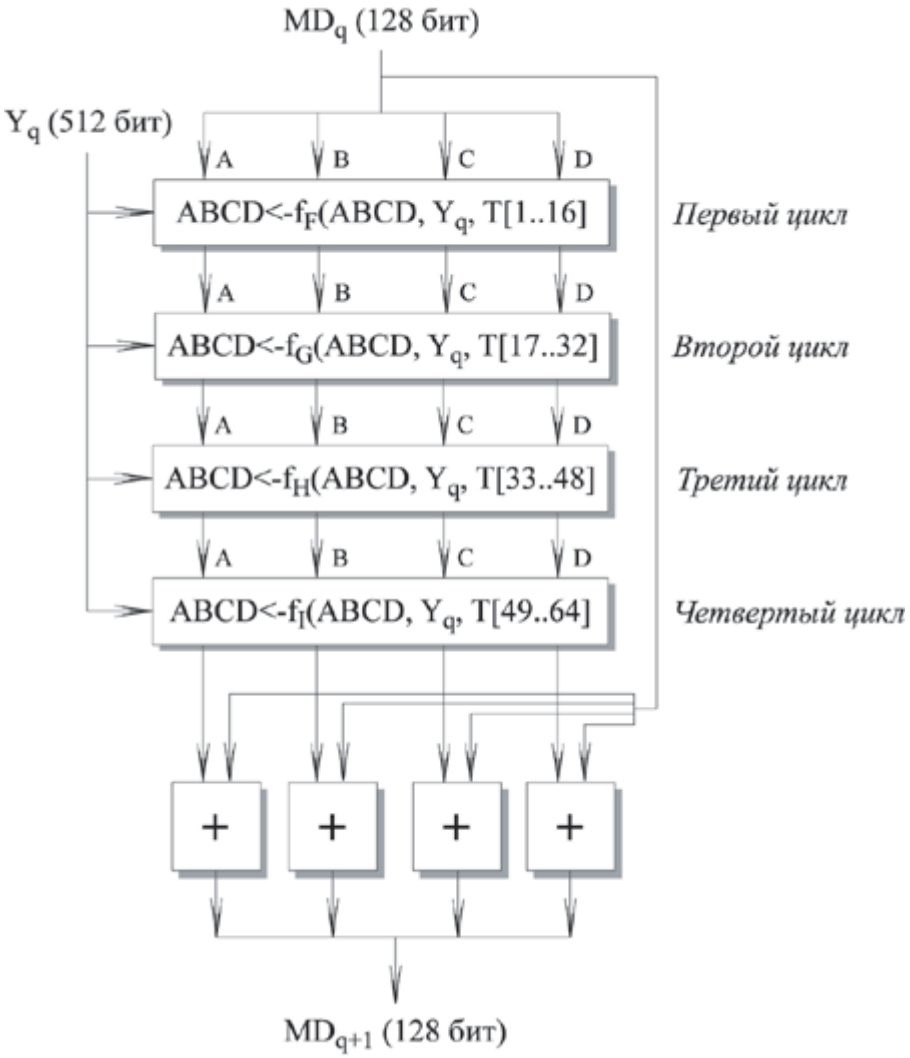

Рис. 8.3. Обработка очередного 512-битного блока

Каждый цикл принимает в качестве входа текущий 512-битный блок Y_q, обрабатывающийся в данный момент, и 128-битное значение буфера `ABCD`, которое является промежуточным значением *дайджеста*, и изменяет содержимое этого буфера. Каждый цикл также использует четвертую часть 64-элементной таблицы `T[1 ... 64]`, построенной на основе функции `sin`. i-ый элемент `T`, обозначаемый `T[i]`, имеет

значение, равное целой части от $2^{32} * abs\ (sin\ (i))$, i задано в радианах. Так как $abs\ (sin\ (i))$ является числом между 0 и 1, каждый элемент T является целым, которое может быть представлено 32 битами. Таблица обеспечивает "случайный" набор 32-битных значений, которые должны ликвидировать любую регулярность во входных данных.

Для получения MD_{q+1} выход четырех циклов складывается по модулю 2^{32} с MD_q. Сложение выполняется независимо для каждого из четырех слов в буфере.

Шаг 5: выход

После обработки всех L 512-битных блоков выходом L-ой стадии является 128-битный *дайджест сообщения*.

Рассмотрим более детально логику каждого из четырех циклов выполнения одного 512-битного блока. Каждый цикл состоит из 16 шагов, оперирующих с буфером ABCD. Каждый шаг можно представить в виде:

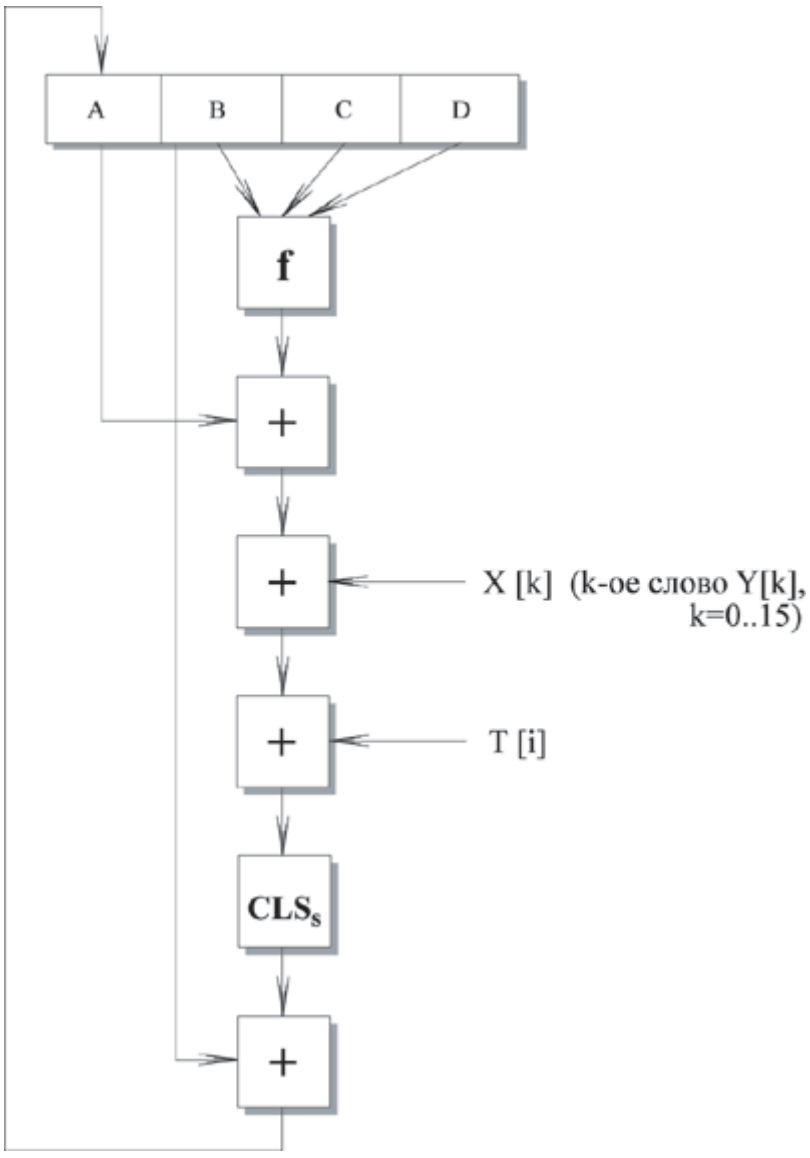

Рис. 8.4. Логика выполнения отдельного шага
$A \leftarrow B + CLS_s\ (A + f\ (B, C, D) + X\ [k] + T\ [i])$

где

A, B, C, D - четыре слова буфера; после выполнения каждого отдельного шага происходит циклический сдвиг влево на одно слово.

f - одна из элементарных функций f_F, f_G, f_H, f_I.

CLS_s - циклический сдвиг влево на s битов 32-битного аргумента.

X [k] = M [q * 16 + k] - k-ое 32-битное слово в q-ом 512 блоке сообщения.

T [i] - i-ое 32-битное слово в матрице T.

+ - сложение по модулю 2^{32}.

На каждом из четырех циклов алгоритма используется одна из четырех элементарных логических функций. Каждая элементарная функция получает три 32-битных слова на входе и на выходе создает одно 32-битное слово. Каждая функция является множеством побитовых логических операций, т.е. n-ый бит выхода является функцией от n-ого бита трех входов. Элементарные функции следующие:

$f_F = (B \& C) \vee (\text{not } B \& D)$
$f_G = (B \& D) \vee (C \& \text{not } D)$
$f_H = B \oplus C \oplus D$
$f_I = C \oplus (B \& \text{not } D)$

Массив из 32-битных слов X [0..15] содержит значение текущего 512-битного входного блока, который обрабатывается в настоящий момент. Каждый цикл выполняется 16 раз, а так как каждый блок входного сообщения обрабатывается в четырех циклах, то каждый блок входного сообщения обрабатывается по схеме, показанной на Рис. 4, 64 раза. Если представить входной 512-битный блок в виде шестнадцати 32-битных слов, то каждое входное 32-битное слово используется четыре раза, по одному разу в каждом цикле, и каждый элемент таблицы T, состоящей из 64 32-битных слов, используется только один раз. После каждого шага цикла происходит циклический сдвиг влево четырех слов A, B, C и D. На каждом шаге изменяется только одно из четырех слов буфера ABCD. Следовательно, каждое слово буфера изменяется 16 раз, и затем 17-ый раз в конце для получения окончательного выхода данного блока.

Можно суммировать алгоритм *MD5* следующим образом:

$MD_0 = IV$
$MD_{q+1} = MD_q + f_I[Y_q, f_H[Y_q, f_G[Y_q, f_F[Y_q, MD_q]]]]$
$MD = MD_{L-1}$

Где

IV - начальное значение буфера ABCD, определенное на шаге 3.

Y_q - q-ый 512-битный блок сообщения.

L - число блоков в сообщении (включая поля дополнения и длины).

MD - окончательное значение *дайджеста сообщения*.

Алгоритм MD4

Алгоритм MD4 является более ранней разработкой того же автора Рона Ривеста. Первоначально данный алгоритм был опубликован в октябре 1990 г., незначительно измененная версия была опубликована в RFC 1320 в апреле 1992 г. Кратко рассмотрим основные цели MD4:

1. Безопасность: это обычное требование к *хэш-коду*, состоящее в том, чтобы было вычислительно невозможно найти два сообщения, имеющие один и тот же *дайджест*.
2. Скорость: программная реализация алгоритма должна выполняться достаточно быстро. В частности, алгоритм должен быть достаточно быстрым на 32-битной архитектуре. Поэтому алгоритм основан на простом множестве элементарных операций над 32-битными словами.
3. Простота и компактность: алгоритм должен быть простым в описании и простым в программировании, без больших программ или подстановочных таблиц. Эти характеристики не только имеют очевидные программные преимущества, но и желательны с точки зрения безопасности, потому что для анализа возможных слабых мест лучше иметь простой алгоритм.
4. Желательна little-endian архитектура: некоторые архитектуры процессоров (такие как линия Intel 80xxx) хранят левые байты слова в позиции младших адресов байта (little-endian). Другие (такие как SUN Sparcstation) хранят правые байты слова в позиции младших адресов байта (big endian). Это различие важно, когда сообщение трактуется как последовательность 32-битовых слов, потому что эти архитектуры имеют инверсное представление байтов в каждом слове. Ривест выбрал использование схемы little-endian для интерпретации сообщения в качестве последовательности 32-битных слов. Этот выбор сделан потому, что big-endian процессоры обычно являются более быстрыми.

Эти цели преследовались и при разработке *MD5*. *MD5* является более сложным и, следовательно, более медленным при выполнении, чем MD4. Считается, что добавление сложности оправдывается возрастанием уровня безопасности. Главные различия между этими двумя алгоритмами состоят в следующем:

1. MD4 использует три цикла из 16 шагов каждый, в то время как *MD5* использует четыре цикла из 16 шагов каждый.
2. В MD4 дополнительная константа в первом цикле не применяется. Аналогичная дополнительная константа используется для каждого из шагов во втором цикле. Другая дополнительная константа используется для каждого из шагов в третьем цикле. В *MD5* различные дополнительные константы, `T [i]`, применяются для каждого из 64 шагов.
3. *MD5* использует четыре элементарные логические функции, по одной на каждом цикле, по сравнению с тремя в MD4, по одной на каждом цикле.
4. В *MD5* на каждом шаге текущий результат складывается с результатом предыдущего шага. Например, результатом первого шага является измененное слово `A`. Результат второго шага хранится в `D` и образуется добавлением `A` к циклически сдвинутому влево на определенное число бит результату элементарной функции. Аналогично, результат третьего шага хранится в `C` и образуется добавлением `D` к циклически сдвинутому влево результату элементарной функции. MD4 это последнее сложение не включает.

Усиление алгоритма в MD5

Алгоритм *MD5* имеет следующее свойство: каждый бит *хэш-кода* является функцией от каждого бита входа. Комплексное повторение элементарных функций f_F, f_G, f_H и f_I обеспечивает то, что результат хорошо перемешан; то есть маловероятно, чтобы два сообщения, выбранные случайно, даже если они имеют явно похожие закономерности, имели одинаковый *хэш-код*. Считается, что *MD5* является наиболее *сильной хэш-функцией* для 128-битного *хэш-кода*, то есть трудность нахождения двух сообщений, имеющих одинаковый *дайджест*, имеет порядок 2^{64} операций. В то время, как трудность нахождения сообщения с данным *дайджестом* имеет порядок 2^{128} операций.

Два результата, тем не менее, заслуживают внимания. Показано, что используя дифференциальный криптоанализ, можно за разумное время найти два сообщения, которые создают один и тот же *дайджест* при использовании только одного цикла *MD5*.

Подобный результат можно продемонстрировать для каждого из четырех циклов. Однако обобщить эту атаку на полный алгоритм *MD5* из четырех циклов пока не удалось.

Существует способ выбора блока сообщения и двух соответствующих ему промежуточных значений *дайджеста*, которые создают одно и то же выходное значение. Это означает, что выполнение *MD5* над единственным блоком из 512 бит приведет к одинаковому выходу для двух различных входных значений в буфере `ABCD`. Пока способа расширения данного подхода для успешной атаки на *MD5* не существует.

9. Лекция: Хэш-функции и аутентификация сообщений. Часть 2

Хэш-функция SHA-1

Безопасный хэш-алгоритм (Secure Hash Algorithm) был разработан национальным институтом стандартов и технологии (NIST) и опубликован в качестве федерального информационного стандарта (FIPS PUB 180) в 1993 году. ***SHA-1***, как и MD5, основан на алгоритме MD4.

Логика выполнения SHA-1

Алгоритм получает на входе сообщение максимальной длины 2^{64} бит и создает в качестве выхода дайджест сообщения длиной `160` бит.

Алгоритм состоит из следующих шагов:

Рис. 9.1. Логика выполнения SHA-1

Шаг 1: добавление недостающих битов

Сообщение добавляется таким образом, чтобы его длина была кратна `448` по модулю `512` (`длина ≡448 mod 512`). Добавление осуществляется всегда, даже если сообщение уже имеет нужную длину. Таким образом, число добавляемых битов находится в диапазоне от `1` до `512`.

Добавление состоит из единицы, за которой следует необходимое количество нулей.

Шаг 2: добавление длины

К сообщению добавляется блок из 64 битов. Этот блок трактуется как беззнаковое 64-битное целое и содержит длину исходного сообщения до добавления.

Результатом первых двух шагов является сообщение, длина которого кратна `512` битам. Расширенное сообщение может быть представлено как последовательность 512-битных блоков Y_0, Y_1, . . . , Y_{L-1}, так что общая длина расширенного сообщения есть `L * 512` бит. Таким образом, результат кратен шестнадцати 32-битным словам.

Шаг 3: инициализация SHA-1 буфера

Используется 160-битный буфер для хранения промежуточных и окончательных результатов хэш-функции. Буфер может быть представлен как пять 32-битных регистров `A`, `B`, `C`, `D` и `E`. Эти регистры инициализируются следующими шестнадцатеричными числами:

```
A = 67452301
B = EFCDAB89
C = 98BADCFE
D = 10325476
E = C3D2E1F0
```

Шаг 4: обработка сообщения в 512-битных (16-словных) блоках

Основой алгоритма является модуль, состоящий из 80 циклических обработок, обозначенный как H_{SHA}. Все 80 циклических обработок имеют одинаковую структуру.

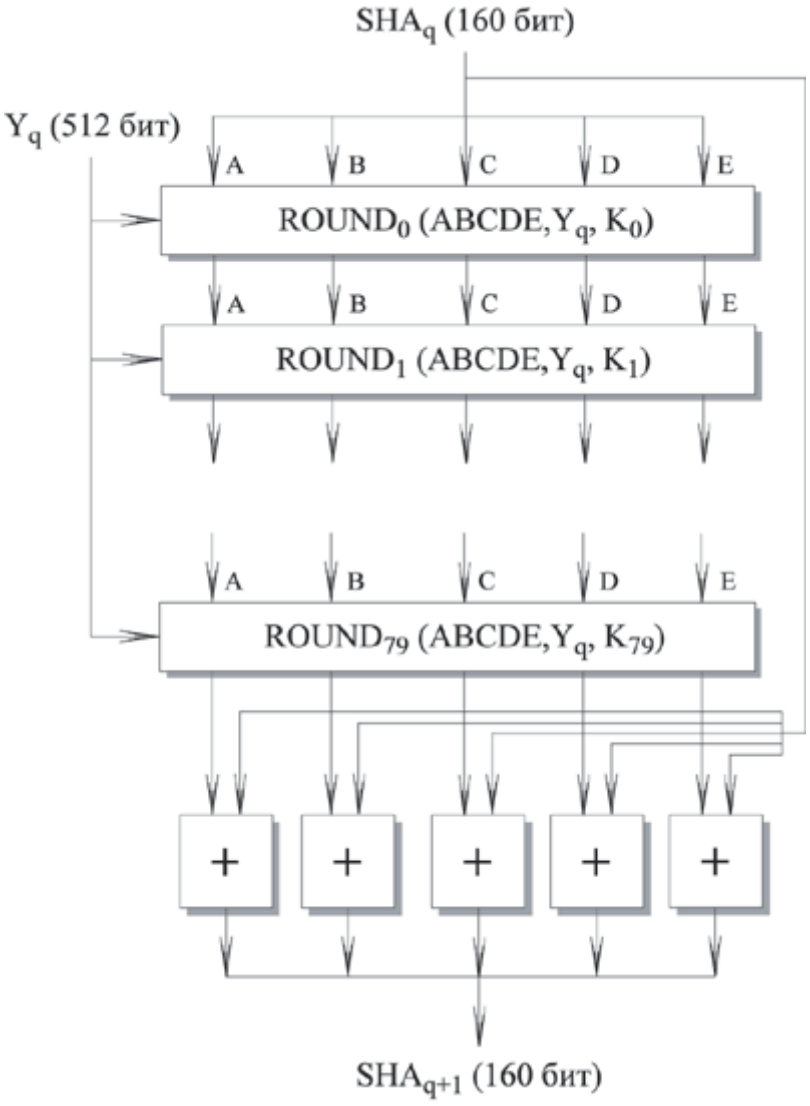

Рис. 9.2. Обработка очередного 512-битного блока

Каждый цикл получает на входе текущий 512-битный обрабатываемый блок Y_q и 160-битное значение буфера `ABCDE`, и изменяет содержимое этого буфера.

В каждом цикле используется дополнительная константа K_t, которая принимает только четыре различных значения:

$0 \leq t \leq 19$ $K_t = 5A827999$
(целая часть числа $[2^{30} \times 2^{1/2}]$)

$20 \leq t \leq 39$ $K_t = 6ED9EBA1$
(целая часть числа $[2^{30} \times 3^{1/2}]$)

$40 \leq t \leq 59$ $K_t = 8F1BBCDC$
(целая часть числа $[2^{30} \times 5^{1/2}]$)

$60 \leq t \leq 79$ $K^t = CA62C1D6$
(целая часть числа $[2^{30} \times 10^{1/2}]$)

Для получения SHA_{q+1} выход 80-го цикла складывается со значением SHA_q. Сложение по модулю 2^{32} выполняется независимо для каждого из пяти слов в буфере с каждым из соответствующих слов в SHA^q.

Шаг 5: выход

После обработки всех 512-битных блоков выходом L-ой стадии является 160-битный дайджест сообщения.

Рассмотрим более детально логику в каждом из 80 циклов обработки одного 512-битного блока. Каждый цикл можно представить в виде:

$$A, B, C, D, E\ (CLS_5\ (A) + f_t\ (B, C, D) + E + W_t + K_t), A, CLS_{30}\ (B), C, D$$

Где

A, B, C, D, E - пять слов из буфера.

t - номер цикла, $0 \leq t \leq 79$.

f_t - элементарная логическая функция.

CLS_s - циклический левый сдвиг 32-битного аргумента на s битов.

W_t - 32-битное слово, полученное из текущего входного 512-битного блока.

K_t - дополнительная константа.

$+$ - сложение по модулю 2^{32}.

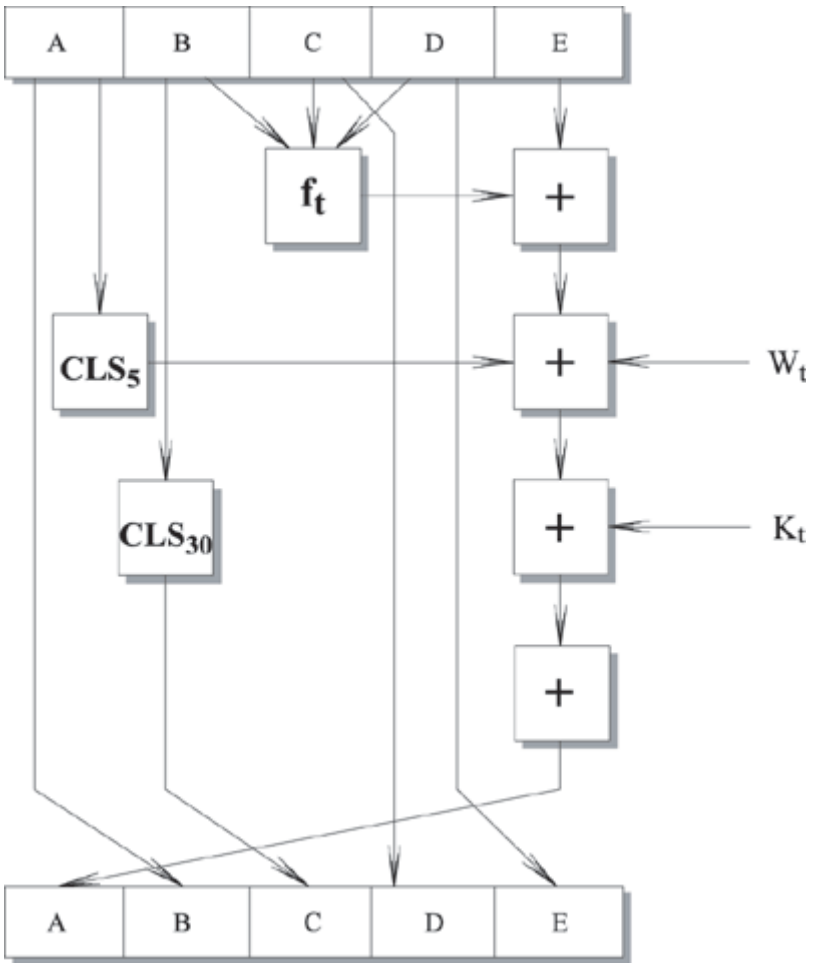

Рис. 9.3. Логика выполнения отдельного цикла

Каждая элементарная функция получает на входе три 32-битных слова и создает на выходе одно 32-битное слово. Элементарная функция выполняет набор побитных логических операций, т.е. n-ый бит выхода является функцией от n-ых битов трех входов. Функции следующие:

Номер цикла	ft (B, C, D)
$(0 \leq t \leq 19)$	$(B \wedge C) \vee (\neg B \wedge D)$
$(20 \leq t \leq 39)$	$B \oplus C \oplus D$
$(40 \leq t \leq 59)$	$(B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$
$(60 \leq t \leq 79)$	$B \oplus C \oplus D$

На самом деле используются только три различные функции. Для $0 \leq t \leq 19$ функция является условной: `if B then C else D`. Для $20 \leq t \leq 39$ и $60 \leq t \leq 79$ функция создает бит четности. Для $40 \leq t \leq 59$ функция является истинной, если два или три аргумента истинны.

32-битные слова W_t получаются из очередного 512-битного блока сообщения следующим образом.

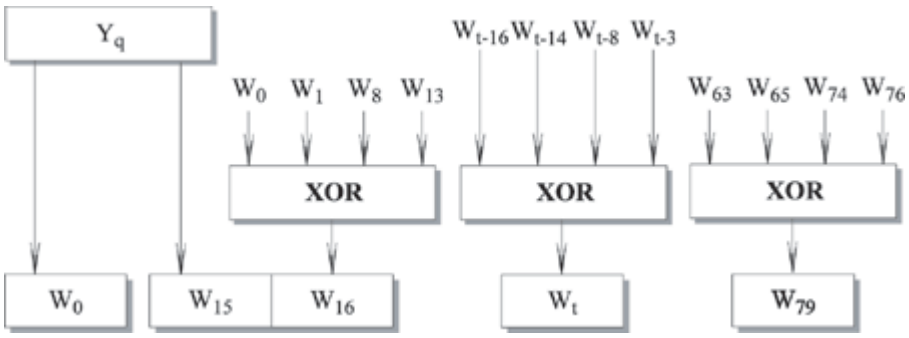

Рис. 9.4. Получение входных значений каждого цикла из очередного блока

Первые 16 значений W_t берутся непосредственно из 16 слов текущего блока. Оставшиеся значения определяются следующим образом:

$$W_t = W_{t-16} \oplus W_{t-14} \oplus W_{t-8} \oplus W_{t-3}$$

В первых 16 циклах вход состоит из 32-битного слова данного блока. Для оставшихся 64 циклов вход состоит из XOR нескольких слов из блока сообщения.

Алгоритм *SHA-1* можно суммировать следующим образом:

$$SHA_0 = IV$$
$$SHA_{q+1} = \Sigma 32\ (SHA_q\ ,\ ABCDE_q\ )$$
$$SHA = SHA_{L-1}$$

Где

IV - начальное значение буфера $ABCDE$.

$ABCDE_q$ - результат обработки q-того блока сообщения.

L - число блоков в сообщении, включая поля добавления и длины.

$\Sigma 32$ - сумма по модулю 2^{32}, выполняемая отдельно для каждого слова буфера.

SHA - значение дайджеста сообщения.

Сравнение SHA-1 и MD5

Оба алгоритма, *SHA-1* и MD5, произошли от MD4, поэтому имеют много общего.

Можно суммировать ключевые различия между алгоритмами.

	MD5	SHA–1
Длина дайджеста	128 бит	160 бит
Размер блока обработки	512 бит	512 бит
Число итераций	64 (4 цикла по 16 итераций в каждом)	80
Число элементарных логических функций	4	3
Число дополнительных констант	64	4

Сравним оба алгоритма в соответствии с теми целями, которые были определены для алгоритма MD4:

1. **Безопасность:** наиболее очевидное и наиболее важное различие состоит в том, что дайджест *SHA-1* на 32 бита длиннее, чем дайджест MD5. Если предположить, что оба алгоритма не содержат каких-либо структурированных данных, которые уязвимы для криптоаналитических атак, то *SHA-1* является более стойким алгоритмом. Используя лобовую атаку, труднее создать произвольное сообщение, имеющее данный дайджест, если требуется порядка 2^{160} операций, как в случае алгоритма *SHA-1*, чем порядка 2^{128} операций, как в случае алгоритма MD5. Используя лобовую атаку, труднее создать два сообщения, имеющие одинаковый дайджест, если требуется порядка 2^{80} как в случае алгоритма *SHA-1*, чем порядка 2^{64} операций как в случае алгоритма MD5.
2. **Скорость:** так как оба алгоритма выполняют сложение по модулю 2^{32}, они рассчитаны на 32-битную архитектуру. *SHA-1* содержит больше шагов (80 вместо 64) и выполняется на 160-битном буфере по сравнению со 128-битным буфером MD5. Таким образом, *SHA-1* должен выполняться приблизительно на 25% медленнее, чем MD5 на той же аппаратуре.
3. `Простота и компактность:` оба алгоритма просты и в описании, и в реализации, не требуют больших программ или подстановочных таблиц. Тем не менее, *SHA-1* применяет одношаговую структуру по сравнению с четырьмя структурами, используемыми в MD5. Более того, обработка слов в буфере одинаковая для всех шагов *SHA-1*, в то время как в MD5 структура слов специфична для каждого шага.
4. **Архитектуры little-endian и big-endian:** MD5 использует little-endian схему для интерпретации сообщения как последовательности 32-битных слов, в то время как *SHA-1* задействует схему big-endian. Каких-либо преимуществ в этих подходах не существует.

Хэш-функции SHA-2

В 2001 году NIST принял в качестве стандарта три хэш-функции с существенно большей длиной хэш-кода. Часто эти хэш-функции называют *SHA-2* или SHA-256, SHA-384 и SHA-512 (соответственно, в названии указывается длина создаваемого ими хэш-кода). Эти алгоритмы отличаются не только длиной создаваемого хэш-кода, но и длиной обрабатываемого блока, длиной слова и используемыми внутренними функциями. Сравним характеристики этих хэш-функций.

Алгоритм	Длина сообщения (в битах)	Длина блока (в битах)	Длина слова (в битах)	Длина дайджеста сообщения (в битах)	Безопасность (в битах)
SHA-1	$<2^{64}$	512	32	160	80
SHA-256	$<2^{64}$	512	32	256	128
SHA-384	$<2^{128}$	1024	64	384	192
SHA-512	$<2^{128}$	1024	64	512	256

Под безопасностью здесь понимается стойкость к атакам типа "парадокса дня рождения".

В данных алгоритмах размер блока сообщения равен `m` бит. Для SHA-256 `m = 512`, для SHA-384 и SHA-512 `m = 1024`. Каждый алгоритм оперирует с w-битными словами. Для SHA-256 `w = 32`, для SHA-384 и SHA-512 `w = 64`. В алгоритмах используются обычные булевские операции над словами, а также сложение по модулю 2^w, правый сдвиг на `n` бит $SHR^n(x)$, где `x` - w-битное слово, и циклические (ротационные) правый и левый сдвиги на `n` бит $ROTR^n(x)$ и $ROTL^n(x)$, где `x` - w-битное слово.

SHA-256 использует шесть логических функций, при этом каждая из них выполняется с 32-битными словами, обозначенными как x, y и z. Результатом каждой функции тоже является 32-битное слово.

$$Ch\ (x,\ y,\ z) = (x \wedge y) \oplus (\neg x \wedge z)$$
$$Maj\ (x,\ y,\ z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$
$$\Sigma_0^{\{256\}}\ (x) = ROTR^2\ (x) \oplus ROTR^{13}\ (x) \oplus ROTR^{22}\ (x)$$
$$\Sigma_1^{\{256\}}\ (x) = ROTR^6\ (x) \oplus ROTR^{11}\ (x) \oplus ROTR^{25}\ (x)$$
$$\sigma_0^{\{256\}}\ (x) = ROTR^7\ (x) \oplus ROTR^{18}\ (x) \oplus SHR^3\ (x)$$
$$\sigma_1^{\{256\}}\ (x) = ROTR^{17}\ (x) \oplus ROTR^{19}\ (x) \oplus SHR^{10}\ (x)$$

SHA-384 и SHA-512 также используют шесть логических функций, каждая из которых выполняется над 64-битными словами, обозначенными как x, y и z. Результатом каждой функции является 64-битное слово.

$$Ch\ (x,\ y,\ z) = (x \wedge y) \oplus (\neg x \wedge z)$$
$$Maj\ (x,\ y,\ z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$
$$\Sigma_0^{\{512\}}\ (x) = ROTR^{28}\ (x) \oplus ROTR^{34}\ (x) \oplus ROTR^{39}\ (x)$$
$$\Sigma_1^{\{512\}}\ (x) = ROTR^{14}\ (x) \oplus ROTR^{18}\ (x) \oplus ROTR^{41}\ (x)$$
$$\sigma_0^{\{512\}}\ (x) = ROTR^1\ (x) \oplus ROTR^8\ (x) \oplus SHR^7\ (x)$$
$$\sigma_1^{\{512\}}\ (x) = ROTR^{19}\ (x) \oplus ROTR^{61}\ (x) \oplus SHR^6\ (x)$$

Предварительная подготовка сообщения, т.е. добавление определенных битов до целого числа блоков и последующее разбиение на блоки выполняется аналогично тому, как это делалось в *SHA-1* (конечно, с учетом длины блока каждой хэш-функции). После этого каждое сообщение можно представить в виде последовательности N блоков $M^{(1)}$, $M^{(2)}$, … , $M^{(N)}$.

Рассмотрим SHA-256. В этом случае инициализируются восемь 32-битных переменных, которые послужат промежуточным значением хэш-кода:

$$a,\ b,\ c,\ d,\ e,\ f,\ g,\ h$$

Основой алгоритма является модуль, состоящий из 64 циклических обработок каждого блока $M^{(i)}$:

$$T_1 = h + \Sigma_1^{\{256\}}(e) + Ch(e,\ f,\ g) + K_t^{\{256\}} + W_t$$
$$T_2 = \Sigma_0^{\{256\}}(a) + Maj(a,\ b,\ c)$$
$$h = g$$
$$g = f$$
$$f = e$$
$$e = d + T_1$$
$$d = c$$
$$c = b$$
$$b = a$$
$$a = T_1 + T_2$$

где $K_i^{\{256\}}$ - шестьдесят четыре 32-битных константы, каждая из которых является первыми 32-мя битами дробной части кубических корней первых 64 простых чисел.

W_t вычисляются из очередного блока сообщения по следующим правилам:

$$W_t = M_t^{(i)}\ ,\ 0 \leq t \leq 15$$
$$W_t = \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16}\ ,$$

$16 \leq t \leq 63$

i-ое промежуточное значение хэш-кода $H^{(t)}$ вычисляется следующим образом:

$$H_0^{(i)} = a + H_0^{(i-1)}$$
$$H_1^{(i)} = b + H_1^{(i-1)}$$
$$H_2^{(i)} = c + H_2^{(i-1)}$$
$$H_3^{(i)} = d + H_3^{(i-1)}$$
$$H_4^{(i)} = e + H_4^{(i-1)}$$
$$H_5^{(i)} = f + H_5^{(i-1)}$$
$$H_6^{(i)} = g + H_6^{(i-1)}$$
$$H_7^{(i)} = h + H_7^{(i-1)}$$

Теперь рассмотрим SHA-512. В данном случае инициализируются восемь 64-битных переменных, которые будут являться промежуточным значением хэш-кода:

a, b, c, d, e, f, g, h

Основой алгоритма является модуль, состоящий из 80 циклических обработок каждого блока $M^{(i)}$:

$$T_1 = h + \Sigma_1^{\{512\}}(e) + Ch(e, f, g) + K_t^{\{512\}} + W_t$$
$$T_2 = \Sigma_0^{\{512\}}(a) + Maj(a, b, c)$$
$$h = g$$
$$g = f$$
$$f = e$$
$$e = d + T_1$$
$$d = c$$
$$c = b$$
$$b = a$$
$$a = T_1 + T_2$$

где $K_i^{\{512\}}$ - восемьдесят 64-битных констант, каждая из которых является первыми 64-мя битами дробной части кубических корней первых восьмидесяти простых чисел.

W_t вычисляются из очередного блока сообщения по следующим правилам:

$$W_t = M_t^{(i)} \ , \ 0 \leq t \leq 15$$
$$W_t = \sigma_1^{\{512\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{512\}}(W_{t-15}) + W_{t-16} \ ,$$
$$16 \leq t \leq 79$$

i-ое промежуточное значение хэш-кода $H(t)$ вычисляется следующим образом:

$$H_0^{(i)} = a + H_0^{(i-1)}$$
$$H_1^{(i)} = b + H_1^{(i-1)}$$
$$H_2^{(i)} = c + H_2^{(i-1)}$$
$$H_3^{(i)} = d + H_3^{(i-1)}$$
$$H_4^{(i)} = e + H_4^{(i-1)}$$
$$H_5^{(i)} = f + H_5^{(i-1)}$$
$$H_6^{(i)} = g + H_6^{(i-1)}$$
$$H_7^{(i)} = h + H_7^{(i-1)}$$

Рассмотрим SHA-384. Отличия этого алгоритма от SHA-512:

1. Другой начальный хэш-код $H^{(0)}$.
2. 384-битный дайджест получается из левых 384 битов окончательного хэш-кода $H^{(N)}$:

$H_0^{(N)} \,||\, H_1^{(N)} \,||\, H_2^{(N)} \,||\, H_3^{(N)} \,||\, H_4^{(N)} \,||\, H_5^{(N)}$.

Хэш-функция ГОСТ 3411

Алгоритм ***ГОСТ 3411*** является отечественным стандартом для хэш-функций. Его структура довольно сильно отличается от структуры алгоритмов *SHA-1,2* или MD5, в основе которых лежит алгоритм MD4.

Длина хэш-кода, создаваемого алгоритмом *ГОСТ 3411*, равна 256 битам. Алгоритм разбивает сообщение на блоки, длина которых также равна 256 битам. Кроме того, параметром алгоритма является стартовый вектор хэширования H - произвольное фиксированное значение длиной также 256 бит.

Алгоритм обработки одного блока сообщения

Сообщение обрабатывается блоками по 256 бит справа налево.

Каждый блок сообщения обрабатывается по следующему алгоритму.

1. Генерация четырех ключей длиной 256 бит каждый.
2. Шифрование 64-битных значений промежуточного хэш-кода H на ключах $K_i (i = 1, 2, 3, 4)$ с использованием алгоритма ГОСТ 28147 в режиме простой замены.
3. Перемешивание результата шифрования.

Для генерации ключей используются следующие данные:

- промежуточное значение хэш-кода H длиной 256 бит;
- текущий обрабатываемый блок сообщения M длиной 256 бит;
- параметры - три значения C_2, C_3 и C_4 длиной 256 бит следующего вида: C_2 и C_4 состоят из одних нулей, а C_3 равно

$$1^8\ 0^8\ 1^{16}\ 0^{24}\ 1_{16}\ 0^8\ (0^8\ 1^8)^2\ 1^8\ 0^8\ (0^8\ 1^8)^4\ (1^8\ 0^8)^4$$

где степень обозначает количество повторений 0 или 1.

Используются две формулы, определяющие перестановку и сдвиг.

Перестановка P битов определяется следующим образом: каждое 256-битное значение рассматривается как последовательность тридцати двух 8-битных значений.

Перестановка P элементов 256-битной последовательности выполняется по формуле $y = \Phi(x)$, где x - порядковый номер 8-битного значения в исходной последовательности; y - порядковый номер 8-битного значения в результирующей последовательности.

$$\Phi(i + 1 + 4(k - 1)) = 8i + k$$
$$i = 0 \div 3,\ k = 1 \div 8$$

Сдвиг A определяется по формуле

$$A(x) = (x_1 \oplus x_2) \,||\, x_4 \,||\, x_3 \,||\, x_2$$

Где

x_i - соответствующие 64 бита 256-битного значения x,

|| обозначает конкатенацию.

Присваиваются следующие начальные значения:

$i = 1, \; U = H, \; V = M.$

$W = U \oplus V, \; K_1 = P(W)$

Ключи K_2, K_3, K_4 вычисляются последовательно по следующему алгоритму:

$U = A(U) \oplus C_i,$
$V = A(A(V)),$

$W = U \oplus V,$
$K_i = P(W)$

Далее выполняется шифрование 64-битных элементов текущего значения хэш-кода H с ключами K_1, K_2, K_3 и K_4. При этом хэш-код H рассматривается как последовательность 64-битных значений:

$H = h_4 \; || \; h_3 \; || \; h_2 \; || \; h_1$

Выполняется шифрование алгоритмом ГОСТ 28147:

$s_i = E_{Ki}[h_i] \qquad i = 1, 2, 3, 4$
$S = s_1 \; || \; s_2 \; || \; s_3 \; || \; s_4$

Наконец на заключительном этапе обработки очередного блока выполняется перемешивание полученной последовательности. 256-битное значение рассматривается как последовательность шестнадцати 16-битных значений. Сдвиг обозначается Ψ и определяется следующим образом:

$\eta_{16} \; || \; \eta_{15} \; || \; \ldots \; || \; \eta_1$ - исходное значение

$\eta_1 \oplus \eta_2 \oplus \eta_3 \oplus \eta_4 \oplus \eta_{13} \oplus \eta_{16} \; || \; \eta_{16} \; || \; \ldots \; || \; \eta_2$ - результирующее значение

Результирующее значение хэш-кода определяется следующим образом:

$X(M, H) = \Psi^{61}(H \oplus \Psi(M \oplus \Psi^{12}(S)))$

где

H - предыдущее значение хэш-кода,

M - текущий обрабатываемый блок,

Ψ^i - i-ая степень преобразования Ψ.

Логика выполнения ГОСТ 3411

Входными параметрами алгоритма являются:

- исходное сообщение M произвольной длины;
- стартовый вектор хэширования H, длина которого равна 256 битам;
- контрольная сумма Σ, начальное значение которой равно нулю и длина равна 256 битам;
- переменная L, начальное значение которой равно длине сообщения.

Сообщение M делится на блоки длиной 256 бит и обрабатывается справа налево. Очередной блок i обрабатывается следующим образом:

1. $H = X(M_i, H)$
2. $\Sigma = \Sigma \boxplus' M_i$
3. L рассматривается как неотрицательное целое число, к этому числу прибавляется 256 и вычисляется остаток от деления получившегося числа на 2^{256}. Результат присваивается L.

Где $\boxplus'$ обозначает следующую операцию: Σ и M_i рассматриваются как неотрицательные целые числа длиной 256 бит. Выполняется обычное сложение этих чисел и находится остаток от деления результата сложения на 2^{256}. Этот остаток и является результатом операции.

Самый левый, т.е. самый последний блок M' обрабатывается следующим образом:

1. Блок добавляется слева нулями так, чтобы его длина стала равна 256 битам.
2. Вычисляется $\Sigma = \Sigma \boxplus' M_i$.
3. L рассматривается как неотрицательное целое число, к этому числу прибавляется длина исходного сообщения M и находится остаток от деления результата сложения на 2^{256}.
4. Вычисляется $H = X(M', H)$.
5. Вычисляется $H = X(L, H)$.
6. Вычисляется $H = X(\Sigma, H)$.

Значением функции хэширования является H.

Коды аутентификации сообщений - MAC

Требования к MAC

Напомним, что обеспечение целостности сообщения - это невозможность изменения сообщения так, чтобы получатель этого не обнаружил. Под аутентификацией понимается подтверждение того, что информация получена от законного источника, и получателем является тот, кто нужно. Один из способов обеспечения целостности - это вычисление *MAC* (Message Authentication Code). В данном случае под *MAC* понимается некоторый аутентификатор, являющийся определенным способом вычисленным блоком данных, с помощью которого можно проверить целостность сообщения. В некоторой степени симметричное шифрование всего сообщения может выполнять функцию аутентификации этого сообщения. Но в таком случае сообщение должно содержать достаточную избыточность, которая позволяла бы проверить, что сообщение не было изменено. Избыточность может быть в виде определенным образом отформатированного сообщения, текста на конкретном языке и т.п. Если сообщение допускает произвольную последовательность битов (например, зашифрован ключ сессии), то симметричное шифрование всего сообщения не может обеспечивать его целостность, так как при дешифровании в любом случае получится последовательность битов, правильность которой проверить нельзя. Поэтому гораздо чаще используется критографически созданный небольшой блок данных фиксированного размера, так называемый аутентификатор или имитовставка, с помощью которого проверяется целостность сообщения. Этот блок данных может создаваться с помощью секретного ключа, который разделяют отправитель и получатель. *MAC* вычисляется в тот момент, когда известно, что сообщение корректно. После этого *MAC* присоединяется к сообщению и передается вместе с ним получателю. Получатель вычисляет *MAC*, используя тот же самый секретный ключ, и сравнивает вычисленное значение с

полученным. Если эти значения совпадают, то с большой долей вероятности можно считать, что при пересылке изменения сообщения не произошло.

$MAC = CK\ (M)$

Рассмотрим свойства, которыми должна обладать функция MAC. Если длина ключа, используемого при вычислении MAC, равна k, то при условии сильной функции MAC противнику потребуется выполнить 2^k попыток для перебора всех ключей. Если длина значения, создаваемого MAC, равна n, то всего существует 2^n различных значений *MAC*.

Предположим, что конфиденциальности сообщения нет, т.е. оппонент имеет доступ к открытому сообщению и соответствующему ему значению *MAC*. Определим усилия, необходимые оппоненту для нахождения ключа *MAC*. Предположим, что $k > n$, т.е. длина ключа больше длины *MAC*. Тогда, зная M_1 и $MAC_1 = C_K\ (M_1)$, оппонент может вычислить $MAC_1 = CK_i\ (M_1)$ для всех возможных ключей K_i. При этом, по крайней мере, для одного из ключей будет получено совпадение $MAC_i = MAC_1$. Оппонент вычислит 2^k значений *MAC*, тогда как при длине *MAC* n битов существует всего 2^n значений *MAC*. Мы предположили, что $k > n$, т.е. $2^k > 2^n$. Таким образом, правильное значение *MAC* будет получено для нескольких значений ключей. В среднем совпадение будет иметь место для $2^k / 2^n = 2^{(k-n)}$ ключей. Поэтому для вычисления единственного ключа оппоненту требуется знать несколько пар сообщение и соответствующий ему *MAC*.

Таким образом, простой перебор всех ключей требует не меньше, а больше усилий, чем поиск ключа симметричного шифрования той же длины.

Функция вычисления *MAC* должна обладать следующими свойствами:

1. Должно быть вычислительно трудно, зная M и $C_K\ (M)$, найти сообщение M', такое, что $C_K(M) = C_K(M')$.
2. Значения $C_K(M)$ должны быть равномерно распределенными в том смысле, что для любых сообщений M и M' вероятность того, что $C_K(M) = C_K(M')$, должна быть равна 2^{-n}, где n - длина значения MAC.

MAC на основе алгоритма симметричного шифрования

Для вычисления *MAC* может использоваться алгоритм симметричного шифрования (например, DES) в режиме CBC и нулевой инициализационный вектор. В этом случае сообщение представляется в виде последовательности блоков, длина которых равна длине блока алгоритма шифрования. При необходимости последний блок дополняется справа нулями, чтобы получился блок нужной длины. Вычисление *MAC* происходит по следующей схеме:

$$MAC_1 = E_K\ [P_1]$$
$$MAC_2 = E_K\ [P_2 \oplus MAC_1]$$
$$\cdot\ \cdot\ \cdot$$
$$MAC_N = E_K\ [P_N \oplus MAC_{N-1}]$$
$$MAC = MAC_N$$

MAC = MACN MAC на основе хэш-функции

Другим способом обеспечения целостности является использование хэш-функции. Хэш-код присоединяется к сообщению в тот момент, когда известно, что сообщение корректно. Получатель проверяет целостность сообщения вычислением хэш-кода полученного сообщения и сравнением его с полученным хэш-кодом, который должен быть передан безопасным способом. Одним из таких безопасных способов может быть

шифрование хэш-кода закрытым ключом отправителя, т.е. создание подписи. Возможно также шифрование полученного хэш-кода алгоритмом симметричного шифрования, если отправитель и получатель имеют общий ключ симметричного шифрования.

HMAC

Еще один вариант использования хэш-функции для получения *MAC* состоит в том, чтобы определенным образом добавить секретное значение к сообщению, которое подается на вход хэш-функции. Такой алгоритм носит название ***HMAC***, и он описан в RFC 2104.

При разработке алгоритма HMAC преследовались следующие цели:

- возможность использовать без модификаций уже имеющиеся хэш-функции;
- возможность легкой замены встроенных хэш-функций на более быстрые или более стойкие;
- сохранение скорости работы алгоритма, близкой к скорости работы соответствующей хэш-функции;
- возможность применения ключей и простота работы с ними.

В алгоритме *HMAC* хэш-функция представляет собой "черный ящик". Это, во-первых, позволяет использовать существующие реализации хэш-функций, а во-вторых, обеспечивает легкую замену существующей хэш-функции на новую.

Введем следующие обозначения:

`H` - встроенная хэш-функция.

`b` - длина блока используемой хэш-функции.

`n` - длина хэш-кода.

`K` - секретный ключ. К этому ключу слева добавляют нули, чтобы получить b-битовый ключ K^+.

Вводится два вспомогательных значения:

`Ipad` - значение `'00110110'`, повторенное `b/8` раз.

`Opad` - значение `'01011010'`, повторенное `b/8` раз.

Далее *HMAC* вычисляется следующим образом:

$$HMAC = H((K^+ \oplus Opad) || H((K^+ \oplus Ipad) || M))$$

10. Лекция: Цифровая подпись

Требования к цифровой подписи

Аутентификация защищает двух участников, которые обмениваются сообщениями, от воздействия некоторой третьей стороны. Однако простая аутентификация не защищает участников друг от друга, тогда как и между ними тоже могут возникать определенные формы споров.

Например, предположим, что Джон посылает Мери аутентифицированное сообщение, и аутентификация осуществляется на основе общего секрета. Рассмотрим возможные недоразумения, которые могут при этом возникнуть:

- Мери может подделать сообщение и утверждать, что оно пришло от Джона. Мери достаточно просто создать сообщение и присоединить аутентификационный код, используя ключ, который разделяют Джон и Мери.
- Джон может отрицать, что он посылал сообщение Мери. Так как Мери может подделать сообщение, у нее нет способа доказать, что Джон действительно посылал его.

В ситуации, когда обе стороны не доверяют друг другу, необходимо нечто большее, чем аутентификация на основе общего секрета. Возможным решением подобной проблемы является использование *цифровой подписи*. *Цифровая подпись* должна обладать следующими свойствами:

1. Должна быть возможность проверить автора, дату и время создания подписи.
2. Должна быть возможность аутентифицировать содержимое во время создания подписи.
3. Подпись должна быть проверяема третьей стороной для разрешения споров.

Таким образом, функция *цифровой подписи* включает функцию аутентификации.

На основании этих свойств можно сформулировать следующие требования к *цифровой подписи*:

1. Подпись должна быть битовым образцом, который зависит от подписываемого сообщения.
2. Подпись должна использовать некоторую уникальную информацию отправителя для предотвращения подделки или отказа.
3. Создавать *цифровую подпись* должно быть относительно легко.
4. Должно быть вычислительно невозможно подделать *цифровую подпись* как созданием нового сообщения для существующей *цифровой подписи*, так и созданием ложной *цифровой подписи* для некоторого сообщения.
5. *Цифровая подпись* должна быть достаточно компактной и не занимать много памяти.

Сильная хэш-функция, зашифрованная закрытым ключом отправителя, удовлетворяет перечисленным требованиям.

Существует несколько подходов к использованию функции *цифровой подписи*. Все они могут быть разделены на две категории: *прямые* и *арбитражные*.

Прямая и арбитражная цифровые подписи

При использовании ***прямой цифровой подписи*** взаимодействуют только сами участники, т.е. отправитель и получатель. Предполагается, что получатель знает открытый ключ отправителя. *Цифровая подпись* может быть создана шифрованием всего сообщения или его хэш-кода закрытым ключом отправителя.

Конфиденциальность может быть обеспечена дальнейшим шифрованием всего сообщения вместе с подписью открытым ключом получателя (асимметричное шифрование) или разделяемым секретным ключом (симметричное шифрование). Заметим, что обычно функция подписи выполняется первой, и только после этого выполняется функция конфиденциальности. В случае возникновения спора некая третья сторона должна просмотреть сообщение и его подпись. Если функция подписи выполняется над зашифрованным сообщением, то для разрешения споров придется хранить сообщение как в незашифрованном виде (для практического использования), так и в зашифрованном (для проверки подписи). Либо в этом случае необходимо хранить ключ симметричного шифрования, для того чтобы можно было проверить подпись исходного сообщения. Если *цифровая подпись* выполняется над

незашифрованным сообщением, получатель может хранить только сообщение в незашифрованном виде и соответствующую подпись к нему.

Все прямые схемы, рассматриваемые далее, имеют общее слабое место. Действенность схемы зависит от безопасности закрытого ключа отправителя. Если отправитель впоследствии не захочет признать факт отправки сообщения, он может утверждать, что закрытый ключ был потерян или украден, и в результате кто-то подделал его подпись. Можно применить административное управление, обеспечивающее безопасность закрытых ключей, для того чтобы, по крайней мере, хоть в какой-то степени ослабить эти угрозы. Один из возможных способов состоит в требовании в каждую подпись сообщения включать отметку времени (дату и время) и сообщать о скомпрометированных ключах в специальный центр.

Другая угроза состоит в том, что закрытый ключ может быть действительно украден у X в момент времени T. Нарушитель может затем послать сообщение, подписанное подписью X и помеченное временной меткой, которая меньше или равна T.

Проблемы, связанные с *прямой цифровой подписью*, могут быть частично решены с помощью арбитра. Существуют различные схемы с применением ***арбитражной подписи***. В общем виде *арбитражная подпись* выполняется следующим образом. Каждое подписанное сообщение от отправителя X к получателю Y первым делом поступает к арбитру A, который проверяет подпись для данного сообщения. После этого сообщение датируется и посылается к Y с указанием того, что оно было проверено арбитром. Присутствие A решает проблему схем *прямой цифровой подписи*, при которых X может отказаться от сообщения.

Арбитр играет важную роль в подобного рода схемах, и все участники должны ему доверять.

Рассмотрим некоторые возможные технологии *арбитражной цифровой подписи*.

Симметричное шифрование, арбитр видит сообщение:

$$X \longrightarrow A: M \,||\, E_{Kxa}\,[\, ID_X \,||\, H\,(M)\,]$$

Предполагается, что отправитель X и арбитр A разделяют секретный ключ K_{XA} и что A и Y разделяют секретный ключ K_{AY}. X создает сообщение M и вычисляет его хэш-значение H (M). Затем X передает сообщение и подпись A. Подпись состоит из идентификатора X и хэш-значения, все зашифровано с использованием ключа K_{XA}. A дешифрует подпись и проверяет хэш-значение.

$$A \longrightarrow Y: E_{Kay}\,[\, ID_X \,||\, M \,||\, E_{Kxa}\,[ID_X \,||\, H\,(M)\,],\ T\,]$$

Затем A передает сообщение к Y, шифруя его K_{AY}. Сообщение включает ID_X, первоначальное сообщение от X, подпись и отметку времени. Y может дешифровать его для получения сообщения и подписи. Отметка времени информирует Y о том, что данное сообщение не устарело и не является повтором. Y может сохранить M и подпись к нему. В случае спора Y, который утверждает, что получил сообщение M от X, посылает следующее сообщение к A:

$$E_{Kay}\,[\, ID_X \,||\, M \,||\, E_{Kxa}\,[ID_X \,||\, H\,(M)\,]\,]$$

Арбитр использует K_{AY} для получения ID_X, M и подписи, а затем, используя K_{XA}, может дешифровать подпись и проверить хэш-код. По этой схеме Y не может прямо проверить

подпись X; подпись используется исключительно для разрешения споров. Y считает сообщение от X аутентифицированным, потому что оно прошло через A. В данном сценарии обе стороны должны иметь высокую степень доверия к A:

1. X должен доверять A в том, что тот не будет раскрывать K_{XA} и создавать фальшивые подписи в форме $E_{Kка}$ [ID_X || H (M)].
2. Y должен доверять A в том, что он будет посылать E_{Kay} [ID_X || M || E_{Kxa} [ID_X || H (M)]] только в том случае, если хэш-значение является корректным и подпись была создана X.
3. Обе стороны должны доверять A в решении спорных вопросов.

Симметричное шифрование, арбитр не видит сообщение:

Если арбитр не является такой доверенной стороной, то X должен добиться того, чтобы никто не мог подделать его подпись, а Y должен добиться того, чтобы X не мог отвергнуть свою подпись.

Предыдущий сценарий также предполагает, что A имеет возможность читать сообщения от X к Y и что возможно любое подсматривание. Рассмотрим сценарий, который, как и прежде, использует арбитраж, но при этом еще обеспечивает конфиденциальность. В таком случае также предполагается, что X и Y разделяют секретный ключ K_{XY}.

$$X \longrightarrow A:\ ID_X \ ||\ E_{Kxy}\ [M]\ ||\ E_{Kxa}\ [ID_X \ ||\ H\ (E_{KXY}\ [M])\ ]$$

X передает A свой идентификатор, сообщение, зашифрованное K_{XY}, и подпись. Подпись состоит из идентификатора и хэш-значения зашифрованного сообщения, которые зашифрованы с использованием ключа K_{XA}. A дешифрует подпись и проверяет хэш-значение. В данном случае A работает только с зашифрованной версией сообщения, что предотвращает его чтение.

$$A \longrightarrow Y:\ E_{Kay}\ [\ ID_X\ ||\ E_{KXY}[M]\ ||\ E_{Kxa}\ [\ ID_X\ ||\ H\ (\ E_{KXY}\ [M])], \ T]$$

A передает Y все, что он получил от X плюс отметку времени, все шифруя с использованием ключа K_{AY}.

Хотя арбитр и не может прочитать сообщение, он в состоянии предотвратить подделку любого из участников, X или Y. Остается проблема, как и в первом сценарии, что арбитр может сговориться с отправителем, отрицающим подписанное сообщение, или с получателем, для подделки подписи отправителя.

Шифрование открытым ключом, арбитр не видит сообщение:

Все обсуждаемые проблемы могут быть решены с помощью схемы открытого ключа.

$$X \longrightarrow A:\ ID_X \ ||\ E_{KRx}\ [\ ID_X\ ||\ E_{KUy}\ [E_{KRx}\ [M]\ ]\ ]$$

В этом случае X осуществляет двойное шифрование сообщения M, сначала своим закрытым ключом KR_X, а затем открытым ключом Y KU_Y. Получается подписанная секретная версия сообщения. Теперь это подписанное сообщение вместе с идентификатором X шифруется KR_X и вместе с ID_X посылается A. Внутреннее, дважды зашифрованное, сообщение недоступно арбитру (и всем, исключая Y). Однако A может дешифровать внешнюю шифрацию, чтобы убедиться, что сообщение пришло от X (так

как только X имеет KR_X). Проверка дает гарантию, что пара закрытый/открытый ключ законна, и тем самым верифицирует сообщение.

$$A \longrightarrow Y: E_{KRa} [ID_X || E_{KUy} [E_{KRx} [M]] || T]$$

Затем A передает сообщение Y, шифруя его KR_A. Сообщение включает ID_X, дважды зашифрованное сообщение и отметку времени.

Эта схема имеет ряд преимуществ по сравнению с предыдущими двумя схемами. Во-первых, никакая информация не разделяется участниками до начала соединения, предотвращая договор об обмане. Во-вторых, некорректные данные не могут быть посланы, даже если KR_X скомпрометирован, при условии, что не скомпрометирован KR_A. В заключение, содержимое сообщения от X к Y неизвестно ни A, ни кому бы то ни было еще.

Стандарт цифровой подписи DSS

Национальный институт стандартов и технологии США (NIST) разработал федеральный стандарт *цифровой подписи DSS*. Для создания *цифровой подписи* используется алгоритм *DSA* (Digital Signature Algorithm). В качестве хэш-алгоритма стандарт предусматривает использование алгоритма SHA-1 (Secure Hash Algorithm). *DSS* первоначально был предложен в 1991 году и пересмотрен в 1993 году в ответ на публикации, касающиеся безопасности его схемы.

Подход DSS

DSS использует алгоритм, который разрабатывался для использования только в качестве *цифровой подписи*. В отличие от RSA, его нельзя использовать для шифрования или обмена ключами. Тем не менее, это технология открытого ключа.

Рассмотрим отличия подхода, используемого в *DSS* для создания *цифровых подписей*, от применения таких алгоритмов как RSA.

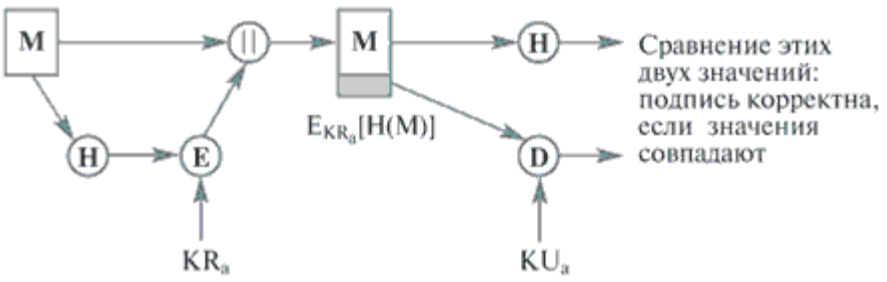

Рис. 10.1. Создание и проверка подписи с помощью алгоритма RSA

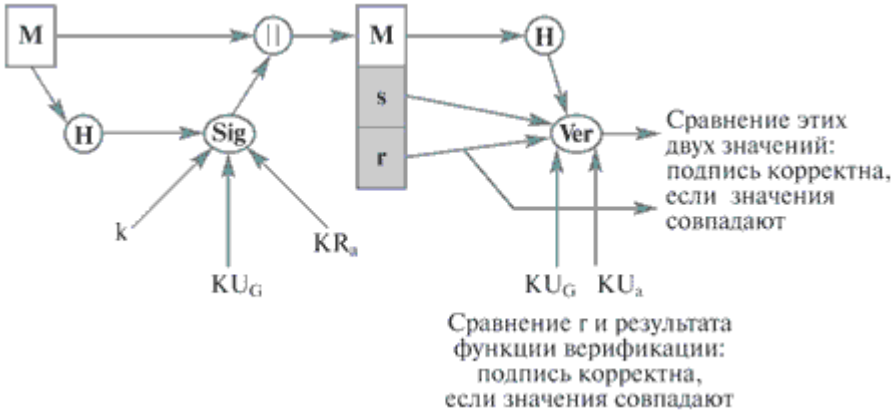

Рис. 10.2. Создание и проверка подписи с помощью стандарта DSS

В подходе RSA подписываемое сообщение подается на вход сильной хэш-функции, которая создает хэш-код фиксированной длины. Для создания подписи этот хэш-код шифруется с использованием закрытого ключа отправителя. Затем сообщение и подпись пересылаются получателю. Получатель вычисляет хэш-код сообщения и проверяет подпись, используя открытый ключ отправителя. Если вычисленный хэш-код равен дешифрованной подписи, то считается, что подпись корректна.

Подход *DSS* также использует сильную хэш-функцию. Хэш-код является входом функции подписи вместе со случайным числом k, созданным для этой конкретной подписи. Функция подписи также зависит от закрытого ключа отправителя KR_a и множества параметров, известных всем участникам. Можно считать, что это множество состоит из глобального открытого ключа KU_G. Результатом является подпись, состоящая из двух компонент, обозначенных как s и r.

Для проверки подписи получатель также создает хэш-код полученного сообщения. Этот хэш-код вместе с подписью является входом в функцию верификации. Функция верификации зависит от глобального открытого ключа KU_G и от открытого ключа отправителя KU_a. Выходом функции верификации является значение, которое должно равняться компоненте r подписи, если подпись корректна. Функция подписи такова, что только отправитель, знающий закрытый ключ, может создать корректную подпись.

Теперь рассмотрим детали алгоритма, используемого в *DSS*.

Алгоритм цифровой подписи

DSS основан на трудности вычисления дискретных логарифмов и базируется на схеме, первоначально представленной ElGamal и Schnorr.

Общие компоненты группы пользователей

Существует три параметра, которые являются открытыми и могут быть общими для большой группы пользователей.

160-битное простое число q, т.е. $2^{159} < q < 2^{160}$.

Простое число p длиной между 512 и 1024 битами должно быть таким, чтобы q было делителем $(p - 1)$, т.е. $2^{L-1} < p < 2^L$, где $512 < L < 1024$ и $(p-1)/q$ является целым.

$g = h^{(p-1)/q} \bmod p$, где h является целым между 1 и $(p-1)$ и g должно быть больше, чем $1,10$.

Зная эти числа, каждый пользователь выбирает закрытый ключ и создает открытый ключ.

Закрытый ключ отправителя

Закрытый ключ x должен быть числом между 1 и $(q-1)$ и должен быть выбран случайно или псевдослучайно.

x - случайное или псевдослучайное целое,
$0 < x < q$,

Открытый ключ отправителя

Открытый ключ вычисляется из закрытого ключа как $y = g^x \bmod p$. Вычислить y по известному x довольно просто. Однако, имея открытый ключ y, вычислительно невозможно определить x, который является дискретным логарифмом y по основанию g.

$$y = g^x \bmod p$$

Случайное число, уникальное для каждой подписи.

k - случайное или псевдослучайное целое, $0 < k < q$, уникальное для каждого подписывания.

Подписывание

Для создания подписи отправитель вычисляет две величины, r и s, которые являются функцией от компонент открытого ключа (p, q, g), закрытого ключа пользователя (x), хэш-кода сообщения $H(M)$ и целого k, которое должно быть создано случайно или псевдослучайно и должно быть уникальным при каждом подписывании.

$$r = (g^k \bmod p) \bmod q$$
$$s = [\, k^{-1} (H(M) + xr) \,] \bmod q$$
$$\text{Подпись} = (r, s)$$

Проверка подписи

Получатель выполняет проверку подписи с использованием следующих формул. Он создает величину v, которая является функцией от компонент общего открытого ключа, открытого ключа отправителя и хэш-кода полученного сообщения. Если эта величина равна компоненте r в подписи, то подпись считается действительной.

$$w = s^{-1} \bmod q$$
$$u_1 = [\, H(M)\, w \,] \bmod q$$
$$u_2 = r\, w \bmod q$$
$$v = [\, (g^{u1}\, y^{u2}) \bmod p \,] \bmod q$$
$$\text{подпись корректна, если } v = r$$

Докажем, что $v = r$ в случае корректной подписи.

Лемма 1. Для любого целого t, если

$$g = h^{(p-1)/q} \bmod p$$
$$\text{то } g^t \bmod p = g^{t \bmod q} \bmod p$$

По теореме Ферма, так как h является взаимнопростым с p, то $h^{p-1} \bmod p = 1$. Следовательно, для любого неотрицательного целого n

$$\begin{aligned} g^{nq} \bmod p &= (h^{(p-1)/q} \bmod p)^{nq} \bmod p \\ &= h^{((p-1)/q)\, nq} \bmod p \\ &= h^{(p-1)n} \bmod p \\ &= ((h^{(p-1)} \bmod p)^n) \bmod p \\ &= 1^n \bmod p = 1 \end{aligned}$$

Таким образом, для неотрицательных целых n и z мы имеем

$$
\begin{aligned}
g^{nq+z} \bmod p &= (g^{nq}\ g^{z}) \bmod p \\
&= ((g^{nq} \bmod p)\ (g^{z} \bmod p)) \bmod p \\
&= g^{z} \bmod p
\end{aligned}
$$

Любое неотрицательное целое t может быть представлено единственным образом как $t = nq + z$, где n и z являются неотрицательными целыми и $0 < z < q$. Таким образом $z = t \bmod q$.

Лемма 2. Для неотрицательных чисел a и b: $g^{(a \bmod q + b \bmod q)} \bmod p = g^{(a+b) \bmod q} \bmod p$.

По лемме 1 мы имеем

$$
\begin{aligned}
&g^{(a \bmod q + b \bmod q)} \bmod p \\
&= g^{(a \bmod q + b \bmod q) \bmod q} \bmod p \\
&= g^{(a + b) \bmod q} \bmod p
\end{aligned}
$$

Лемма 3. $y^{(rw) \bmod q} \bmod p = g^{(xrw) \bmod q} \bmod p$

По определению $y = g^{x} \bmod p$. Тогда:

$$
\begin{aligned}
&y^{(rw) \bmod q} \bmod p \\
&\quad = (g^{x} \bmod p)^{(rw) \bmod q} \bmod p && \text{по правилам} \\
&\quad = g^{x\ ((rw) \bmod q)} \bmod p && \text{модульной арифметики} \\
&\quad = g^{(x\ ((rw \bmod q))) \bmod q} \bmod p && \text{по лемме 1} \\
&\quad = g^{(xrw) \bmod q} \bmod p
\end{aligned}
$$

Лемма 4. $((H(M) + xr)\ w) \bmod q = k$

По определению $s = (k^{-1}\ (H(M) + xr)) \bmod q$. Кроме того, так как q является простым, любое неотрицательное целое меньшее q имеет мультипликативную инверсию. Т.е. $(k\ k^{-1}) \bmod q = 1$. Тогда:

$$
\begin{aligned}
(ks) \bmod q &= (k((k^{-1}(H(M) + xr)) \bmod q)) \bmod q \\
&= (k\ (k^{-1}(H(M) + xr))) \bmod q \\
&= ((k^{k-1}) \bmod q)\ ((H(M) + xr) \bmod q) \bmod q \\
&= (H(M) + xr) \bmod q
\end{aligned}
$$

По определению $w = s^{-1} \bmod q$, следовательно, $(ws) \bmod q = 1$. Следовательно:

$$
\begin{aligned}
&((H(M) + xr)\ w) \bmod q \\
&\quad = (((H(M) + xr) \bmod q)\ (w \bmod q)) \bmod q \\
&\quad = (((ks) \bmod q)\ (w \bmod q)) \bmod q \\
&\quad = (kws) \bmod q \\
&\quad = (k \bmod q)\ ((ws) \bmod q)) \bmod q \\
&\quad = k \bmod q
\end{aligned}
$$

Так как $0 < k < q$, то $k \bmod q = k$.

Теорема. Используя определения для v и r, докажем, что $v=r$.

$$
\begin{aligned}
v &= ((g^{u1}\ y^{u2}) \bmod p) \bmod q \\
&= ((g^{(H(M)\ w) \bmod q}\ y^{(rw) \bmod q}) \bmod p) \bmod q \\
&= ((g^{(H(M)\ w) \bmod q}\ g^{(xrw) \bmod q}) \bmod p) \bmod q \\
&= ((g^{(H(M)\ w) \bmod q + (xrw) \bmod q}) \bmod p) \bmod q \\
&= ((g^{(H(M)\ w + xrw) \bmod q}) \bmod p) \bmod q
\end{aligned}
$$

$$= ((g^{w\ (H(M)\ +\ xr)\ mod\ q})\ mod\ p)\ mod\ q$$
$$= (g^{k\ mod\ p})\ mod\ q$$
$$= r$$

Отечественный стандарт цифровой подписи ГОСТ 3410

В отечественном стандарте *ГОСТ 3410*, принятом в 1994 году, используется алгоритм, аналогичный алгоритму, реализованному в стандарте *DSS*. Оба алгоритма относятся к семейству алгоритмов ElGamal.

В стандарте *ГОСТ 3410* используется хэш-функция ГОСТ 3411, которая создает хэш-код длиной 256 бит. Это во многом обуславливает требования к выбираемым простым числам p и q:

1. p должно быть простым числом в диапазоне
2. $2^{509} < p < 2^{512}$
3. либо
   $2^{1020} < p < 2^{1024}$

4. q должно быть простым числом в диапазоне

   $2^{254} < q < 2^{256}$

q также должно быть делителем $(p-1)$.

Аналогично выбирается и параметр g. При этом требуется, чтобы $g^q \pmod p = 1$.

В соответствии с теоремой Ферма это эквивалентно условию в *DSS*, что $g = h^{(p-1)/q} \bmod p$.

Закрытым ключом является произвольное число x

$$0 < x < q$$

Открытым ключом является число y

$$y = g^x \bmod p$$

Для создания подписи выбирается случайное число k

$$0 < k < q$$

Подпись состоит из двух чисел (r, s), вычисляемых по следующим формулам:

$$r = (g^k \bmod p) \bmod q$$
$$s = (k\ H(M)\ +\ xr) \bmod q$$

Еще раз обратим внимание на отличия *DSS* и *ГОСТ 3410*.

1. Используются разные хэш-функции: в *ГОСТ 3410* применяется отечественный стандарт на хэш-функции ГОСТ 3411, в *DSS* используется SHA-1, которые имеют разную длину хэш-кода. Отсюда и разные требования на длину простого числа q: в *ГОСТ 3410* длина q должна быть от 254 бит до 256 бит, а в *DSS* длина q должна быть от 159 бит до 160 бит.
2. По-разному вычисляется компонента s подписи. В *ГОСТ 3410* компонента s вычисляется по формуле

$$s = (k\ H(M) + xr) \bmod q$$

В *DSS* компонента s вычисляется по формуле

$$s = [k^{-1}\ (H(M) + xr)] \bmod q$$

Последнее отличие приводит к соответствующим отличиям в формулах для проверки подписи.

Получатель вычисляет

$$w = H(M)^{-1} \bmod q$$
$$u_1 = w\ s \bmod q$$
$$u_2 = (q-r)\ w \bmod q$$
$$v = [(g^{u1}\ y^{u2}) \bmod p] \bmod q$$
Подпись корректна, если $v = r$.

Структура обоих алгоритмов довольно интересна. Заметим, что значение r совсем не зависит от сообщения. Вместо этого r есть функция от k и трех общих компонент открытого ключа. Мультипликативная инверсия k ($\bmod p$) (в случае *DSS*) или само значение k (в случае ГОСТ 3410) подается в функцию, которая, кроме того, в качестве входа имеет хэш-код сообщения и закрытый ключ пользователя. Эта функция такова, что получатель может вычислить r, используя входное сообщение, подпись, открытый ключ пользователя и общий открытый ключ.

В силу сложности вычисления дискретных логарифмов нарушитель не может восстановить k из r или x из s.

Другое важное замечание заключается в том, что экспоненциальные вычисления при создании подписи необходимы только для $g^k \bmod p$. Так как это значение от подписываемого сообщения не зависит, оно может быть вычислено заранее. Пользователь может заранее просчитать некоторое количество значений r и использовать их по мере необходимости для подписи документов. Еще одна задача состоит в определении мультипликативной инверсии k^{-1} (в случае *DSS*). Эти значения также могут быть вычислены заранее.

Подписи, созданные с использованием стандартов *ГОСТ 3410* или *DSS*, называются ***рандомизированными***, так как для одного и того же сообщения с использованием одного и того же закрытого ключа каждый раз будут создаваться разные подписи (r,s), поскольку каждый раз будет использоваться новое значение k. Подписи, созданные с применением алгоритма RSA, называются ***детерминированными***, так как для одного и того же сообщения с использованием одного и того же закрытого ключа каждый раз будет создаваться одна и та же подпись.

11. Лекция: Криптография с использованием эллиптических кривых

Математические понятия

Преимущество подхода на основе *эллиптических кривых* в сравнении с задачей факторизации числа, используемой в RSA, или задачей целочисленного логарифмирования, применяемой в алгоритме Диффи-Хеллмана и в DSS, заключается в том, что в данном случае обеспечивается эквивалентная защита при меньшей длине ключа.

В общем случае уравнение ***эллиптической кривой*** E имеет вид:

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

В качестве примера рассмотрим *эллиптическую кривую* E, уравнение которой имеет вид:

$$y^2 + y = x^3 - x^2$$

На этой кривой лежат только четыре точки, координаты которых являются целыми числами. Это точки

$A\ (0,\ 0),\ B\ (1,\ -1),\ C\ (1,\ 0)$ и $D\ (0,\ -1)$

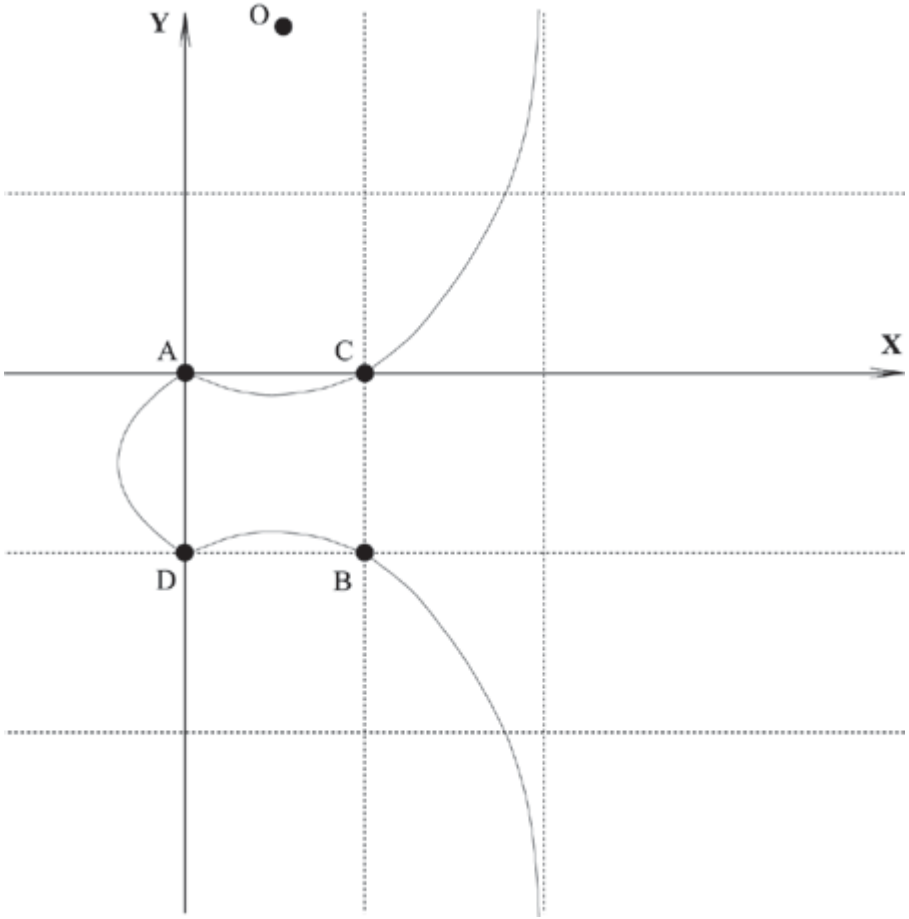

Рис. 1. Пример эллиптической кривой с четырьмя точками

Для определения *операции сложения для точек на эллиптической кривой* сделаем следующие предположения:

- На плоскости существует бесконечно удаленная точка $0 \in E$, в которой сходятся все вертикальные прямые.
- Будем считать, что касательная к кривой пересекает точку касания два раза.
- Если три точки *эллиптической кривой* лежат на прямой линии, то их сумма есть 0.

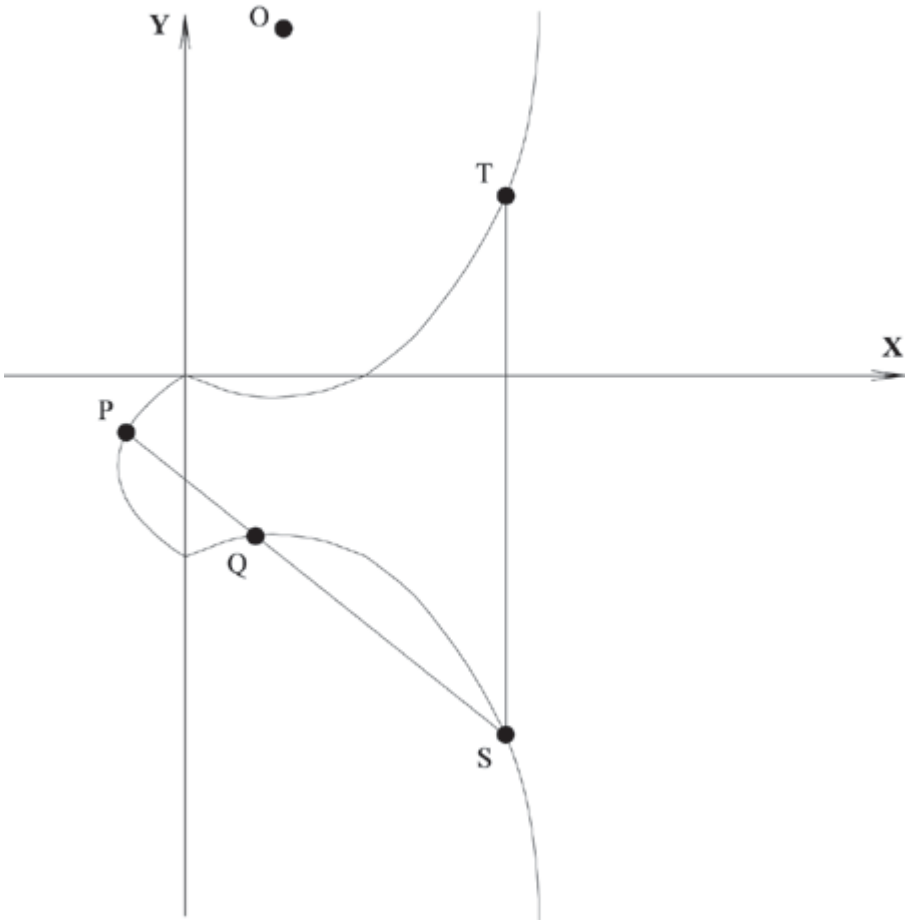

Рис. 2. Сложение точек на эллиптической кривой

Введем следующие правила сложения точек на *эллиптической кривой*:

- Точка `0` выступает в роли *нулевого элемента*. Так, `0 = -0` и для любой точки `P` на *эллиптической кривой* `P + 0 = P`.
- Вертикальная линия пересекает кривую в двух точках с одной и той же координатой `x` - скажем, `S = (x, y)` и `T = (x, -y)`. Эта прямая пересекает кривую и в бесконечно удаленной точке. Поэтому $P_1 + P_2 + 0 = 0$ и $P_1 = -P_2$.
- Чтобы сложить две точки `P` и `Q` (см. рисунок 11.2) с разными координатами `x`, следует провести через эти точки прямую и найти точку пересечения ее с *эллиптической кривой*. Если прямая не является касательной к кривой в точках `P` или `Q`, то существует только одна такая точка, обозначим ее `S`. Согласно нашему предположению

  ```
  P + Q + S = O
  ```

Следовательно,

```
P + Q = -S
или
P + Q = T
```

Если прямая является касательной к кривой в какой-либо из точек `P` или `Q`, то в этом случае следует положить `S = P` или `S = Q` соответственно.

- Чтобы удвоить точку `Q`, следует провести касательную в точке `Q` и найти другую точку пересечения `S` с *эллиптической кривой*. Тогда `Q + Q = 2 × Q = -S`.

Введенная таким образом *операция сложения* подчиняется всем обычным правилам сложения, в частности коммутативному и ассоциативному законам. Умножение точки P *эллиптической кривой* на положительное число k определяется как сумма k точек P.

В криптографии с использованием *эллиптических кривых* все значения вычисляются по модулю p, где p является простым числом. Элементами данной *эллиптической кривой* являются пары неотрицательных целых чисел, которые меньше p и удовлетворяют частному виду *эллиптической кривой*:

$$y^2 \equiv x^3 + ax + b \pmod p$$

Такую кривую будем обозначать $E_p(a,b)$. При этом числа a и b должны быть меньше p и должны удовлетворять условию $4a^3 + 27b^2 \pmod p \neq 0$. Множество точек на *эллиптической кривой* вычисляется следующим образом.

1. Для каждого такого значения x, что $0 \leq x \leq p$, вычисляется $x^3 + ax + b \pmod p$.
2. Для каждого из полученных на предыдущем шаге значений выясняется, имеет ли это значение квадратный корень по модулю p. Если нет, то в $E_p(a,b)$ нет точек с этим значением x. Если корень существует, имеется два значения y, соответствующих операции извлечения квадратного корня (исключением является случай, когда единственным значением оказывается $y = 0$). Эти значения (x,y) и будут точками $E_p(a,b)$.

Множество точек $E_p(a,b)$ обладает следующими свойствами:

1. $P + 0 = P$
2. Если $P = (x,y)$, то $P + (x,-y) = 0$. Точка $(x,-y)$ является отрицательным значением точки P и обозначается $-P$. Заметим, что $(x,-y)$ лежит на *эллиптической кривой* и принадлежит $E_p(a,b)$.
3. Если $P = (x_1,y_1)$ и $Q = (x_2,y_2)$, где $P \neq Q$, то $P + Q = (x_3,y_3)$ определяется по следующим формулам:
4. $x_3 \equiv \lambda^2 - x_1 - x_2 \pmod p$
5. $y_3 \equiv \lambda (x_1 - x_3) - y_1 \pmod p$

где

$$\lambda = \begin{cases} (y_2 - y_1)/(x_2 - x_1), & \text{если } P \neq Q \\ (3x_1^2 + a)/2y_1, & \text{если } P = Q \end{cases}$$

Число λ есть угловой коэффициент секущей, проведенной через точки $P = (x_1, y_1)$ и $Q = (x_2, y_2)$. При $P = Q$ секущая превращается в касательную, чем и объясняется наличие двух формул для вычисления λ.

Задача, которую должен решить в этом случае атакующий, есть своего рода задача ***"дискретного логарифмирования на эллиптической кривой"***, и формулируется она следующим образом. Даны точки P и Q на *эллиптической кривой* $E_p(a,b)$. Необходимо найти коэффициент $k < p$ такой, что

$$P = k \times Q$$

Относительно легко вычислить P по данным k и Q, но довольно трудно вычислить k, зная P и Q.

Рассмотрим три способа использования *эллиптических кривых* в криптографии.

Аналог алгоритма Диффи-Хеллмана обмена ключами

Обмен ключами с использованием *эллиптических кривых* может быть выполнен следующим образом. Сначала выбирается простое число $p \approx 2^{180}$ и параметры a и b для уравнения *эллиптической кривой*. Это задает множество точек $E_p\ (a,b)$. Затем в $E_p\ (a,b)$ выбирается генерирующая точка $G = (x_1, y_1)$. При выборе G важно, чтобы наименьшее значение n, при котором $n \times G = 0$, оказалось очень большим простым числом. Параметры $E_p\ (a,b)$ и G криптосистемы являются параметрами, известными всем участникам.

Обмен ключами между пользователями A и B производится по следующей схеме.

1. Участник A выбирает целое число n_A, меньшее n. Это число является закрытым ключом участника A. Затем участник A вычисляет открытый ключ $P_A = n_A \times G$, который представляет собой некоторую точку на $E_p\ (a,b)$.
2. Точно так же участник B выбирает закрытый ключ n_B и вычисляет открытый ключ P_B.
3. Участники обмениваются открытыми ключами, после чего вычисляют общий секретный ключ K

Участник A:

$$K = n_A \times P_B$$

Участник B:

$$K = n_B \times P_A$$

Следует заметить, что общий секретный ключ представляет собой пару чисел. Если данный ключ предполагается использовать в качестве сеансового ключа для алгоритма симметричного шифрования, то из этой пары необходимо создать одно значение.

Алгоритм цифровой подписи на основе эллиптических кривых ECDSA

Алгоритм ECDSA (Elliptic Curve Digest Signature Algorithm) принят в качестве стандартов ANSI X9F1 и IEEE P1363.

Создание ключей:

1. Выбирается *эллиптическая кривая* $E_p\ (a,b)$. Число точек на ней должно делиться на большое целое n.
2. Выбирается точка $P \in E_p\ (a,b)$.
3. Выбирается случайное число $d \in [1,\ n-1]$.
4. Вычисляется $Q = d \times P$.
5. Закрытым ключом является d, открытым ключом - $(E,\ P,\ n,\ Q)$.

Создание подписи:

1. Выбирается случайное число $k \in [1, n-1]$.
2. Вычисляется
3. $k \times P = (x_1, y_1)$
4. и
5. $r = x_1 \pmod n$.

   Проверяется, чтобы r не было равно нулю, так как в этом случае подпись не будет зависеть от закрытого ключа. Если $r = 0$, то выбирается другое случайное число k.

6. Вычисляется
7. $k^{-1} \bmod n$
8. Вычисляется
9. $s = k^{-1} (H(M) + dr) \pmod n$

   Проверяется, чтобы s не было равно нулю, так как в этом случае необходимого для проверки подписи числа $s^{-1} \bmod n$ не существует. Если $s = 0$, то выбирается другое случайное число k.

10. Подписью для сообщения M является пара чисел (r,s).

Проверка подписи:

1. Проверить, что целые числа r и s принадлежат диапазону чисел $[0, n-1]$. В противном случае результат проверки отрицательный, и подпись отвергается.
2. Вычислить $w = s^{-1} \pmod n$ и $H(M)$
3. Вычислить
4. $u_1 = H(M) \ w \pmod n$
5. $u_2 = rw \pmod n$
6. Вычислить
7. $u_1P + u_2Q = (x_0, y_0)$
8. $v = x_0 \pmod n$
9. Подпись верна в том и только том случае, когда $v = r$

Шифрование/дешифрование с использованием эллиптических кривых

Рассмотрим самый простой подход к шифрованию/дешифрованию с использованием *эллиптических кривых*. Задача состоит в том, чтобы зашифровать сообщение M, которое может быть представлено в виде точки на эллиптической кривой $P_m \ (x,y)$.

Как и в случае обмена ключом, в системе шифрования/дешифрования в качестве параметров рассматривается *эллиптическая кривая* $E_p \ (a,b)$ и точка G на ней. Участник B выбирает закрытый ключ n_B и вычисляет открытый ключ $P_B = n_B \times G$. Чтобы зашифровать сообщение P_m используется открытый ключ получателя $B \ \ P_B$. Участник A выбирает случайное целое положительное число k и вычисляет зашифрованное сообщение C_m, являющееся точкой на *эллиптической кривой*.

$C_m = \{k \times G, \ P_m + k \times P_B\}$

Чтобы дешифровать сообщение, участник B умножает первую координату точки на свой закрытый ключ и вычитает результат из второй координаты:

$P_m + k \times P_B - n_B \times (k \times G) =$
$P_m + k \times (n_B \times G) - n_B \times (k \times G) = P_m$

Участник A зашифровал сообщение P_m добавлением к нему kxP_B. Никто не знает значения k, поэтому, хотя P_B и является открытым ключом, никто не знает $k \times P_B$. Противнику для восстановления сообщения придется вычислить k, зная G и $k \times G$. Сделать это будет нелегко.

Получатель также не знает k, но ему в качестве подсказки посылается $k \times G$. Умножив $k \times G$ на свой закрытый ключ, получатель получит значение, которое было добавлено отправителем к незашифрованному сообщению. Тем самым получатель, не зная k, но имея свой закрытый ключ, может восстановить незашифрованное сообщение.

12. Лекция: Алгоритмы обмена ключей и протоколы аутентификации

Алгоритмы распределения ключей с использованием третьей доверенной стороны

Понятие мастер-ключа

При симметричном шифровании два участника, которые хотят обмениваться конфиденциальной информацией, должны иметь один и тот же ключ. Частота изменения ключа должна быть достаточно большой, чтобы у противника не хватило времени для полного перебора ключа. Следовательно, сила любой криптосистемы во многом зависит от технологии распределения ключа. Этот термин означает передачу ключа двум участникам, которые хотят обмениваться данными, таким способом, чтобы никто другой не мог ни подсмотреть, ни изменить этот ключ. Для двух участников A и B распределение ключа может быть выполнено одним из следующих способов.

1. Ключ может быть создан A и физически передан B.
2. *Третья сторона* может создать ключ и физически передать его A и B.
3. A и B имеют предварительно созданный и недолго используемый ключ, один участник может передать новый ключ другому, применив для шифрования старый ключ.
4. Если A и B каждый имеют безопасное соединение с третьим участником C, C может передать ключ по этому безопасному каналу A и B.

Первый и второй способы называются ручным распределением ключа. Это самые надежные способы распределения ключа, однако во многих случаях пользоваться ими неудобно и даже невозможно. В распределенной системе любой хост или сервер должен иметь возможность обмениваться конфиденциальной информацией со многими аутентифицированными хостами и серверами. Таким образом, каждый хост должен иметь набор ключей, поддерживаемый динамически. Проблема особенно актуальна в больших распределенных системах.

Количество требуемых ключей зависит от числа участников, которые должны взаимодействовать. Если выполняется шифрование на сетевом или IP-уровне, то ключ необходим для каждой пары хостов в сети. Таким образом, если есть N хостов, то необходимое число ключей $[N (N - 1)]/2$. Если шифрование выполняется на прикладном уровне, то ключ нужен для каждой пары прикладных процессов, которых гораздо больше, чем хостов.

Третий способ распределения ключей может применяться на любом уровне стека протоколов, но если атакующий получает возможность доступа к одному ключу, то вся последовательность ключей будет раскрыта. Более того, все равно должно быть проведено первоначальное распространение большого количества ключей.

Поэтому в больших автоматизированных системах широко применяются различные варианты четвертого способа. B этой схеме предполагается существование так

называемого центра распределения ключей (Key Destribution Centre - ***KDC***), который отвечает за распределение ключей для хостов, процессов и приложений. Каждый участник должен разделять уникальный ключ с *KDC*.

Использование центра распределения ключей основано на использовании иерархии ключей. Как минимум используется два типа ключей: *мастер-ключи* и *ключи сессии*.

Для обеспечения конфиденциальной связи между конечными системами используется временный ключ, называемый ***ключом сессии***. Обычно *ключ сессии* используется для шифрования транспортного соединения и затем уничтожается. Каждый *ключ сессии* должен быть получен по сети из центра распределения ключей. *Ключи сессии* передаются в зашифрованном виде, используя *мастер-ключ*, который разделяется между центром распределения ключей и конечной системой.

Эти *мастер-ключи* также должны распределяться некоторым безопасным способом. Однако при этом существенно уменьшается количество ключей, требующих ручного распределения. Если существует `N` участников, которые хотят устанавливать соединения, то в каждый момент времени необходимо `[N (N - 1)]/2` *ключей сессии*. Но требуется только `N` *мастер-ключей*, по одному для каждого участника.

Время жизни *ключа сессии* как правило равно времени жизни самой сессии.

Чем чаще меняются *ключи сессии*, тем более безопасными они являются, так как противник имеет меньше времени для взламывания данного *ключа сессии*. С другой стороны, распределение *ключей сессии* задерживает начало любого обмена и загружает сеть. Политика безопасности должна сбалансировать эти условия для определения оптимального времени жизни конкретного *ключа сессии*.

Если соединение имеет долгое время жизни, то должна существовать возможность периодически менять *ключ сессии*.

Для протоколов, не поддерживающих соединение, таких как протокол, ориентированный на транзакции, нет явной инициализации или прерывания соединения. Следовательно, неясно, как часто надо менять *ключ сессии*. Большинство подходов основывается на использовании нового *ключа сессии* для каждого нового обмена. Наиболее часто применяется стратегия использования *ключа сессии* только для фиксированного периода времени или только для определенного количества транзакций.

Протоколы аутентификации

Рассмотрим основные протоколы, обеспечивающие как взаимную *аутентификацию* участников, так и *аутентификацию* только одного из участников.

Взаимная аутентификация

Данные протоколы применяются для взаимной *аутентификации* участников и для обмена *ключом сессии*.

Основной задачей таких протоколов является обеспечение конфиденциального распределения *ключа сессии* и гарантирование его своевременности, то есть протокол не должен допускать повторного использования старого *ключа сессии*. Для обеспечения конфиденциальности *ключи сессии* должны передаваться в зашифрованном виде. Вторая задача, обеспечение своевременности, важна, потому что существует угроза перехвата передаваемого сообщения и повторной его пересылки. Такие повторения в худшем случае могут позволять взломщику использовать скомпрометированный *ключ сессии*, при этом успешно подделываясь под другого

участника. Успешное повторение может, как минимум, разорвать операцию *аутентификации* участников.

Такие повторы называются *replay-атаками*. Рассмотрим возможные примеры подобных *replay-атак*:

1. Простое повторение: противник просто копирует сообщение и повторяет его позднее.
2. Повторение, которое не может быть определено: противник уничтожает исходное сообщение и посылает скопированное ранее сообщение.

Один из возможных подходов для предотвращения *replay-атак* мог бы состоять в присоединении последовательного номера (sequence number) к каждому сообщению, используемому в *аутентификационном* обмене. Новое сообщение принимается только тогда, когда его последовательный номер правильный. Трудность данного подхода состоит в том, что каждому участнику требуется поддерживать значения sequence number для каждого участника, с которым он взаимодействует в данный момент. Поэтому обычно sequence number не используются для *аутентификации* и обмена ключами. Вместо этого применяется один из следующих способов:

1. *Отметки времени*: участник A принимает сообщение как не устаревшее только в том случае, если оно содержит *отметку времени*, которая, по мнению A, соответствует текущему времени. Этот подход требует, чтобы часы всех участников были синхронизированы.
2. Запрос/ответ: участник A посылает в запросе к B случайное число (***nonce*** - number only once) и проверяет, чтобы ответ от B содержал корректное значение этого *nonce*.

Считается, что подход с *отметкой времени* не следует использовать в приложениях, ориентированных на соединение, потому что это технически трудно, так как таким протоколам, кроме поддержки соединения, необходимо будет поддерживать синхронизацию часов различных процессоров. При этом возможный способ осуществления успешной атаки может возникнуть, если временно будет отсутствовать синхронизация часов одного из участников. В результате различной и непредсказуемой природы сетевых задержек распределенные часы не могут поддерживать точную синхронизацию. Следовательно, процедуры, основанные на любых *отметках времени*, должны допускать окно времени, достаточно большое для приспособления к сетевым задержкам, и достаточно маленькое для минимизации возможности атак.

С другой стороны, подход запрос/ответ не годится для приложений, не устанавливающих соединения, так как он требует предварительного рукопожатия перед началом передач, тем самым отвергая основное свойство транзакции без установления соединения. Для таких приложений доверие к некоторому безопасному серверу часов и постоянные попытки каждой из частей синхронизировать свои часы с этим сервером может быть оптимальным подходом.

Использование симметричного шифрования

Для обеспечения *аутентификации* и распределения *ключа сессии* часто используется двухуровневая иерархия ключей симметричного шифрования. В общих чертах эта стратегия включает использование доверенного центра распределения ключей (*KDC*). Каждый участник разделяет секретный ключ, называемый также *мастер-ключом*, с *KDC*. *KDC* отвечает за создание ключей, называемых *ключами сессии*, и за распределение этих ключей с использованием *мастер-ключей*. *Ключи сессии* применяются в течение короткого времени для шифрования только данной сессии между двумя участниками.

Большинство алгоритмов распределения секретного ключа с использованием *KDC*, включает также возможность *аутентификации* участников.

Протокол Нидхэма и Шредера

Предполагается, что секретные *мастер-ключи* K_A и K_B разделяют соответственно A и *KDC* и B и *KDC*. Целью протокола является безопасное распределение *ключа сессии* KS между A и B. Протокол представляет собой следующую последовательность шагов:

1. $A \longrightarrow KDC: \quad ID_A \;||\; ID_B \;||\; N_1$
2. $KDC \longrightarrow A: \quad E_{Ka}\ [K_S \;||\; ID_B \;||\; N_1 \;||\; E_{Kb}\ [K_S \;||\; ID_A]\ ]$
3. $A \longrightarrow B: \quad E_{Kb}\ [K_S \;||\; ID_A]$
4. $B \longrightarrow A: \quad E_{KS}\ [N_2]$
5. $A \longrightarrow B: \quad E_{KS}\ [f\ (N_2)]$

1. A запрашивает у *KDC* *ключ сессии* для установления защищенного соединения с B. Сообщение включает идентификацию A и B и уникальный идентификатор данной транзакции, который обозначен как N_1 и называется *nonce*. *Nonce* может быть временной меткой, счетчиком или случайным числом; минимальное требование состоит в том, чтобы он отличался для каждого запроса. Кроме того, для предотвращения подделки желательно, чтобы противнику было трудно предугадать *nonce*. Таким образом, случайное число является лучшим вариантом для *nonce*.
2. *KDC* отвечает сообщением, зашифрованным ключом K_A. Таким образом, только A может расшифровать сообщение, и A уверен, что оно получено от *KDC*, так как предполагается, что кроме A и *KDC* этот ключ не знает никто. Это сообщение включает следующие элементы, предназначенные для A:
   - Одноразовый *ключ сессии*.
   - Идентификатор B.
   - *nonce*, который идентифицирует данную сессию .

   A должен убедиться, что полученный *nonce* равен значению *nonce* из первого запроса. Это доказывает, что ответ от *KDC* не был модифицирован при пересылке и не является повтором некоторого предыдущего запроса. Кроме того, сообщение включает два элемента, предназначенные для B:

   - Одноразовый *ключ сессии* K_S.
   - Идентификатор A ID_A.

   Эти два последних элемента шифруются *мастер-ключом*, который *KDC* разделяет с B. Они посылаются B при установлении соединения и доказывают идентификацию A.

3. A сохраняет у себя *ключ сессии* и передает B информацию от *KDC*, предназначенную B: $E_{Kb}\ [K_S \;||\; ID_A]$. Так как эта информация зашифрована K_B, она защищена от просмотра. Теперь B знает *ключ сессии* (K_S), знает, что другим участником является A, (ID_A) и что начальная информация передана от *KDC*, т.к. она зашифрована с использованием K_B.

   В этой точке *ключ сессии* безопасно передан от A к B, и они могут начать безопасный обмен. Тем не менее, существует еще два дополнительных шага:

4. Используя созданный *ключ сессии*, B пересылает A *nonce* N_2.
5. Также используя K_S, A отвечает $f\ (N_2)$, где f - функция, выполняющая некоторую модификацию N_2.

Эти шаги гарантируют B, что сообщение, которое он получил, не изменено и не является повтором предыдущего сообщения.

Заметим, что реальное распределение ключа включает только шаги 1 - 3, а шаги 4 и 5, как и 3, выполняют функцию *аутентификации*.

A безопасно получает *ключ сессии* на шаге 2. Сообщение на шаге 3 может быть дешифровано только B. Шаг 4 отражает знание B ключа K_S, и шаг 5 гарантирует B знание участником A ключа K_S и подтверждает, что это не устаревшее сообщение, так как используется *nonce* N_2. Шаги 4 и 5 призваны предотвратить общий тип *replay-атак*. B частности, если противник имеет возможность захватить сообщение на шаге 3 и повторить его, то это должно привести к разрыву соединения.

Разрывая рукопожатие на шагах 4 и 5, протокол все еще уязвим для некоторых форм атак повторения. Предположим, что противник X имеет возможность скомпрометировать старый *ключ сессии*. Маловероятно, чтобы противник мог сделать больше, чем просто копировать сообщение шага 3. Потенциальный риск состоит в том, что X может заставить взаимодействовать A и B, используя старый *ключ сессии*. Для этого X просто повторяет сообщение шага 3, которое было перехвачено ранее и содержит скомпрометированный *ключ сессии*. Если B не запоминает идентификацию всех предыдущих *ключей сессий* с A, он не сможет определить, что это повтор. Далее X должен перехватить сообщение рукопожатия на шаге 4 и представиться A в ответе на шаге 5.

Протокол Деннинга

Деннинг предложил преодолеть эту слабость модификацией протокола Нидхэма и Шредера, которая включает дополнительную *отметку времени* на шагах 2 и 3:

1. A -> KDC: $ID_A \,||\, ID_B$
2. KDC -> A: $E_{Ka}\,[K_S \,||\, ID_B \,||\, T \,||$
   $E_{Kb}\,[K_S \,||\, ID_A \,||\, T]\,]$
3. A -> B: $E_{Kb}\,[K_S \,||\, ID_A \,||\, T]$
4. B -> A: $E_{KS}\,[N_1]$
5. A -> B: $E_{KS}\,[f\ (N_1)]$

T - это *отметка времени*, которая гарантирует A и B, что *ключ сессии* является только что созданным. Таким образом, и A, и B знают, что распределенный ключ не является старым. A и B могут верифицировать *временную отметку* проверкой, что

$$|Clock - T| < \Delta t_1 + \Delta t_2$$

где $\Delta\ t_1$ - оцениваемое нормальное расхождение между часами *KDC* и локальными часами (у A или B) и t_2 - ожидаемая сетевая задержка времени. Каждый участник может установить свои часы, ориентируясь на определенный доверенный источник. Поскольку *временная отметка* T шифруется с использованием секретных *мастер-ключей*, взломщик, даже зная старый *ключ сессии*, не сможет достигнуть цели повторением шага 3 так, чтобы B не заметил искажения времени.

Шаги 4 и 5 не были включены в первоначальное представление, но были добавлены позднее. Эти шаги подтверждают A, что B получил *ключ сессии*.

Протокол Деннинга обеспечивает большую степень безопасности по сравнению с протоколом Нидхэма и Шредера. Однако данная схема требует доверия к часам, которые должны быть синхронизированы в сети. B этом есть определенный риск, который состоит в том, что распределенные часы могут рассинхронизироваться в результате диверсии или повреждений. Проблема возникает, когда часы отправителя

спешат по отношению к часам получателя. В этом случае противник может перехватить сообщение от отправителя и повторить его позднее, когда *отметка времени* в сообщении станет равной времени на узле получателя. Это повторение может иметь непредсказуемые последствия.

Один способ вычисления атак повторения состоит в требовании, чтобы участники регулярно сверяли свои часы с часами *KDC*. Другая альтернатива, при которой нет необходимости всем синхронизировать часы, состоит в доверии протоколам рукопожатия, использующим *nonce*.

Протокол аутентификации с использованием билета

Данный протокол пытается преодолеть проблемы, возникшие в предыдущих двух протоколах. Он выглядит следующим образом:

$$
\begin{aligned}
&1.\ A \rightarrow B: && ID_A \,||\, N_a \\
&2.\ B \rightarrow KDC: && ID_B \,||\, N_b \,||\, E_{Kb}\,[ID_A \,||\, N_a \,||\, T_b] \\
&3.\ KDC \rightarrow A: && EK_a\,[ID_B \,||\, N_a \,||\, K_S \,||\, T_b] \,||\, \\
& && \quad E_{Kb}\,[ID_A \,||\, K_S \,||\, T_b] \,||\, N_b \\
&4.\ A \rightarrow B: && E_{Kb}\,[ID_A \,||\, K_S \,||\, T_b] \,||\, E_{KS}\,[N_b]
\end{aligned}
$$

1. A инициализирует *аутентификационный* обмен созданием *nonce* N_a и посылкой его и своего идентификатора к B в незашифрованном виде. Этот *nonce* вернется к A в зашифрованном сообщении, включающем *ключ сессии*, гарантируя A, что *ключ сессии* не старый.
2. B сообщает *KDC*, что необходим *ключ сессии*. Это сообщение к KDC включает идентификатор B и *nonce* N_b. Данный *nonce* вернется к B в зашифрованном сообщении, которое включает *ключ сессии*, гарантируя B, что *ключ сессии* не устарел. Сообщение B к *KDC* также включает блок, зашифрованный секретным ключом, разделяемым B и *KDC*. Этот блок используется для указания *KDC*, когда заканчивается время жизни данного *ключа сессии*. Блок также специфицирует намеченного получателя и содержит *nonce*, полученный от A. Этот блок является своего рода "верительной грамотой" или "билетом" для A.
3. *KDC* получил *nonces* от A и B и блок, зашифрованный секретным ключом, который B разделяет с *KDC*. Блок служит билетом, который может быть использован A для последующих *аутентификаций*. *KDC* также посылает A блок, зашифрованный секретным ключом, разделяемым A и *KDC*. Этот блок доказывает, что B получил начальное сообщение A (ID_B), что в нем содержится допустимая *отметка времени* и нет повтора (N_a). Этот блок обеспечивает A *ключом сессии* (K_S) и устанавливает ограничение времени на его использование (T_b).
4. A посылает полученный билет B вместе с *nonce* B, зашифрованным *ключом сессии*. Этот билет обеспечивает B *ключом сессии*, который тот использует для дешифрования и проверки *nonce*. Тот факт, что *nonce* B расшифрован *ключом сессии*, доказывает, что сообщение пришло от A и не является повтором.

Данный протокол аутентифицирует A и B и распределяет *ключ сессии*. Более того, протокол предоставляет в распоряжение A билет, который может использоваться для его последующей *аутентификации*, исключая необходимость повторных контактов с *аутентификационным* сервером. Предположим, что A и B установили сессию с использованием описанного выше протокола и затем завершили эту сессию. Впоследствии, но до истечения лимита времени, установленного протоколом, A может создать новую сессию с B. Используется следующий протокол:

$$
\begin{aligned}
&1.\ A \rightarrow B: && E_{Kb}\,[ID_A \,||\, K_S \,||\, T_b],\ N_a' \\
&2.\ B \rightarrow A: && N_b',\ E_S\,[N_a'] \\
&3.\ A \rightarrow B: && E_S\,[N_b']
\end{aligned}
$$

Когда B получает сообщение на шаге 1, он проверяет, что билет не просрочен. Заново созданные *nonces* N_a' и N_b' гарантируют каждому участнику, что не было атак повтора. Время T_b является временем относительно часов B. Таким образом, эта временная метка не требует синхронизации, потому что B проверяет только им самим созданные *временные отметки*.

Использование шифрования с открытым ключом
Протокол аутентификации с использованием аутентификационного сервера.

Рассмотрим протокол, использующий *отметки времени* и *аутентификационный* сервер:

$$
\begin{aligned}
&1.\ A \rightarrow AS: && ID_A \,||\, ID_B \\
&2.\ AS \rightarrow A: && E_{KRas}\,[ID_A \,||\, KU_a \,||\, T] \,||\, \\
& && E_{KRas}\,[ID_B \,||\, KU_b \,||\, T] \\
&3.\ A \rightarrow B: && E_{KRas}\,[ID_A \,||\, KU_a \,||\, T] \,||\, \\
& && E_{KRas}\,[ID_B \,||\, KU_b \,||\, T] \,||\, \\
& && E_{KUb}\,[E_{KRa}\,[K_S \,||\, T]]
\end{aligned}
$$

В данном случае *третья доверенная сторона* является просто *аутентификационным* сервером AS, потому что *третья сторона* не создает и не распределяет секретный ключ. AS просто обеспечивает сертификацию открытых ключей участников. *Ключ сессии* выбирается и шифруется A, следовательно, не существует риска, что AS взломают и заставят распределять скомпрометированные *ключи сессии*. *Отметки времени* защищают от повтора скомпрометированных *ключей сессии*.

Данный протокол компактный, но, как и прежде, требует синхронизации часов.

Протокол аутентификации с использованием KDC

Другой подход использует *nonces*. Этот протокол состоит из следующих шагов:

$$
\begin{aligned}
&1.\ A \rightarrow KDC: && ID_A \,||\, ID_B \\
&2.\ KDC \rightarrow A: && E_{KRkdc}\,[ID_B \,||\, KU_b] \\
&3.\ A \rightarrow B: && E_{KUb}\,[N_a \,||\, ID_A] \\
&4.\ B \rightarrow KDC: && ID_B \,||\, ID_A \,||\, E_{KUkdc}\,[N_a] \\
&5.\ KDC \rightarrow B: && E_{KRkdc}\,[ID_A \,||\, KU_a] \,||\, \\
& && E_{KUb}\,[E_{KRkdc}\,[N_a \,||\, K_S \,||\, ID_B]] \\
&6.\ B \rightarrow A: && E_{KUa}\,[E_{KRkdc}\,[N_a \,||\, K_S \,||\, ID_B] \,||\, N_b] \\
&7.\ A \rightarrow B: && E_{KS}\,[N_b]
\end{aligned}
$$

На первом шаге A информирует *KDC*, что хочет установить безопасное соединение с B. *KDC* возвращает A сертификат открытого ключа B (шаг 2). Используя открытый ключ B, A информирует B о создании защищенного соединения и посылает *nonce* N_a (шаг 3). На 4-м шаге B спрашивает *KDC* о сертификате открытого ключа A и запрашивает *ключ сессии*. B включает *nonce* A, чтобы *KDC* мог пометить *ключ сессии* этим *nonce*. *Nonce* защищен использованием открытого ключа *KDC*. На 5-м шаге *KDC* возвращает B сертификат открытого ключа A плюс информацию $\{N_a, K_S, ID_B\}$. Эта информация означает, что K_S является секретным ключом, созданным *KDC* в интересах B и связан с N_a. Связывание K_S и N_a гарантирует A, что K_S не устарел. Эта тройка шифруется с использованием закрытого ключа *KDC*, это гарантирует B, что тройка действительно получена от *KDC*. Она также шифруется с использованием открытого ключа B, чтобы никто другой не мог подсмотреть *ключ сессии* и использовать эту тройку для установления соединения с A. На шаге 6 тройка $\{N_a, K_S, ID_B\}$, зашифрованная закрытым ключом *KDC*, передается A вместе с *nonce* N_b, созданным B. Все сообщение шифруется открытым ключом A. A восстанавливает *ключ сессии* K_S, использует его для шифрования N_b, который возвращает B. Это последнеее сообщение гарантирует B, что A знает *ключ сессии*.

Это достаточно безопасный протокол при различного рода атаках. Однако авторы предложили пересмотренную версию данного алгоритма:

1. A -> KDC: $ID_A \| ID_B$
2. KDC -> A: $E_{KRauth}[ID_B \| KU_b]$
3. A -> B: $E_{KUb}[N_a \| ID_A]$
4. B -> KDC: $ID_B \| ID_A \| E_{KUauth}[N_a]$
5. KDC -> B: $E_{KRauth}[ID_A \| KU_a] \| E_{KUb}[E_{KRauth}[N_a \| K_S \| ID_A \| ID_B]]$
6. B -> A: $E_{KUa}[E_{KRauth}[N_a \| K_S \| ID_A \| ID_B] \| N_b]$
7. A -> B: $E_{KS}[N_b]$

Добавляется идентификатор A ID_A к данным, зашифрованных с использованием закрытого ключа *KDC* на шагах 5 и 6 для идентификации обоих участников сессии. Это включение ID_A приводит к тому, что значение *nonce* N_a должно быть уникальным только среди всех *nonces*, созданных A, но не среди *nonces*, созданных всеми участниками. Таким образом, пара $\{ID_A, N_a\}$ уникально идентифицирует соединение, созданное A.

Односторонняя аутентификация

Существует специфическое приложение - электронная почта, для которого шифрование также имеет большое значение. Особенность e-mail состоит в том, что отправителю и получателю нет необходимости быть на связи в одно и то же время. Вместо этого сообщения направляются в почтовый ящик получателя, где они хранятся до тех пор, пока у того не появится возможность получить их.

Заголовок сообщения должен быть незашифрованным, чтобы сообщение могло пересылаться протоколами e-mail, такими как X.400 или SMTP. Однако желательно, чтобы протоколы управления почтой не имели бы доступа к самому сообщению. Соответственно, e-mail сообщение должно быть зашифровано так, чтобы система управления почтой могла бы не знать ключ шифрования.

Вторым требованием является *аутентификация* сообщения. Обычно получателю нужна определенная гарантия того, что сообщение пришло от законного отправителя.

Использование симметричного шифрования

При использовании симметричного шифрования сценарий централизованного распределения ключей в полном объеме непригоден. Эти схемы требуют, чтобы в двух заключительных шагах отправитель посылал запрос получателю, ожидая ответа с созданным *ключом сессии*, и только после этого отправитель может послать сообщение.

С учетом перечисленных ограничений протоколы использования *KDC* являются возможными кандидатами для шифрования электронной почты. Для того чтобы избежать требования к получателю B находиться на связи в то же самое время, когда и отправитель A, шаги 4 и 5 должны быть опущены. Таким образом, остается последовательность шагов:

1. A -> KDC: $ID_A \| ID_B \| N_1$
2. KDC -> A: $E_{Ka}[K_S \| ID_B \| N_1 \| E_{Kb}[K_S \| ID_A]]$
3. A -> B: $E_{Kb}[K_S , ID_A] \| E_{KS}[M]$

Данный подход гарантирует, что только требуемый получатель сообщения сможет прочитать его. Это также обеспечивает определенный уровень *аутентификации*, что отправителем является A. Очевидно, что протокол не защищает от атак повтора. Некоторая мера защиты может быть обеспечена включением *отметки времени* в

сообщение. Однако поскольку существуют потенциальные задержки в процессе передачи e-mail сообщений, такие *временные отметки* имеют ограниченный срок действия.

Использование шифрования с открытым ключом

При использовании шифрования с открытым ключом требуется, чтобы отправитель знал открытый ключ получателя (для обеспечения конфиденциальности), получатель знал открытый ключ отправителя (для обеспечения *аутентификации*), или и то, и другое (для обеспечения конфиденциальности и *аутентификации*).

Если требуется конфиденциальность, то может быть использована следующая схема:

$A \rightarrow B: \quad E_{KUb}\ [K_S]\ ||\ E_{KS}\ [M]$

В этом случае сообщение шифруется одноразовым секретным ключом. A шифрует этот одноразовый ключ открытым ключом B. Только B имеет соответствующий закрытый ключ для получения одноразового ключа и использования этого ключа для дешифрования сообщения. Эта схема более эффективна, чем простое шифрование всего сообщения открытым ключом B.

Если требуется *аутентификация*, то цифровая подпись может быть создана по такой схеме:

$A \rightarrow B: \quad M\ ||\ E_{KRa}\ [H\ (M)]$

Этот метод гарантирует, что A не сможет впоследствии отвергнуть полученное сообщение. Однако данная технология открыта для другого типа подделок. Например, можно получить доступ к почтовой очереди перед доставкой, вырезать подпись отправителя, вставить свою и опять поставить сообщение в очередь на доставку.

Чтобы этого не допустить, сообщение и подпись можно зашифровать ключом симметричного шифрования, который в свою очередь шифруется открытым ключом получателя:

$A \rightarrow B: \quad E_{KS}\ [M\ ||\ E_{KRa}\ [H\ (M)]]\ ||\ E_{KUb}\ [K_S]$

Следующая схема не требует, чтобы B знал открытый ключ A. B этом случае должен использоваться сертификат открытого ключа:

$$A \rightarrow B: \quad M\ ||\ E_{KRa}\ [H\ (M)\ ]\ ||\ E_{KRas}\ [T\ ||\ ID_A\ ||\ KU_a]$$

В конце сообщения A посылает B подпись, зашифрованную закрытым ключом A, и сертификат A, зашифрованный закрытым ключом *аутентификационного* сервера. Получатель применяет сертификат для получения открытого ключа отправителя и затем использует открытый ключ отправителя для проверки самого сообщения. Конфиденциальность может быть добавлена аналогично предыдущей схеме.

Литература по курсу

- 1. Edward Roback, Elaine Barker, James Foti, James Nechvatal, Lawrence Bassham, Morris Dworkin, William Burr

  Report on the Development of the Advanced Encryption Standard (AES)

  Computer Security Division Information Technology Laboratory National Institute of Standards and Technology Technology Administration U.S. Department of Commerce. 2000г. 116с.

- 2. Государственный Стандарт Российской Федерации

  ИНФОРМАЦИОННАЯ ТЕХНОЛОГИЯ. КРИПТОГРАФИЧЕСКАЯ ЗАЩИТА ИНФОРМАЦИИ. Процедуры выработки и проверки электронной цифровой подписи на базе асимметричного криптографического алгоритма

  1994г.
- 3. Государственный Стандарт Российской Федерации

  ИНФОРМАЦИОННАЯ ТЕХНОЛОГИЯ. КРИПТОГРАФИЧЕСКАЯ ЗАЩИТА ИНФОРМАЦИИ. Функция хэширования

  1994г.
- 4. RFC 3280

  Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile

  2002г. 129с.
- 5. RFC 3281

  An Internet Attribute Certificate Profile for Authorization

  2002г. 40с.
- 6. RFC 2510

  Internet X.509 Public Key Infrastructure Certificate Management Protocols

  1999г. 72с.
- 7. RFC 2511

  Internet X.509 Certificate Request Message Format

  1999г. 25с.
- 8. RFC 3369

  Cryptographic Message Syntax

  2002г. 60с.
- 9. RFC 2560

  X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP

  1999г. 23с.
- 10. RFC 2797

  Certificate Management Messages over CMS

  2000г. 47с.
- 11. RFC 3379

  Delegated Path Validation and Delegated Path Discovery Protocol Requirements

  2002г. 15с.
- 12. RFC 2633

  S/MIME Version 3 Message Specification

  1999г. 32с.
- 13. RFC 2632

  S/MIME Version 3 Certificate Handling

  1999г. 13с.

- 14. **Security Architecture for the Internet Protocol**

  RFC 2401 1998г. 66с.
- 15. **Internet Security Association and Key Management Protocol (ISAKMP)**

  RFC 2408 1998г. 86с.
- 16. **The Internet Key Exchange (IKE)**

  RFC 2409 1998г. 41с.
- 17. RFC 2412

  The OAKLEY Key Determination Protocol

  1998г. 55с.