Андрей А.Терехов

Конспект лекций "Криптографическая защита информации"

Санкт-Петербург
1999

1.Основные понятия – криптография, криптоанализ, криптология

Криптография – наука о сохранении секретности сообщений.
Криптоанализ – наука о методах взлома зашифрованных сообщений.
Криптология – отрасль математики, включающая в себя криптографию и криптоанализ.
Криптографический алгоритм – математические функции, используемые для шифрации и дешифрации.

Если надежность алгоритма основана на хранении алгоритма в секрете, то такая надежность называется *ограниченной* (очень легко ломается).

Для секретности все современные алгоритмы используют *ключ* (обозначим его k). Набор возможных значений ключа – *пространство ключей* (keyspace).

$$E_k(p) = c$$
$$D_k(c) = p$$

симметричный ключ

$$E_{k_1}(p) = c$$
$$D_{k_2}(c) = p$$

алгоритм с открытым ключом

Симметричные алгоритмы делятся на 2 типа: *поточные* (шифруют побитно) и *блочные* (текст разбивается на блоки и шифруется поблочно).

Основная задача криптографии – сохранить исходный текст от *противника*

Атака – это попытка криптоанализа. Успешная атака – *метод*. Атака предполагает знание криптографического алгоритма.

Участники протоколов:

Alice, Bob, Carol – участники практически всех протоколов. (Может быть добавлен Dave).
Eve (Ева) – подслушивающая сторона (eavesdropper).
Mallet – злоумышленник, активный взломщик (malicious active attacker).
Trent – доверенный арбитратор (trusted arbitrator).
Walter – садовник, привратник (warden) – защищает A в некоторых протоколах.
Peggy – доказывающая сторона (prover).
Victor – проверяющая сторона (verifier).

2.Классическая криптография (шифр Цезаря, перестановочные шифры и т.д.)

В докомпьютерные времена криптография была основана на буквах и их перестановке и замене.
Подстановочные шифры – такие шифры, в которых одни буквы заменяются на другие.

4 типа подстановочных шифров:

1. простая подстановка;
2. гомофоническая подстановка (одной букве могут соответствовать несколько различных значений: “A” $\rightarrow$ 5, 13, 25, 56; “B” $\rightarrow$ 7, 19, 31, 42 …);
3. многоалфавитная постановка (несколько постановок шифров, например, в зависимости от номера буквы в тексте);
4. полиграммный шифр (шифрование группами “ABA” $\rightarrow$ “RTQ”, “ABB” $\rightarrow$ “SLL” и т.д.).

Шифр Цезаря – сдвиг на 3 вправо по модулю 26.

Легко ломаются, так как не скрывают частоты букв. Шеннон: требуется порядка 25 букв для восстановления текста [806].

Общий алгоритм взлома шифров [689]: S. Peleg, A. Rosenfield “Breaking Substitution Ciphers Using a Relaxation Algorithm” CACM, v.22, n.11, Nov.1979, pp. 598-605.

Гомофонические подстановки (1401г., герцогиня Мантуйская) – более сложные, но не скрывают всех статистических показателей текста. Тривиальны для known-plaintext, но и cyphertext only только несколько минут (см. [710]).

Многоалфавитная подстановка (1568г., Леон Баптиста). Объединенная Армия использует их во время Гражданской войны в Америке. Легко ломаются, но до сих пор используются, например, в Word Perfect. *Период* такого шифра [475, 355, 360].

Полиграммный шифр – PlayFair (1854), использовался Англией в Первой Мировой. Кодирует парами. См. [360, 832, 500]

Перестановочные шифры – такие шифры, в которых буквы текста остаются неизменными, но меняется их порядок.

Например: текст пишется колонками, а затем читается по вертикали. Немецкий шифр ADFGVX был сломан французами в Первую Мировую войну. Требует времени и (иногда) фиксированного размера (по модулю).

Роторные машины – Enigma (немцы во Вторую Мировую войну) была сломана англичанами с участием Тьюринга (документы рассекретили только в 1996!).

Алгоритм XOR /* программка*/

Симметричный алгоритм : текст XOR'ится с ключевым словом. Т.к. XOR, примененный дважды, восстанавливает исходный текст, то алгоритм симметричен:

$$P \ \ XOR \ \ K = c$$

$$\Leftarrow (P \ XOR \ K) \ XOR \ K = p$$

$$C \ \ XOR \ \ K = p$$

Как его ломать?

Предположим, что текст на английском, причем длина ключа сравнительно небольшая.

1. "Подсчет совпадений" [355]. Если два куска шифрованного текста используют один и тот же ключ, то совпадет более 6% байтов. Если нет, то меньше 0.4% (пердполагали ASCII текст; другой текст – другие цифры). Наименьшее смещение, показывающее совпадение и есть длина повторяющегося ключа (сдвигаем текст направо по одному байту).
2. Сдвинем шифртекст на найденную длину и поXORим его с самим собой. Это уничтожает ключ и оставляет нас с исходным текстом, поXORенным с тем же текстом, но сдвинутым на длину ключа. Т.к. в английском языке всего лишь около 1 бита реальной информации на байт, то у нас достаточно избыточности, чтобы выбрать единственную расшифровку.

3.Одноразовый блокнот и доказательство его информативной надежности

Как это ни странно, существует единая шифровая схема.

Одноразовый блокнот (one-time pad) был изобретен в 1917 майор Джозеф Моборн (Mauborgne) и Гильбертом Вернамом.

Одноразовый блокнот – это большой неповторяющийся набор случайных чисел. Каждый набор используется ровно для одного сообщения. Каждая буква одноразового блокнота используется для шифрации ровно одной буквы сообщения.

У Получателя такой же блокнот. При посылке и после расшифровки копии блокнота уничтожаются.

Если нет доступа к ключу, то взломать такую схему невозможно.

Т.к. для любого сообщения данной длины можно подобрать одноразовый блокнот: из зашифрованного сообщения получить данные, то взломать это невозможно (т.к. любой текст равновероятен).

Проблема только в генерации одноразового блокнота (точнее в случайности). Любая атака против схемы – это атака против генератора.

Сложности:

1. длина ключа равна длине текста;
2. распространение ключей;
3. хранение ключей;

Определение

Односторонняя хэш-функция:
Вход M некоторой длины; выход – h, фиксированной длины (хэш).

1. h(M) считается легко
2. по h трудно найти M
3. по M трудно найти M': h(M) = h(M'). "Трудно" $\approx 2^{64}$ операций или больше.

4. Односторонние хэш-функции, функции ловушки и т.д.

Определение

Односторонняя функция (one-way function) – это функция, которая достаточно легко вычислить, но вычисление обратной функции – это вычислительно сложная задача.

Проблема односторонней функции в том, что с ее помощью нельзя осуществлять шифрацию (расшифровать не удастся никому). Поэтому более широко используется.

Пример односторонней функции: разбить молотком часы.

Определение

Односторонняя функция с потайным ходом (trap-door one-way function) – это такая односторонняя функция, которую легко обратить, если известен некоторый секрет.

Пример односторонней функции с потайным ходом: поточтовый ящик

Определение

Односторонняя хэш-функция (она же message digest cryptographic checksum) – это функция, берущая строку на вход и выдающая строку фиксированного размера (обычно меньшую), обладающая при этом свойствами односторонности. Бывают с ключом и без.

Основные требования к односторонним хэш-функциям

Выход не должен зависеть от входа каким-либо разумным образом; изменение одного бита на входе должно изменять, в среднем, половину выходной строки; по хэш-значению должно быть вычислительно невозможно найти входную строку.

Главное в односторонней хэш-функции

По данному M должно быть трудно найти M': h(M) = h(M').

Часто требуется даже более сильное условие:

Определение

Сильная односторонняя хэш-функция - это такая хэш-функция, для которой вычислительно трудно найти M, M': h(M) = h(M').

5. DES – Data Encryption Standard

1970-е. Криптография в загоне. Знания – только у ???? и NSA.

1972. National Bureau of Standards (NBS), теперь National Institute of Standards and Technology (NIST) организовал программу защиты данных. Как часть программы, они хотели создать единый алгоритм для надежной передачи и хранения данных. Достоинства единого алгоритма.

15 мая 1973, в Federal Register NBS опубликовало публичный запрос предложений по алгоритмам.

Требования к алгоритму:

1. Высокий уровень секретности;
2. Полная спецификация алгоритма и легкость в понимании;
3. Секретность должна заключаться в ключе, а не в знании алгоритма;
4. Алгоритм должен быть доступен всем;
5. Алгоритм должен быть легко адаптируем для использования в различных приложениях;
6. Алгоритм должен быть экономичен в реализации;
7. Алгоритм должен быть эффективен;
8. Верифицируемость алгоритма;
9. Экспортируемость.

Август 27, 1974 – второй запрос.

IBM предложило собственный алгоритм LUCIFER. Использовал только простые операции над битами и вообще был прямолинеен. Запросили NSA. IBM согласилось поделиться интеллектуальной собственностью.

17 марта 1975г. NBS опубликовало алгоритм, а 1 августа запросило комментарии.

Основные претензии – сокращение ключа с 112 бит до 56; опасения о наличии в DES “trap door”.

Короче, 23 ноября 1976г. DES стал стандартом.

На 1991г. существовало 45 различных, утвержденных NIST, реализаций DES. Существовал также Federal Standard, по которому hardware должен быть еще запрятан за замками и сигнализациями.

В 1992 г. алгоритм был ресертифицирован на 5 лет (последний раз).

В 1995г. был построен DES Cracker стоимостью 250000$, сломавший данный фирмой RSA ключ за 3 дня. Утверждается, что скорость поиска равна 88 млрд. ключей в секунду.

В 1997г. начался тендер AES (Advanced Encryption Standard). Выпуск стандарта назначен на середину 2001 г.

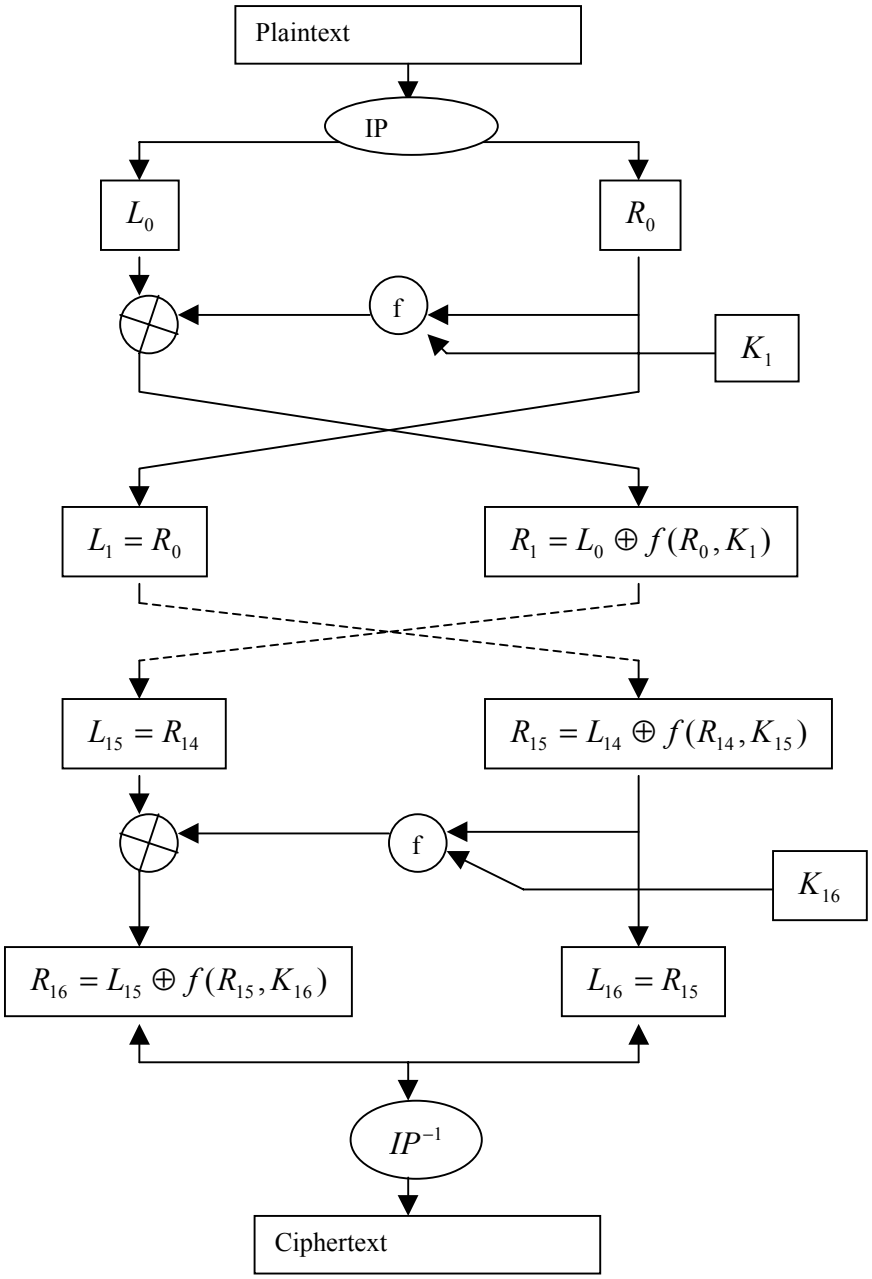
Обзор алгоритма

DES является блочным алгоритмом и шифрует данные, разбитые на 64-битовые блоки (на выходе тоже 64-битовые блоки). Шифрация и дешифрация происходит по одинаковому алгоритму (за исключением выбора ключей).

Длина ключа равна 56 бит (ключ обычно 64, но каждый восьмой бит – проверка четности). Ключ – любое число. Вся секретность в ключе.

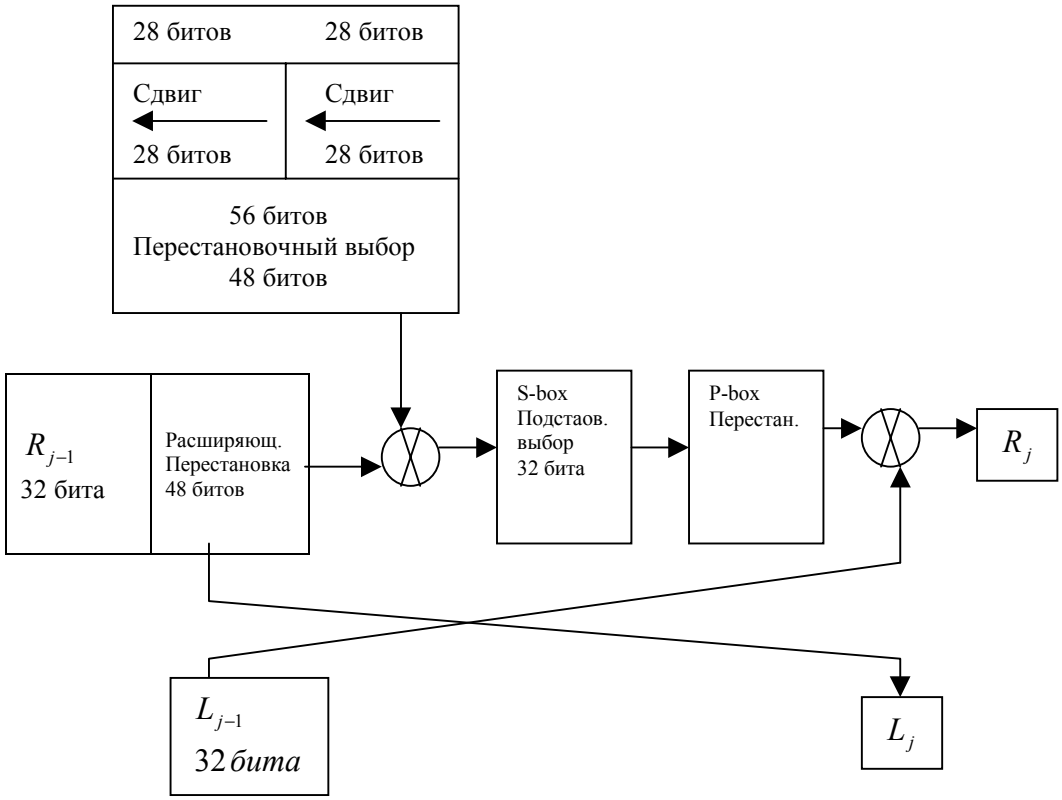
Алгоритм состоит из перемешивания и диффузии (подстановка + перестановка). Это один раунд. Всего их 16.

Общая схема



Описание алгоритма

После начальной перестановки (IP), блок разбивается на левую и правую половины (Li и Ri) по 32 бита. Затем происходит 16 раундов идентичных операций (функция f), в которой данные комбинируются с ключом. После 16-го раунда правая и левая половины объединяются и финальная перестановка (IP^{-1}), являющаяся инверсией начальной перестановки завершает алгоритм.



В каждом раунде: ключевые биты сдвигаются, а затем 48 битов выбираются из 56. Правая часть данных расширяется до 48 битов путем расширяющей перестановки, затем XOR-ится с 48 битами полученного ключа. Результат попадает на вход в S – box, оттуда получаем 32 бита (подстановка), которые снова переставляются.

Эти действия образуют функцию f.

Выход функции f XOR-ится с левой частью.

Пусть B_i - результат i-ой итерации; $B_i = (L_i, R_i)$.

k_i - 48-битный ключ для i-ого раунда; f – функция DES-а.

Тогда раунд выглядит как

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

$$R_i = L_i \ XOR \ f(R_{i-1}, k_i)$$

Исходная подстановка (initial permutation)

(Таблица перестановки 10.1). IP и IP^{-1} не влияют на надежность (стойкость) DES. Многие software версии этого не делают, т.к. битовые операции сложны.

Преобразование ключа (key transformation)

Изначально, 64-битный ключ сужается до 56 бит (убираются биты четности) и переставляются. (Таблица 10.2).

Затем для каждого i-ого раунда генерится ключ k_i : сначала 56-битный ключ делится на две половины по 28 бит. Затем половины сдвигаются налево на 1 или 2 разряда, в зависимости от раунда (см. таблицу 10.3). После сдвига, выбирается 48 бит из 56 бит ключа (вместе с перестановкой, поэтому называют compression permutation или permuted choice) – см. таблицу 10.4.

Расширяющая перестановка (expansion permutation)

В этой операции правая половина данных, R_i, расширяется с 32 до 48 бит (с перестановкой).

Основная цель этой операции – в убыстрении зависимости выходных битов.

Критерий лавинообразности – как можно быстрее добиться условия зависимости каждого бита шифра от каждого бита текста и каждого бита ключа.

См. таблицу 10.5

Подстановочный выбор (S-box substitution)

На вход Пер ???? выбору подается 48-битный результат XOR-енья сжатого ключа и расшифрованного текста. Замены выполняются восемью S-box-ами. 48 бит делится на 8 6-битных блоков. 1-ый блок $\rightarrow$ 1-ый S-box, …, 8-ой блок $\rightarrow$ 8-ой S-box…

Каждый S-box является таблицей из 4 строк и 16 колонок. Рассмотрим 6-битный вход : (b1,b2,…b6). Биты b1 и b6 образуют 2-битное число (от 0 до 3), обозначающее строку таблицы. По строке и столбцу находим искомое число, на которое заменяем входное.

Это критический этап алгоритма. Все прочие операции – линейны и легко поддаются анализу. А S-box-ы не линейны и более, чем другие операции дают DES-у криптостойкость.

Результатом являются 8 4-битных блоков, которые собирают в единый 32-битный блок.

Пространство P-box-а

32-битный вход переставляется согласно P-box-у. В этой операции ни один бит не теряется и ни один бит не используется дважды (прямая перестановка). См. таблицу 10.7

Наконец, результат из P-box-а XOR-ится с левой частью начального 64-битного блока. Затем меняются правая и левая часть и начинается следующий раунд.

Финальная перестановка

Финальная перестановка является инверсией начальной перестановки и описывается в таблице 10.8. Отметим, что левая и правая части не меняются после последнего раунда DES-а, вместо этого конкатенированный блок $R_{16}L_{16}$ используются в качестве входа для IP^{-1}

Расшифровка DES-а

После всех этих подстановок, перестановок, XOR-ов и сдвигов, можно подумать, что алгоритм расшифровки другой и так же запутан, как и алгоритм шифрации. Однако, операции подобраны таким образом, что алгоритм работает в обе стороны.

Единственная разница в том, что ключи надо использовать в обратном порядке, т.е. ключи расшифрации $k_{16}\ldots k_1$. Алгоритм генерации ключей для раунда тоже обратим. Сдвиг ключей – направо, а количество бит сдвига – определяется по таблице справа налево.

Слабые ключи

Алгоритм выбора ключей для раундов допускает некоторые ключи, называемые *слабыми*. Как уже говорилось, ключ делится на две половины и каждая из них сдвигается независимо. В связи с этим может получится так, что каждый раундовый ключ будет одинаковым. Это может произойти, если ключ состоит только из 0, только из 1 или , если одна половина из 0, а другая из 1.

Есть и другие свойства, ослабляющие надежность слабых ключей.

Ключи приведены в таблице 10.10. Напомним, что каждый восьмой бит выкидывается, а начальная перестановка меняет порядок.

Кроме того, существуют пары ключей, дающих одинаковый результат при шифровке текстов (взаимозаменяемые ключи). Это тоже связано с генерацией подключей - вместо 16 различных, генерятся только 2 различных, используемые 8 раз в алгоритмах. Это так называемые *полуслабые ключи* (см таблицу 10.11).

Есть также ключи, производящие только 4 подключа, каждый из которых используется 4 раза в алгоритме (см таблицу 10.12).

В общем-то, 64 слабых ключа из 72, 057, 594, 037, 927, 936 возможных ключей – это немного. Шансы напороться на такой ключ ничтожно малы. Если все-таки страшно, то ключ можно проверить.

Больше слабых ключей не найдено.

Ключи дополнения

Возьмем побитовое дополнение ключа (т.е. заменим все 0 на 1 и все 1 на 0). Тогда, если ключ шифрует блок текста, то дополнение ключа зашифрует дополнение текста в дополнение шифровки.

$$J_{k'} \text{ - дополнение к x} \Rightarrow \begin{matrix} E_k(P) = C \\ E_{k'}(P') = C' \end{matrix}$$

Это значит, что chosen-plaintext attack против DES должен проверить только половину возможных ключей, т.е. 2^{55} вместо 2^{56}. Biham & Shamir показали также, что существуют known-plaintext attack той же сложности, при наличии как минимум 2^{33} known-plaintexts/

Сомнительно, что это существенная слабость, т.к. большинство сообщений не имеют дополнений (вероятность низкая) и пользователей можно предупредить не использовать ключи дополнения.

Является ли DES группой?

Количество отображений всех 64-битных текстов во все возможные 64-битные шифры (= 2^{64}!). DES с его 56-битным ключом, дает 2^{56} (10^{16}) таких отображений. Используя множественную шифрацию, можно достичь большого количества подобных отображений. Но это верно только в том случае, если DES является группой. (Аналогия: сложение над целыми – группа; умножение с делением – не группа).

Если DES группа, то криптоанализ упрощается.

Тогда существовал бы k_3: $E_{k_2}(E_{k_1}(P)) = E_{k_3}(P)$.

Хуже того, DES поддавался бы meet-in-the-middle атаке (2^{28} щитов вместо 2^{56} для полного перебора).

Популярный вопрос. Вагон статей. В 1992г. это было доказано [184].

Длина ключа

Почему 56 бит? В Lucifer было 128 бит.

Опасность brute force attack.

В 1979г. Diffe & Hellman: специальная ??????? машинка сломает DES за день и будет стоить 20 млн.$

1981: 2 дня взлома и 50 млн.$.

Для NSA и французской разведки это уже давно доступно.

Число итераций

Почему 16 раундов? Почему не 32? Alan Konheim показал, что после 8 итераций шифр практически является случайной функцией любого бита текста и любого бита ключа [500]. Так почему бы не ограничиться 8 раундами?

Тем не менее, варианты DES-а с меньшим числом раундов успешно ломались: в 1982 легко сломали 3-4 раунда DES; на шесть лет позже сломался DES с 6 раундами.

В 1990г. Biham & Shamir изобрели дифференциальный криптоанализ, который объяснил этот вопрос. DES с меньшим 16 числом раундов может быть сломан с помощью known-plaintext attack более эффективно, чем brute force.

Устройство S-box-ов

NSA обвиняли также во вмешательство в устройство S-box-ов (подозревали их во встраивании trap door), поэтому поэтому устроили мощный анализ этого дела.

На 2-м семинаре, посвященном DES-у, NSA раскрыло несколько критериев:

1. Ни один S-box не является линейной аффинной функцией входных данных, т.е. не существует системы линейных уравнений, которыми можно выразить 4 выходных бита в терминах 6 входных.
2. Изменение одного бита во входе изменяет как минимум два бита на выходе, следовательно, максимизация диффузии.
3. Если один и бит равен const, то S-box-ы должны минимизировать разницу между числом 0 и 1.

Интересные результаты

Два различных, но специально подобранных, входа могут производить одинаковый выход.

Можно добиться равенства входа и выхода изменением только в трех соседних S-box-ах по биту. Еще один критерий NSA заключался в устойчивости к дифференциальному криптоанализу, который тогда еще никто не знал.

Дифференциальный криптоанализ

1990г. Eli Biham & Adi Shamir

Дифференциальный криптоанализ – анализ, рассматривающий пары (текст, шифр), точнее рассматривает пары шифров, тексты которых имеют определенные различия. Сами тексты могут выбирать слуг, лишь бы они отвечали некоторым условиям (смысл не нужен).

Некоторые различия в тексте имеют высокую вероятность возникновения в шифрах. Такие различия называются *характеристиками*.

Например, если разница между текстами равна 0080 8200 6000 0000, то имеется примерно 5% шанс, что разница между шифрами будет такой же.

Дифференциальный криптоанализ использует такие характеристики (их много) для назначения вероятностей отдельным ключам и впоследствии для обнаружения наиболее вероятного ключа.

Такая атака работает против DES-а и других алгоритмов с постоянными S-box-ами. Атака сильно зависит от структуры S-box-ов. Так получилось, что в DES-е S-boxes оптимизированы против таких атак.

Для 16 раундов:

Ch.Pl. = 2^{47} kn.pl. = 2^{55}, Analyzed pl. = 2^{36}, Complexity of analysis = 2^{37}

Для 17-18 раундов время примерно равно времени перебора. При 19 раундах перебор выгоднее.

В общем-то, угроза скорее теоретическая – огромные временные и емкостные потребности для атаки. Для ch.-pl. Требуется шифровать поток 1.5 Мбит/с в течение 3 лет для получения необходимой информации.

Почему DES так устойчив к дифференциальному криптоанализу?

IBM's Don Coppersmith: "Мы в IBM знали о дифференциальном криптоанализе в 1974г., поэтому и оптимизировали его для этого". Шамир в ответ попробовал добиться утверждения, что в IBM не было известно более мощных атак, но Coppersmith отмолчался.

Related-key Criptanalysis

См. таблицу 10.3. DES-овский ключ сдвигается на 2 бита после любого раунда, кроме 1, 2, 9 и 16 раундов (на 1 бит). Почему?

Модифицированный DES, в котором ключ сдвигается на 2 бита поле любого раунда менее надежен. Biham изобрел related-key criptanalysis, который ломает такой вариант с использованием 2^{17} chosen-key plaintexts или known-plaintexts.

Related-key criptanalysis изучает разницу между двумя ключами и рассматривает при этом plaintext и шифр. Атака не зависит от числа раундов (хоть 1000).

Еще менее реальная угроза. Единственный сценарий – криптоаналитик знает некоторые свойства датчика случайных чисел, используемого для создания ключей DES, но не все, и пи этом во вражеской организации сидит наш агент, скармливающий этому шифратору кипу текста.

DES оптимизирован и против этой атаки.

Аналогичный российский стандарт

ГОСТ 28147-89 “Системы обработки информации. Защита криптографическая. Алгоритм криптографического преобразования”.

См. также статью Андрей Винокуров “Алгоритм шифрования ГОСТ 28147-89, его использование и реализация для компьютеров платформы Intel x86”.

6. Режимы использования блочных алгоритмов.

Блочные алгоритмы можно использовать по-разному, в зависимости от конкретных задач.

Electronic Codebook Mode (режим электронной кодовой книги)

Наиболее очевидный способ использования блочного алгоритма любому тексту соответствует некоторый шифр. Отсюда название, т.к. один блок всегда зашифр. в один и тот же шифр. Electronic Codebook (ECB). Обычно размер блока 64 бита, следовательно, в кодовой книге будет 2^{64} записей, т.е. достаточно много. Для любого ключа – своя «кодовая книга».

Достоинством ECB является независимость шифрации блоков. Это важно для файлов со случайным доступом (напр., БД). Записи можно удалять и добавлять в середину БД (если считать, что запись состоит из целого числа блоков).

Проблемой ECB является то, что криптоаналитик может начать собирать кодовую книгу без знания ключа (при наличии plaintext и шифр. текста). В реальной жизни сообщения имеют повторяющуюся структуру (заголовки, обращения, подписи, письма эл. почты). Естественно, наиболее уязвимы начала и окончания писем (=> информация об отправителе, дате и т.д.). Эта проблема решается в других режимах.

Другая, потенциально более опасная проблема, называется **воспроизведение блоков** (block replay) и связана с тем, что противник может изменять зашифрованные сообщения без знания ключа (и даже деталей алгоритма) так, чтобы надуть получателя. Впервые обсуждена в [183].

Рассм. сиситему банковских расчетов. Для обеспечения работы различн. сиситем, сеть банков использует стандартный формат сообщений, напр.:

Название банка-отправителя	12 байтов	
Название банка-получателя	12 байтов	
Имя вкладчика	48 байтов	Сообщения шифруются блочным
Р/С вкладчика	16 байтов	алгоритмом в режиме ECB.
Сумма вклада	8 байт	

Маллет подслушивает на линии между Банком 1 и Банком 2. Он посылает 100$ из одного в другой, затем еще раз. Просматривает сообщения, находит пару идентичных (если много, то операция повторяется). Теперь все, что нужно – это послать данное сообщение еще раз (=> +100$). Если проделать это с суммами >100$ и несколько раз, то можно спокойно получить достаточно для спокойного проживания где-нибудь в Латинской Америке.

На первый взгляд, достаточно вставить timestamp, но это не помогает:

1	2	3	4	5	6	7	8	9	10	11	12	13
Time Stamp	Sending Bank	Receiving Bank		Depositor’s Name						Account		Amount

Итого 13 блоков по 64 бита. Если Маллет выясняет, что 5-12 блоки содержат нужную ему информацию, то он подменяет **только их**. Если же он выяснит еще и блок для нужной суммы, то может спереть и больше, чем реально послано. Это и есть block replay.

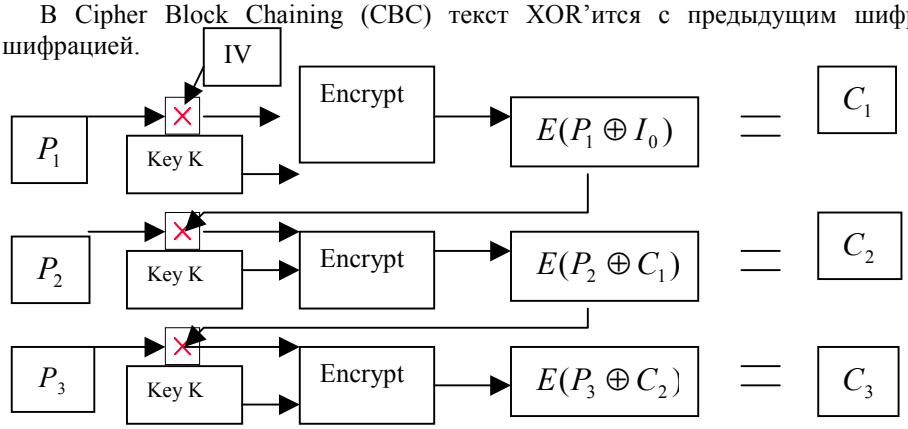
Причем такая атака сложнее для отлова. Вероятно, это вскроется только тогда, когда всполошатся реальные вкладчики, не получившие переведенных денег.

Банки могут уменьшить риск, меняя ключи, но это значит только то, что Маллету придется работать быстрее. Решением является техника, называемая сцеплением (chaining).

Cipher Block Chaining Mode (режим сцепления шифрованных блоков)

Для сцепления используется механизм обратной связи, поскольку результат шифрации предыдущих блоков используется для шифрации текущего блока. Таким образом любой блок шифра зависит не только от исходного текста, но и от всех предыдущих блоков текста.

В Cipher Block Chaining (CBC) текст XOR'ится с предыдущим шифр. блоком перед шифрацией.



Дешифрация аналогично. Математическая запись выглядит так:

$$C_i = E_k(P_i \; XOR \; C_{i-1})$$

$$P_i = C_{i-1} \; XOR \; Д_k(C_i)$$

Вектор инициализации (IV) используется для того, чтобы любое сообщение было по-настоящему уникальным (иначе будут трудности со стандартным заголовком). IV должен быть случайным числом. Его не обязательно хранить в секрете, можно передавать его вместе с сообщением.

Большинство сообщений не делятся нацело на 64-битные блоки, обычно остается короткий блок в конце. Можно по-разному бороться с этим. Простейший метод – padding (дополнение до полного блока). Если надо потом убирать мусор, то достаточно просто последним байтом обозначить количество лишних байтов. Можно также обозначать последний байт текста символом конца файла.

Это не всегда можно сделать (напр., если надо расшифровать блок и поставить его куда-нибудь в память). Тогда применяется следующая схема шифрования: предположим, в последнем блоке j бит. После зашифровки последнего целого блока, зашифруем шифрованный блок еще раз, выберем j начальных битов полученного текста и поXOR'им с исходным текстом. Это и есть шифр для неполного блока.

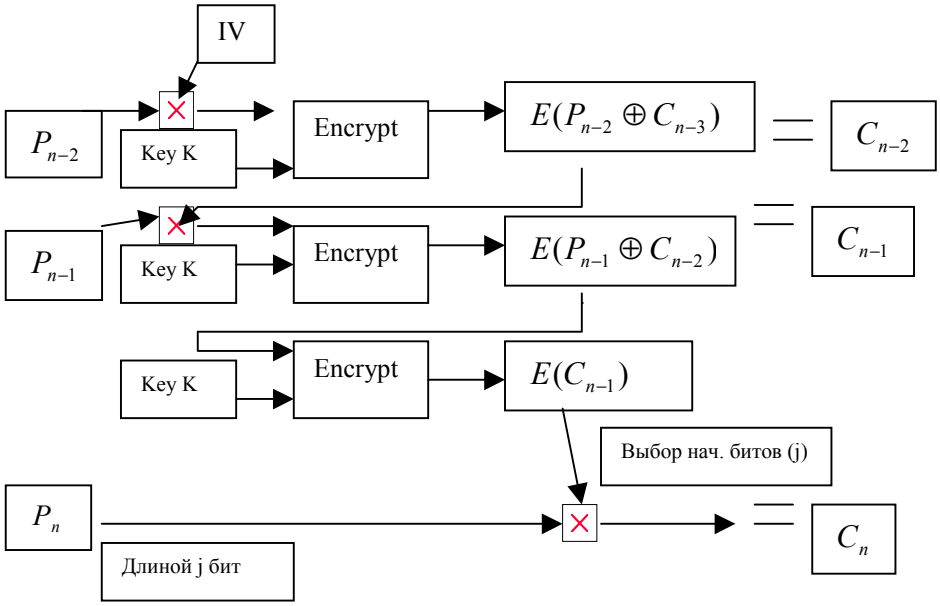


Рис. Метод шифрования посл. блока.

Восстановление после ошибок (error propagation) в CBC.

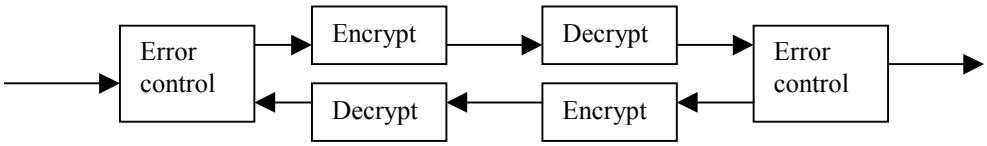
CBC может быть охарактеризована как feedback шифр. теста при шифрации и feedforward шифр. теста при дешифрации. Это значит, что одна ошибка в исх. тексте повлияет на шифр блок и все последующие шифр. блоки. Это неважно, т.к. дешифрация обратит этот эффект и в восстановленном тексте снова будет только одна ошибка.

Ошибки в шифрованном тексте более распространены (из-за плохих линий связи или ошибок диска). В режиме CBC, ошибка 1 бит в шифрованном тексте целиком портит этот блок +1 бит след блока (в той же позиции, что и предыдущий). Послед. блоки не повреждаются, т.е. CBC является самовосстанавливающимся режимом (self-recovering).

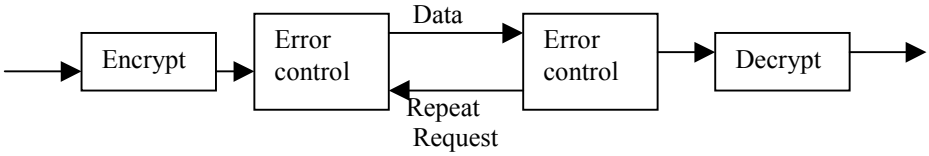
Несмотря на то, что CBC восст. от ошибок в битах, оно совсем не защищено от ошибок синхронизации. Если добавить или потерять бит, то система будет производить бесконечный бред. Поэтому любая система, использующая CBC должна обеспечить сохранение структуры исходного текста (framing).

Свойство превращения маленькой ошибки в шифре в большую ошибку текста называется error extention. Для ее избежания необх. ср-ва error detection & recovery (причем после шифрации и до дешифрации, иначе только хуже).

Полная схема error control:



Хорошая схема error control:



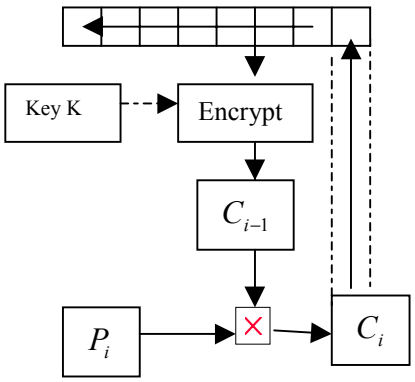
Gipher Feedback Mode (режим обратной связи с шифром)

Проблема с CBC, в частности, заключается в том, что шифрация не может начаться до получения целого блока данных. Это может быть неудобно, напр., для приложений, работающих с удаленным терминалом.

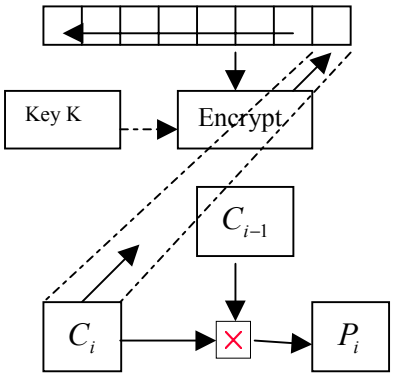
В Gipher Feedback Mode (GFM) данные шифруются единицами меньшими, чем целый блок. В принципе будет рассм. шифрация одного ASCII символа (т.н. 8-битн. GFM), но его можно обобщить и на 1-битный случай.

Рис. 8-битный CFB mode

(а) шифрация



(б) дешифрация



На рисунке показан 8-битный CFB с 64-битным блочным алгоритмом. Здесь блочный алгоритм работает с очередью размером со вх. блок (64). В начале, очередь заполняется инициализационным вектором, как в CBC. Очередь шифруется, а самые левые 8 битов

результата XOR'ятся с первым 8-битным символом текста. Это и есть первые 8 битов шифрованного текста, их уже можно передавать. Эти же 8 битов становятся самыми правыми 8 битами очереди (все остальное сдвигается влево на 8 битов). Самые левые 8 битов при этом отбрасываются. Затем все повторяется.

Расшифровка является обратным процессом. Отметим, что на обеих сторонах блочный алгоритм исполняется в режиме шифрации.

Как и в CBC, в CFB шифрованный текст зависит от всего предыдущего текста.

Восстановление после ошибок в CFB.

В CFB ошибка в тексте обращается при расшифровке (все, как в CBC).

Ошибка в шифре более интересна. Первый эффект однобитовой ошибки – в ошибке исходного текста. После этого ошибка попадает на регистр сдвига и портит шифр до тех пор, пока не выпадет с другого конца регистра. Т.о. в 8-битном CFB 9 байт шифра портятся от ошибки в 1 бит, а весь последующий шифр расшифровывается корректно.

Тонкая проблема: если Маллет знает текст сообщения, то он может так подправить биты в данном блоке, что блок расшифруется во все, что он хочет. Следующий блок будет расшифрован в мусор, но вред уже может быть причинен.

CFB является self-recovering и по отношению к ошибкам синхронизации. Ошибка попадает на регистр, портит 8 байт и выпадает с другого конца (т.к. синхронизация всегда начинается от последнего зашифрованного байта).

Output Feedback Mode (режим обратной связи с выводом)

Output Feedback Mode (OFM) похож на CFB за исключением того, что в регистр сдвига помещается C_{i-1}, а не C_i. Иногда поэтому данный режим называется internal feelback (внутренняя обратная связь), т.к. механизм обратной связи не зависит от исходного текста и зашифрованного текста.

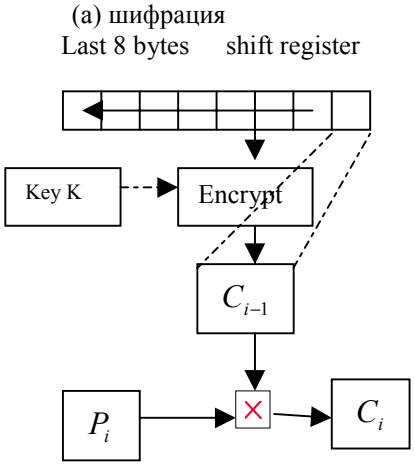
Восстановление ошибок в OFB.

Достоинство OFB в отсутствии error extension. Ошибка в одном бите в шифре вызывает 1-битную ошибку при восстановлении текста. Это удобно для передачи оцифрованного видео или аудио, где случайная ошибка терпима, а error extension – нет.

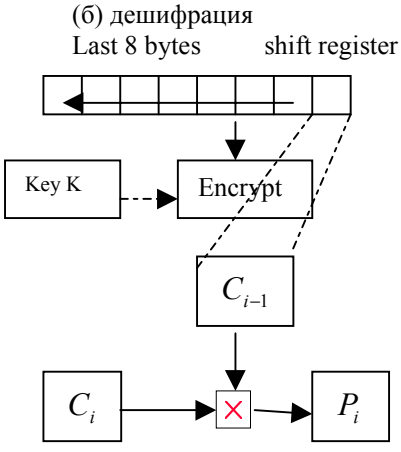
С другой стороны, потеря синхронизации фатальна, поэтому любая система, использующая OFB должна иметь механизм обнаружения потери синхронизации и механизм загрузки регистров сдвига новым IV для восстановления синхронизации.

Рис. Output Feedback Mode.

(а) шифрация



(б) дешифрация



Недавний анализ OFB доказал, что OFB должен быть использован только в том случае, когда размер блока = размеру feedback'а, т.е. 64-битный алгоритм должен быть использован с 64-битным OFB. Даже несмотря на то, что ГОСТ США допускает другие режимы для DES'а, этого следует избегать.

Отбор подходящего режима для блочного алгоритма.

ECB – простейший и слабейший. CBC немного сложнее, но много надежнее. OFB и CAИ сильнее медленнее; CFB обычно используется для удаленных терминалов, OFB для высокоскоростных синхр. систем. Прочие режимы лучше не использовать. DES разрешено использовать с любым из этих четырех режимов.

6. Ключи. Длина ключа. Хранение, распространения и генерация ключей.

Ключи и длина ключей.

Надежность симм. криптосистемы есть функция от надежности алгоритма и длины ключа. Предположим, что алгоритм идеален, т.е. нет лучшего способа взлома криптосистемы, чем полный перебор ключей (т.н. метод грубой силы brute-force attack).

Если ключ длиной 8 бит, то потребуется 2^8=256 попыток (а в среднем, 128). Если ключ длиной 56 бит => 2^{56} возможных ключей. Если предположить, что скорость перебора = 1 млн. ключей/сек (когда-то это была внушительной цифрой), то потребуется 2000 лет для нахождения правильного ключа. Если ключ длиной 64 бита, то на той же скорости потребуется ~600 000 лет. Если ключ длиной 128 бит, то это потребует 10^{25} лет. Учитывая, что Вселенная существует всего 10^{10} лет, то такая цифра вероятно, достаточно много.

С ключом длиной 2048 бит, миллион машин со скоростью 1 млн. ключей/ сек, работающих параллельно, потратят 10^{597} лет на поиски ключей.

Т.о., если предположить, что brute-force attack – самая эффективная атака против алгоритма (что уже достаточно сильное предположение), то ключ должен быть достаточно большим, чтобы сделать эту атаку вычислительно невозможной. Скорость brute-force attack определяется двумя факторами: количеством ключей и скоростью тестирования (1). Скоростью тестирования одного ключа можно пренебречь как конст. фактором.

Можно составить таблицу, которая будет давать необходимое число процессов для взлома алгоритма за заданное время при заданном количестве шифраций в секунду. (см. таблицу 7.1). Но эти числа дают только часть ответа. Дело в том, что машина заведомо раскрывающая ключ за год, имеет 8% шанс раскрыть ключ за месяц. Если при этом ключ меняют 1 раз в месяц, то есть 8% вероятность раскрыть ключ еще во время его использования.

Более того, пусть есть машина, отыскивающая ключ за месяц, а ключ меняется каждый час. Несмотря на то, что вероятность найти данный ключ за час всего 0,14%, вероятность найти правильный ключ до его смена за месяц использования такой схемы = 63%, причем эта цифра не зависит от частоты смена ключа.

Т.о. частая смена ключей позволяет разве что минимизировать последствия взлома системы, но во многих системах это все равно недопустимый риск.

Другим параметром, определяющим возможность brute-force attack, является стоимость.

Поэтому в вопросе определения необходимой длины ключа можно задать исходные данные так:

Длина ключа должна быть такой, чтобы вероятность взлома была не более 1 из 2^{32} при наличии у противника 100 млн. $, даже предполагается рост его возможностей в 30% в год за данный период.

См. табл. 7.6. для оценки требований к шифрации различной информации:

Вид сообщений	Жизн. интервал	Минимальная длина ключа
Тактическая военная информация	Мин/часы	56 бит
Сообщения о продуктах, слияниях компаний и т.д.	Дни/недели	56-64 бит
Торговые секреты (рецепт Coca Cola)	Декады	64 бита
Секреты водородной бомбы	>40 лет	128 бит
Дипломатические взаимоотношения	>65 лет	Как минимум 128 бит
Результаты переписи населения	100 лет	Как минимум 128 бит

Труднее оценить рост вычислительных мощностей. Некое полезное правило: эффективность компьютерного оборудования, деленная на цену, возрастает, каждые пять лет в 10 раз. Т.о., через 50 лет самый быстрый компьютер будет в 10^{10} раз быстрее, чем сейчас! При этом речь идет только об универсальных компьютерах, а специализированные компьютеры могут развиваться еще быстрее.

Работа с ключами.

Даже если все используемые алгоритмы и протоколы идеальны, ничто не поможет в том случае, когда ваши ключи становятся известны противнику. В реальном мире, работа с ключами – самая сложная проблема надежности криптографических систем. Ключи должны быть защищены, по крайней мере, так же надежно, как и сами данные.

Работа с ключами делится на несколько разделов.

Генерация ключей.

Например, в программе Disklock for Macintosh ver. 2.1 ключ хранился вместе с зашифрованным файлом; т.о. если знать, куда смотреть, то не надо взламывать DES.

Сокращенные пространства ключей.

Рассм. DES. Его полное пространство ключей = 2^{56} (10^{16}). Известна одна реализация DES'а, в которой ключом являлось только сочетание букв (причем только маленьких) и цифр. Это дает только 10^{12} ключей. Т.о. плохая реализация сделала DES в 10,000 раз слабее.

См. таблицы 7.7, 7.8.

Плохие ключи.

Когда люди выбирают ключи, они обычно выбирают плохие ключи. Они, скорее всего, выберут ключ «Дрюня», а не «*9(hH\A-». Поэтому умный алгоритм brute-force не станет перебирать ключи в последовательном порядке. Вначале следует перебрать тривиальные ключи: слова, имена, простейшие исправления англ. слов и т.д. Это т.н. атака со словарем, т.к. атакующий использует словарь распространенных ключей. Подробно описано в статье Д. Кляйна [482] (см. стр. 141-143). С помощью этой атаки удается взломать в среднем порядка 25% паролей. Эта атака много эффективнее, когда применяется на файл с ключами, чем на один

конкретный ключ. Один пользователь может выбрать абе хороший ключ, но 1000 человек не может выбрать себе хорошие ключи (т.е. дурак найдется).

Как генерировать хорошие ключи.

Хороший ключ – случайный ключ.

1) Если уж брать легко запоминающийся ключ, то следует сделать его странным, напр.:
   - слова, разделенные символом пунктуации;
   - строка из букв, являющихся начальными к более длинной фразе.
2) Лучше использовать технику, называемую key crunching, которая конвертирует строку легко запоминающихся символов в случайные ключи. Для этого берут строку (длинную) напускают на нее одностороннюю хэш-функцию и получают псевдослучайную строку битов. Если строка достаточно большая, то результат случаен. Достаточно брать ~ по одной букве на один бит ключа.

Символ Лежандра.

Удобен для проверки того, является ли целое число a квадратическим вычетом по модулю простого числа p. (a $\in \mathbf{Z}_n^*$ -- квадратичн. вычет по модулю n, если $\exists x \in \mathbf{Z}_n^* : x^2 \equiv a(\mathrm{mod}\, n)$. Множество всех квадратичн. вычетов обозн. Q_n)

Определение: Пусть p – простое, a – целое. Тогда символ Лежандра $(\frac{a}{p})$ определяется следующим образом:

$$(\frac{a}{p}) = \begin{cases} -1, если\ a \in Q_p \\ 1, если\ a \in \overline{Q_p} \\ 0, если\ p/a \end{cases}$$

Свойства символа Лежандра:

(1) $(\frac{a}{p}) \equiv a^{\frac{p-1}{2}} (\mathrm{mod}\, p)$

(2) $(\frac{ab}{p}) = (\frac{a}{p})(\frac{b}{p})$

(3) если $a \equiv b(\mathrm{mod}\, p)$, то $(\frac{a}{p}) = (\frac{b}{p})$

(4) $(\frac{2}{p}) = (-1)^{\frac{p^2-1}{8}}$

(5) если q – простое нечетное число, отличное от p, то $(\frac{p}{q}) = (\frac{q}{p})(-1)^{(p-1)(q-1)/4}$ -- закон квадр. взаимности.

Символ Якоби.

Является обобщением чисел Лежандра на нечетные, но необязательно простые числа.

Определение: пусть $n \geq 3$ -- нечетное, с разложением на простые множители $n = p_1^{l_1} \ldots p_k^{l_k}$. Тогда символ Якоби определяется как $(\frac{a}{n}) = (\frac{a}{p_1})^{l_1} \ldots (\frac{a}{p_k})^{l_k}$. Отметим, что если n – простое, то символы Якоби и Лежандра совпадают.

Свойства символа Якоби:

Пусть $m \geq 3, n \geq 3$ -- нечетные числа; $a, b \in Z$. Тогда символ Якоби имеет следующие свойства:

(1) $(\frac{a}{n}) = 0,1 или -1. Кроме того, (\frac{a}{n}) = 0 \Leftrightarrow НСД(a,n) \neq 1$

(2) $(\frac{ab}{n}) = (\frac{a}{n})(\frac{b}{n})$

(3) $(\frac{a}{mn}) = (\frac{a}{m})(\frac{a}{n})$

(4) $если\ a \equiv b (\mathrm{mod}\, n), то (\frac{a}{n}) = (\frac{b}{n})$

(5) $(\frac{1}{n}) = 1$

(6) $(\frac{-1}{n}) = (-1)^{\frac{n-1}{2}}$

(7) $(\frac{2}{n}) = (-1)^{\frac{n^2-1}{8}}$

(8) $(\frac{m}{n}) = (\frac{n}{m})(-1)^{\frac{(m-1)(n-1)}{4}}$

Алгоритм вычисления символа Якоби (=> символа Лежандра)

Jacobi (a,n) // n>=3, нечетное; $a \in [0, n)$, целое

1. if a=0 then return 0;
2. if a=1 then return 1;
3. write $a = 2^l a_1, where\ a_1 is\ odd$
4. if e is even then s:=1. Otherwise set s:=1 if n≡1 or 7(mod 8) or set s:=-1 if n≡3 or 5(mod 8)
5. if n≡3 (mod 4) and $a_1 \equiv 3 (\mathrm{mod}\, 4)$ then s:=-s
6. $n_1 := n \bmod a_1$
7. if $a_1 = 1\ then return\ s; else return\ (s * Jacobi(n_1, a_1))$;

Данный алгоритм работает за $o((\ln n)^2)$ битовых операций.

Эйлеровская φ-функция

Определение: для n≥1 φ(n) обозн. число целых ∈ [1,n], взаимн. простых с n.

Свойства φ(n):

(1) если p-простое, то φ(p)=p-1

(2) φ(n) – мультипликативна, т.е. если НОД(m,n)=1, то φ(m,n) = φ(m)φ(n)

(3) если

$$n = p_1^{l_1} \ldots p_k^{l_k}, разложение\ на\ простые\ множители, то\ \varphi(n) = n(1 - \frac{1}{p_1}) \ldots (1 - \frac{1}{p_k})$$

(4) Теорема: для всех n≥5 – целых $\varphi(n) > \frac{n}{6 \ln \ln n}$

9 Задача факторизации составного числа.

Для взлома системы RSA достаточно разложить n на простые множители. Поэтому задача факторизации составного числа приобрела огромное практическое значение. Отметим, что задача факторизации выступает как «обратная» задача к определению простоты данного числа n. Факторизация оказывается значительно сложнее.

Кроме того, нахождение НОД'а двух чисел также значительно проще, чем поиск делителей, поэтому, где возможно, надо обходиться им; НОД широко используется в алгоритмах факторизации.

Очевидно, что для разложения числа достаточно перебрать $o(\sqrt{n})$ чисел, попробовав поочередно разделить число n на каждое из них.

Более нетривиален следующий метод:

Метод Ферма (1643 г.)

Пусть n = UV, где U ≤ V. Допустим, что n – нечетно => U,V – нечетны. Поэтому можно положить $x = \frac{U+V}{2}, y = \frac{V-U}{2}, n = x^2 - y^2, 0 \le y < x \le n$. Метод Ферма заключается в поиске x, y удовлетворяющим этим соотношениям. Тогда n = (x - y)(x + y). Следующий алгоритм показывает, как, не выполняя операций деления, можно реал. мет. Ферма. (см. Кнут, т.2, стр. 414):

(1) $x' \leftarrow 2\lfloor\sqrt{n}\rfloor + 1; y' \leftarrow 1; r \leftarrow \lfloor\sqrt{n}\rfloor^2 - n;$ // во время работы алгоритма x', y', r отвечают соответственно 2x+1, 2y+1, $x^2 + y^2 - n^2$ из формул выше.

(2) Если r ≤ 0, то перейти в шаг (4)

(3) $r \leftarrow r - y'; y' \leftarrow y'+2; возвратиться\ в\ шаг\ (2)$

(4) если r=0, то конец алгоритма: имеем n=((x'-y')/2)((x'+y'-2)/2) и (x'-y')/2 есть наиб. множитель числа n≤ $\sqrt{n}$

(5) $r \leftarrow r + x'; x' \leftarrow x'+2; возвратиться\ в\ шаг\ (3)$

Число шагов, выполняемых алгоритмом для нахождения U, V пропорционально $(x'-y'-2)/2 - \lfloor\sqrt{n}\rfloor = V - \lfloor\sqrt{n}\rfloor, т.е. тоже\ o(\sqrt{n}).$

Ферма, конечно же, действовал не так. Он обычно рассматривал величину $x^2 - n$ и, исходя из последних цифр, делал вывод, является она полным квадратом (последние две цифры полного квадрата должны быть 00, a1, a4, 25, b6, a9, где a – четное, a b – нечетная цифра). Поэтому Ферма избегал шагов (2) и (3).

Метод Ферма эффективен, если делители n близки друг к другу.

Метод Ферма является, по сути, переборным, т.е. неэффективным. Рассмотрим следующий метод:

Метод базы разложения (метод Диксона)

Определение: Для данной базы разложения Д={ $p_1, p_2, ..., p_k$ } различных простых чисел за исключением p_1=-1, назовем число b Д-числом для данного n, если абсолютное значение $b^2 \pmod n$ раскладывается на простые множители из базы Д.

Пример: пусть n=4633; Д = {-1, 2, 3}. Тогда числа 67, 68, 69 являются Д-числами.

$67^2 \equiv -144 (4633) \equiv -1 \cdot 2^4 \cdot 3^2 (4633)$

$68^2 \equiv -9 (4633) \equiv -1 \cdot 3^2 (4633)$

$69^2 \equiv 128 (4633) \equiv 2^5 (4633)$

Определение: пусть F_2^h -- векторное пространство всех двоичных чисел длиной h бит. Тогда Д—числам можно сопоставить элементы из F_2^h: если

$b^2 = \prod_{j=1}^{h} p_j^{\alpha_j} \pmod n \Rightarrow V_j = \alpha_j \bmod 2.$ Таким образом строится $\vec{V} = (V_1 ... V_h)$

Пример: 67 ~ (100), 68 ~ (100), 69 ~ (010)

Идея алгоритма приблизительно та же, что и в методе Ферма. Мы будем искать s, t:
$s^2 \neq \pm t(n), тогда [(s^2 - t^2) \vdots n]\ или\ [(s+t)(s-t) \vdots n]$ и делитель n равен НОД(s+t, n).

Пусть у нас есть набор векторов $\vec{V}_i$: при покомпонентном сложении по mod 2 они дают нулевой вектор. Тогда произведение соответствующих b_i содержит только четные степени $p_i \in Д.Обозначим\ a : b^2 \equiv a(\mathrm{mod}\, n).$

Тогда для таких b_i имеем $Пa_i = \prod_{j=1}^{h} p_i^{\sum \alpha_{ij}}, где \sum_i \alpha_{ij}$ -- четные.

Тогда правая часть является квадратом числа = $\prod_{j=1}^{h} p_j^{\gamma_j}, где\ \gamma_j = 1/2 \sum_i \alpha_{ij}$

Теперь $b \equiv \prod_{j=1}^{h} b_i (\mathrm{mod}\, n); c \equiv \prod_{j=1}^{h} p_j^{\gamma_j} (\mathrm{mod}\, n).$

Если получится b=c или b=-c, то попытка неудачна и надо начинать сначала, с поиска нулевых векторов. В противном случае, делитель n есть НОД(b+c, n).

Сложность метода: $o(e^{c\sqrt{\log n \cdot \log\log n}})$.

Существует множество других способов с такой же сложностью. Наиболее популярный из них (и, до недавнего времени, наиболее быстрый) – Quadratic Sieve [707,890, 709]. Для 664-битного (200-разр.) числа такая оценка сложности дает 10^{23} операций. Для машины с 1 млн. итераций в сек. это дает 3.7 млрд. лет.

<u>Ро-метод Полларда.</u>

Перебор всех чисел до B гарантирует факторизацию всех чисел до B^2. За это же время следующий алгоритм факторизует любое число до B^4 (при удаче). Алгоритм эвристический => мы не можем гарантировать время работы или успешное окончание, но на практике дает хорошие результаты.

Pollard-Rho(n)

1. i←1
2. $x_1 \leftarrow Random(0, n-1)$ // инициализация случайным элементом из Z_n
3. $y \leftarrow x_1$
4. k←2 // инициализация к
5. while True do //бесконечный цикл поиска делителей n
6. i ← i + 1 //увеличение индекса
7. $x_i \leftarrow (x_{i-1}^2 - 1) \bmod n$ //вычисление следующего элемента бесконечной //последовательности x'ов
8. $d \leftarrow \gcd(y - x_i, n)$ //попытка нахождения делителя n с помощью
9. if d≠1 and d≠n //сохраненного y и текущего x_i если удалось, то печатаем d
10. then print d
11. if i = k //сохранение элемента с номером кратным
12. then $y \leftarrow x_i$ //степени двойки и увеличение к в 2 раза
13. k←2k

Что характерно, алгоритм работает <u>без</u> индексов тоже, т.к. нам необходимо сохранять только последние значения x_i.

Процедура Pollard-Rho выглядит несколько загадочно. Отметим, однако, что процедура никогда не печатает неверный ответ. Процедура может ничего не напечатать, но мы покажем, что есть основания ожидать напечатания делителя p после приблизительно $\sqrt{p}$ итераций цикла while. Т.о. если n – составное, то можно ожидать нахождения делителей после $n^{1/4}$ операций (т.к. все делители $\le \sqrt{n}$.)

Анализ процедуры начнем с изучения вопроса: как много шагов требуется случайной последовательности по модулю n для повторения себя? Т.к. x_n -- конечное и

последовательность зависит только от предыдущего элемента, то последовательность непременно начнет повторяться. Как только найдутся $j < i : x_i = x_j$, мы попали в цикл, т.к. $x_{i+1} = x_{j+1}$ и т.д. Поэтому и название: «ρ-эвристика», т.к. последовательность $x_1, x_2, ..., x_{j-1}$ может быть нарисована как «хвостик» ρ, а цикл $x_j, x_{j+1}, ..., x_i$ -- как «тело» ρ (см. 33.7).

Так вот, если последовательность случайная, то каждый x_i можно воспринимать как случайно вынутый из x_n, согласно равномерному распределению на x_n согласно «парадоксу дней рождения» (см. 6.6.1), ожидаемое число шагов до зацикливания последовательности = $\Theta(\sqrt{n})$.

Что такое «парадокс дней рождения»? 183; 23

Будем считать, что функция $(x^2 - 1) \bmod n$ действует как «случайная» функция (это даст правдоподобные результаты) => $\{x_i\}$ повторяется через $\Theta(\sqrt{n})$.

⌉ p – нетривиальный делитель n: $\gcd(p, \frac{n}{p}) = 1$. Напр., если n: $n = p_1^{l_1} \cdot ... \cdot p_n^{l_n}$, то можно взять $p_1^{l_1}$ вместо нашего p_1.

Тогда последовательности $\{x_i\}$ соответствует последовательность $\{x_i'\} : x_i' = x_i \bmod p \; \forall i$. Отсюда по варианту китайской теоремы об остатках: $x_{i+1}' = (x_i'^2 - 1) \bmod p$, т.к. (x mod n) mod p = x mod p.

Рассуждая так же, как и раньше, получаем, что количество шагов до зацикливания $\{x_i'\} = \Theta(\sqrt{n})$. Если p мало по сравнению с n, то $\{x_i'\}$ может зациклиться много быстрее, чем $\{x_i\}$. (см. рис. 33.7)

⌉ t обозначим индекс первого повторяющегося эл-та в $\{x_i\}$, а U>0 обозначим длину цикла, т.е. t, U – наименьшие значения: $x_{t+i}' = x_{t+u+i}' \; \forall i \geq 0.$ Ясно, что ожидаемые значения t, U равняются $\Theta(\sqrt{p})$.

Отметим, что если $x_{t+i}' = x_{t+u+i}' \Rightarrow x_{t+u+i} - x_{t+i} \vdots p \Rightarrow НОД(x_{t+u+i} - x_{t+i}, n) > 1.$

Таким образом, как только алгоритм сохранил в y любое значение x_k : R≥t, значение (y mod p) всегда лежит на цикле по модулю p. Рано или поздно, к установят в значение большее, чем U. Тогда процедура проделает полный круг по циклу и по модулю p без изменений y, и как только x_i «добегает» до сохраненного значения (y mod p), т.е. когда $x_i \equiv$y mod p, мы находим делитель n (согласно замечанию, обведенному рамкой).

Т.к. ожидаемые значения t, U = $\Theta(\sqrt{p})$, ожидаемое количество шагов для нахождения делителя p тоже равно $\Theta(\sqrt{p})$.

<u>Комментарии к ρ-методу Полларда.</u>

Существует две причины, по которым алгоритм может работать не совсем так, как ожидалось.

1) Эвристический анализ времени работы алгоритма не строг, и возможно, что цикл значений по модулю p может оказаться много больше, чем $\sqrt{p}$. В этом случае алгоритм работает правильно, но много медленнее, чем хочется. На практике, однако, это не является реальной угрозой.
2) Делители, производимые алгоритмом, могут всегда быть одним из тривиальных делителей –1 или n. Напр., пусть U = pq, где p, q – простые. Может случиться так, что t, U для p будут совпадать с t, U для q => делитель p будет всегда возникать одновременно

с делителем q => всегда будем получать делитель pq = n. На практике, при необходимости алгоритм может рестартовать в виде $x_{i+1} \leftarrow (x_i^2 - c) \bmod n$. Значений c=0 и c=2 надо избегать по некоторым дополнительным соображениям, но ∀ другое с сойдет.

Приведенный анализ эвристичен и нестрог, но на практике алгоритм ведет себя именно так, как описано. Метод особенно хорош для нахождения маленьких делителей больших чисел.

Для разбиения сост. чисел β-битовых: $O(n^{1/4}) = O(2^{\beta/4})$ арифметических операций.

Прочие методы.

На данный момент самым быстрым алгоритмом факторизации является Number Field Sieve [527, 11, 538]. Он эффективнее, чем QS для больших чисел (равная скорость у них достигается на числах от 110 до 135 десятичных знаков). Для меньших чисел QS эффективнее, для больших чисел NFS эффективнее. Его сложность: $e^{\sqrt[3]{\ln n}\sqrt[3]{(\ln \ln n)^2}}$.

В наше время, 110-разрядные числа регулярно факторизуют. Каждое 10-кратное увеличение компьютерной мощности позволяет факторизовать числа на 10 разрядов длиннее. По оценке Ленстры, 155-разряд. (512-бит) число можно факторизовать с NFS за год на 50 больших машинах.

Так что 1024-битные числа вполне надежны, но область находится в развитии.

Сильные простые числа.

Для избежания некоторых методов факторизации, рекомендуют использовать для системы RSA сильные простые числа (srong primes), т.е. такие числа, чтобы

А) (p-1) и (q-1) имели большие простые делители;

Б) (p + 1) и (q + 1) имели большие простые делители.

(Есть и другие определения, предъявляемые более суровые требования к p и q). В общем, лучше использовать сильные простые числа, хоть это и затрудняет генерацию простых чисел.

10. RSA: дешифрация и генерация ключей. Скорость RSA. Гибридные криптосистемы. Надежность RSA.

Напомним основные идеи RSA:

Пусть p, q – простые; n = pq; e: НОД(e, (p - 1)(q - 1)) = 1; d: ed ≡ 1 mod (p - 1)(q - 1).
Формула итерации: $e = M^e \bmod n$, формула дешифрации: $M = c^d \bmod n$
Открытый ключ: (e,n)Закрытый ключ: (d,n)

Генерация ключей.

Итак, пусть у нас уже есть необходимые p,q. Теперь надо выбрать e. В принципе, это несложно, т.к. единственное требование – это взаимная простота с (p-1)(q-1). Раньше поэтому говорили, что лучше всего выбрать небольшое e, т.к. тогда будет более быстрая шифрация. Однако , недавно J. Hasted показал, что такая схема и ненадежна (см. [417], J. Hasted “Solving simultaneous modular equations of low degree”, SIAM J. Comput, 17(2), 1988, pp.336-341). Другая атака, придуманная Майклом Винером, может отыскать d в тех случаях, когда e – число порядка до n/4 [878]. То же самое касается числа d.

Так что надо выбирать большие d, e. Но условие НОД(e, (p-1)(q-1))=1 не представляет существенной проблемы. Вообще, вероятность того, что два случайно выбранных числа взаимно просты ~3/5 (см. Кнут, т.ч. стр. 366, теор. Чезаро; точное значение = $6/\pi^2$).

После выбора e, d может быть найдено за полином времени. Процедура поиска d:

Определение: $Z_n = \{0,1,...,n-1\}, Z_{n^*} = \{x \in Z_n : x > 0, НОД(x,n) = 1\}$, т.е. Z_{n^*} состоит из ненулевых элементов Z_n, взаимно простых с n.

```
Function Inverse (n, x) returns integer;
/* для n>0 и x∈ Z_n* возвращает U∈ Z_n* : Ux ≡ 1 mod n */
procedure Update (a, b);
        temp: = b; b: = a - y * temp; a: = temp;
    end Update;
    g: = n; h: = x; w: = 1; z: = 0; v: = 0; r: = 1;
    while (h>0) do
    y:=[g/h]; Update (g, h); Update (w, z); Update (v, r);
    end while;
    return (v mod n);
end Inverse;
```

Дешифрация в RSA.

Пусть есть зашифрованный текст C. Мы хотим эффективно посчитать $M = C^d \bmod n$. Обозначим $a = C^d \bmod p, b = C^d \bmod q$. Разобьем d: d = k * (p - 1) + r. Вспомним т. Эйлера-Ферма (малая теорема Ферма): пусть p – простое a∈ Z_{n^*}. Тогда $a^{p-1} \equiv 1 \bmod p$. Используя эту теорему, выполним преобразования:

$a = (C^{p-1})^k \cdot C^r \bmod p = 1^k \cdot C^r \bmod p = (C \bmod p)^r \bmod p$.

Аналогично, если $d = j*(q-1)+s, mo\, b = (C \bmod q)^s \bmod q$.

Наконец, учитывая то, что r = d mod p - 1, a s = d mod q - 1, получим алгоритм дешифрации текста:

1. посчитать $a = (C \bmod p)^{d \bmod(p-1)} \bmod p, b = (C \bmod q)^{d \bmod(q-1)} mdq$
2. найти U:0<U<p & uq ≡ 1 mod p .
3. исследовать одну из формул:

$$M = (((a - (b \bmod p)) * U) \bmod p) * q + b, если\, a \geq b \bmod p$$

$$M = (((a + p - (b \bmod p)) * U) \bmod p) * q + b, если\, a < b \bmod p$$

Отметим, что п.3 есть в чистом виде формулировка т.н. китайской теоремы об остатках.

Это практически оптимальная формула и легко найти с помощью использованной выше функции Inverse. M легко считается, если найдены a и b, а нахождение a и b отнимет не больше, чем 2logp и 2logq умножений соответственно (см. Кнут т.2).

Т.о. шифрация и дешифрация может быть реализованы за O(log n), т.е. за линейное время. При этом они все равно остаются, хоть и элемент, но time-consuming операциями, т.к. связаны с умножением и делением по модулю 100 – и более разр. чисел.

Скорость RSA. Гибридные криптосистемы.

Самые быстрые реализации RSA в 1000 раз медленнее, чем DES. В 1989 г. самые быстрые VKSI-реализации давали скорость 64 Кб/с. Сейчас скорость ~1 Мб/с. (для сравнения DES(ГОСТ) – от 10 до 100 Мб/с).

В software, RSA ~ в 100 раз медленнее DES'a.

Числа могут меняться, но RSA никогда не достигнет скорости симметричных алгоритмов. Вот почему на практике большинство систем использует RSA исключительно для обмена DES'овскими ключами, а затем шифруют все прочее DES'ом. Такой подход называется гибридными криптосистемами.

Криптография с открытым ключом идеальна для шифрации и распределения ключей, аутентификации и пр. Симметричная криптография идеальна для шифрования файлов и каналов связи.

Ненадежные варианты RSA.

Помимо уже упоминавшейся RSA с малыми экспонентами, существуют и другие ненадежные варианты RSA, связанные с ошибками в реализации или в использовании.

RSA с общим модулем.

Т.к. вычислять n для каждого пользователя накладно, то возникает соблазн дать всем пользователям системы одинаковое n. К сожалению, это сильно понижает криптостойкость системы.

Наиболее очевидная проблема в том, что если одно и то же сообщение когда-либо будет зашифровано двумя разными ключами по одинаковому модулю, и эти два ключа будут взаимно простыми (что, скорее всего, так и будет), то исходный текст может быть восстановлен без обоих ключей расшифровки. [820]

Другая схема взлома: пусть p – исходное сообщение; e1, e2 – ключи шифрования, u – общий модуль.

Два шифртекста: $c1 = p^{e1} \bmod n; c2 = p^{e2} \bmod n.$

Криптоаналитик знает n, e1, e2, c1, c2. Тогда p может быть восстановлен следующим образом: т.к. e1, e2 – взаимно просты, то по расширенному алгоритму Евклида можно найти r, s: rxe1+sxe2=1. Предположим, что r – отрицательное (либо r либо s должно быть отрицательным), то евклид. алгоритм может быть использован для отыскания e_1^{-1}. Тогда:

$(c_1^{-1})^{-r} \times c_2^s = p \bmod n$.

Chosen Ciphtrtext атаки против RSA.

Сценарий 1. Ева подслушивает канал Алисы и перехватывает сообщение C, зашифрованное с помощью открытого ключа Алисы. Ева хочет прочитать это сообщение, т.е. p: p= c^d .

Для восст. p, Ева вначале выбирает случайное число r: r<n и добывает открытый ключ Алисы e. После этого вычисляется:

$x = r^r \bmod n, y = x \times c \bmod n, t = r^{-1} \bmod n$

Т.к. $x = r^e \bmod n, тогда\ r = x^d \bmod n \Rightarrow t = x^{-d} \bmod n.$

Теперь Ева подсовывает у на подпись Алисе (необходимо при этом, чтобы Алиса подписала сообщение целиком, а не хэш от сообщения). Алиса никогда не видела у раньше и посылает Еве: $u = y^d \bmod n.$ После этого Ева высчитывает:

$t \times u \bmod n = x^{-d} \times y^d \bmod n = x^{-d} \times x^d \times c^d \bmod n = c^d \bmod n = p$.

Сценарий 2. Предположим, что Тренд – компьютерный нотариус. Если Алисе требуется заверить документ, то она посылает его Тренту. Трент подписывает его цифр. подписью RSA и отсылает обратно (при этом не использует односторон. хэш функции).

Маллет хочет, чтобы Трент подписал сообщение N, которое Трент не стал бы подписывать в обычных условиях. (может быть там неправильная временная отметка или документ написан от имени другого лица и т.д.)

Вначале, Маллет берет некоторое значение X и считает $Y = X^e$ (e – открытый ключ Трента и должен быть доступен для проверки его подписей). Далее вычисляем M=YN и M посылается на подпись Тренту. Трент возвращает $M^d \bmod n$. Теперь Маллет считает

$(M^d \bmod n)x^{-1} = N^d \bmod n$, является подписью N.

Вообще имеется несколько методов, которые Маллет может использовать для вычисления подобных вещей [240, 270, 289]. Слабость, кот. использовать все эти алгоритмы состоит в следующем: $(XM)^d = X^d M^d \bmod n$ (т.е. возведение в степень сохр. мультипликат.)

Наиболее общим образом это отражено в

Сценарии 3. Еве хочется, чтобы Алиса подписала некоторое сообщение M_3. Она генерит

$M_1, M_2: M_3 \equiv M_1 \times M_2 \pmod n$

Если ей удастся заставить Алису подписать M_1 и M_2, то она имеет и подпись к M_3:

$$M_3^d \pmod n = M_1^d \pmod n \times M_2^d \pmod n.$$

Мораль: не использовать RSA для подписи случайных документов, представляемых вам незнакомцами. Обязательно использовать одностороннюю хэш функцию или другой алгоритм подписи.

Теоремы о надежности RSA.

Как мы уже видели, RSA является мультипликативной системой: E(X)E(Y)=E(XY).

Следовательно, если злоумышленник сумеет расшифр. любую полиномиальную часть шифртекста за полином время, то он сможет расшифр. все шифртекты за случайное полином. время: для расшифровки E(X) достаточно найти (методом проб и ошибок) Y: E(XY), равное E(X)E(Y) может быть расшифровано (получая XY), а затем поделив результат на Y, получая X.

Т.о. можно сказать, что RSA либо равномерно надежно, либо равномерно ненадежно.

Были доказаны даже более сильные результаты. Наример, было показано, что если полиномиальная часть шифртекста не может быть расшифрована за полином. время, то не удастся угадать даже наименьший значимый бит текста с вероятностью, больше, чем ε (см. Alexi W.B., Chor B., Goldreich O., Schnorr C.B. RSA and Rabin functions: certain parts are as hard as a while, SIAM J. Comput. 17(2), 1988, pp. 194-209; 18, 83 Handboor Th. C.Sc. References to 13).

В системе RSA необходимо учитывать также след. возможность , выбрав произведение s, можно вычислить значение $M' = s^l$, т.е. произвольное значение можно представить как подпись некоторого сообщения. Такие фикт. сообщения, естественно, является случайным. Если же требуется подпис. случ. сообщения, то надо добавлять заранее уговор. вектор длиной θ (тогда вероятность подделки = $2^{-\theta}$).

11. Цифровая подпись.

Что хорошего в обычной подписи на документах?

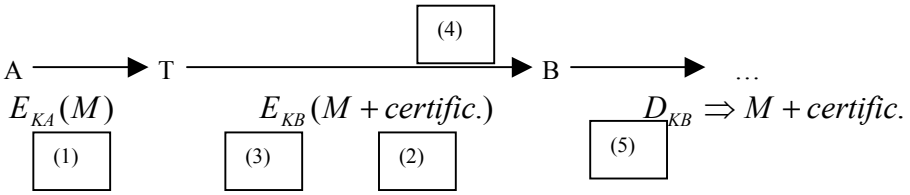
1. Подпись нельзя подделать (=> является доказательством того, что подписывающий добровольно подписал документ).
2. Подпись аутентична (ею можно убедить получателя в том, что документ именно от отправителя).
3. Подпись нельзя переиспользовать (подпись является частью документа и ее нельзя перенести на другой документ).
4. Подписанный документ неизменяем (после того, как документ подписан, он не может быть изменен)
5. От подписи нельзя отказаться (документ и подпись к нему является физическими вещами. Подписывающий не может позже заявлять, что он или она не подписывали документ).

Конечно, в реальной жизни все эти положения не выполняются, но нам приходится жить с этими проблемами; в конце концов, жульничать трудно и есть риск быть обнаруженным.

Нам хотелось бы иметь электронную подпись с такими же свойствами, но есть проблемы (легкость копирования + возможность изменения постфактум).

Протокол 1. (подпись с симм. криптосистемой и арбитратором)

Алиса хочет послать сообщение Бобу. Трент – арбитратор.



Протокол работает, но Трент – узкое место для любой коммуникационной системы; еще труднее создать доверие к Тренту. Его базы данных должны быть абсолютно надежны. Если эти данные попадут наружу, то начнется хаос. На практике это не работает.

Протокол 2. Подпись документов с помощью криптографии с открытым ключом.

(1) A использовать секретный ключ алг. цифр. подписи и подпис. Сообщение
(2) A посыл. док-т B
(3) B исп. откр. ключ A для проверки подписи.

Проблема: B можно переиспользовать (документ + подпись). Это не страшно, если подписан контракт, но что, если это электр. чек? Тогда B можно получать деньги дважды, трижды и т.д.

Для избежания этого используют отметку о времени подписи. (=> банк записывает все timestamps в базу данных).

Протокол 3. Подпись с крипт. с откр. ключом и одност. хэш-функцией.

Обычно криптогр. с открытым ключом слишком медленна (+ соображения безопасности, см. предыдущ. лекцию). Поэтому:

(1) A считает одност. хэш-ф. от документа;
(2) A подпис. хэш своим секретным ключом;
(3) A посылает (документ, подписанный хэш) Бобу;
(4) B счит. хэш от документа и проверяет подпись с помощью открытого ключа A. Если оба хэша сошлись, то подпись легальна.

Отсюда мораль – центральная БД может хранить хэши, не зная содержимого файлов (подаются только хэши; фиксирует user и время подачи). Т.о. можно зафиксировать C, не разглашая документ. Документ тогда потребуется публиковать только в случае разбирательств по C.

Жульничества с цифровой подписью.

Алиса может жульничать со своей цифровой подписью, и с этим ничего не сделаешь. Если она хочет подписать документ и затем заявить, что она не делала, то вот что она будет говорить: подписывает документ, затем намеренно теряет ключ или анонимно его публикует. Теперь любой нашедший ключ может выдавать себя за Алису. Поэтому Алиса заявляет, что ее ключ скомпрометирован и что она на самом деле ничего не подписывала.

Можно пытаться использовать timestamps, но это не очень помогает.

Одно из решений – получателю документа давать его для отметки времени арбитратору (=> документ был подписан ко времени отметки). Теперь если Алиса притворяется, что потеряла ключ или действительно его теряет, то не имеющими силы признаются только те документы, которые подписаны после объявления о пропаже (такая же схема используется при утере или краже кредитных карт).

Другое решение – хранить секретные ключи в защищенных от взлома модулях (tamper-resistant) => Алисе не удастся добраться до своего ключа для противозаконного его использования.

Применение цифровых подписей.

Одно из самых первых применений цифровых подписей – обеспечение контроля за выполнением запрета на ядерные испытания. США и СССР установили датчики на территории друг друга. Проблем две:

1) наблюдающая страна должна удостовериться в том, что ей не подсовывают ложные данные;
2) страна, на территории которой установлены датчики должна убедиться в том, что датчик посылает только данные, необходимые для мониторинга.

С помощью цифровой подписи тестируемая страна может читать, но не изменять данные сейсмометров; тестирующая страна удостоверяется, что данные не изменялись.

Digital Signature Algorithm (DSA)

В 1993 г. в США был принят новый стандарт для цифровой подписи (Digital Signature Standard (DSS)), включающий в себя следующий алгоритм подписи:

В алгоритме используются следующие параметры:

p – простое число длиной L битов, где L∈ [512, 1024], L⋮64

q – 160-битный простой делитель числа (p-1)

$g = h^{(p-1)/q} \bmod p, где\, h : h < p-1 : h^{(p-1)q} \bmod p > 1$

к: 0<к<q

$y = g^x \bmod p$

M – исходное сообщение, которое требуется подписать.

k—случайное целое число: 0<k<q

H – одност. хэш-функция, реализуемая по алгоритму SHA (Secure Hash Algorithm)

Первые три параметра (p, q, g) – открыты и могут быть общими для сети пользователей;

x – секретный ключ (разный для различных пользователей)

y – открытый ключ (разный для различных пользователей)

Алгоритм подписи:

(1) выбрать случайное число k<q

(2) посчитать:

$r = (g^k \bmod p) \bmod q, s = (k^{-1}(H(M) + xr) \bmod q, где (r,s) - - подпись M$

Алгоритм проверки подписи:

(1) посчитать

$$W = s^{-1} \bmod q,\ u_1 = (H(M) \cdot W) \bmod q,\ u_2 = (r \cdot W) \bmod q,\ v = ((g^{u_1} \cdot y^{u_2}) \bmod p) \bmod q$$

(2) Если v=r, то подпись достоверна.

Доказательство того, что v=r при совпадении M и M', r и r', s и s' -- см. статью «Digital Signature Standard proposed by NIST, CASM, July 1992, vol. 35, NA 7»

Предвычисления для DSA.

В реальных применениях DSA может быть ускорен за счет предвычислений . Отметим, что значение r не зависит от сообщения => можно образовать последовательность из случайно выбранных значений k и для каждого из них посчитать r. Кроме того, можно заранее посчитать k^{-1}.

Тогда можно считать s для данных r, k^{-1}. Это значительно ускорит алгоритм.

Критика DSA.

Основная критика – DSA не RSA! К 1993 г. в RSA были вложены огромные деньги, и крупные компании не хотели терять свои деньги (IBM, Microsoft, Sun, DEC, Novell, Apple, Lotus…).

Всего NIST получило 109 комментариев. Из них 4 были опубликованы в CASM, 1992, vol. 35, no. 7.

Основное достоинство DSA в том, что он ?????-free.

Основным недостатком DSA было то, что в изначальном предложении длина ключа была зафиксирована в 512 бит. Поэтому его длину увеличили до 1024 бит и сделали изменяемой.

Генерация простых чисел в DSA.

Lenstra & Haber отметили, что некоторые модули значительно проще взломать, чем другие [535]. Если кто-нибудь использует такой «плохой» модуль, то его подпись будет легко подделать.

В принципе, это не проблема, т.к. 1) такие модули легко отслеживать; 2) такие модули редки и вероятность их использования чрезвычайно мала.

NIST рекомендует метод генерации простых чисел p, q: p-1⋮q. (при этом p длиной 512-1024 бит, q – 160 бит).

Пусть L-1=n*160+b, где L – длина p, a n, b – два числа.

(1) Выбрать начальную последовательность длиной как минимум 160 бит и назвать ее S. Пусть g=длина(S) в битах.
(2) U=SHA(S) XOR SHA((S+1) mod 2^g)
(3) Сформировать q из U путем установки наибольшего и наименьшего значащих битов в 1.
(4) Проверить, является ли q простым.
(5) Если q – простое, то вернуться в шаг 1ю
(6) Пусть C=0, N=0
(7) Для k=0,1,…,n положить $V_k = SHA((S + N + k) \bmod 2^g)$
(8) Посчитать $W = V_0 + V_1 \times 2^{160} + ... + (V_n \bmod 2^b) \times 2^{n \times 160}$ *и* $X = W + 2^{L-1}$ (отметим, что X – это число длиной 2^L бит)
(9) Положить p = X - ((X mod 2q) - 1). (отметим, что p ≡ 1 mod 2q)
(10) Если $p < 2^{L-1}$, то перейти в (13)
(11) Проверить, является ли p простым
(12) Если p простое, то перейти в (15)
(13) C = C + 1; N = N + n + 1
(14) Если C=4096, то перейти в (1), иначе в (7)
(15) Сохраняем S и C, используемые для генерации p, q.

12. Односторонняя хэш-функция на примере Secure Hash Algorithm. Методы поиска коллизий односторонней хэш-функции.

Односторонние хэш-функции производят выход фиксированной длины (от 128 до 160 бит). Большинство односторонних х.-ф. построены на одностор. функции, которая производит значение длины n по двум входным блокам длиной n каждый. Обычно, входом функции служат блок текста и хэш предыдущего блока:

Хэш последнего блока становится хэшем всего сообщения. Т.о. удается произвести хэш фиксированной длины независимо от длины входного сообщения. Эта техника помогает избежать потенциальных проблем с сообщениями с одинаковым хэшем, но различной длиной [587, 236].

Есть некоторая теория о том, что если х.-ф. надежна для одного блока, то х.-ф. также надежна для строки произвольной длины (с использованием данной техники). Доказанные результаты значительно менее сильны [632, 589, 236].

Secure Hash Algorithm (SHA).

Специфицирован NIST и NSA для использования в DSA. По сообщению длиной < 2^{64} битов выдает 160-битное значение (message digest). Затем digest подают на вход DSA, который считает подпись сообщения. На проверяемой стороне по сообщению должны получить тот же дайджест. SHA надежен, т.к. вычислительно невозможно найти сообщение по его дайджесту либо найти два различных сообщения с одинаковым дайджестом. Любое изменение (при передаче) сообщения с очень высокой вероятностью приведет к другому дайджесту (=> подпись не совпадает).

Описание алгоритма SHA.

В начале, сообщение достраивается до длины, кратной 512 (padding): к сообщению добавляется «1», затем столько нулей, сколько надо до длины, кратной 512 минус 64 бита, затем 64-битное представление длины сообщения до padding'а (алгоритм padding'а такой же, как в MD5).

Далее инициализируются пять переменных:
A=67 45 23 01
B=EF CD AB 89
C=98 BA DC FE
D=10 32 54 76
E=C3 D2 E1 F0
(в MD5 четыре первых, но там и выход 128-битный)

Затем начинается основной цикл, в нем обрабатывается 512 бит за раз (и так со всеми блоками сообщения).

Затем: AA=A; BB=B; CC=C; DD=D; EE=E;

В основном цикле 4 раунда по 20 операций (в MD5 4 раунда по 16). В каждой операции производятся нелинейные операции, сдвиги и сложения.

Пусть $f_t(X,Y,Z) = XY\ OR\ (NOT\ X)Z$ для первых 20 операций

$f_t(X,Y,Z) = X\ XOR\ Y\ XOR\ Z$ для вторых 20

$f_t(X,Y,Z) = XY\ OR\ XZ\ OR\ YZ$ третьих 20

$f_t(X,Y,Z) = X\ XOR\ Y\ XOR\ Z$ четвертых 20

В алгоритме также используется 4 константы:

$K_t = 5A827999$	первые 20	$f_1(x,y,z) = ((x\ \&\ y)\|(\sim x\ \&\ z))$
$K_t = 6ED9EBA1$	вторые 20	$f_2(x,y,z) = (x\hat{}\ y\hat{}\ z)$
$K_t = 8F1BBCDC$	третьи 20	$f_3(x,y,z) = ((x\ \&\ y)\|(x\ \&\ z)\|(y\ \&\ z))$
$K_t = CA62C1D1$	четвертые 20	$f_4(x,y,z) = (x\hat{}\ y\hat{}\ z)$

Блок сообщения преобраз. из 16 32-битн. слов ($M_0,..,M_{15}$) в 80 32-битн. слов ($W_0,...,W_{79}$) по следующему алгоритму:

$W_t = M_t, t = 0..15$

$W_t = W_{t-3}\ XOR\ W_{t-8}\ XOR\ W_{t-14}\ XOR\ W_{t-16}, t = 16..79$

Если t – номер операции (1..80) M_j -- j-ый подблок сообщения (0..15), <<<S – циклический сдвиг влево на S бит, то 80 операций выглядят как:

TEMP=(A<<<5) + $f_t(B,C,D) + E + W_t + K_t$

E = D
D = C
C = B<<<30
B = A
A = TEMP

Далее A, B, C, D, E прибавляют к AA, BB, CC, DD, EE и берется следующий блок. Результат – конкатенация AA, BB, CC, DD, EE.

Надежность SHA.

SHA очень похож на MD4, скорее вариант MD4, чем его перепланировка. С другой стороны, Рон Ривест придумал MD5 (взамен MD4) и опубликовал причины, побудившие его сделать это. Проведем сравнение:

1. «4-й раунд добавлен» -- в SHA тоже (хоть и использует ту же f, что и во втором раунде).
2. «В каждом шаге используется собственная константа» – в SHA оставлена как в MD4 – константа используется 20 раундов подряд.
3. « f_2 было изменено с (XY or XZ or YZ) на (XZ or Y or not(Z)) для того, чтобы сделать ее менее симметричной» -- в SHA оставлено как было.
4. «Результат зависит теперь от каждого шага => быстрее эффект лавины» -- в SHA тоже + добавлена пятая пер-ая (это мешает использовать атаку de Boer-Bossebers).
5. Порядок, в котором входные слова используются в раундах 2, 3 изменен, чтобы они меньше походили друг на друга» -- в SHA вообще по-другому (циклический сдвиг)
6. «Сдвиги в каждом раунде разные => быстрее эффект лавины» -- в SHA оставлено как в MD4 – сдвиг постоянен, взаимно простой с длиной слова.

T.o.: SHA = MD4 + expand transform + extra round + better-avalanche
MD5 = MD4 + improved bit-bashing + extra round + better- avalanche

Замечание: можно использовать блочный алгоритм как одностороннюю хэш функцию (если блочный алгоритм надежен, то и х.-ф. надежна, но надо хранить в секрете ключ).

Метод увеличения хэша.

Для получения хэш-значения с большей длиной, чем это позволяет выбранная хэш-функция, был предложен следующий метод:

1) Сгенерировать хэш от сообщения
2) Добавить хэш в конец сообщения (append)
3) Сгенерировать хэш от конкатенации сообщения и хэша
4) Получить длинное хэщ-значение путем конкатенации хэш-значения из пункта 1) и хэш-значения из пункта 3)
5) Повторять шаги (1-3) до получения необходимой длины.

Надежность или ненадежность данного метода не доказаны.

Основные методы криптоанализа односто. хэш-функций.

1. «Лобовая».

Имея h(M), злоумышленник должен найти $M' : h(M) = h(M')$. (=> может заявлять, что подписано M'). Если х.-ф. дает m-битную строку, то этот метод требует 2^m случайных сообщений.

2. «Парадокс дней рождения»

Должно быть, трудно найти два случайных сообщения M и M': h(M) = h(M'), это значит легче, чем предыдущая задача.

«Парадокс дней рождения» -- известная статистическая задача. Сколько человек должно находиться в комнате, чтобы с вероятность >0.5 у кого-то был одинаковый с вами день рождения? Ответ: 183. Теперь, сколько человек должно находиться в комнате, чтобы с вероятностью >0.5 хоть у кого-то из них совпали дни рождения? Ответ удивительно мал: 23, т.к. 23 человека ~253 парам.

Нахождение кого-то с определенным днем рождения аналогично «лобовой атаке». Нахождения двух людей с одинаковым случайным ДР аналогично данной атаке.

Предположим, что наилучший способ атаки одност. х.-ф. – грубая сила. Одност. х.-ф. производит значение длиной m, тогда для данного метода потребуется $2^{m/2}$ значение. Если машина хэширует 1 млн. сообщений в секунду, то потребуется 600,000 лет для нахождения второго сообщения с таким же 64-битным хэшем. Та же машина может найти пару сообщений (случайных) с одинаковым хэшем за час.

Следующий протокол впервые описан Ювалем (Yuval, [899]):

(1) Алиса готовит 2 версии контракта – один выгодный для Боба, а другой разоряющий его;
(2) Алиса делает несколько малозначительных изменений к каждому документу и считает хэш-значения каждый раз (напр., заменить SPACE на SPACE-BACKSPACE-SPACE, добавить один или два пробела перед переводом строки и т.д. Только путем одного «изменения / оставления как было» на каждой строке Алиса может сгенерировать 2^{32} документов).
(3) Алиса сравнивает набор хэшей для обоих документов, подыскивая одинаковые пары (если х.-ф. = 64 бит, то обычно хватит 2^{32} пар). Выбирается та пара, что дает одинаковые хэши.
(4) Алиса подсовывает Бобу выгодный контракт; Боб подписывает его хэш
(5) Алиса может теперь доказать, что Боб подписал невыгодный ему контракт.

Есть и другие типы атак, основанные на «атаке дней рождения». Злоумышленники могут посылать на автоматическую систему управления (напр., спутник) случайное сообщение со случайной подписью. Рано или поздно, у одного из этих сообщений будет правильная подпись.

Враги не будут знать, что делает эта команда, но если их задачей было только навредить, то этого может быть достаточно.

Аналогичные Российские стандарты.

ГОСТ Р34.10-94 «Процедуры выработки и проверки электронной цифровой подписи на базе асимметричного криптографического алгоритма».

ГОСТ Р34.11-94 «Функция хэширования»

Эти стандарты объединены под общим заголовком «Информационная технология. Криптографическая защита информации».

13. Случайные числа. Генераторы случайных чисел.

Зачем вообще разговаривать о случайных числах при обсуждении криптографии? В каждом компиляторе есть датчик случайных чисел, но, к сожалению, большинство таких датчиков почти наверняка криптографически нестойки и, возможно, даже не очень случайны.

Почему? Такие датчики не случайны, т.к. в этом нет особой потребности. Большинству обычных приложений не требуется большого количества случайных чисел (симуляторы, компьютерные игры и т.п.). Но криптография особенно чувствительна к свойствам датчиков случайных чисел. Зачастую криптографическая стойкость алгоритма напрямую зависит от случайности датчика случайных чисел.

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

Лучшее, что может произвести компьютер – это псевдослучайная последовательность.

Что такое псевдослучайная последовательность? Нечто сложно определяемое (мы не будем давать точного определения). Период такой последовательности должен быть таким, чтобы конечная последовательность разумной длины не была периодической. Относительно короткие непериодические подпоследовательности должны быть как можно более неотличимы от случайных последовательностей, в частности, соответствовать различным критериям случайности. Итак,

Определение: Генератор последовательности псевдослучаен, если он выглядит случайным, т.е. проходит все статистические тесты случайности (начиная с χ^2 --критерия – см. Кнут. т.2, стр. 52 и далее).

Для криптографических приложений статистической случайности недостаточно, хотя это и необходимое свойство.

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

KSPRSG тоже подвержены криптоанализу – как любой шифр. алгоритм.

Наконец, наиболее сильное определение:

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

Вопрос существования случайных последовательностей – философский вопрос. Мы знаем, что на микроуровне случайность существует (квантовая механика), но сохранит ли эта случайность при переходе на макроуровень.

Дополнительное свойство случайной последовательности: случайная последовательность не может быть сжата. KSPRSG не может быть сжаты (на практике).

Генератор со своими определениями 1-3 достаточно хорош для одноразовых блокнотов, генерации ключей и любых других криптографических приложений.

Теперь рассмотрим эти категории генераторов подробнее.

1. Генераторы псевдослучайных последовательностей.

Линейный конгруэнтный датчик псевдослучайных чисел.

ЛКГ – это последовательности вида $X_n = (aX_{n-1} + b) \bmod m.$ X_0 -- начальное значение, a – множитель, b – приращение, m – модуль.

У такого генератора период ≤m.

Если a, b, m правильно выбраны, то генератор является генератором максимальной длины и имеет период m (напр., НОД(m, b) = 1).

В таблице 15.1 (взятой из [713]) приведен список хороших констант для ЛКГ. Они все дают ЛКГ максимальной длины и, что даже важнее, проходят Кнутовский спектральный тест случайности для размерностей 2, 3, 4, 5 и 6. [488]. Табл. 15.1 организована по максимальному результату ЛКГ, не переполняющему слово определенной длины.

Преимущества ЛКГ в том, что они быстрее и требуют мало операций для производства одного бита.

К сожалению, ЛКГ не может быть использованы для поточных шифров – они предсказуемы, впервые были взломаны Joan Boyar [701]. Она также взломала квадратичные генераторы:

$X_n = (aX_{n-1}^2 + bX_{n-1} + C) \bmod m$ и кубические генераторы

$X_n = (aX_{n-1}^3 + bX_{n-1}^2 + cX_{n-1} + d) \bmod n$.

Другие исследователи обобщили работу Bayar'a на случай общего полином. конгруэнтн. генератора. Stern & Boyar показали как взломать ЛКГ даже если известна не вся последовательность.

Тем не менее, ЛКГ полезны для обычных приложений (таких как симуляторы и компьютерные игры).

Комбинация ЛКГ.

Wishmann & Hill [877], а позже Pierre L'Ecuger [530] изучили комбинации ЛКГ. Результаты не являются более стойкими криптографически, но имеют большие периоды и лучше ведут себя на некоторых критериях случайности.

Приведем программу, реализующую такой генератор для 32-битного компьютера (стр. 319-350).

…..

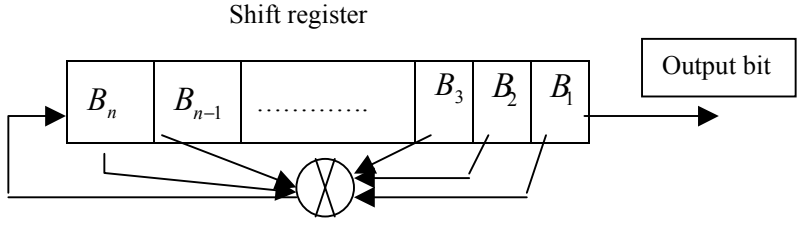
Генератор работает, если машина может представлять все целые от $-2^{31} + 85$ до $2^{31} + 85$. Переменные S1 и S2 глобальны, в них содержится текущие значения генератора. Перед первым вызовом они должны быть инициализированы.

Начальные значения S1∈ [1, 2147483562]; начальное значение S2∈ [1, 2147483398].

Период генератора ~ 10^{18}.

Линейные регистры с обратной связью (Linear Feedback Shift Registers).

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



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

Теоретически, n-битный LFSR может сгенерировать псевдослучайную последовательность длиной $2^n - 1$ бит перед зацикливанием [385]. Для этого регистр сдвига должен побывать во всех $2^n - 1$ внутренних состояниях (кстати, количество состояний именно $2^n - 1$, а не 2^n, т.к. регистр сдвига, состоящий из нулей, вызовет бесконечную последовательность нулей, что не особо удобно).

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

Пример: LFSR длиной 4 бита, tapped at 1st & 4th bits…

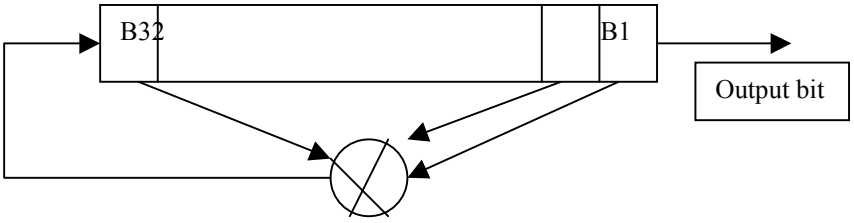
Теорема: Для того, чтобы LFSR был LFSR максимальной длины, необходимо и достаточно, чтобы полином, образованный из элементов tap sequence плюс единица был примитивным полиномом по модулю 2. (на самом деле, примитивный полином – это генератор в данном поле)

Определение: Примитивный полином степени n – это неприводимый полином, который делит $x^{2^{n-1}} + 1, \text{но не делит } x^d + 1 \text{для любого } d : (2^{n-1} \vdots d)$.

Определение: Полином неприводим, если он не может быть выражен как произведение двух других полиномов (кроме 1 и самого себя, конечно же).

В табл. 15.2 приведен список некоторых примитивных полиномов по модулю 2 [870]. Например: запись (32, 7, 5, 3, 2, 1, 0) означает, что следующий полином примитивный. Тогда: $x^{32} + x^7 + x^5 + x^3 + x^2 + x + 1$.

Этот полином легко превратить в LFSR максимальной длины (последнее число всегда 0). Числа, кроме последнего куля, обозначают tap sequence. В этом примере, числа означают, что взяв 32-битный регистр сдвига и генерируя новый бит путем XOR'а 32-го, 7-го, 5-го, 3-го, 2-го, 1-го бит, мы получим LFSR максимальной длины (с $2^{32} - 1$ состояниями):



Приведем программу на языке C для этого LFSR:

```
Int LFSR() {
      Static unsigned long ShiftRegister=1; //Anything but 0
ShiftRegister = (((( ShiftRegister>>31)
                  ^( ShiftRegister>>6)
                  ^( ShiftRegister>>4)
                  ^( ShiftRegister>>2)
                  ^( ShiftRegister>>1)
                  ^( ShiftRegister)
                  &0x00000001)
                  <<31
                  | ShiftRegister>>1);
      return ShiftRegister & 0x00000001;
}
```

Код несколько усложняется, когда регистр сдвига длиннее, чем размер слова, но незначительно.

В табл. 15.2 приведено так много полиномов, т.к. LFSR очень часто используется для потоковой криптографии и хочется, чтобы люди использовали различные полиномы.

Кстати, любая запись в таблице представляет собой 2 разных примитивных полинома, т.к. если p(x) – примитивный, то $x^n p(1/x)$ -- примитивный.

Например, если (a, b, 0) – примитивный, то (a, a-b,0) – примитивный. Если (a, b, c, d, 0) примитивный, то и (a, a-d, a-c, a-b,0) – примитивный и т.д.

Примитивные трехчлены особенно удобны, т.к. только 2 бита регистра сдвига должны быть отXORены. Но при этом они и более уязвимы к атакам.

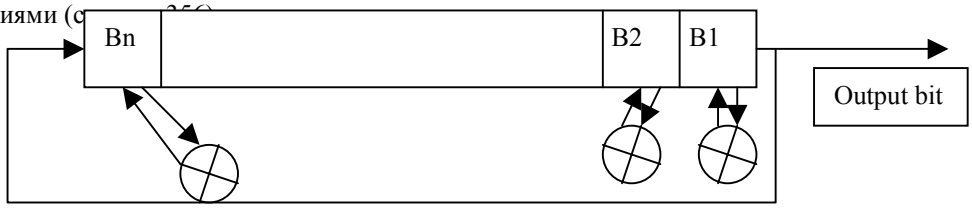
LFSR – неплохие генераторы случайных чисел, но имеют неприятные свойства. Последоват. биты линейны, что делает их бесполезными для шифрации. Для LSFR длины n внутреннее

состояние можно узнать по n выходным битам генератора. Даже если схема обратной связи неизвестна, то достаточно 2n выходных бита, чтобы определить ее.

Большие случайные числа, сгенерированы из последовательных битов LFSR, сильно коррелированны и иногда даже не совсем случайны. Тем не менее, LFSR достаточно часто используется как базовые алгоритмы шифрации.

Модифицир. LFSR.

Можно переписать программу, реализующую LFSR для того, чтобы не возиться с битовыми операциями (с



14. Криптографически стойкие датчики случайных чисел. Методы получения настоящих случайных чисел.

2. Криптографически стойкие ДСЧ.

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

Известно 4 подхода к конструированию CSPRSG:

1) системно-теоретический подход;
2) сложностно-теоретический подход;
3) информационно-теоретический подход;
4) рандомизированный подход.

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

2.1. Системно-теоретический подход.

В этом подходе, криптограф создает генератор ключевого потока, у которого есть проверяемые свойства – период, распределение битов, линейная сложность и т.д. криптограф изучает также различные методы криптоанализа и оптимизирует ДСЧ против этих атак.

Этот подход выработал набор критериев для потоковых шифров. Они были сформулированы Рюппелем [761]:

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

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

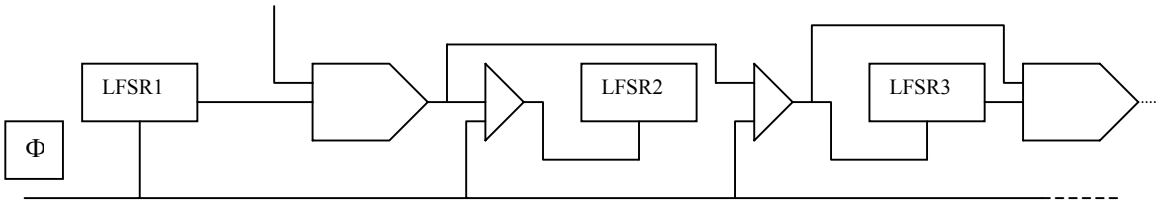
Основная проблема подобных криптосистем в том, что для них трудно доказать какие-либо факты об их криптостойкости. Дело в том, что для всех этих критериев не была доказана их

необходимость или достаточность. Потоковый шифр может удовлетворять всем этим принципам и все-таки оказаться нестойким.

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

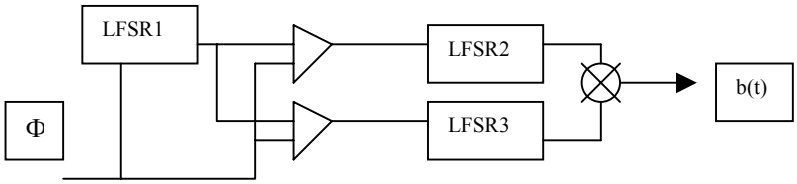
Примеры:

2.1.1. Каскад Голлманна.



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

2.1.2. Alternating Stop-and-Go Generator/



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

Сложно-теоретический подход.

В этом подходе, криптограф пытается использовать теорию сложности для доказательства стойкости генератора => генераторы более сложные; основаны на тех же проблемах, что мы видели в криптографии с открытым ключом (=> медленные).

Примеры:

2.2.1 RSA.

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

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

Результат генератора – наименьший значимый бит x_i. Стойкость этого генератора ~ стойкости RSA. Если N достаточно большое, то генератор надежен.

2.2.2 Blum Blum Shub

На данный момент – самый простой и эффективный. Назван по фамилиям изобретателей. Мы милосердно назовем его BBS. Теория этого генератора – квадратичные вычеты по модулю n.

Найдем два больших простых числа p, q, дающие при делении на 4 остаток 3. Произведение $n = p \times q$ назовем числом Блюма. Выберем x: НОД(n, x) = 1. Посчитаем $x_0 = x^2 \bmod n$ -- это начальное значение генератора.

Теперь i-ым псевдослучайным числом является наименьший значимый бит x_i, где $x_i = x_{i-2}^2 \bmod n$.

Интересно, что нам не надо проходить через (i-1) состояние для того, чтобы получить i-й бит. Если мы знаем p, q, то мы можем сосчитать его сразу:

$b_i \; естьнаименьшее значение бит \; x_i = x_0^{(r^i) \bmod ((p-1)(q-1))} \bmod n$.

Значит, можно использовать этот CSPRG для random-access файла.

Число n можно распространять свободно => каждый сможет генерить биты, используя этот генератор. Но если криптоаналитик не взломает n, они не смогут предсказать следующий бит – даже утв. типа «с вероятностью 51% следующий бит = 1».

Более того, BBS генератор непредсказуем налево и направо (т.е. имея часть последовательности нельзя предсказать предыдущий или последующий бит).

Кроме того, можно использовать не только наименьший значащий бит, но и l битов x_i (где l – длина x_i).

Информационно-теоретический подход.

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

2.4. Рандомизированный подход.

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

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

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

3. Генераторы настоящих случайных чисел.

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

Итак, наша цель – произвести числа, которые невозможно воспроизвести.

3.1. Таблица RAND.

В 1955 Rand Corporation опубликовала книжку с миллионом случайных цифр.

3.2. Использование таймера компьютера.

Если необходим один случайный бит (или даже несколько), то можно взять наименьший значимый бит таймера. Это может плохо работать под Unix'ом из-за различий синхронизацией, но на PC будет нормально.

Нельзя получать таким образом много битов. Например, если каждый вызов процедуры генерации бита занимает четное число тиков таймера, то мы будем получать одни и те же значения, если нечетное – то последовательность 101010… Даже если зависимость не настолько очевидна, результирующая строка будет далека от случайности.

3.3. Использование случайного шума.

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

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

Проблема не в том, чтобы найти источники случайности, но в том, чтобы сохранить случайность при измерениях.

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

Отклонения и корреляции.

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

1) XOR'еньем: если случайный бит смещен к 0 на величину e, то вероятность появления 0 может быть записана как P(0) = 0.5+e. XOR'енье двух битов даст: $P(0) = (0.5+e)^2 + (0.5-e)^2 = 0.5 + 2 \cdot e^2$. XOR'енье четырех битов: $P(0) = 0.5 + 8 \cdot e^4$ и т.д. Процесс сходится к равновероятным 0 и 1.
2) Данный метод уничтожает все смещ. со случ. в остальном источника. Рассмотрим пару битов. Если это одинаковые биты, то отбросим их и рассмотрим следующую пару. Если биты различны, то берем первый бит.

Потенциальная проблема обоих методов в том, что при наличии корреляции между соседними битами данные методы увеличивают смещен. Один из способов избежать этого – использовать различные источники случайных чисел (XOR'ить биты 4-х различных источников или смотреть на пары битов из разных источников).

Сам по себе факт наличия смещения у генератора случайных чисел не означает его непригодность. Например, рассмотрим проблему генерации 112-битного ключа для троированного DES'а. Пусть у Алисы есть только генератор со смещ. к нулю: P(0) = 0.55; P(1)=0.45 (=> всего 0ю99277 битов энтропии на бит ключа по сравнению с 1 для идеального генератора). Тогда Маллет при попытке взлома ключа может оптимизировать процедуру поиска ключа методом грубой силы, пытаясь начать с наиболее вероятного ключа (00…0) и заканчивая наименее вероятным (11…1). Из-за смещения, Маллет может ожидать найти ключ в среднем за 2^{109} попыток. Если бы смещения не было, то потребовалось бы 2^{111} попыток. Выигрыш есть, но несущественный.

Распределение случайности с помощью односторонней хэш-функции.

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

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

Как сгенерировать случайные числа более сложного вида (распределения)?

Смотри вкладку (или стр. 372-375).

15. Простейшие криптографические протоколы.

Криптографический протокол – это последовательность шагов, выполняемая двумя или более сторонами, спроектированная для выполнения некоторых криптографических задач.

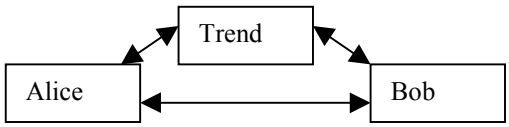
Свойства криптографических протоколов:

1. Каждый участник должен знать протокол и все необходимые шаги заранее.
2. Каждый участник протокола должен быть согласен ему следовать.
3. Протокол должен быть четко и однозначно определен.
4. Протокол должен быть полным (должны быть определены действия для любых ситуаций)

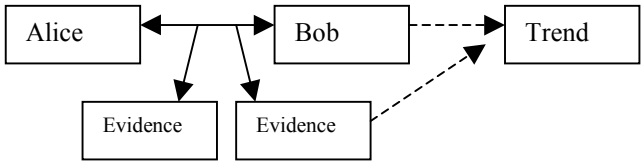
Протоколы бывают:

- арбитрируемые (arbitrated);
- со свидетельствами (adjudicated);
- самодостаточные (self-enforcing).

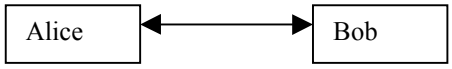
Arbitrated



Adjudicated



Self-enforcing



Атаки против протоколов:

- пассивные (подслушивание протокола в целях получения информации); соотв. Ciphertext-only attack ~ Eve
- активные (попытки изменить протокол в свою пользу – вставить сообщение, изменять сообщения и т.п.) = Mallet
- пассивные обманщики (cheaters) – одна из сторон, пытающаяся получить больше информации, но следуя протоколу;
- активные обманщики – могут менять протокол.

Обмен ключами и посылка сообщений.

Протокол, использующий симм. криптографию.

1) A и B договариваются о криптосистеме;
2) A и B договариваются о ключе;
3) A шифрует сообщение;
4) A посылает шифр. сообщение B;
5) B расшифровывает и читает.

Проблемы:

- возможность компрометирования ключа;
- необходимость секретного распространения ключа;

- быстрый рост необходимого числа ключей.

Протокол, использующий криптографию с открытым ключом.

1) A и B договариваются о криптосистеме;
2) B посылает A свой открытый ключ;
3) A шифрует сообщение, используя открытый ключ B и посылает его;
4) B расшифровывает сообщение A, используя секретный ключ.

Man-in-the-Middle Attack.

1) A посылает B свой открытый ключ. M перехватывает ключ и посылает B собственный открытый ключ.
2) B посылает A свой открытый ключ. M … посылает A собственный ….
3) A посылает сообщение B; т.к. это открытый ключ M, то он расшифровывает его, читает, зашифровывает с помощью открытого ключа B и посылает B
4) Аналог. с B.

Man-in-the-Middle работает, если у A и B нет других способов проверить, что они говорят друг с другом. Простейший способ борьбы – зачитать по телефону одност. х.-ф. своего ключа. Но что делать, если это невозможно?

Interlock protocol (блокировка).

Изобретен Ривестом и Шамиром [748] для предотвращения описанной выше атаки (см. следующую страницу):

1) A посылает B свой открытый ключ
2) B посылает A свой открытый ключ
3) A шифрует свое сообщение для B и посылает половину шифровки.
4) B шифрует сообщение для A и тоже посылает половину шифровки.
5) A посылает вторую половину сообщения.
6) B складывает половинки и расшифровывает их своим секретным ключом.

Важно, чтобы половинки не читались друг без друга (брать каждый второй бит). При этой схеме Маллету придется изобретать совершенно другие сообщения между A и B. В любом случае, это сложнее, чем просто перехват и перешифровывание сообщений.

Аутентификация.

Как определить подлинность человека, садящегося за компьютер? (или ATM, или телефонную банковскую систему и т.д.)

Традиционно это делается с помощью паролей – человек и компьютер знают пароль и компьютер запрашивает его при каждом login'е.

На самом деле, компьютеру не обязательно знать пароль. Компьютеру необходимо различать правильные пароли от неправильных. Это легко реализуется с помощью односторон. функций. Теперь вместо паролей компьютер будет хранить односторон. ф-ции от пароля.

1) A посылает компьютеру свой пароль
2) Компьютер берет односторон. ф-цию от пароля
3) Компьютер сравнивает результат со значением, хранящемся в БД.

Т.к. компьютер теперь не хранит пароли, угроза взлома и кражи листа паролей значительно ослабляется. Даже если список украдут, по нему еще нужно восстановить пароли, что является криптографич. сложной задачей.

Атака со словарем и соль.

Даже файл с односторон. ф-циями паролей уязвим. Если Маллет раздобудет его, а затем составит список 100,000 наиболее распространенных паролей, затем применит к этому списку односторон. ф-цию и сравнит результат с нашим файлом паролей, то у него есть шанс отыскать совпадения (dictionary attack).

Для того, чтобы бороться с такой атакой, используется соль – случайная строка, которая приписывается к паролю, а затем уже применяется односторон. ф-ция. В БД попадает и соль, и

значение ф-ции. Если количество различных значений соли достаточно велико, то это практически сводит на нет возможные атаки со словарем на наиболее распространенные пароли.

В большинстве Unix-системах используются 12 битов соли.

Соль – не панацея и увеличение числа битов не решит всех проблем. Соль защищает от атак со словарем на файл паролей, но не от атаки на один отдельно взятый пароль. Это запутывает результат у тех, кто использует одни и те же пароли на разных машинах, но не делает плохие пароли лучше.

Упомянем также идентификацию пользователя с помощью криптографии с открытым ключом (стр. 48-49).

Разделение секретов (secret splitting)

Пусть в нашей компании имеется некоторый секрет. Мы боимся перебежчиков и потому никому не разглашаем секрет целиком. Более того, мы хотим, чтобы для конструирования секрета требовались 4 наших сотрудника.

1) Трент генерирует 3 случайные строки S_1, S_2, S_3 той же длины, что и сам секрет M
2) $P = M \; xor \; S_1 \; xor \; S_2 \; xor \; S_3$
3) Трент отдает P первому сотруднику, S_1 -- второму сотруднику и т.д.
4) При необходимости реконструирования секрета: $P \; xor \; S_1 \; xor \; S_2 \; xor \; S_3 = M$.

Это протокол, в котором Трент имеет абсолютную власть – он может создать бессмысленные сообщения, и никто этого не заметит до попытки реконструирования секрета. В конце концов, это его секрет.

Разделение секретов (secret sharing).

Простейший случай – некоторое сообщение делится на n частей, называемых тенями (shadows) так, что любых m этих сообщений достаточно для реконструирования сообщения (это так называемая (m, n)-threshold scheme).

С помощью таких схем можно строить сколь угодно сложные условия.

Идея была изобретена независимо Шамиром [788] и G.R. Bladey [117]. Существует несколько разных алгоритмов (см. секцию 16.5).

Криптографическая защита баз данных.

У нас есть БД адресов организации. С одной стороны, мы хотим раздать эту базу данных всем членам организации, чтобы они могли обмениваться письмами и т.д. С другой стороны, если БД попадет в руки рассыльщиков спама, то организация потонет в мусорных письмах.

Оказывается, мы можем зашифровать БД таким образом, чтобы было легко извлечь адрес отдельного человека, но тяжело извлечь список рассылки всех сотрудников (схема из [337, 336]):

Каждая запись в БД состоит из двух полей. Ключевое поле – фамилия сотрудника, от которой берется одност. хэш-функция. Второе поле – полное имя сотрудника и его адрес, зашифрованный с ключом = фамилия.

Т.о. искать по конкретной фамилии легко, но воспроизвести всю БД сложно.

Timestamping services.

Linking Protocol (Distributed Protocol) – идея в том, чтобы связать данный документ с предыдущими и последующими сообщениями (=> данное возникло не раньше предыдущего и не позже последующего).

Алгоритм рукопожатия (по Диффи-Хеллману).

Задача: сформировать общий сеансовый ключ.

Выберем большое простое число p; a: a<p. Пусть также p: разложение числа p-1 на множители должно содержать, по крайней мере, один большой простой множитель, а размер p ≥ 512 бит.

1. A и B договариваются о значениях a, p (несекретно, передается в открытом виде).
2. A генерирует большое случайное целое x_A (закрытый ключ) и вычисляет свой открытый ключ $y_A = a^{x_A} \bmod p$. Аналогично, B генерирует x_B и вычисляет $y_B = a^{x_B} \bmod p$.
3. A и B обмениваются значениями y_A, y_B.
4. A вычисляет $k_A = (y_B)^{x_A} \bmod p; B\,вычисляет\, k_B = (y_A)^{x_B} \bmod p.$ Отметим, что $k_A = (y_B)^{x_A} \bmod p = (a^{x_B})^{x_A} \bmod p = (a^{x_B x_A}) \bmod p = a^{x_A x_B} \bmod p = (y_A)^{x_B} \bmod p = k_B \Rightarrow k_A = k_B = k_{AB}$

   это число и является общим сеансовым ключом, сформированным абонентами A, B.

Злоумышленник может перехватить $y_A, y_B, a, p.$ Ему нужно найти такие $x_A^{'}, x_B^{'} : y_A = a^{x_A^{'}} \bmod p, y_B = a^{x_B^{'}} \bmod p$. Это задача дискретного логарифмирования. (взято из книги В. М. Зима, А. А. Молдовян, Н. А. Молдовян «Компьютерные сети и защита предаваемой информации», СПб, 1998).

16. Проблемы многократного применения блочных шифров.

Многократная шифрация возникает в тех случаях, когда один и тот же блок текста шифруется несколько раз.

Двойная шифрация.

Наивный подход к задаче увеличения надежности блочного шифра – это зашифровать блок дважды с двумя разными ключами.

$$c = E_{K_2}(E_{K_1} P);$$

$$P = D_{K_1}(D_{K_2} c).$$

По идее, если блочный алгоритм не является группой, то необязательно существует $K_3 : c = E_{K_2}(E_{K_1} P) = E_{K_3} P.$ Следовательно, полученный блок должен быть значительно сложнее взломать перебором. Вместо 2^n попыток (где n – число битов в ключе) потребуется 2^{2n} попыток.

Однако, это не так. Merkle & Hellman [594] изобрели схему, в которой при known-plaintext attack удается взломать двойную шифрацию не за 2^{2n} попыток, а за 2^{n+1}. Такая атака называется встреча посередине (meet-in-the-middle) и работает по следующей схеме:

Криптоаналитик знает c_1, P_1, c_2, P_2.

$$c_1 = E_{K_2}(E_{K_1}(P_1)),$$

$$c_2 = E_{K_2}(E_{K_1}(P_1))$$

Для каждого возможного K сосчитаем $E_K(P_1)$ и сохраним результаты в памяти. После этого, сосчитаем $D_K(c_1) для \forall K$ и будем искать одинаковые результаты. Если нашлись, то, возможно, что текущий ключ = K_2, а ключ в памяти = K_1. Попробуем зашифровать P_2 с помощью K_1 и K_2, если да, то с вероятностью 2^{2n-2b} мы имеем K (b – размер блока). Если нет, то продолжим поиск. Среднее число фальшивых совпадений ≈ $2^{2n} - 2^b$. Максимальное число шифраций в таком случае = $2 \times 2^n = 2^{n+1}$. Если ошибка слишком вероятна, то можно взять третий текст (=> вероятность ошибки = 2^{2n-3b}). Есть и другие оптимизации.

Эта атака требует огромного количества памяти, 2^n блоков. Для 56-битного алгоритма это значит 2^{56} 64-битных блоков или 2^{59} байт. Это, конечно же, много больше, чем можно позволить, но достаточно, чтобы убедить криптографов в том, что двойная шифрация ничего не стоит.

Тройная шифрация.

Более удачная идея предложена Tuchman [857]: шифровать блок 3 раза двумя ключами, причем отправитель вначале шифрует первым ключом, затем расшифровывает вторым ключом, затем снова шифрует первым ключом – т.н. encrypt-decrypt-encrypt (EDE) mode.

$$c = E_{K_1}(D_{K_2}(E_{K_1}P)),$$

$$P = D_{K_1}(E_{K_1}(D_{K_1}P))$$

Если у блочного алгоритма n-битный ключ, то такая схема дает 2n-битный ключ. EDE mode обеспечивает совместимость с обычной реализацией (достаточно подставить одинаковый ключ и мы имеем обычную шифрацию).

Такая техника не подвержена атаке meet-in-the-middle, но Merkle & Hellman придумали другой вариант, который использует 2^n блоков памяти и работает за 2^{n-1} шагов [594]. Это chosen-plaintext attack, требующая огромного количества текста. Paul van Dorschot & Michael Wiener сделали из нее known-plaintext attack с 2^{2n-14} шагами.

Для абсолютной безопасности и используется 3 разных ключа. Лучший метод взлома работает за 2^{2n} шагов и требует 2^n блоков памяти.

Т.о. тройная шифрация с 3 разными ключами дает ровно такую надежность, какую можно наивно ожидать от двойной шифрации.

Тройная шифрация со сдвигом.

Если сообщение достаточно длинное (> нескольких блоков), то можно улучшить тройную шифрацию padding'ом. Вставим случайную строку длиной в полблока между первой и второй и между второй и третьей шифрациями.

Такой сдвиг не только уничтожает повторения, но и разбивает блоки.

Тройная шифрация с разными алгоритмами.

Может уменьшать надежность алгоритма!?

17. Теория информации и ее использование в криптографии.

Современная теория информации впервые появилась в работах Клода Шэннона в 1948-49 гг. [803, 804]. Подробный трактат по этой теме – [363].

Энтропия и неопределенность.

Определение: Количество информации сообщения – это минимальное количество битов, требуемое для записи всех возможных значений (смыслов) сообщения.

Например: поле, содержащее день недели, содержит не более чем 3 бита информации (т.к. данная информация может быть упакована в 3 бита, причем одно значение будет не использоваться). Если записывать эту информацию строкой, то потребуется больше памяти, но информации не прибавится. Точно так же поле «пол» БД содержит только 1 бит информации.

Формально, количество информации сообщения M измеряется энтропией сообщения, обозначаемой H(M). В целом, энтропия сообщения = $\log_2 n$, где n – количество возможных значений.

Энтропия сообщения также измеряет его неопределенность, т.е. количество битов текста, необходимое для восстановления зашифрованного текста. Например, если блок шифра = «Q*56HM», а значение может быть «MALE» или «FEMALE», то неопределенность сообщения = 1 (криптоаналитику достаточно выяснить один нужный бит для восст. сообщения).

Если неопределенность исходного текста слишком мала, то криптосистема с открытым ключом уязвима к ciphertext-only attack. Например, если c = E(P), где P – один из n различных возможных текстов, то криптоаналитику требуется только зашифровать все n возможных текстов и сравнить результаты с c (т.к. ключ зашифровки -- открытый). Секретный ключ так не найти, но P можно найти. Симметричные криптосистемы не подвержены такой опасности (нельзя производить пробные шифрации без ключа).

Вероятностная шифрация помогает решить эту проблему.

Rate of a Language.

Определение: Для данного языка, интенсивность языка (rate of a language) есть величина r = H(M)/N, где N – длина сообщения.

Интенсивность нормального английского принимает значения от 1.0 до 1.5 бит/букву для больших значений N (мы будем использовать Шэнноновскую оценку 1.2 [806])

Определение: Абсолютная интенсивность языка есть максимальное количество бит, которое может быть закодировано каждой буквой (в предположении, что каждая последовательность букв одинаково вероятна).

Если в алфивите L символов, то абсолютная интенсивность $R = \log_2 L$, это максимальная энтропия отдельного символа.

В английском языке, с 26 буквами, абсолютная интенсивность = $\log_2 26 = 4.7$ битов/символ. Конечно же, реальная интенсивность английского языка много меньше абсолютной интенсивности – английский чрезвычайно избыточен.

Определение: Избыточность языка D = R – r.

Учитывая интенсивность английского = 1.2, избыточность = 3.5 битов/символ. Т.о. в каждой английской букве всего 1.2 бита информации – все остальное избыточно. В ASCII тот же английский все равно имеет 1.2 бита информации на 8 бит символа => 6.8 бит избыточны => общая избыточность = 0.15 бит информации/бит ASCII-текста.

Надежность криптосистем.

Шеннон определил точную математическую модель надежности криптосистемы. Цель криптоанализа – определить ключ K, текст P или оба. Тем не менее, интересна и некоторая вероятностная информация о P – текст ли это или аудио и т.п.

В обычном мире, криптоаналитик имеет некоторую вероятностную информацию о P до начала (язык => избыточность). Цель криптоанализа – путем анализа изменять вероятности, связанные в каждым возможным текстом. Т.о. рано или поздно один из текстов станет заведомо верным (или достаточно вероятным).

Существует криптосистема с совершенной секретностью (perfect secret), т.е. такая криптосистема, в которой шифртекст не дает никакой информации об исходном тексте (кроме, может быть, длины) – это одноразовый блокнот.

Если же секретность не совершенна, то шифртекст дает некоторую информацию о соотв. тексте и это неизбежно. Криптоаналитики используют естественную избыточность языка для того, чтобы уменьшить количество исходных текстов. Чем больше избыточен язык, тем его легче анализировать. Вот почему в реальной жизни зачастую используются архиваторы – компрессия снижает избыточность сообщения.

Энтропия криптосистемы – это мера пространства ключей: $H(K) \approx \log_2(\textit{число ключей})$. Чем больше энтропия, тем надежнее криптосистема.

Расстояние единственности (unicity distance).

Для сообщения длины n, число различных ключей, которыми расшифруют в какой-либо разумный текст на том же языке, дается следующей формулой:

$U = 2^{H(K)-nD} - 1$ [419, 50]

Шеннон определил расстояние единственности как приблизительное количество шифртекстов, такое, что сумма реальной информации (энтропии) в соответствующем исходном тексте плюс энтропия ключа шифрации равняется количеству битов шифртекста. Он показал далее, что шифртексты длиннее, чем это расстояние с достаточной степенью вероятности имеют только одно осмысленное значение. Шифртексты значительно короче вполне могут иметь

различное количество одинаково правдоподобных расшифровок и, следовательно, могут иметь повышенную секретность за счет трудности выбора правильного.

Для большинства симм. криптосистем U = H(K)/D.

Расстояние единственности, как все статистические или информационно-теоретические величины, не дает детерминистических предсказаний, но дает вероятностные результаты.

Расстояние единственности указывает на минимальную длину шифртекста, для которого, скорее всего, существует только одна правильная расшифровка.

Чем больше расстояние единственности, тем лучше криптосистема. Для DES с 56-битным ключом и английского языка U = 56/68 = 8.2, т.е. 8.2 ASCII-символа или 66 бит (расстояние единственности для классических криптосистем – [257]).

Шеннон определил криптосистему с расстоянием единственности равным бесконечности как идеальную секретную (совершенно секретная система обязательно идеально секретна, но не наоборот). Если криптосистема идеально секретна, то даже успешный криптоанализ оставит некоторую неопределенность, является ли полученный текст реальным исходным текстом.

Информационная теория на практике.

Все сказанное имеет великую теоретическую ценность, но на практике используется редко. Расстояние единственности гарантирует ненадежность, если оно слишком мало, но не гарантирует надежности, если оно велико. Очень мало криптоалгоритмов действительно не поддаются анализу.

Тем не менее, теор. информации используется для определения рекомендованного интервала смены ключей для определения алгоритма. Кроме того, для криптоанализа используется набор статистических и теоретически-информационных тестов (для определения направления движения).

К сожалению, множество работ остается засекреченными, включая работы Тьюринга 1940 года.

18. Квантовая криптография.

Квантовая криптография использует естественную неопределенность квантового мира. С ее помощью можно создать канал связи, в котором нельзя подслушать, не нарушив передачи. Charles Bennett и Gilles Brassard упорно разрабатывают эту тему. Лучшее описание квантовой криптографии – [76]; другое – [907].

Согласно квантовой механике, частицы не существуют в каком-либо определенном месте. Они существуют в нескольких местах сразу с вероятностью быть в различных местах, если кто-то станет измерять. Пока не придет ученый с измерялкой, частица не «свернется» до одного положения. Но нельзя мерить каждый аспект частицы одновременно, например координаты и скорость.

Эта неопределенность может быть использована для генерации секретных ключей. Фотоны вибрируют во время своих передвижений. Когда большая группа протонов вибрируют в одном и том же направлении, то они поляризованы. Поляризационные фильтры позволяют пройти только специально поляризованным фотонам и блокируют все остальное.

Пусть у нас есть поток горизонтально поляризованных фотонов. Если они попытаются пройти сквозь горизонтально-поляризованный фильтр, то все пройдут. Медленно поворачивая фильтр до 90 градусов, мы заметим постепенное уменьшение числа проходящих фотонов, пока не перестанут проходить все.

Это противоречит интуиции – можно подумать, что малейший поворот фильтра заблокирует все фотоны, т.к. они горизонтально поляризованы. Так было бы в макромире, но не в квантовой механике. Каждая частица имеет вероятность внезапно поменять поляризацию и совпасть с фильтром. Если угол изменился мало, то вероятность высока; если угол = 90 градусам, то вероятность равно нулю; если угол = 45 градусов, то вероятность = 50%.

Вот это свойство мы и будем использовать для генерации секретного ключа:

(1) A посылает B последовательность фотонных потоков (импульсов). Каждый импульс случайно поляризован в одном из 4 направлений: горизонтальном, вертикальном, левом диагональном или правом диагональном, например: | | / -- -- \ -- | -- /

(2) У B есть поляризационный детектор. Он может установить детектор на измерение горизонт/вертикальной поляризации или диагональной поляризации, но не может сделать и то, и другое, одновременно. Поэтому он устанавливает детектор случайным образом, например: X + + X X X + X + +. Если B установил детектор правильно, то он

узнает правильный результат. Если же нет, то он получит случайный результат, причем не будет знать разницу между ними. Пусть в нашем примере он получит результат | | / -- -- \ -- | -- /

(3) B говорит A по несекретному каналу, какие установки он использовал.
(4) A говорит B какие установки были верными. В нашем примере, это были 2, 6, 7 и 9 импульсы.
(5) A и B оставляют только те результаты, в которых установки совпадали. В нашем примере это * | * * * \ -- * -- *. По заранее достигнутым договоренностям, A и B переводят это в 0 и 1, например, здесь 0011. Так A и B сгенерили 4 бита. Они могут нагенерить сколько хотят. В среднем, Боб будет угадывать в 50% случаев => Алисе придется послать ~2n битов для генерации n битов. Эти биты могут быть использованы как секретный ключ или одноразовый блокнот.

При использовании данной системы Ева не может подслушать. Если она попытается определить фотоны по мере их прохождения, то Боб заметит. Ее единственная надежда – поймать и измерить фотоны, а затем послать Бобу такие же. Но как и Боб, ей придется угадывать поляризацию, и половина ее попыток будет неудачной => появятся ошибки в передаче. А если так, то у A и B получатся разные битовые строки. Поэтому протокол оканчивается так:

(6) A и B сравнивают несколько битов в строчках. Если есть несоответствие, то они знают, что их подслушивают. Если нет, то сравненные биты выкидываются и используются оставшиеся.

Есть и улучшения этого протокола, позволяющие A и B использовать биты, даже если Ева подслушивает их или даже только пытается поймать несколько бит из тысяч.

Bennett & Brassard построили работающую модель. По последним данным, в British Telecom посылали биты по 10-километровому оптоволоконному кабелю [177].

19. Теория сложности и ее использование в криптографии.

Теория сложности может быть использована для анализа вычислительной сложности различных криптографических алгоритмов. В некотором роде, теор. информации говорит нам, что все криптографические алгоритмы (кроме одноразового блокнота) могут быть взломаны. Теория сложности говорит, могут ли они быть взломаны до полного охлаждения Вселенной.

Сложность алгоритма.

Мы будем использовать O-нотацию. Эта нотация позволяет определить каким образом время (T) и память (S) зависит от входных данных. Например, если T = O(n), то удвоение входа удвоит время работы алгоритма; если же $T = O(2^n)$, то добавление одного бита удвоит время работы.

Предположим, что наш компьютер может выполнить конст. алгоритм за микросекунду => линейный – за секунду, квадратичный – за 11.6 дней. Тогда кубический алгоритм потребует 32,000 лет для завершения, а про экспоненциальный не стоит даже говорить:

Class	Complexity	#of operations for n = 10^6	Time at 10^6 o/s
Constant	O(1)	1	1 мкс
Linear	O(n)	10^6	1 сек
Quadratic	O(n^2)	10^{12}	11.6 дней
Cubic	O(n^3)	10^{18}	32,000 лет
Exponential	O(2^n)	$10^{301,000}$	$10^{301,006}$ · время жизни Вселенной

Понятно, что в идеале криптограф хотел бы сказать, что все алгоритмы взлома носят экспоненциальный характер. На практике же все сводится к утверждениям типа «все известные алгоритмы взлома -- суперполиномиальны».

Классы сложностей.

Теория сложности также классифицирует проблемы согласно алгоритмам, необход. для их решения. Для этого исслед. миним. время и память, требуемые для решения самой сложной задачи класса на так называемой машине Тьюринга.

$$P \le NP(NP-complete) \le PSPACE(PSPACE-complete) < EXPTIME.$$

Единственный доказанный факт – это то, что EXPTIME≠P. Кроме этого, существуют проблемы из класса NP-complete; если их можно решить за минимальное время, то P = NP. Например, задача о коммивояжере (travelling salesman).

С другой стороны, как подметил Arnold Reinhold, даже если алгоритм является полиномиальным, то он не обязательно практичен. Полиномиальный алгоритм с оценкой O(n^{20}) на самом деле не вычислим даже для маленьких значений n. При этом алгоритм, работающий за O($e^{0.1n}$) позволит легко решать задачи вплоть до миллиардов бит.

20. Вероятностная шифрация

1) Гомофонические шифры.

Основная идея – в выравнивании частности знаков криптограммы, т.е. использование такого механизма криптографич. преобразований, который вырабатывал бы криптограммы с одинаковой частотой появления каждого знака шифртекста.

Пусть дан источник сообщений с известными статистическими свойствами. Выразим частоты появления каждой буквы исходного алфавита целым числом f_i, где i – номер буквы в алфавите: $f_1, f_2, ..., f_L$, где L – число букв в исходном алфавите. Каждой букве T_i, где i = 1, 2, …, L исходного алфавита сопоставим подмножество ψ_i знаков выходного алфавита, причем потребуем выполнения следующих условий:

A) никакая пара подмножеств Ψ_i, Ψ_j не содержит одинаковых элементов;

B) количество различных знаков в ψ_i равно f_i.

Шифрование будем осуществлять путем замены каждой буквы исходного текста T_i на случайно выбираемый знак из подмножества ψ_i => при многократной замене заданной буквы каждый знак выходного алгоритма будет использоваться в среднем одинаковое число раз, а именно ~ 1/ f_i => в среднем частоты появления в криптограмме всех знаков вых. алфавита равны.

Дешифр. очевидно: по знаку криптограммы определяем подмножество, а по подмножеству – букву исходного алфавита.

Описанный способ требует использования $f_1 + f_2 + ... + f_L$ знаков в вых. алфавите. Наиболее существенный момент шифрования – ввод вероятностного процесса.

2) Шифры с простым вероятностным механизмом.

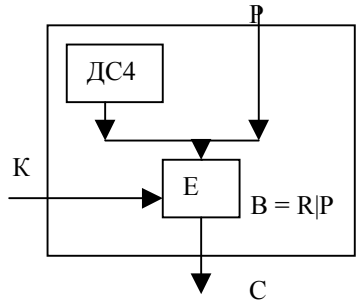
«Подмешивание» случайных данных к шифруемому сообщению позволяет задать вероятностный характер операций преобразования и тем самым повысить выч. стойкость криптосистемы.

Пусть E есть b-битовая функция шифрования, P есть p-битовый блок исходного текста, R-r-битовый случайный блок, где b = r + p. Подадим на вход шифрующей ф-ции блок B = R | P, где | обозначает конкатенацию двоичных векторов R и P:

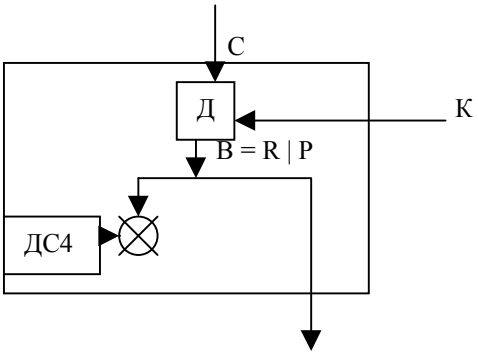
$$P \to B = R \mid P \to C = E(B, K),$$

где K – ключ шифрования. При шифровании размер входного блока увеличивается. Такое шифрование отобр. данный блок открытого текста P на большое множество блоков шифртекста $\{C_1, C_2, ..., C_n\}, где\, n = 2^r$.

(а) шифрация



(б) дешифрация



ДСЧ является внутренней частью шифратора, также как и функция шифр. Е. Предположим, что он расположен в защищенной части шифр. аппаратуры и не может быть подменен злоумышленником (это нормально, т.к. шифраторы проектируются таким образом, чтобы обеспечить защиту от подмены алгоритма и считывания ключа).

При дешифрации законный пользователь восстанавливает B = R | P и отбрасывает R => P. Отсутствие информации о R делает все виды атаки на E более сложными.

Выбирая различные значения отношения (b/p), можно регулировать степень усиления шифрования: чем больше это отношение, тем больше усиление. С другой стороны, происходит замедление шифрования: если функция E имеет исходное значение скорости преобразования S_0, то при использовании вер. шифрования она составит $S' = S_0(b-r)/b$.

Преимущества вероятностной шифрации:

- можно существенно повысить стойкость известных блочных шифров;
- можно управлять стойкостью путем измерения r/b;
- можно использовать новые механизмы задания зависимости процедур шифрования от секретного ключа.

Недостатки вероятн. шифрации:

- скорость уменьшается в (b/p) раз;
- блоки шифртекста длиннее блоков исходного текста.

Для компенсации эффекта расширения можно использовать предварительное сжатие исходного текста (кстати, это существенно повышает стойкость к атакам на основе шифртекста, но не повышает стойкости по отношению к known-plaintext или chosen-plaintext attack).

В принципе, ДС4 можно заменить на ПСЧ, вырабатывающий длинную последовательность (брать некоторые участки). В качестве затравки можно брать значения некоторых параметров, связанных с работой оператора (интервалы между нажатиями).

3) Вероятностные механизмы в двухключевых шифрах.

Т.к. в двухключевых схемах шифрование выполняется по известному алгоритму шифрования E_z, то возможна следующая атака: при наличии шифртекста $C = E_z(T)$ для того,

чтобы найти исходный текст T, криптоаналитик может выбирать случ. разл. варианты возм. откр. текстов $T_i^{'}$ и вычислять соответствующие им криптограммы $C_1^{'} = E_z(T_1^{'}),...,C_m^{'} = E_z(T_m^{'})$. Если исходный текст угадан, то $C = C^{'}$. В принципе, каждая неудачная попытка дает некоторую информацию (сужает множество вариантов исходных текстов).

На практике это мало применимо, однако можно придумать сценарий, когда это реально (фиксированный формат сообщений).

Вероятностные механизмы позволяют предотвратить утечку информации при таком методе криптоанализа: исходному тексту соответствует множество возможных криптограмм $\{C_1,...,C_N\}$, каждый элемент которого дешифруется с помощью секретного ключа в один и тот же текст T: $T = D_Z(C_1) = D_Z(C_2) = ... = D_Z(C_N)$.

Это возможно только в том случае, если длина шифртекстов > длины исходного текста. Если длина шифртекста > длины исходного текста на r бит, то можно настроить такой вероятностный механизм, для которого число шифртекстов, соответствующих входному тексту, равно 2^r.

Теперь криптоаналитик может выбрать исходный текст, но не сможет проверить этот факт: криптоаналитик, зашифровав T, получит $C_j = E_z(T)$, но вероятность $(C_j = C_i) = 1/N$.

Таким образом надо угадать не только текст, но и значение случайного параметра. (Например, можно воспользоваться схемой, описанной в предыдущем пункте).

(взято из книги Н.А. Молдовян «Проблематика и методы криптографии», Санкт-Петербург, 1998)

21. Тесты на простоту числа.

Для многих криптографических систем необходимо сгенерировать случайное большое простое число. Наиболее тривиальный способ – случайно выбрать большое нечетное число и проверить его на простоту. Это реально, т.к. большие простые числа встречаются достаточно часто, о чем свидетельствует теорема:

Теорема: $\lim_{n->\infty} \frac{\pi(n)}{n/\ln n} = 1, где \pi(n) --число\ простых чисел < n$

Оценка n / ln(n) дает вполне точные оценки даже для небольших n. Например, при n $=10^9$ она дает всего лишь 6% погрешность, т.к. $\pi(n)$ =50,847,478, а n/ln(n)=48,254,942 (а для алгебраиста 10^9 -- это небольшое число).

Т.о. вероятность угадывания простого числа n ≈ 1/ln(n) => необходимо проверить приблизительно ln(n) чисел около n для нахождения простого числа одного порядка с n. Например, для нахождения 100-разрядного простого числа среднее число попыток ≈ $\ln 10^{100} \approx 230$ (или 115, если брать только нечетные).

Кроме того, простые числа распределены вполне равномерно.

Теорема: (оценка n-го простого числа). Обозначим n-ое простое число p_n. Тогда $p_n \sim n\ln n, а\ точнее\ n\ln n < p_n < n(\ln n + \ln\ln n)\ для\ n \geq 6.$

Вероятностные тесты на простоту.

Общая схема – проводить некоторые эвристические тесты над числом кандидатом, с каждым новым тестом увеличивая вероятность того, что число простое.

Для этого конструируются следующие множества:

Определение: Для любого нечетного n множество $W(n) \subset Z_n$ определяется так, чтобы

А) по данному $a \in Z_n$ за полином времени можно детерм. определить, верно ли, что $a \in W(n)$

Б) если n – простое, то W(n) = ∅

В) если n – составное, то #W(n) ≥ n / 2

Определение: Элементы W(n) называются свидетелями того, что n – составное, а элементы дополнительного множества $L(n) \subset Z_n \setminus W(n)$ называются лжецами.

Используется это так: выбирается случайное число a $\in Z_n$ и проверяется, верно ли, что a $\in W(n)$. Если да => число определенно составное. Иначе выдается, что число простое. Т.к. тесты независимы, то многократное выполнение тестов увеличивает вероятность того, что число простое (t попыток => P(ошибки)=1/2)

Метод Ферма.

Теорема Ферма: $n - простое \Rightarrow \forall a \in Z_n^+ \; a^{n-1} \equiv 1 (\mathrm{mod}\, n)$ (*)

Т.о. если нам удастся найти $\forall a \in Z_n^+$: n не удовлетворяет (*), то n определенно составное. Удивительным образом обратное утверждение почти верно. Поэтому метод Ферма заключается в следующем:

```
Fermat(n, t)    //n – число для проверки, t – параметр, число итераций
For i:=1 to t do            //n ≥ 3
    A:=Random (2, n-2);
    R := A^(n-1) mod n;
    if R≠1 return COMPOSITE;
return PRIME
```

Понятно, что алгоритм срабатывает не всегда, обламываясь на псевдопростых числах:

Определение: пусть n – нечетное составное число, a $\in$ [1, n-1]; n называется псевдопростым по базе a, если $a^{n-1} \equiv 1 (\mathrm{mod}\, n)$. Число a называется лжецом для алгоритма Ферма для n.

Пример: n = 341=11*31 псевдопростое по базе 2, т.к. $2^{340} \equiv 1 (\mathrm{mod}\, 341)$.

К сожалению, существуют числа, составные, но псевдопростые по всем базам a для всех a: НОД(a, n) = 1. Такие числа называются числами Кармайкла. Если n – число Кармайкла, то единственными свидетелями для n будут a: НОД(a, n) > 1. Т.о. если делители n достаточно большие, то, скорее всего, они не будут выбраны даже при большом числе попыток. Тем не менее, числа Кармайкла достаточно редки:

Теорема: составное число n является числом Кармайкла ⇔

1) n не делится на квадрат любого простого числа;
2) $(n-1) \vdots (p-1)$ для любого простого делителя $p (n \vdots p)$

Следствие: любое число Кармайкла есть произведение как минимум трех простых чисел.

Т.о. их достаточно немного: всего 225 чисел Кармайкла < 100,000,000, всего 105212 чисел Кармайкла $\le 10^{15}$. Более того, вероятность выбрать число Кармайкла β бит $\underset{\beta \to \infty}{\to} 0$.

Более того, 100-разрядное случайное число имеет всего 1 шанс из 10^{13} быть псевдопростым по базе 2.

С другой стороны, чисел Кармайкла бесконечное множество; в интервале [2, n] их больше $n^{2/7}$ для достаточно больших n. Первые числа Кармайкла: 561, 1105 и 1729 (561=3*11*17).

Тем не менее, эти недостатки заставили обратиться к более надежным критериям.

Метод Соловей-Штрассена.

В этом тесте используется символ Якоби (a/n), эквивалентный символу Лежандра для простого n. Это основано на следующем факте:

Критерий Эйлера: пусть n – нечетное простое число => $a^{\frac{n-1}{2}} \equiv (\frac{a}{n})(\mathrm{mod}\, n)$ для всех целых a: НОД(a, n) = 1.

Определение: пусть n – нечетное простое, пусть a – целое, a ∈ [1, n-1]

1) если НОД(a, n) > 1 или $a^{\frac{n-1}{2}} \neq (\frac{a}{n})(\mathrm{mod}\, n)$, то a – Эйлеровский свидетель непростоты n
2) если НОД(a, n) = 1 и $a^{\frac{n-1}{2}} \equiv (\frac{a}{n})(\mathrm{mod}\, n)$, то n – псевдопростое по Эйлеру по базе a и называется лжецом для теста Эйлера.

Пример: сосчитать число 91 = 7 * 13 – Эйлеровское псевдопростое по базе 9, т.к. $9^{45} \equiv 1(\mathrm{mod}\, 91)\, u\, (\frac{9}{91}) = 1$.

Теорема: пусть n – составное нечетное число => не больше φ(n)/2 чисел ∈ [1, n-1] являются лжецами для теста Эйлера (φ(n) – функция Эйлера).

```
Solovay-Strassen (n, t)          //n – нечетное ≥ 3, t – параметр ≥1
For i:=1 to t do
    A:=Random(2, n-2);
    R := A^((n-1)/2) mod n; if R ≠ 1 and R ≠ n-1 then return COMPOSITE;
    S:= (A/n);          //если НОД(a, n) = d => R:d => проверка R ≠ 1 эквивалентна проверке
    //НОД(a, n) ≠ 1
    If R ≠ S (mod n) then return COMPOSITE;
Return PRIME;
```

Утверждение: пусть n – нечетное составное число. Вероятность того, что Solovay-Strassen (n, t) объявит n простым < $(\frac{1}{2})^t$.

4. Метод Миллера-Рабина.

Наиболее эффективный тест, известный также под названием тест с сильными псевдопр. Основан на следующем факте:

Теорема: пусть n – нечетное простое: $n-1 = 2^s r$, где r – нечетное. Пусть a – любое целое число: НОД(a, n) = 1 => $a^r \equiv 1(\mathrm{mod}\, n)\, или\, a^{2^j r} \equiv -1(\mathrm{mod}\, n)$ для некоторого j ∈ [0, s-1]

Определение: пусть n – нечетное составное целое и пусть $n-1 = 2^s r$, где r – нечетное. Пусть a ∈ [1, n-1]

1) если $a^r \neq 1(\mathrm{mod}\, n)$ и если $a^{2^j r} \neq -1(\mathrm{mod}\, n)$ для всех j ∈ [0, s-1], то a называется сильным свидетелем (о непростоте) числа n
2) Иначе (т.е. если $a^r \equiv 1(\mathrm{mod}\, n)\, или\, a^{2^j r} \equiv -1(\mathrm{mod}\, n)$ для некоторого j ∈ [0, s-1]) n называется сильным псевдопростым по базе a (т.е. число n выступает как простое в том смысле, что удовлетворяет теореме для некоторого a). Число a называется сильным лжецом (для простоты) n.

Пример: n = 91 = 7 * 13. Т.к. 91 – 1 = 90 = 2 * 45, s =1, r = 45. Т.к. $9^r = 9^{45} \equiv 1(\mathrm{mod}\, 91)$, 91 является сильным псевдопростым по базе 9. Набор всех сильных лжецов для 91 таков: {1, 9, 10, 12, 16, 17, 22, 29, 38, 53, 62, 69, 74, 75, 79, 81, 82, 90}. Заметим, что число всех сильных лжецов для 91 = 18 = φ(91) / 4.

Теорема: Если n – составное нечетное число, то не больше ¼ всех чисел a ∈ [1, n-1] являются сильными лжецами для n. Более того, если n ≠ 9, то количество сильных лжецов не больше, чем φ(n) / 4.

Miller-Rabin (n, t) //n – нечетное ≥ 3, t – параметр ≥1

Write $n-1=2^s r$, where r is odd

For i:=1 to t do

A:=random(2, n-2);

$Y := A^r \bmod n;$

if Y ≠ 1 and Y ≠ n-1 then

J:=1

While J ≤ s - 1 and Y ≠ n – 1 do

$Y := Y^2 \bmod n;$

if Y = 1 then return COMPOSITE; // => $A^{2^j r} \equiv 1(n); A^{2^{j-1} r} \neq \pm 1(n)$

J: = J + 1; // => n – сост. с делит. НОД($A^{2^{j-1} r} - 1, n$)

If Y ≠ n – 1 then return COMPOSITE; //a – сильный свидетель для n

Return PRIME;

Утверждение: Для любого нечетного состояния n вероятность того, что Miller-Rabin (n, t) объявит n простым < $(\frac{1}{4})^t$.

Замечание: Для большого сост. n, число сильных лжецов много меньше верхней оценки φ(n)/4 => вероятность ошибки Miller-Rabin много меньше $(\frac{1}{4})^t$.

Пример: Для состояния n = 105 = 3 * 5 * 7, единственными сильными лжецами являются 1 и 104. Более того, если k ≥ 2 и n есть произв. первых k нечетных простых, то есть только 2 сильных лжеца для n, а именно 1 и n-1.

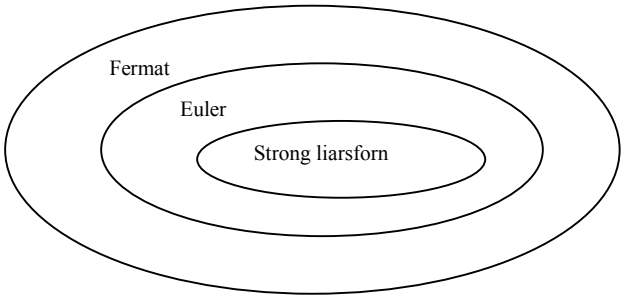
Т.о. можно и дальше оптимизировать схему Миллера-Рабина, выбирая базу не случайным образом, а начиная с тривиальных простых 2, 3, 5, 7 …

Сравнение описанных методов.

Утверждение: пусть n – нечетное составное число.

1) если a – лжец для теста Эйлера для n => a – лжец для теста Ферма для n
2) если a – сильный лжец для n => a – лжец для теста Эйлера для n

Т.о. очевидно, что Miller-Rabin никогда не хуже Solovay-Strassen'a, а тот никогда не хуже Fermat. Однако, есть случаи, когда первые два одинаково хороши.



Утверждение: Если n ≡ 3 (mod 4), то a – Эйлеров лжец ⇔ a – сильный лжец.

Кроме того, Miller-Rabin наиболее эффективен вычислительно, т.к. не требует вычислительных символов Якоби и вероятность у него лучше.

Наконец, перед применением Miller-Rabin надо проверять наличие небольших делителей, напр., проверяя делители < 256 мы выбросим 80% составных чисел до применения дорогостоящего теста Миллер-Рабина.

Random-Search (r, t) // k – целое, t – параметр ≥ 1

1. Generate odd R-bit random integer n;
2. Use trial division to determine whether n is divisible by any odd prime ≤ B. If it is return to step 1.
3. If Miller-Rabin (n, t) = PRIME => return (n);

Else return to step 1

Замечание: В большинстве приближений нас устроит вероятность ошибки $(\frac{1}{2})^{80}$. Для генерации вероятнопростых 1000-битных целых достаточно использовать Miller-Rabin с параметром t = 3.