Глава 15
Объединение блочных шифров

Существует множество способов объединять блочные алгоритмы для получения новых алгоритмов. Стимулом создавать подобные схемы является желание повысить безопасность, не пробираясь через тернии создания нового алгоритма. DES является безопасным алгоритмом, он подвергался криптоанализу добрых 20 лет и, тем не менее, наилучшим способом вскрытия остается грубая сила. Однако ключ слишком короток. Разве не плохо было бы использовать DES в качестве компонента другого алгоритма с более длинным ключом? Это позволило бы получить преимущества длинного ключа с гарантией двух десятилетий криптоанализа.

Одним из способов объединения является **многократное шифрование** - для шифрования одного и того же блока открытого текста алгоритм шифрования используется несколько раз с несколькими ключами. Шифрование каскадом похоже на многократное шифрование, но использует различные алгоритмы. Существуют и другие методы.

Повторное шифрование блока открытого текста одним и тем же ключом с помощью того же или другого алгоритма неразумно. Повторное использование того же алгоритма не увеличивает сложность вскрытия грубой силой. (Не забывайте, мы предполагаем, что алгоритм, включая количество шифрований, известен криптоаналитику.) При различных алгоритмах сложность вскрытия грубой силой может возрастать, а может и остаться неизменной. Если вы собираетесь использовать методы, описанные в этой главе, убедитесь, что ключи для последовательных шифрований различны и независимы.

15.1 Двойное шифрование

Наивным способом повысить безопасность алгоритма является шифрование блока дважды с двумя различными ключами. Сначала блок шифруется первым ключом, а затем получившийся шифротекст шифруется вторым ключом. Дешифрирование является обратным процессом.

$$C = E_{K_2}(E_{K_1}(P))$$

$$P = D_{K_1}(D_{K_2}(C))$$

Если блочный алгоритм образует группу (см. раздел 11.3), то всегда существует K_3, для которого

$$C = E_{K_2}(E_{K_1}(P)) = E_{K_3}(P)$$

Если алгоритм не образует группу, то при помощи исчерпывающего поиска взломать получающийся дважды зашифрованный блок шифротекста намного сложнее. Вместо 2^n (где n - длина ключа в битах), потребуется 2^{2n} попыток. Если алгоритм использует 64-битовый ключ, для обнаружения ключей, которыми дважды зашифрован шифротекст, потребуется 2^{128} попыток.

Но при вскрытии с известным открытым текстом это не так. Меркл и Хеллман [1075] придумали способ обменять память на время, который позволяет вскрыть такую схему двойного шифрования за 2^{n+1} шифрований, а не за 2^{2n}. (Они использовали эту схему против DES, но результаты можно обобщить на все блочные алгоритмы.) Это вскрытие называется "**встреча посередине**", с одной стороны выполняется шифрование а с другой - дешифрирование, получившиеся посередине результаты сравниваются.

В этом вскрытии криптоаналитику известны P_1, C_1, P_2 и C_2, такие что

$$C_1 = E_{K_2}(E_{K_1}(P_1))$$

$$C_2 = E_{K_2}(E_{K_1}(P_2))$$

Для каждого возможного K (или K_1, или K_2), криптоаналитик рассчитывает $E_K(P_1)$ и сохраняет результат в памяти. Собрав все результаты, он для каждого K вычисляет $D_K(C_1)$ и ищет в памяти такой же результат. Если такой результат обнаружен, то возможно, что текущий ключ - K_2, а ключ для результата в памяти - K_1. Затем криптоаналитик шифрует P_1 с помощью K_1 и K_2. Если он получает C_2, то он может гарантировать (с вероятностью успеха 1 к 2^{2n-2m}, где m - размер блока), что он узнал и K_1, и K_2. Если это не так, он продолжает поиск. Максимальное количество попыток шифрования, которое ему, возможно, придется предпринять, равно $2*2^n$, или 2^{n+1}. Если вероятность ошибки слишком велика, он может использовать третий блок шифротекста, обеспечивая вероятность успеха 1 к 2^{2n-3m}. Существуют и другие способы оптимизации [912].

Для такого вскрытия нужен большой объем памяти: 2^n блоков. Для 56-битового ключа нужно хранить 2^{56} 64-битовых блоков, или 10^{17} байтов. Такой объем памяти пока еще трудно себе представить, но этого хватает, чтобы убедить самых параноидальных криптографов в том, что двойным шифрованием пользоваться не стоит.

При 128-битовом ключе для хранения промежуточных результатов потребуется 10^{39} байтов. Если предположить, что есть способ хранить бит информации, используя единственный атом алюминия, устройство памяти, нужное для выполнения такого вскрытия, будет представлять собой алюминиевый куб с ребром, длиной 1 км. Кроме того, вам понадобится куда-то его поставить! Вскрытие "встреча посередине" кажется невозможным для ключей такого размера.

Другим способом двойного шифрования, который иногда называют **Davies-Price**, является вариантом CBC [435].

$$C_i = E_{K_1}(P_1 \oplus E_{K_2}(C_{i-1}))$$
$$P_i = D_{K_1}(C_i) \oplus E_{K_2}(C_{i-1}))$$

Утверждается, что "у этого режима нет никаких особых достоинств", к тому же он, по видимому, так же чувствителен ко вскрытию "встреча посередине" как и другие режимы двойного шифрования.

15.2

Тройное шифрование с двумя ключами

В более интересном методе, предложенном Тачменом в [1551], блок обрабатывается три раза с помощью двух ключей: первым ключом, вторым ключом и снова первым ключом. Он предлагает, чтобы отправитель сначала шифровал первым ключом, затем дешифрировал вторым, и окончательно шифровал первым ключом. Получатель расшифровывает первым ключом, затем шифрует вторым и, наконец, дешифрирует первым.

$$C = E_{K_1}(D_{K_2}(E_{K_1}(P)))$$
$$P = D_{K_1}(E_{K_2}(D_{K_1}(C)))$$

Иногда такой режим называют **шифрование-дешифрирование-шифрование** (encrypt-decrypt-encrypt, EDE) [55]. Если блочный алгоритм использует n-битовый ключ, то длина ключа описанной схемы составляет $2n$ бит. Любопытный вариант схемы шифрование-дешифрирование-шифрование был разработан в IBM для совместимости с существующими реализациями алгоритма: задание двух одинаковых ключей эквивалентно одинарному шифрованию. этим ключом. Схема шифрование-дешифрирование-шифрование сама по себе не обладает никакой безопасностью, но этот режим был использован для улучшения алгоритма DES в стандартах X9.17 и ISO 8732 [55, 761].

K_1 и K_2 чередуются для предотвращения описанного выше вскрытия "встреча посередине". Если $C = E_{K_1}(E_{K_1}(E_{K_1}(P)))$, то криптоаналитик для любого возможного K_1 может заранее вычислить $E_{K_1}(E_{K_1}(P))$ и затем выполнить вскрытие. Для этого потребуется только 2^{n+2} шифрований.

Тройное шифрование с двумя ключами устойчиво к такому вскрытию. Но Меркл и Хеллман разработали другой способ размена памяти на время, который позволяет взломать этот метод шифрования за 2^{n-1} действий, используя 2^n блоков памяти [1075].

Для каждого возможного K_2 расшифруйте 0 и сохраните результат. Затем расшифруйте 0 для каждого возможного K_1, чтобы получить P. Выполните тройное шифрование P, чтобы получить C, и затем расшифруйте C ключом K_1. Если полученное значение совпадает с значением (хранящемся в памяти), полученным при дешифровании 0 ключом K_2, то пара K_1 K_2 является возможным результатом поиска. Проверьте, так ли это. Если нет, продолжайте поиск.

Выполнение этого вскрытия с выбранным открытым текстом требует огромного объема памяти. Понадобится 2^n времени и памяти, а также 2^m выбранных открытых текстов. Вскрытие не очень практично, но все же чувствительность к нему является слабостью алгоритма.

Пауль ван Оорсчот (Paul van Oorschot) и Майкл Винер (Michael Wiener) преобразовали это вскрытие ко вскрытию с известным открытым текстом, для которого нужно p известных открытых текстов. В примере предполагается, что используется режим EDE.

(1) Предположить первое промежуточное значения a.

(2) Используя известный открытый текст, свести в таблицу для каждого возможного K_1 второе промежуточное значение b, при первом промежуточном значении, равном a:

$b = D_{K_1}(C)$

где C - это шифротекст, полученный по известному открытому тексту.

(3) Для каждого возможного K_2 найти в таблице элементы с совпадающим вторым промежуточным значение

b:

$b = E_{K_2}(a)$

(4) Вероятность успеха равно p/m, где p - число известных открытых текстов, а m - размер блока. Если совпадения не обнаружены, выберите другое a и начните сначала.

Вскрытие требует $2^{n+m}/p$ времени и p - памяти. Для DES это равно $2^{120}/p$ [1558]. Для p, больших 256, это вскрытие быстрее, чем исчерпывающий поиск.

Тройное шифрование с тремя ключами

Если вы собираетесь использовать тройное шифрование, я рекомендую три различных ключа. Общая длина ключа больше, но хранение ключа обычно не является проблемой. Биты дешевы.

$$C = E_{K_3}(D_{K_2}(E_{K_1}(P)))$$
$$P = D_{K_1}(E_{K_2}(D_{K_3}(C)))$$

Для наилучшего вскрытия с разменом памяти на время, которым является "встреча посередине", потребуется 2^{2n} действий и 2^n блоков памяти [1075]. Тройное шифрование с тремя независимыми ключами безопасно настолько, насколько на первый взгляд кажется безопасным двойное шифрование.

Тройное шифрование с минимальным ключом (TEMK)

Существует безопасный способ использовать тройное шифрование с двумя ключами, противостоящий описанному вскрытию и называемый Тройным шифрованием с минимальным ключом (Triple Encryption with Minimum Key, TEMK) [858]. Фокус в тои, чтобы получить три ключа из: X_1 и X_2.

$$K_1 = E_{X_1}(D_{X_2}(E_{X_1}(T_1)))$$
$$K_2 = E_{X_1}(D_{X_2}(E_{X_1}(T_2)))$$
$$K_3 = E_{X_1}(D_{X_2}(E_{X_1}(T_3)))$$

T_1, T_2 и T_3 представляют собой константы, которые необязательно хранить в секрете. Эта схема гарантирует, что для любой конкретной пары ключей наилучшим будет вскрытие с известным открытым текстом.

Режимы тройного шифрования

Недостаточно просто определить тройное шифрование, нужно выбрать один из способов его использования. Решение зависит от требуемых безопасности и эффективности. Вот два возможных режима тройного шифрования:

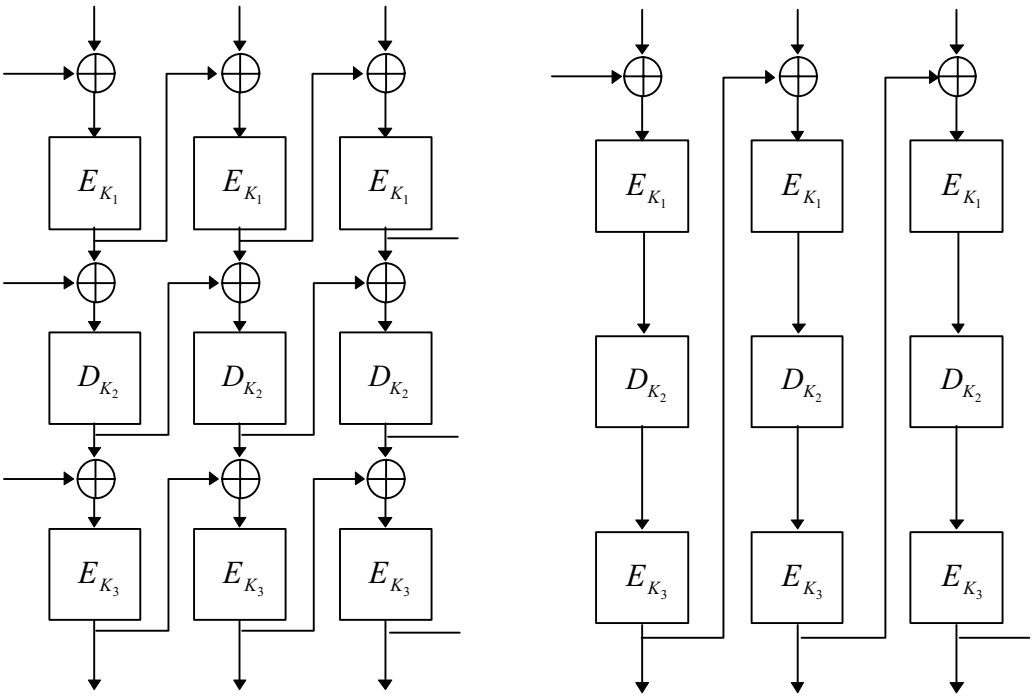
Внутренний CBC: Файл три раза шифруется в режиме CBC (см. 14tha). Для этого нужно три различных IV.

$$C_i = E_{K_3}(S_i \oplus C_{i-1}); S_i = D_{K_2}(T_i \oplus S_{i-1}); T_i = E_{K_1}(P_i \oplus T_{i-1})$$
$$P_i = T_{i-1} \oplus D_{K_1}(T_i); T_i = S_{i-1} \oplus E_{K_2}(S_i); S_i = C_{i-1} \oplus D_{K_3}(C_i)$$

C_0, S_0 и T_0 являются IV.

Внешний CBC: Файл троекратно шифруется в режиме CBC (см. 14thb). Для этого нужен один IV.

$$C_i = E_{K_3}(D_{K_2}(E_{K_1}(P_i \oplus C_{i-1})))$$
$$P_i = C_{i-1} \oplus D_{K_1}(E_{K_2}(D_{K_3}(C_i)))$$



(a) Внутренний CBC

(b) Внешний CBC

Рис. 15-1. Тройное шифрование в режиме CBC.

Для обоих режимов нужно больше ресурсов, чем для однократного шифрования: больше аппаратуры или больше времени. Однако при трех шифрующих микросхемах производительность внутреннего CBC не меньше, чем при однократном шифровании. Так как три шифрования CBC независимы, три микросхемы могут быть загружены постоянно, подавая свой выход себе на вход.

Напротив во внешнем CBC обратная связь находится снаружи по отношению к трем шифрованиям. Это означает, что даже с тремя микросхемами производительность будет равна только одной трети производительности при однократном шифровании. Чтобы получить ту же производительность для внешнего CBC, потребуется чередование IV (см. раздел 9.12):

$$C_i = E_{K_3}(D_{K_2}(E_{K_1}(P_i \oplus C_{i-3})))$$

В этом случае C_0, C_{-1} и C_{-2} являются IV. Это не поможет при программной реализации, разве только при использовании параллельного компьютера.

К сожалению менее сложный режим является также и менее безопасным. Бихам проанализировал различные режимы по отношению к дифференциальному криптоанализу и обнаружил, что безопасность внутреннего CBC по сравнению с однократным шифрованием увеличивается незначительно. Если рассматривать тройное шифрование как единый большой алгоритм, то внутренние обратные связи позволяют вводить внешнюю и известную информацию внутрь алгоритма, что облегчает криптоанализ. Для дифференциальных вскрытий нужно огромное количество выбранных шифротекстов, что делает эти вскрытия не слишком практичными, но этих результатов должно хватить, чтобы насторожить параноидальных пользователей. Анализ устойчивости алгоритмов к вскрытиям грубой силой и "встречей посередине" показал, что оба варианта одинаково безопасны [806].

Кроме этих существуют и другие режимы. Можно зашифровать файл один раз в режиме ECB, затем дважды в CBC, или один раз в CBC, один в ECB и еще раз в CBC, или дважды в CBC и один раз в ECB. Бихам показал, что эти варианты не безопаснее, чем однократный DES, против вскрытия дифференциальным криптоанализом с выбранным открытым текстом [162]. Он не оставил больших надежд и для других вариантов. Если вы собираетесь применять тройное шифрование, используйте внешнюю обратную связь.

Варианты тройного шифрования

Прежде, чем появились доказательства того, что DES не образует группу, для многократного шифрования предлагались различные схемы. Одним из способов обеспечить то, что тройное шифрование не выродится в однократное, было изменение эффективной дины блока. Простым методом является добавление бита-заполнителя. Между первым и вторым, а также между вторым и третьим шифрованиями текст дополняется

строкой случайных битов (см. Рис. 15.2). Если *PP* - это функция дополнения, то:

$$C = E_{K_3}(PP(E_{K_2}(PP(E_{K_1}(P)))))$$

Это дополнение не только разрушает шаблоны, но также обеспечивает перекрытие блоков шифрования, как кирпичей в стене. К длине сообщения добавляется только один блок.

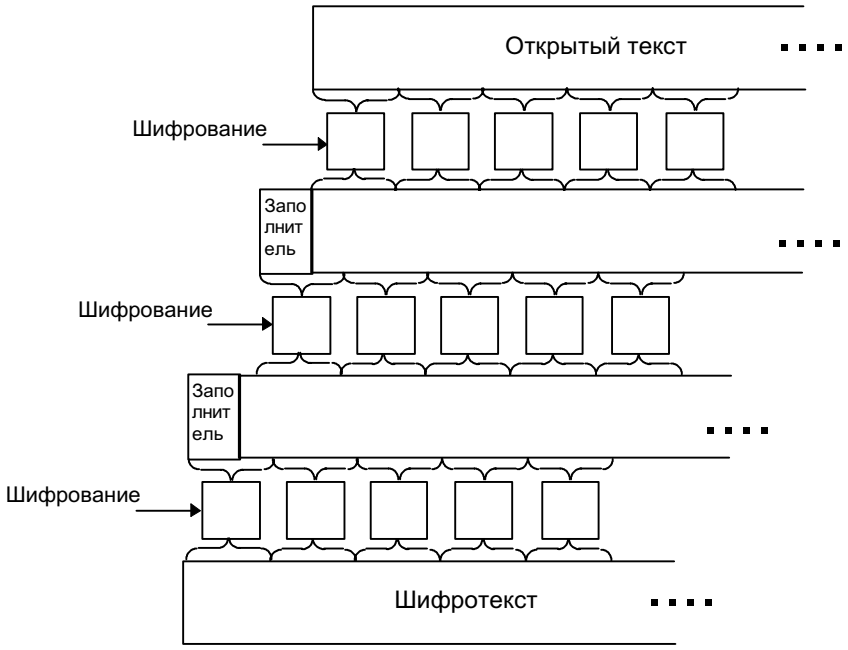


Рис. 15-2. Тройное шифрование с заполнением.

Другой метод, предложенный Карлом Эллисоном (Carl Ellison), использует некоторую функцию независимой от ключа перестановки между тремя шифрованиями. Перестановка должна работать с большими блоками - 8 Кбайт или около этого, что делает эффективный размер бока для этого варианта равным 8 Кбайтам. При условии, что перестановка выполняется быстро, этот вариант ненамного медленнее, чем базовое тройное шифрование.

$$C = E_{K_3}(T(E_{K_2}(T(E_{K_1}(P)))))$$

T собирает входные блоки (до 8 Кбайт в длину) и использует генератор псевдослучайных чисел для их перемешивания. Изменение одного бита входа приводит к изменению 8 байтов результата первого шифрования, к изменению до 64 байтов результата второго шифрования и к изменению до 512 байтов результата третьего шифрования. Если каждый блочный алгоритм работает в режиме CBC, как было первоначально предложено, то изменение единичного бита входа скорее всего приведет к изменению всего 8-килобайтового блока, даже если этот блок не является первым.

Самый последний вариант этой схемы отвечает на вскрытие внутреннего CBC, выполненное Бихамом, добавлением процедуры отбеливания, чтобы замаскировать структуру открытых текстов. Эта процедура представляет собой потоковую операцию XOR с криптографически безопасным генератором псевдослучайных чисел и ниже обозначена как *R*. *T* мешает криптоаналитику определить *a priori*, какой ключ используется для шифрования любого заданного байта входа последнего шифрования. Второе шифрование обозначено *nE* (шифрование с циклическим использованием *n* различных ключей):

$$C = E_{K_3}(R(T(nE_{K_2}(T(E_{K_1}(P))))))$$

Все шифрования выполняются в режиме ECB, используется не меньше *n*+2 ключей шифрования и криптографически безопасный генератор псевдослучайных чисел.

Эта схема была предложена для использования вместе с DES, но она работает с любым блочным алгоритмом. Результаты криптоанализа такой схемы мне неизвестны.

15.3 Удвоение длины блока

В академическом сообществе давно спорят на тему, достаточна ли 64-битовая длина блока. С одной стороны 64-битовый блок обеспечивает диффузию открытого текста только в 8 байтах шифротекста. С другой стороны более длинный блок затрудняет безопасную маскировку структуры, кроме того, больше возможностей ошибить-

ся.

Существуют предложения удваивать длину блока алгоритма с помощью многократного шифрования [299]. Прежде, чем реализовывать одно из них, оцените возможность вскрытия "встреча посередине" . Схема Ричарда Аутбриджа (Richard Outerbridge) [300], показанная на 12-й, не более безопасна, чем тройное шифрование с одинарным блоком и двумя ключами [859].

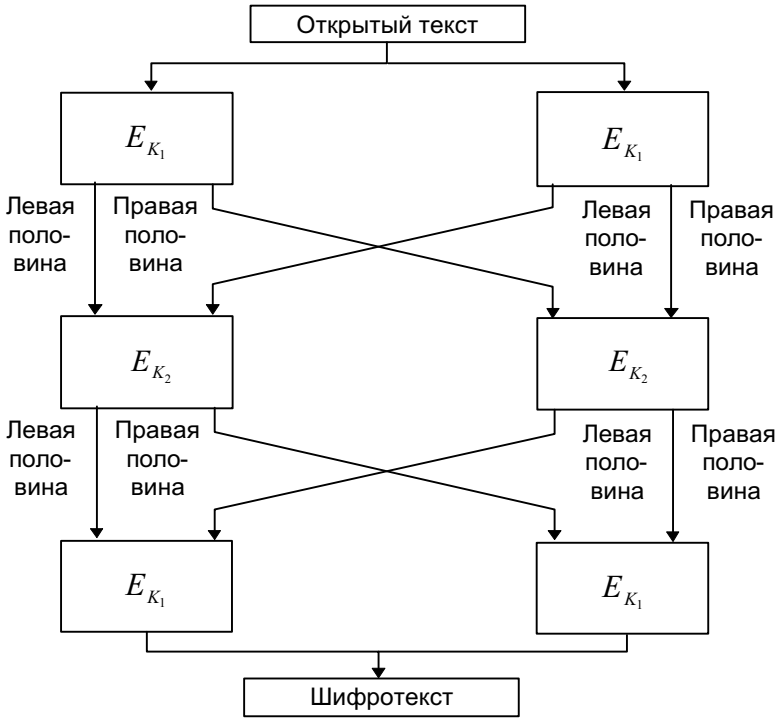


Рис. 15-3. Удвоение длины блока.

Однако я не рекомендую использовать подобный прием . Он не быстрее обычного тройного шифрования : для шифрования двух блоков данных все также нужно шесть шифрований . Характеристики обычного тройного шифрования известны, а за новыми конструкциями часто прячутся новые проблемы .

15.4 Другие схемы многократного шифрования

Проблемой тройного шифрования с двумя ключами является то, что для увеличения вдвое пространства ключей нужно выполнять три шифрования каждого блока открытого текста . Разве не здорово было бы найти какой-нибудь хитрый способ объединить два шифрования, которые удвоили бы пространство ключей ?

Двойной OFB/счетчик

Этот метод использует блочный алгоритм для генерации двух потоков ключей, которые используются для шифрования открытого текста .

$$S_i = E_{K_1}(S_{i-1} \oplus I_1); I_1 = I_1 + 1$$
$$T_i = E_{K_2}(T_{i-1} \oplus I_2); I_2 = I_2 + 1$$
$$C_i = P_i \oplus S_i \oplus T_i$$

S_i и T_i - внутренние переменные, а I_1 и I_2 - счетчики. Две копии блочного алгоритма работают в некотором гибридном режиме OFB/счетчик, а открытый текст, S_i и T_i объединяются с помощью XOR. Ключи K_1 и K_2 независимы. Результаты криптоанализа этого варианта мне неизвестны .

ECB + OFB

Этот метод был разработан для шифрования нескольких сообщений фиксированной длины, например, бл оков диска [186, 188]. Используются два ключа: K_1 и K_2. Сначала для генерации маски для блока нужной длины используется выбранный алгоритм и ключ . Эта маска будет использована повторно для шифрования сообщений теми же ключами. Затем выполняется XOR открытого текста сообщения и маски . Наконец результат XOR шифруется с помощью выбранного алгоритма и ключа K_2 в режиме ECB.

Анализ этого метода проводился только в той работе, в которой он и был опубликован . Понятно, что он не слабее одинарного шифрования ECB и возможно также силен, как и двойное применение алгоритма . Вероятно, криптоаналитик может выполнять поиск ключей независимо, если он получит несколько открытых текстов фа йлов, зашифрованных одним ключом.

Чтобы затруднить анализ идентичных блоков в одних и тех же местах различных сообщений, можно испол ьзовать IV. В отличии от использования IV в других режимах в данном случае перед шифрованием ECB выполняется XOR каждого блока сообщения с IV.

Мэтт Блэйз (Matt Blaze) разработал этот режим для своей UNIX Cryptographic File System (CFS, криптографическая файловая система). Это хороший режим, поскольку скрытым состоянием является только одно ши фрование в режиме ECB, маска может быть сгенерирована только один раз и сохранена . В CFS в качестве блочного алгоритма используется DES.

xDES^i

В [1644, 1645] DES используется как компонент ряда блочных алгоритмов с увеличенными размерами кл ючей и блоков. Эти схемы никак не зависят от DES, и в них может использоваться любой блочный алгоритм .

Первый, xDES^1, представляет собой просто схему Luby-Rackoff с блочным шифром в качестве базовой функции (см. раздел 14.11). Размер блока в два раза больше размера блока используемого блочного фильтра, а размер ключа в три раза больше, чем у используемого блочного фильтра . В каждом из 3 этапов правая полов ина шифруется блочным алгоритмом и одним из ключей, затем выполняется XOR результата и левой половины, и половины переставляются.

Это быстрее, чем обычное тройное шифрование, так как тремя шифрованиями шифруется блок, длина кот орого в два раза больше длины блока используемого блочного алгоритма . Но при этом существует простое вскрытие "встреча посередине", которое позволяет найти ключ с помощью таблицы размером 2^k, где k - это размер ключа блочного алгоритма. Правая половина блока открытого текста шифруется с помощью всех во зможных значений K_1, выполняется XOR с левой половиной открытого текста и полученные значения сохран яются в таблице. Затем правая половина шифротекста шифруется с помощью всех возможных значений K_3, и выполняется поиск совпадений в таблице. При совпадении пара ключей K_1 и K_3 - возможный вариант правого ключа. После нескольких повторений вскрытия останется только один кандидат. Таким образом, xDES^1 не является идеальным решением. Даже хуже, существует вскрытие с выбранным открытым текстом, доказывающее, что xDES^1 не намного сильнее используемого в нем блочного алгоритма [858].

В xDES^2 эта идея расширяется до 5-этапного алгоритма, размер блока которого в 4 раза, а размер ключа в 10 раз превышают размеры блока и ключа используемого блочного шифра . На 11th показан один этап xDES^2, каждый из четырех подблоков по размеру равен блоку используемого блочного шифра, а все 10 ключей независимы.

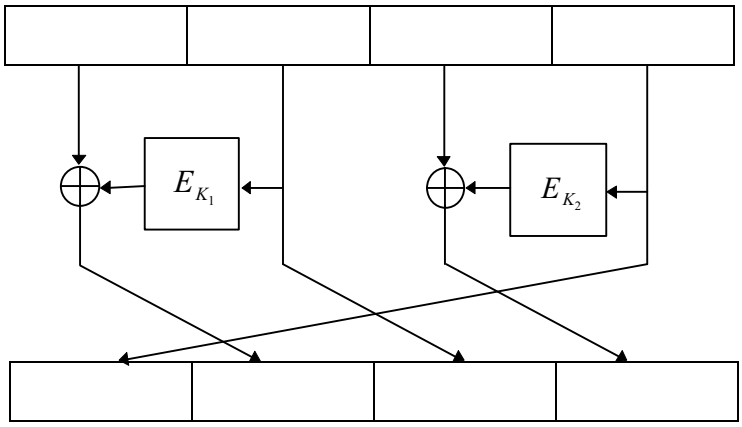


Рис. 15-4. Один этап xDES^2.

К тому же, эта схема быстрее, чем тройное шифрование: для шифрования блока, который в четыре раза больше блока используемого блочного шифра, нужно 10 шифрований . Однако этот метод чувствителен к ди ф-ференциальному криптоанализу [858] и использовать его не стоит. Такая схема остается чувствительной к ди ф-ференциальному криптоанализу, даже если используется DES с независимыми ключами этапов.

Для $i \geq 3$ xDES^i вероятно слишком велик, чтобы использовать его в качестве блочного алгоритма . Например, размер блока для xDES^3 в 6 раз больше, чем у лежащего в основе блочного шифра, ключ в 21 раз длиннее, а для шифрования блока, который в 6 раз длиннее блока лежащего в основе блочного шифра, нужно 21 шифрование .

Это медленнее, чем тройное шифрование.

Пятикратное шифрование

Если тройное шифрование недостаточно безопасно - может быть, вам нужно шифровать ключи тройного шифрования, используя еще более сильный алгоритм - то кратность шифрования можно увеличить. Очень устойчиво к вскрытию "встреча посередине" пятикратное шифрование. (Аргументы, аналогичные рассмотренным для двойного шифрования, показывают, что четырехкратное шифрование по сравнению с тройным лишь незначительно повышает надежность.)

$$C = E_{K_1}(D_{K_2}(E_{K_3}(D_{K_2}(E_{K_1}(P)))))$$
$$P = D_{K_1}(E_{K_2}(D_{K_3}(E_{K_2}(D_{K_1}(C)))))$$

Эта схема обратно совместима с тройным шифрованием, если $K_1 = K_2$, и с однократным шифрованием, если $K_1 = K_2 = K_3$. Конечно, она будет еще надежней, если использовать пять независимых ключей.

15.5 Уменьшение длины ключа в CDMF

Этот метод был разработан IBM для продукта CDMF (Commercial Data Masking Facility, Коммерческое средство маскирования данных) (см. раздел 24.8), чтобы превратить 56-битовый ключ DES в 40-битовый, разрешенный для экспорта [785]. Предполагается, что первоначальный ключ DES содержит биты четности.

(1) Обнуляются биты четности: биты 8, 16, 24, 32, 40, 48, 56, 64.

(2) Результат этапа (1) шифруется с помощью DES ключом 0xc408b0540ba1e0ae, результат шифрования об ъединяется посредством XOR с результатом этапа (1).

(3) В результате этапа (2) обнуляются следующие биты: 1, 2, 3, 4, 8, 16, 17, 18, 19, 2.0, 2.4, 32, 33, 34, 35, 36, 40, 48, 49, 50, 51, 52, 56, 64.

(4) Результат этапа (3) шифруется с помощью DES ключом 0xef2c041ce6382fe6. Полученный ключ использ уется для шифрования сообщения.

Не забывайте, что этот метод укорачивает ключ и, следовательно, ослабляет алгоритм.

15.6 Отбеливание

Отбеливанием (whitening) называется способ, при котором выполняется XOR части ключа с входом блочного алгоритма и XOR другой части ключа с выходом блочного алгоритма. Впервые этот метод был применен для варианта DESX, разработанного RSA Data Security, Inc., а затем (по-видимому, независимо) в Khufu и Khafre. (Ривест и дал имя этому методу, это необычное использование слова.)

Смысл этих действий в том, чтобы помешать криптоаналитику получить пару "открытый текст/шифротекст" для лежащего в основе блочного алгоритма. Метод заставляет криптоаналитика угадывать не только ключ алгоритма, но и одно из значений отбеливания. Так как XOR выполняется и перед, и после блочного алгоритма, считается, что этот метод устойчив против вскрытия "встреча посередине".

$$C = K_3 \oplus E_{K_2}(P \oplus K_1)$$
$$P = K_1 \oplus D_{K_2}(C \oplus K_3)$$

Если $K_1 = K_2$, то для вскрытия грубой силой потребуется $2^{n+m/p}$ действий, где n - размер ключа, m - размер блока, и p - количество известных открытых текстов. Если K_1 и K_2 различны, то для вскрытия грубой силой с тремя известными открытыми текстами потребуется 2^{n+m+1} действий. Против дифференциального и линейного криптоанализа, такие меры обеспечивают защиту только для нескольких битов ключа. Но с вычислительной точки зрения это очень дешевый способ повысить безопасность блочного алгоритма.

15.7 Многократное последовательное использование блочных алгоритмов

А как насчет шифрования сначала алгоритмом A и ключом K_A, а затем еще раз алгоритмом B и ключом K_B? Может быть у Алисы и Боба различные представления о том, какой алгоритм безопаснее : Алиса хочет пользоваться алгоритмом A, а Боб - алгоритмом B. Этот прием, иногда называемый **последовательным использованием** (cascading), можно распространить и на большее количество алгоритмов и ключей.

Пессимисты утверждали, что совместное использование двух алгоритмов не гарантирует повышения без опасности. Алгоритмы могут взаимодействовать каким-то хитрым способом, что на самом деле даже *уменьшит*. Даже тройное шифрование тремя различными алгоритмами может не быть настолько безопасным, насколько

вам это кажется. Криптография - достаточно темное искусство, если вы не совсем понимаете, что делаете, то можете легко попасть в беду.

Действительность намного светлее. Упомянутые предостережения верны, только если различные ключи зависят друг от друга. Если все используемые ключи независимы, то сложность взлома последовательности алгоритмов по крайней мере не меньше, чем сложность взлома первого из применяемых алгоритмов [1033]. Если второй алгоритм чувствителен к вскрытию с выбранным открытым текстом, то первый алгоритм может облегчить это вскрытие и при последовательном использовании сделать второй алгоритм чувствительным к вскрытию с известным открытым текстом. Такое возможное облегчение вскрытия не ограничивается только алгоритмами шифрования: если вы позволите кому-то другому определить любой из алгоритмов, делающих что-то с вашим сообщением до шифрования, стоит удостовериться, что ваше шифрование устойчиво по отношению к вскрытию с выбранным открытым текстом. (Обратите внимание, что наиболее часто используемым алгоритмом для сжатия и оцифровки речи до модемных скоростей, применяемым перед любым алгоритмом шифрования, является CELP, разработанный NSA.)

Это можно сформулировать и иначе: При использовании вскрытия с выбранным открытым текстом последовательность шифров взломать не легче, чем любой из шифров последовательности [858]. Ряд результатов показал, что последовательное шифрование взломать по крайней мере не легче, чем самый сильный из шифров последовательности, но в основе этих результатов лежат некоторые несформулированные предположения [528]. Только если алгоритмы коммутативны, как в случае каскадных потоковых шифров (или блочных шифров в режиме OFB), надежность их последовательности не меньше, чем у сильнейшего из используемых алгоритмов.

Если Алиса и Боб не доверяют алгоритмам друг друга, они могут использовать их последовательно. Для потоковых алгоритмов их порядок не имеет значения. При использовании блочных алгоритмов Алиса может сначала использовать алгоритм A, а затем алгоритм B. Боб, который больше доверяет алгоритму B, может использовать алгоритм B перед алгоритмом A. Между алгоритмами они могут вставить хороший потоковый шифр. Это не причинит вреда и может значительно повысить безопасность.

Не забудьте, что ключи для каждого алгоритма последовательности должны быть независимыми. Если алгоритм A использует 64-битовый ключ, а алгоритм B - 128-битовый ключ, то получившаяся последовательность должна использовать 192-битовый ключ. При использовании зависимых ключей у пессимистов гораздо больше шансов оказаться правыми.

15.8 Объединение нескольких блочных алгоритмов

Вот другой способ объединить несколько блочных алгоритмов, безопасность которого гарантировано будет по крайней мере не меньше, чем безопасность обоих алгоритмов. Для двух алгоритмов (и двух независимых ключей):

(1) Генерируется строка случайных битов R того же размера, что и сообщение M.

(2) R шифруется первым алгоритмом.

(3) $M \oplus R$ шифруется вторым алгоритмом.

(4) Шифротекст сообщения является объединением результатов этапов (2) и (3).

При условии, что строка случайных битов действительно случайна, этот метод шифрует M с помощью одноразового блокнота, а затем содержимое блокнота и получившееся сообщение шифруются каждым из двух алгоритмов. Так как и то, и другое необходимо для восстановления M, криптоаналитику придется взламывать оба алгоритма. Недостатком является удвоение размера шифротекста по сравнению с открытым текстом.

Этот метод можно расширить для нескольких алгоритмов, но добавление каждого алгоритма увеличивает шифротекст. Сама по себе идей хороша, но, как мне кажется, не очень практична.

Глава 16
Генераторы псевдослучайных последовательностей и потоковые шифры

16.1 Линейные конгруэнтные генераторы

Линейными конгруэнтными генераторами являются генераторы следующей формы

$X_n = (aX_{n-1} + b) \bmod m$

в которых X_n - это n-ый член последовательности, а X_{n-1} - предыдущий член последовательности. Переменные a, b и m - постоянные: a - **множитель**, b - **инкремент**, и m - модуль. Ключом, или затравкой, служит значение X_0.

Период такого генератора не больше, чем m. Если a, b и m выбраны правильно, то генератор будет **генератором с максимальным периодом** (иногда называемым максимальной длиной), и его период будет равен m. (Например, b должно быть взаимно простым с m.) Подробное описание выбора констант для получения максимального периода можно найти в [863, 942]. Еще одной хорошей статьей по линейным конгруэнтным генераторам и их теории является [1446].

В 15-й, взятой из [1272,], перечисляются хорошие константы линейных конгруэнтных генераторов. Все они обеспечивают генераторы с максимальным периодом и, что даже более важно, удовлетворяют спектральному тесту на случайность для размерностей 2, 3, 4, 5 и 6 [385, 863]. Таблица организована по максимальному произведению, которое не вызывает переполнения в слове указанной длины.

Преимуществом линейных конгруэнтных генераторов является их быстрота за счет малого количества операций на бит.

К несчастью линейные конгруэнтные генераторы нельзя использовать в криптографии, так как они предсказуемы. Впервые линейные конгруэнтные генераторы были взломаны Джимом Ридсом (Jim Reeds) [1294, 1295, 1296], а затем Джоан Бояр (Joan Boyar) [1251]. Ей удалось также вскрыть квадратичные генераторы:

$Xn = (aX_{n-1}^2 + bX_{n-1} + c) \bmod m$

и кубические генераторы:

$Xn = (aX_{n-1}^3 + bX_{n-1}^2 + c\,X_{n-1} + d) \bmod m$

Другие исследователи расширили идеи Бояр, разработав способы вскрытия любого полиномиального генератора [923, 899, 900]. Были взломаны и усеченные линейные конгруэнтные генераторы [581, 705, 580], и усеченные линейные конгруэнтные генераторы с неизвестными параметрами [1500, 212]. Таким образом была доказана бесполезность конгруэнтных генераторов для криптографии.

Табл. 16-1.
Константы для линейных конгруэнтных генераторов

Переполняется при	a	b	m
2^{20}	106	1283	6075
2^{21}	211	1663	7875
2^{22}	421	1663	7875
2^{23}	430	2531	11979
	936	1399	6655
	1366	1283	6075
2^{24}	171	11213	53125
	859	2531	11979
	419	6173	29282
	967	3041	14406
2^{25}	141	28411	134456
	625	6571	31104
	1541	2957	14000
	1741	2731	12960
	1291	4621	21870
	205	29573	139968
2^{26}	421	17117	81000
	1255	6173	29282

	281	28411	134456
2^{27}	1093	18257	86436
	421	54773	259200
	1021	24631	116640
	1021	25673	121500
2^{28}	1277	24749	117128
	741	66037	312500
	2041	25673	121500
2^{29}	2311	25367	120050
	1807	45289	214326
	1597	51749	244944
	1861	49297	233280
	2661	36979	175000
	4081	25673	121500
	3661	30809	145800
2^{30}	3877	29573	139968
	3613	45289	214326
	1366	150889	714025
2^{31}	8121	28411	134456
	4561	51349	243000
	7141	54773	259200
2^{32}	9301	49297	233280
	4096	150889	714025
2^{33}	2416	374441	1771875
2^{34}	17221	107839	510300
	36261	66037	312500
2^{35}	84589	45989	217728

Однако, линейные конгруэнтные генераторы сохраняют свою полезность для некриптографических прил о-жений, например, для моделирования. Они эффективны и в большинстве используемых эмпирических тестах демонстрируют хорошие статистические характеристики. Важную информацию о линейных конгруэнтных г е-нераторах и их теории можно найти в [942].

Объединение линейных конгруэнтных генераторов

Был предпринят ряд попыток объединения линейных конгруэнтных генераторов [1595, 941]. Криптографи-ческая безопасность полученных результатов не повышается, но они обладают более длинными периодами и лучшими характеристиками в некоторых статистических тестах. Для 32-битовых компьютеров можно испол ь-зовать следующий генератор [941]:

```
static long sl = 1 ; /* "long" должно быть 32-битовым целым. */ static long s2 = 1 ;

#define MODMULT(a,b,c,m,s) q = s/a; s = b*(s-a*q) - c*q; if (s<0) s+=m ;
/* MODMIJLT(a,b,c,nl,s) рассчитывает s*b mod m при условии, что m=a*b+c и 0 <= c < m. */

/* combinedLCG возвращает действительное псевдослучайное значение в диапазоне
* (0,1). Она объединяет линейные конгруэнтные генераторы с периодами
* 2^31-85 и 2^31-249, и ее период равен произведению этих двух простых чисел. */

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

/* В общем случае перед использованием combinedLCG вызывается initLCG. */
void initLCG( long InitS1, long InitS2 )
{
     sl = InitS1;
     s2 = InitS2;
}
```

Этот генератор работает при условии, что компьютер может представить все целые числа между $-2^{31}+85$ и $2^{31}-249$. Переменные s_1 и s_2 глобальны и содержат текущее состояние генератора. Перед первым вызовом их необходимо проинициализировать. Для переменной s_1 начальное значение должно лежать в диапазоне между 1

и 2147483562, для переменной s_2 - между 1 и 2147483398. Период генератора близок к 10^{18}.

На 16-битовом компьютере используйте другой генератор:

```
static int s1 = 1 ; /* "int" должно быть 16-битовым целым. */
static int s2 = 1 ;
static int s3 = 1 ;

#define MODMULT(a,b,c,m,s) q = s/a; s = b*(s-a*q) - c*q; if (s<0) s+=m ;
/* combinedLCG возвращает действительное псевдослучайное значение в диапазоне
* (0,1). Она объединяет линейные конгруэнтные генераторы с периодами 2^15-405,
* 2^15-1041 и 2^15-1111, и ее период равен произведению этих трех простых чисел. */

double combinedLCG ( void )
{
     long q ;
     long z ;

     MODMULT ( 206, 157, 21, 32363, s1 )
     MODMULT ( 217, 146, 45, 31727, s2 )
     MODMULT ( 222, 142, 133, 31657, s3 )
     z = s1 - s2 ;
     if ( z < 1 )
             z -= 32362 ;
     z += s3 ;
     if ( z < 1 )
             z += 32362 ;
     return z * 3.0899e-5 ;

}

/* В общем случае перед использованием combinedLCG вызывается initLCG. */
void initLCG( long InitS1, long InitS2, long InitS3)
{
     s1 = InitS1;
     s2 = InitS2;
     s3 = InitS3;
}
```

Этот генератор работает при условии, что компьютер может представить все целые числа между -32363 и 32363. Переменные s_1, s_2 и s_3 глобальны и содержат текущее состояние генератора. Перед первым вызовом их необходимо проинициализировать. Для переменной s_1 начальное значение должно лежать в диапазоне между 1 и 32362, для переменной s_2 - между 1 и 31726, для переменной s_3 - между 1 и 31656. Период генератора равен $1.6*10^{13}$. Для обоих генераторов константа b равна 0.

16.2 Сдвиговые регистры с линейной обратной связью

Последовательности сдвиговых регистров используются как в криптографии, так и в теории кодирования. Их теория прекрасно проработана, потоковые шифры на базе сдвиговых регистров являлись рабочей лошадкой военной криптографии задолго до появления электроники.

Сдвиговый регистр с обратной связью состоит из двух частей: сдвигового регистра и **функции обратной связи** (см. 15th). Сдвиговый регистр представляет собой последовательность битов. (Количество битов определяется **длиной** сдвигового регистра. Если длина равна n битам, то регистр называется n-битовым сдвиговым регистром.) Всякий раз, когда нужно извлечь бит, все биты сдвигового регистра сдвигаются вправо на 1 позицию. Новый крайний левый бит является функцией всех остальных битов регистра. На выходе сдвигового регистра оказывается один, обычно младший значащий, бит. **Периодом** сдвигового регистра называется длина получаемой последовательности до начала ее повторения.

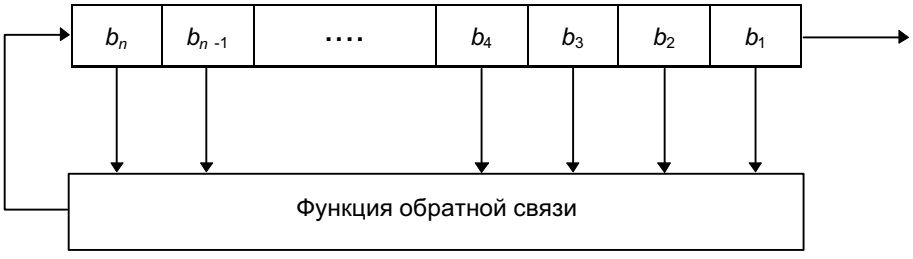


Рис. 16-1. Сдвиговый регистр с обратной связью

Криптографам нравились потоковые шифры на базе сдвиговых регистров: они легко реализовывались с помощью цифровой аппаратуры. Я лишь слегка затрону математическую теорию. В 1965 году Эрнст Селмер (Ernst Selmer), главный криптограф норвежского правительства, разработал теорию последовательности сдвиговых регистров [1411]. Соломон Голомб (Solomon Golomb), математик NSA, написал книгу, излагающие ее некоторые свои резальтаты и результаты Селмера [643]. См. также [970, 971, 1647].

Простейшим видом сдвигового регистра с обратной связью является **линейный сдвиговый регистр с обратной связью** (linear feedback shift register, или LFSR) (см. 14th). Обратная связь представляет собой просто XOR некоторых битов регистра, перечень этих битов называется **отводной последовательностью** (tap sequence). Иногда такой регистр называется **конфигурацией Фиббоначи**. Из-за простоты последовательности обратной связи для анализа LFSR можно использовать довольно развитую математическую теорию. Криптографы любят анализировать последовательности, убеждая себя, что эти последовательности достаточно случа йны, чтобы быть безопасными. LFSR чаще других сдвиговых регистров используются в криптографии.

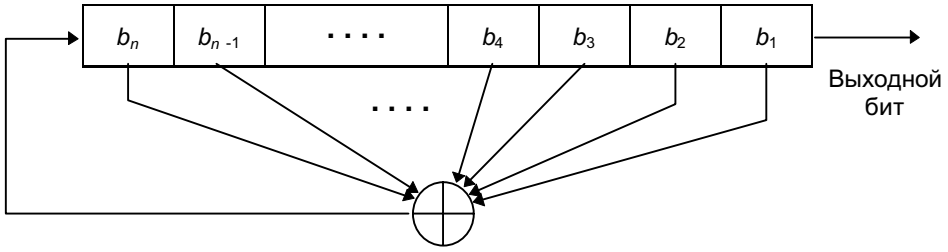


Рис. 16-2. Сдвиговый регистр с линейной обратной связью.

На 13-й показан 4-битовый LFSR с отводом от первого и четвертого битов. Если его проинициализировать значением 1111, то до повторения регистр будет принимать следующие внутренние состояния :

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

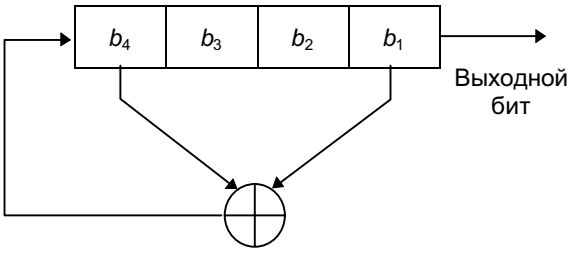


Рис. 16-3. 4-битовый LFSR.

Выходной последовательностью будет строка младших значащих битов :

1 1 1 1 0 1 0 1 1 0 0 1 0 0 0. . . .

n-битовый LFSR может находиться в одном из 2^n-1 внутренних состояний. Это означает, что теоретически такой регистр может генерировать псевдослучайную последовательность с периодом 2^n-1 битов. (Число внутренних состояний и период равны 2^n-1, потому что заполнение LFSR нулями, приведет к тому, что сдвиговый регистр будет выдавать бесконечную последовательность нулей, что абсолютно бесполезно .) Только при определенных отводных последовательностях LFSR циклически пройдет через все 2^n-1 внутренних состояний, такие

LFSR являются LFSR с максимальным периодом. Получившийся результат называется **M-последовательностью**.

Для того, чтобы конкретный LFSR имел максимальный период, многочлен, образованный из отводной последовательности и константы 1, должен быть примитивным по модулю 2. **Степень** многочлена является длиной сдвигового регистра. Примитивный многочлен степени n - это неприводимый многочлен, который является делителем $x^{2^n-1}+1$, но не является делителем x^d+1 для всех d, являющихся делителями 2^n-1 (см. раздел 11.3). Соответствующую математическую теорию можно найти в [643, 1649, 1648].

В общем случае не существует простого способа генерировать примитивные многочлены данной степени по модулю 2. Проще всего выбирать многочлен случайным образом и проверять, не является ли он примитивным. Это нелегко - и чем-то похоже на проверку, не является ли простым случайно выбранное число - но многие математические пакеты программ умеют решать такую задачу. Ряд методов приведен в [970, 971].

Некоторые, но, конечно же, не все, многочлены различных степеней, примитивные по модулю 2, приведены в 14-й [1583, 643, 1649, 1648, 1272, 691]. Например, запись (32, 7, 5, 3, 2, 1, 0) означает, что следующий многочлен примитивен по модулю 2:

$$x^{32} + x^7 + x^5 + x^3 + x^2 + x + 1$$

Это можно легко обобщить для LFSR с максимальным периодом. Первым числом является длина LFSR. Последнее число всегда равно 0, и его можно опустить. Все числа, за исключением 0, задают отводную последовательность, отсчитываемую от левого края сдвигового регистра. То есть, члены многочлена с меньшей степенью соответствуют позициям ближе к правому краю регистра.

Продолжая пример, запись (32, 7, 5, 3, 2, 1, 0) означает, что для взятого 32-битового сдвигового регистра новый бит новый бит генерируется с помощью XOR тридцать второго, седьмого, пятого, третьего, второго и первого битов (см. 12th), получающийся LFSR будет иметь максимальную длину, циклически проходя до повторения через 2^{32}-1 значений.

Код для этого LFSR на языке C выглядит следующим образом:

```
int LFSR ( ) {
     static unsigned long ShiftRegister = 1;
     /* Все, кроме 0. */
     ShiftRegister = ((((ShiftRegister >> 31)
            ^(ShiftRegister >> 6)
            ^(ShiftRegister >> 4)
            ^(ShiftRegister >> 2)
            ^(ShiftRegister >> 1)
            ^ShiftRegister))
            & 0x00000001)
            <<31)
            | (ShiftRegister >> 1) ;
     return ShiftRegister & 0x00000001;
}
```

Если сдвиговый регистр длиннее компьютерного слова, код усложняется, но не намного.

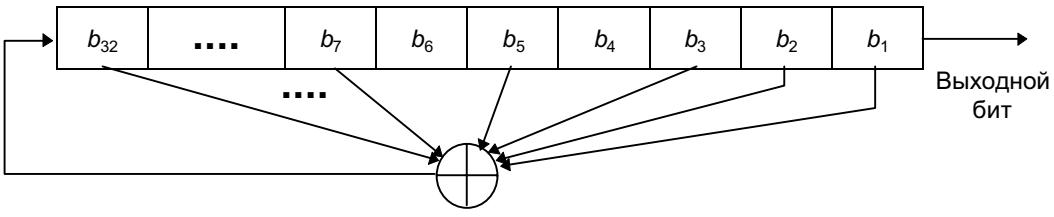


Рис. 16-4. 32-битовый LFSR с максимальной длиной.

Табл. 16-2.
Некоторые примитивные многочлены по модулю 2

(1, 0)	(7, 3, 0)	(14, 5, 3, 1, 0)	(18, 5, 2, 1, 0)
(2, 1, 0)	(8, 4, 3, 2, 0)	(15, 1, 0)	(19, 5, 2, 1, 0)
(3, 1, 0)	(9, 4, 0)	(16, 5, 3.2, 0)	(20, 3, 0)
(4, 1, 0)	(10, 3, 0)	(17, 3, 0)	(21, 2, 0)
(5, 2, 0)	(11, 2, 0)	(17, 5, 0)	(22, 1, 0)
(6, 1, 0)	(12, 6, 4, 1, 0)	(17, 6, 0)	(23, 5, 0)
(7, 1, 0)	(13, 4, 3, 1, 0)	(18, 7, 0)	(24, 4, 3, 1, 0)

(25, 3, 0)
(26, 6, 2, 1, 0)
(27, 5, 2, 1, 0)
(28, 3, 0)
(29, 2, 0)
(30, 6, 4, 1.0)
(31, 3, 0)
(31, 6, 0)
(31, 7, 0)
(31, 13, 0)
(32, 7, 6, 2, 0)
(32, 7, 5, 3, 2, 1, 0)
(33, 13, 0)
(33, 16, 4, 1, 0)
(34, 8, 4, 3, 0)
(34, 7, 6, 5, 2, 1, 0)
(35, 2, 0)
(135, 11, 0)
(135, 16, 0)
(135, 22, 0)
(136, 8, 3, 2, 0)
(137, 21, 0)
(138, 8, 7, 1, 0)
(139, 8, 5, 3, 0)
(140, 29, 0)
(141, 13, 6, 1, 0)
(142, 21, 0)
(143, 5, 3, 2, 0)
(144, 7, 4, 2, 0)
(145, 52, 0)
(145, 69, 0)
(146, 5, 3, 2, 0)
(147, 11, 4, 2, 0)
(148, 27, 0)
(149, 10, 9, 7, 0)
(150, 53, 0)
(151, 3, 0)
(151, 9, 0)
(151, 15, 0)
(151, 31, 0)
(151, 39, 0)
(151, 43, 0)
(151, 46, 0)
(151, 51, 0)
(151, 63, 0)
(151, 66, 0)
(151, 67, 0)
(151, 70, 0)
(36, 11, 0)
(36, 6, 5, 4, 2, 1, 0)
(37, 6, 4, 1, 0)
(37, 5, 4, 3, 2, 1, 0)
(38, 6, 5, 1, 0)
(39, 4, 0)
(40, 5, 4, 3, 0)
(41, 3, 0)
(42, 7, 4, 3, 0)
(42, 5, 4, 3, 2, 1, 0)
(43, 6, 4, 3, 0)
(44, 6, 5, 2, 0)
(45, 4, 3, 1, 0)
(46, 8, 7, 6, 0)
(46, 8, 5, 3, 2, 1, 0)
(47, 5, 0)
(48, 9, 7, 4, 0)
(48, 7, 5, 4, 2, 1, 0)
(49, 9, 0)
(49, 6, 5, 4, 0)
(50, 4, 3, 2, 0)
(51, 6, 3, 1, 0)
(52, 3, 0)
(53, 6, 2, 1, 0)
(54, 8, 6, 3, 0)
(54, 6, 5, 4, 3, 2, 0)
(55, 24, 0)
(55, 6, 2, 1, 0)
(56, 7, 4, 2, 0)
(57, 7, 0)
(57, 5, 3, 2, 0)
(58, 19.0)
(58, 6, 5, 1, 0)
(59, 7, 4, 2, 0)
(59, 6, 5, 4, 3, 1, 0)
(60, 1, 0)
(61, 5, 2, 1, 0)
(62, 6, 5, 3, 0)
(63, 1, 0)
(64, 4, 3, 1, 0)
(65, 18, 0)
(65, 4, 3, 1, 0)
(66, 9, 8, 6, 0)
(66, 8, 6, 5, 3, 2, 0)
(67, 5, 2, 1, 0)
(152, 6, 3, 2, 0)
(153, 1, 0)
(153, 8, 0)
(154, 9, 5, 1, 0)
(155, 7, 5, 4, 0)
(156, 9, 5, 3, 0)
(157, 6, 5, 2, 0)
(158, 8, 6, 5, 0)
(159, 31, 0)
(159, 34, 0)
(159, 40, 0)
(160, 5, 3, 2, 0)
(161, 18, 0)
(161, 39, 0)
(161, 60, 0)
(162, 8, 7, 4, 0)
(163, 7, 6, 3, 0)
(164, 12, 6, 5, 0)
(165, 9, 8, 3, 0)
(166, 10, 3, 2, 0)
(167, 6, 0)
(170, 23, 0)
(172, 2, 0)
(174, 13, 0)
(175, 6, 0)
(175, 16, 0)
(175, 18, 0)
(175, 57, 0)
(177, 8, 0)
(177, 22, 0)
(1 77, 88, 0)
(68, 9, 0)
(68, 7, 5, 1, 0)
(69, 6, 5, 2, 0)
(70, 5, 3, 1, 0)
(71, 6, 0)
(71, 5, 3, 1, 0)
(72, 10, 9, 3, 0)
(72, 6, 4, 3, 2, 1, 0)
(73, 25, 0)
(73, 4, 3, 2, 0)
(74, 7, 4, 3, 0)
(75, 6, 3, 1, 0)
(76, 5, 4, 2, 0)
(77, 6, 5, 2, 0)
(78, 7, 2, 1, 0)
(79, 9, 0)
(79, 4, 3, 2, 0)
(80, 9, 4, 2, 0)
(80, 7, 5, 3, 2, 1, 0)
(81, 4, 0)
(82, 9, 6, 4, 0)
(82, 8, 7, 6, 1, 0)
(83, 7, 4, 2, 0)
(84, 13, 0)
(84, 8, 7, 5, 3, 1, 0)
(85, 8, 2, 1, 0)
(86, 6, 5, 2, 0)
(87, 13, 0)
(87, 7, 5, 1, 0)
(88, 11, 9, 8, 0)
(88, 8, 5, 4, 3, 1, 0)
(89, 38, 0)
(89, 51, 0)
(89, 6, 5, 3, 0)
(90, 5, 3, 2, 0)
(91, 8, 5, 1, 0)
(91, 7, 6, 5, 3, 2, 0)
(92, 6, 5, 2, 0)
(93, 2, 0)
(94, 21, 0)
(94, 6, 5, 1, 0)
(95, 11, 0)
(95, 6, 5, 4, 2, 1, 0)
(96, 10, 9, 6, 0)
(96, 7, 6, 4, 3, 2, 0)
(178, 87, 0)
(183, 56, 0)
(194, 87, 0)
(198, 65, 0)
(201, 14, 0)
(201, 17, 0)
(201, 59, 0)
(201, 79, 0)
(202, 55, 0)
(207, 43, 0)
(212, 105, 0)
(218, 11, 0)
(218, 15, 0)
(218, 71, 0)
(218.83, 0)
(225, 32, 0)
(225, 74, 0)
(225, 88, 0)
(225, 97, 0)
(225, 109, 0)
(231, 26, 0)
(231, 34, 0)
(234, 31, 0)
(234, 103, 0)
(236, 5, 0)
(250, 103, 0)
(255, 52, 0)
(255, 56, 0)
(255, 82, 0)
(258, 83, 0)
(266, 47, 0)
(97, 6, 0)
(98, 11, 0)
(98, 7, 4, 3, 1, 0)
(99, 7, 5, 4, 0)
(100, 37, 0)
(100, 8, 7, 2, 0)
(101, 7, 6, 1, 0)
(102, 6, 5, 3, 0)
(103, 9, 9)
(104, 11, 10, 1, 0)
(105, 16, 0)
(106, 15, 0)
(107, 9, 7, 4, 0)
(108, 31, 0)
(109, 5, 4, 2.0)
(110, 6, 4, 1, 0)
(111, 10, 0)
(111, 49, 0)
(113, 9, 0)
(113, 15, 0)
(113, 30, 0)
(114, 11, 2, 1, 0)
(115, 8, 7, 5, 0)
(116, 6, 5, 2, 0)
(117, 5, 2, 1, 0)
(118, 33, 0)
(119, 8, 0)
(119, 45, 0)
(120, 9, 6, 2, 0)
(121, 18, 0)
(122, 6, 2, 1, 0)
(123, 2, 0)
(124, 37, 0)
(125, 7, 6, 5, 0)
(126, 7, 4, 2, 0)
(127, 1, 0)
(127, 7, 0)
(127, 63, 0)
(128, 7, 2, 1, 0)
(129, 5, 0)
(130, 3, 0)
(131, 8, 3, 2, 0)
(132, 29, 0)
(133, 9, 8, 2, 0)
(134, 57, 0)
(270, 133, 0)
(282, 35, 0)
(282, 43, 0)

(286, 69, 0)	(378, 43, 0)	(521, 168, 0)	(2281, 915, 0)
(286, 73, 0)	(378, 107, 0)	(607, 105, 0)	(2281, 1029, 0)
(294, 61, 0)	(390, 89, 0)	(607, 147, 0)	(3217, 67, 0)
(322, 67, 0)	(462, 73, 0)	(607, 273, 0)	(3217, 576, 0)
(333, 2, 0)	(521, 32, 0)	(1279, 216, 0)	(4423, 271, 0)
(350, 53, 0)	(521, 48, 0)	(1279, 418, 0)	(9689, 84, 0)
(366, 29, 0)	(521, 158, 0)	(2281, 715, 0)	

Обратите внимание, что у всех элементов таблицы нечетное число коэффициентов . Я привел такую длинную таблицу, так как LFSR часто используются для криптографии с потоковыми шифрами, и я хотел, чтобы разные люди могли подобрать различные примитивные многочлены . Если $p(x)$ примитивен, то примитивен и $x^n p(1/x)$, поэтому каждый элемент таблицы на самом деле определяет два примитивных многочлена .

Например, если (a, b, 0) примитивен, то примитивен и (a, a - b, 0). Если примитивен (a, b, c, d, 0), то примитивен и (a, a - d, a - c, a - b, 0). Математически:

если примитивен $x^a + x^b + 1$, то примитивен и $x^a + x^{a-b} + 1$

если примитивен $x^a + x^b + x^c + x^d + 1$, то примитивен и $x^a + x^{a-d} + x^{a-c} + x^{a-b} + 1$

Быстрее всего программно реализуются примитивные трехчлены, так как для генерации нового бита тужно выполнять XOR только двух битов сдвигового регистра. Действительно, все многочлены обратной связи, приведенные в 14-й, являются **разреженными**, то есть, у них немного коэффициентов. Разреженность всегда представляет собой источник слабости, которой иногда достаточно для вскрытия алгоритма . Для криптографических алгоритмов гораздо лучше использовать **плотные** примитивные многочлены, те, у которых много коэффициентов. Применяя плотные многочлены, особенно в качестве части ключа, можно использовать значительно более короткие LFSR.

Генерировать плотные примитивные многочлены по модулю 2 нелегко . В общем случае для генерации примитивных многочленов степени k нужно знать разложение на множители числа 2^k-1. Примитивные многочлены можно найти в следующих трех хороших работах: [652, 1285, 1287].

Сами по себе LFSR являются хорошими генераторами псевдослучайных последовательностей, но они обладают некоторыми нежелательными неслучайными свойствами . Последовательные биты линейны, что делает их бесполезными для шифрования. Для LFSR длины n внутреннее состояние представляет собой предыдущие n выходных битов генератора. Даже если схема обратной связи хранится в секрете, она может быть определена по $2n$ выходным битам генератора с помощью высоко эффективного алгоритма Berlekamp-Massey [1082,1083]: см. раздел 16.3.

Кроме того, большие случайные числа, генерируемые с использованием идущих подряд битов этой последовательности, сильно коррелированны и для некоторых типов приложений вовсе не являются случайными . Несмотря на это LFSR часто используются для создания алгоритмов шифрования .

Программная реализация LFSR

Программные реализации LFSR медленны и быстрее работают, если они написаны на ассемблере, а не на C. Одним из решений является использование параллельно 16 LFSR (или 32, в зависимости от длины слова вашего компьютера). В этой схеме используется массив слов, размер которого равен длине LFSR, а каждый бит слова массива относится к своему LFSR. При условии, что используются одинаковые многочлены обратной связи , это может дать заметный выигрыш производительности . Вообще, лучшим способом обновлять сдвиговые регистры является умножение текущего состояния на подходящие двоичные матрицы [901].

Схему обратной связи LFSRможно модифицировать. Получающийся генератор не будет криптографически более надежным, но он все еще будет обладать максимальным периодом, и его легче реализовать программно [1272]. Вместо использования для генерации нового крайнего левого бита битов отводной последовательности выполняется XOR каждого бита отводной последовательности с выходом генератора и замена его результатом этого действия, затем результат генератора становится новым крайним левым битом (см. 11th). Иногда эту модификацию называют **конфигурацией Галуа**. На языке C это выглядит следующим образом:

```
#define mask 0x80000057

static unsigned long ShiftRegister=1;
void seed_LFSR (unsigned long seed)
{
     if (seed == 0) /* во избежание беды */
             seed = 1 ;
     ShiftRegister = seed;
}

int modified_LFSR (void)
```

```
{
    if (ShiftRegister & 0x00000001) {
           ShiftRegister = (ShiftRegister ^ mask >> 1) | 0x8000000 ;
    return 1;
    } else {
           ShiftRegister >>= 1;
           return 0;
    }
}
```

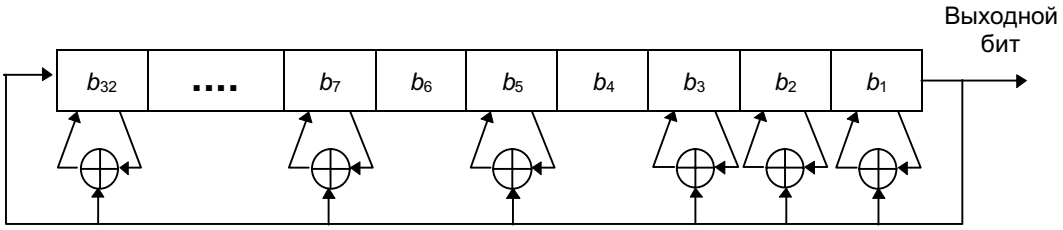


Рис. 16-5. LFSR Галуа.

Выигрыш состоит в том, что все XOR можно сделать за одну операцию. Эта схема также может быт распараллелена, а полиномы различных обратных связей могут быть различны. Такая конфигурация Галуа может дать выигрыш и при аппаратной реализации, особенно в виде СБИС. Вообще, при использовании аппаратуры, которая хорошо выполняет сдвиги применяйте конфигурацию Фиббоначи, если есть возможность использовать параллелизм, применяйте конфигурацию Галуа.

16.3 Проектирование и анализ потоковых шифров

Большинство реальных потоковых шифров основаны на LFSR. Даже в первые дни электроники построить их было несложно. Сдвиговый регистр не представляет из себя ничего большего, чем массив битов, а последовательность обратной связи - набор вентилей XOR. Даже при использовании СБИС потоковый шифр на базе LFSR обеспечивает немалую безопасность с помощью нескольких логических вентилей.

Проблема LFSR состоит в том, что их программная реализация очень неэффективна. Вам приходится избегать разреженных многочленов обратной связи - они облегчают корреляционные вскрытия [1051, 1090, 350] - а плотные многочлены обратной связи неэффективны. Выход любого потокового шифра является побитовым, для шифрования того, что можно выполнить за одну итерацию DES, необходимо выполнить 64 итерации потокового алгоритма. Действительно, программная реализация простого алгоритма LFSR, подобного описываемому ниже сжимающему генератору, не быстрее, чем DES.

Эта отрасль криптографии быстро развивается и very politically charged. Большинство разработок засекречены - множество используемых сегодня военных систем шифрования основаны на LFSR. Действительно, у большинства компьютеров Cray (Cray 1, Cray X-MP, Cray Y-MP) есть весьма любопытная инструкция, обычно называемая как "счетчик совокупности" (population count). Она подсчитывает количество единиц в регистре и может быть использована как для эффективного вычисления расстояния Хэмминга между двумя двоичными словами и для реализации векторизированной версии LFSR. Я слышал, что эта инструкция считается канонической инструкцией NSA, обязательно фигурирующей почти во всех контрактах, касающихся компьютеров.

С другой сторон было взломано удивительно большое число казавшихся сложными генераторов на базе сдвиговых регистров. И, конечно же, число таких генераторов, взломанных военными криптоаналитическими учреждениями, такими как NSA, еще больше. Иногда удивляешься тому, что самые простые из них предлагаются снова и снова.

Линейная сложность

Анализировать потоковые шифры часто проще, чем блочные. Например, важным параметром, используемым для анализа генераторов на базе LFSR, является **линейная сложность** (linear complexity), или линейный интервал. Она определяется как длина n самого короткого LFSR, который может имитировать выход генератора. Любая последовательность, генерированная конечным автоматом над конечным полем, имеет конечную линейную сложность [1006]. Линейная сложность важна, потому что с помощью простого алгоритма, называемого **алгоритмом Berlekamp-Massey**, можно определить этот LFSR, проверив только $2n$ битов потока ключей [1005]. Воссоздавая нужный LFSR, вы взламываете потоковый шифр.

Эта идея можно расширить с полей на кольца [1298] и на случаи, когда выходная последовательность рассматривается как числа в поле нечетной характеристики [842]. Дальнейшее расширение приводит к вводу понятия профиля линейной сложности, который определяет линейную сложность последовательности по мере ее удлинения [1357, 1168, 411, 1582]. Другой алгоритм вычисления линейной сложности прост только в очень сп е-

цифических условиях [597, 595, 596, 1333]. Обобщение понятия линейной сложности выполнено в [776]. Существую также понятия сферической и квадратичной сложности [844].

В любом случае помните, что высокая линейная сложность не обязательно гарантирует безопасность генератора, но низкая линейная сложность указывает на недостаточную безопасность генератора [1357, 12.49].

Корреляционная независимость

Криптографы пытаются получить высокую линейную сложность, нелинейно объединяя результаты некоторых выходных последовательностей. При этом опасность состоит в том, что одна или несколько внутренних выходных последовательностей - часто просто выходы отдельных LFSR - могут быть связаны общим ключевым потоком и вскрыты при помощи линейной алгебры. Часто такое вскрытие называют **корреляционным вскрытием** или вскрытием разделяй-и-властвуй. Томас Сигенталер (Thomas Siegenthaler) показал, что можно точно определить **корреляционную независимость**, и что существует компромисс между корреляционной независимостью и линейной сложностью [1450].

Основной идеей корреляционного вскрытия является обнаружение некоторой корреляции между выходом генератора и выходом одной из его составных частей. Тогда, наблюдая выходную последовательность, можно получить информацию об этом промежуточном выходе. Используя эту информацию и другие корреляции, можно собирать данные о других промежуточных выходах до тех пор, пока генератор не будет взломан.

Против многих генераторов потоков ключей на базе LFSR успешно использовались корреляционные вскрытия и их вариации, такие как быстрые корреляционные вскрытия, предлагающие компромисс между вычислительной сложностью и эффективностью [1451, 278, 1452, 572, 1636, 1051, 1090, 350, 633, 1054, 1089, 995]. Ряд интересных новых идей в этой области можно найти в [46, 1641].

Другие вскрытия

Существуют и другие способы вскрытия генераторов потоков ключей. Тест на **линейную корректность** (linear consistency) пытается найти некоторое подмножество ключа шифрования с помощью матричной техники [1638]. Существует и **вскрытие корректности "встречей посередине"** (meet-in-the-middle consistency attack) [39, 41]. **Алгоритм линейного синдрома** (linear syndrome algorithm) основан на возможности записать фрагмент выходной последовательности в виде линейного уравнения [1636, 1637]. Существует **вскрытие лучшим аффинным приближением** (best afflne approximation attack) [502] и **вскрытие выведенным предложением** (derived sequence attack) [42]. К потоковым шифрам можно применить также методы дифференциального [501] и линейного [631] криптоанализа.

16.4 Потоковые шифры на базе LFSR

Основной подход при проектировании генератора потока ключей на базе LFSR прост. Сначала берется один или несколько LFSR, обычно с различными длинами и различными многочленами обратной связи. (Если длины взаимно просты, а все многочлены обратной связи примитивны, то у образованного генератора будет максимальная длина.) Ключ является начальным состоянием регистров LFSR. Каждый раз, когда необходим новый бит, сдвиньте на бит регистры LFSR (это иногда называют **тактированием** (clocking)). Бит выхода представляет собой функцию, желательно нелинейную, некоторых битов регистров LFSR. Эта функция называется **комбинирующей функцией**, а генератор в целом - **комбинационным генератором**. (Если бит выхода является функцией единственного LFSR, то генератор называется **фильтрующим генератором**.) Большая часть теории подобного рода устройств разработана Селмером (Selmer) и Нилом Цирлером (Neal Zierler) [1647].

Можно ввести ряд усложнений. В некоторых генераторах для различных LFSR используется различная тактовая частота, иногда частота одного генератора зависит от выхода другого. Все это электронные версии идей шифровальных машин, появившихся до Второй мировой войны, которые называются генераторами с **управлением тактовой частотой** (clock-controlled genelators) [641]. Управление тактовой частотой может быть с прямой связью, когда выход одного LFSR управляет тактовой частотой другого LFSR, или с обратной связью, когда выход одного LFSR управляет его собственной тактовой частотой.

Хотя все эти генераторы чувствительны, по крайней мере теоретически, к вскрытиям вложением и вероятной корреляцией [634, 632], многие из них безопасны до сих пор. Дополнительную теорию сдвиговых регистров с управляемой тактовой частотой можно найти в [89].

Ян Касселлс (Ian Cassells), ранее возглавлявший кафедру чистой математики в Кембридже и работавший криптоаналитиком в Bletchly Park, сказал, что "криптография - это смесь математики и путаницы, и без путаницы математика может быть использована против вас." Он имел в виду, что в потоковых шифрах для обеспечения максимальной длины и других свойств необходимы определенные математические структуры, такие как LFSR, но, чтобы помешать кому-то получить содержание регистра и вскрыть алгоритм, необходимо внести некоторый сложный нелинейный беспорядок. Этот совет справедлив и для блочных алгоритмов.

Поэтому не стоит серьезно увлекаться генераторами потока ключей на базе LFSR, описания которых появились в литературе. Я не знаю, используется ли хоть один из них в реальных криптографических продуктах. Большей частью они представляют лишь теоретический интерес. Некоторые были взломаны, некоторые могли остаться безопасными.

Так как шифры на базе LFSR обычно реализуются аппаратно, на рисунках используются символы электронной логики. В тексте, $\oplus$ означает XOR, $\wedge$ - AND, $\vee$ - OR, и $\neg$ - NOT.

Генератор Геффа

В этом генераторе потока ключей используются три LFSR, объединенные нелинейным образом (см. 10th) [606]. Два LFSR являются входами мультиплексора, а третий LFSR управляет выходом мультиплексора. Если a_1, a_2 и a_3 - выходы трех LFSR, выход генератора Геффа (Geffe) можно описать как:

$b = (a_1 \wedge a_2) \oplus((\neg a_1) \wedge a_3)$

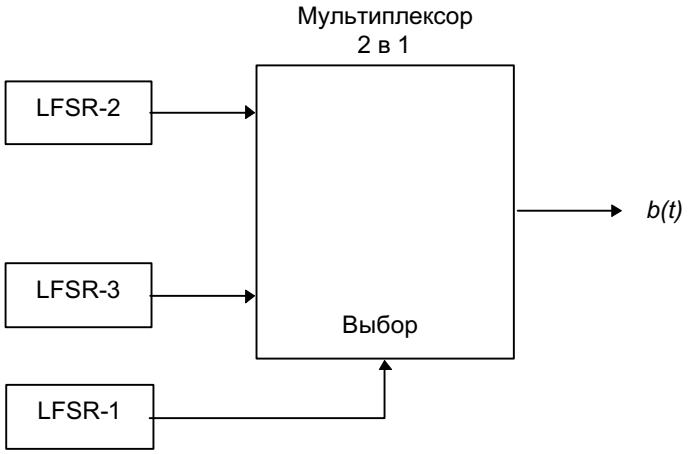


Рис. 16-6. Генератор Геффа.

Если длины LFSR равны n_1, n_2 и n_3, соответственно, то линейная сложность генератора равна

$(n_1 + 1)\, n_2 + n_1 n_3$,

Период генератора равен наименьшему общему делителю периодов трех генераторов. При условии, что степени трех примитивных многочленов обратной связи взаимно просты, период этого генератора будет равен произведению периодов трех LFSR.

Хотя этот генератор неплохо выглядит на бумаге, он криптографически слаб и не может устоять против корреляционного вскрытия [829, 1638]. В 75 процентах времени выход генератора равен выходу LFSR-2. Поэтому, если известны отводные последовательности обратной связи, можно догадаться о начальном значении LFSR-2 и сгенерировать выходную последовательность этого регистра. Тогда можно подсчитать, сколько раз выход LFSR совпадает с выходом генератора. Если начальное значение определено неверно, две последовательности будут согласовываться в 50 процентах времени, а если правильно, то в 75 процентах времени.

Аналогично, выход генератора равен выходу LFSR в 75 процентах времени. С такими корреляциями генератор потока ключей может быть легко взломан. Например, если примитивные многочлены состоят только из трех членов, и длина самого большого LFSR равна n, для восстановления внутренних состояний всех трех LFSR нужен фрагмент выходной последовательности длиной $37n$ битов [1639].

Обобщенный генератор Геффа

Вместо выбора между двумя LFSR в этой схеме выбирается один из k LFSR, где k является степенью 2. Всего используется $k + 1$ LFSR (см. 9th). Тактовая частота LFSR-l должна быть в $\log_2 k$ раз выше, чем у остальных k LFSR.

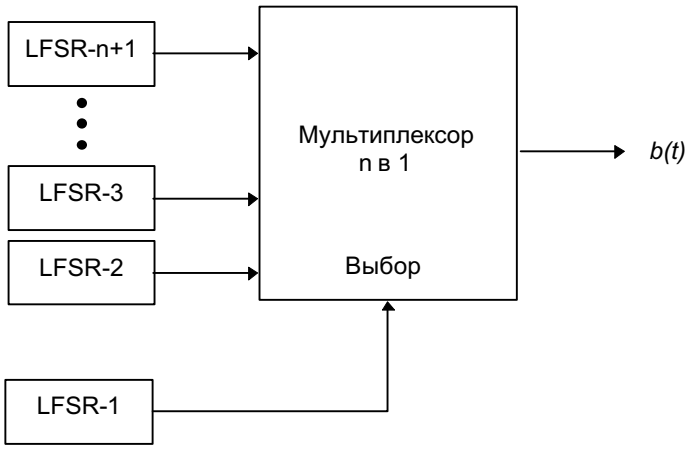


Рис. 16-7. Обобщенный генератор Геффа.

Несмотря на то, что эта схема сложнее генератора Геффа, для взлома можно использовать то же корреляционное вскрытие. Не рекомендую этот генератор.

Генератор Дженнингса

В этой схеме мультиплексор используется для объединения двух LFSR [778, 779, 780]. Мультиплексор, управляемый LFSR-1, выбирает 1 бит LFSR-2 в качестве очередного выходного бита. Кроме того, используется функция, которая отображает выход LFSR-2 на вход мультиплексора (см. 8th).

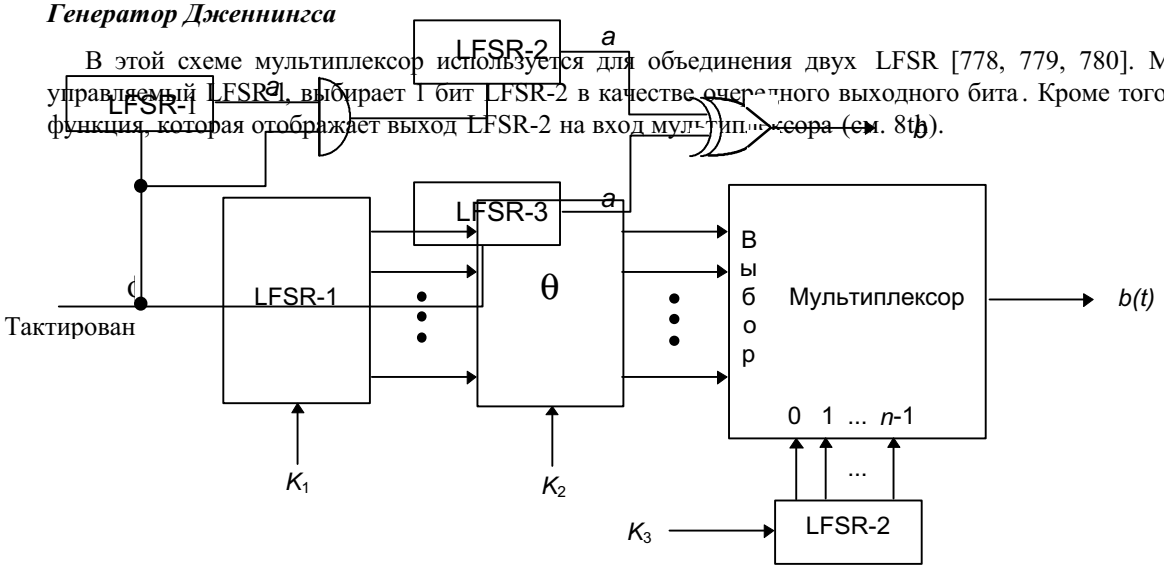


Рис. 16-8. Генератор Дженнингса.

Ключом является начальное состояние двух LFSR и функции отображения. Хотя у этого генератора замечательные статистические свойства, он пал перед выполненным Россом Андерсоном (Ross Anderson) вскрытием корректности встречей посередине [39] и вскрытием линейной корректности [1638,442]. Не используйте этот генератор.

Генератор "стоп-пошел" (Stop-and-Go) Both-Piper

Этот генератор, показанный на 7th, использует выход одного LFSR для управления тактовой частотой другого LFSR [151]. Тактовый вход LFSR-2 управляется выходом LFSR-l, так что LFSR-2 может изменять свое состояние в момент времени *t* только, если выход LFSR-l в момент времени *t* - 1 был равен 1.

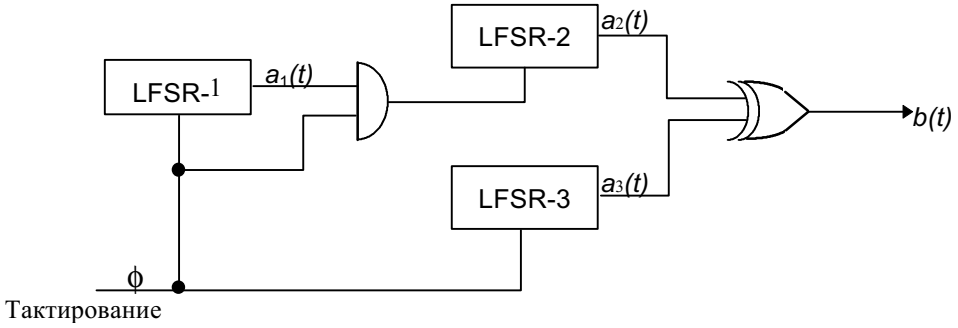


Рис. 16-9. Генератор "стоп-пошел" Beth-Piper.

Никому не удалось привести для общего случая достоверные данные о линейной сложности этого генератора. Однако он не устоял перед корреляционным вскрытием [1639].

Чередующийся генератор "стоп-пошел"

В этом генераторе используются три LFSR различной длины. LFSR-2 тактируется, когда выход LFSR-l равен 1, LFSR-3 тактируется, когда выход LFSR-l равен 0. Выходом генератора является XOR LFSR-2 и LFSR-3 (см. Рис. 16.10) [673].

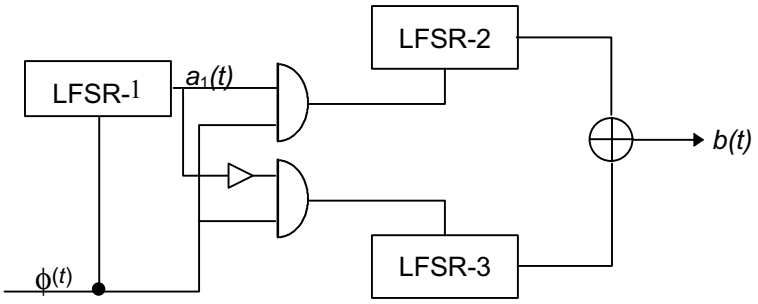


Рис. 16-10. Чередующийся генератор "стоп-пошел"

У этого генератора большой период и большая линейная сложность. Авторы показали способ корреляционного вскрытия LFSR-1, но это не сильно ослабляет генератор. Были предложены и другие генераторы такого типа [1534, 1574, 1477].

Двусторонний генератор "стоп-пошел"

В этом генераторе используется два LFSR с одинаковой длиной n (см. Рис. 16.11) [1638]. Выходом генератора является XOR выходов каждого LFSR. Если выход LFSR-l в момент времени t-1 равен 0, а в момент времени t-2 - 1, то LFSR-2 не тактируется в момент времени t. Наоборот, если выход LFSR-2 в момент времени t-1 равен 0, а в момент времени t-2 - 1, и если LFSR-2 тактируется в момент времени t, то LFSR-l не тактируется в момент времени t.

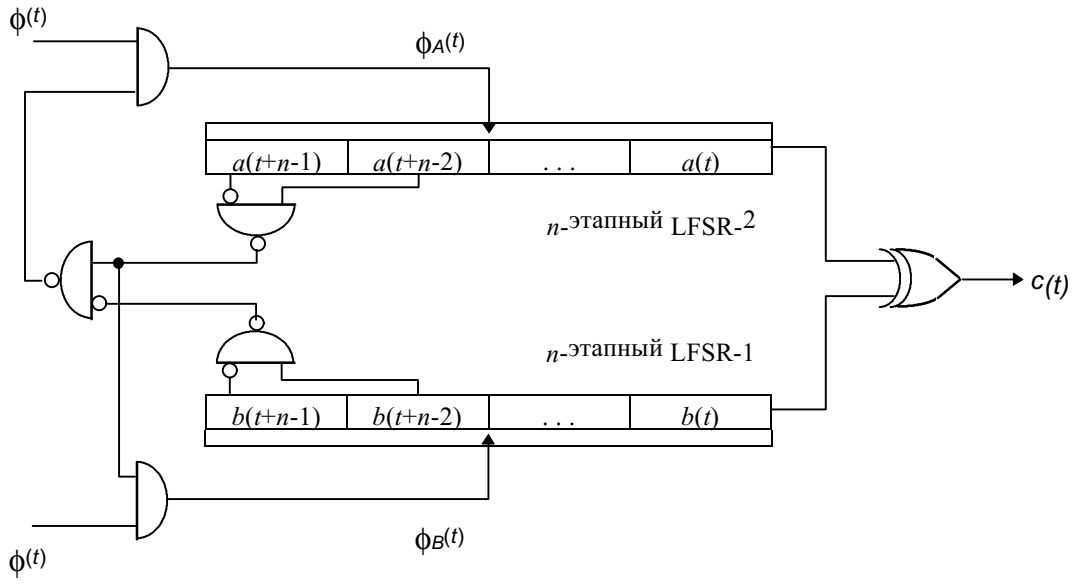


Рис. 16-11. Двусторонний генератор "стоп-пошел".

Линейная сложность такой системы примерно равна ее периоду. Согласно [1638], "в такой системе не очевидная избыточность ключа не наблюдается".

Пороговый генератор

Этот генератор пытается обойти проблемы безопасности, характерные для предыдущих генераторов, с помощью переменного числа LFSR [277]. По теории при использовании большего количества LFSR вскрыть шифр сложнее.

Этот генератор показан на 4-й. Возьмите выход большого числа LFSR (используя нечетное число регистров). Для получения максимального периода убедитесь, что длины всех LFSR взаимно просты, а многочлены обратной связи - примитивны.. Если более половины выходных битов LFSR - 1, то выходом генератора является 1. Если более половины выходных битов LFSR - 0, то выходом генератора является 0.

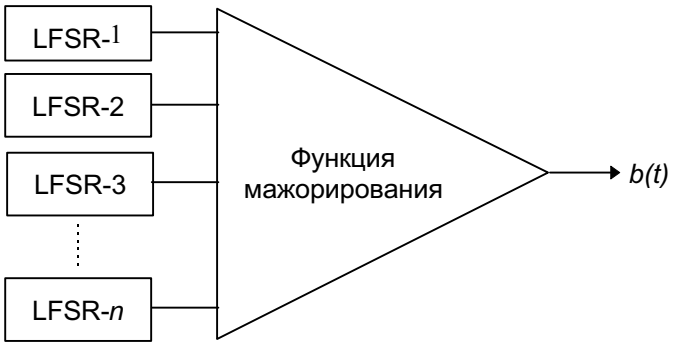


Рис. 16-12. Пороговый генератор.

Для трех LFSR выход генератора можно представить как:

$b = (a_1 \wedge a_2) \oplus (a_1 \wedge a_3) \oplus (a_2 \wedge a_3)$

Это очень похоже на генератор Геффа за исключением того, что пороговый генератор обладает большей линейной сложностью

$n_1n_2 + n_1n_3 + n_2n_3$

где n_1, n_2 и n_3 - длины первого, второго и третьего LFSR.

Этот генератор не слишком хорош. Каждый выходной бит дает некоторую информацию о состоянии LFSR - точнее 0.189 бита - и генератор в целом не может устоять перед корреляционным вскрытием. Я не советую использовать такой генератор.

Самопрореживающие (Self-Decimated) генераторы

Самопрореживающими называются генераторы, которые управляют собственной тактовой частотой . Было предложено два типа таких генераторов, один Рэйнером Рюппелом (Ranier Rueppel) (см. 3-й) [1359] другой Биллом Чамберсом (Bill Chambers) и Дитером Коллманом (Dieter Collmann) [308] (см. 2nd). В генераторе Рюппела если выход LFSR равен 0, LFSR тактируется *d* раз. Если выход LFSR равен 0, LFSR тактируется *k* раз. Генератор Чамберса и Коллмана сложнее, но идея остается той же . К сожалению оба генератора не безопасны [1639], хотя был предложен ряд модификаций, которые могут исправить встречающиеся проблемы [1362.].

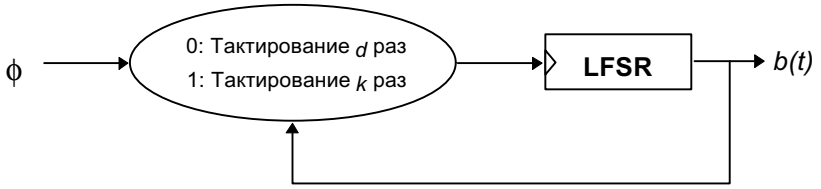


Рис. 16-13. Самопрореживающий генератор Рюппела.

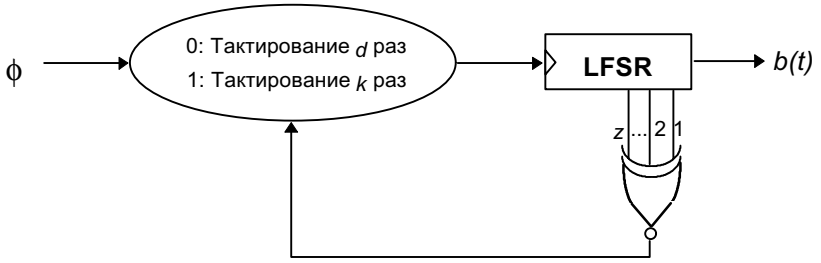


Рис. 16-14. Самопрореживающий генератор Чамберса и Голлмана.

Многоскоростной генератор с внутренним произведением (inner-product)

Этот генератор, предложенный Массеем (Massey) и Рюппелом [1014], использует два LFSR с разными тактовыми частотами (см. 1st). Тактовая частота LFSR-2 в *d* раз больше, чем у LFSR-l. Отдельные биты этих LFSR объединяются операцией AND, а затем для получения выходного бита генератора они объединяются посредс т-вом XOR.

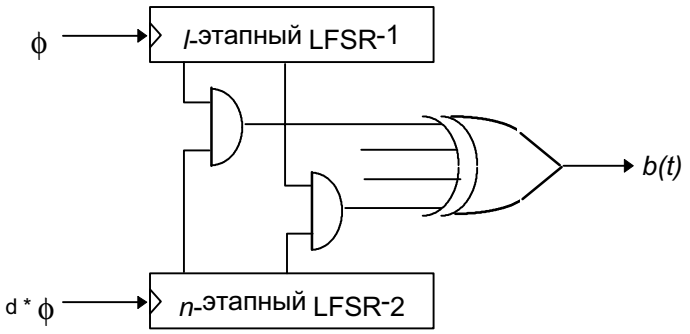


Рис. 16-15. Многоскоростной генератор с внутренним произведением.

Хотя этот генератор обладает высокой линейной сложностью и великолепными статистическими характер истиками, он все же не может устоять перед вскрытием линейной согласованности [1639]. Если n_1 - длина LFSR-l, n_2 - длина LFSR-2, а *d* - отношение тактовых частот, то внутреннее состояние генератора может быть получено по выходной последовательн ости длиной

$n_2 + n_2 + \log_2 d$

Суммирующий генератор

Еще одно предложение Рэйнер Рюппела , этот генератор суммирует выходы двух LFSR (с переносом) [1358, 1357]. Это в высокой степени нелинейная операция . В конце 80-х этот генератор был лидером в отношении безопасности, но он пал перед корреляционным вскрытием [1053, 1054, 1091]. Кроме того, было показано, что этот генератор является частным случаем обратной связи, использующей сдвиговый регистр с переносом (см.

раздел 17.4), и может быть взломан [844].

DNRSG

Это означает "динамический генератор случайной последовательности" ("dynamic random-sequence generator") [1117]. Идея состоит в том, чтобы взять два различных фильтруемых генератора - пороговых, суммирующих, и т.п. - использующих один набор LFSR, а управляемых другим LFSR.

Сначала тактируются все LFSR. Если выходом LFSR-0 является 1, то вычисляется выход первого фильтрующего генератора. Если выходом LFSR-0 является 0, то вычисляется выход второго фильтрующего генератора. Окончательным результатом является XOR выходов первого и второго генераторов.

Каскад Голлманна

Каскад Голлманна (см. 0-й), описанный в [636, 309], представляет собой усиленную версию генератора "стоп-пошел". Он состоит из последовательности LFSR, тактирование каждого из которых управляется предыдущим LFSR. Если выходом LFSR-l в момент времени *t* является 1, то тактируется LFSR-2. Если выходом LFSR-2 в момент времени *t* является 1, то тактируется LFSR-3, и так далее. Выход последнего LFSR и является выходом генератора. Если длина всех LFSR одинакова и равна *n*, линейная сложность системы из *k* LFSR равна

$n(2^n - 1)^{k-1}$

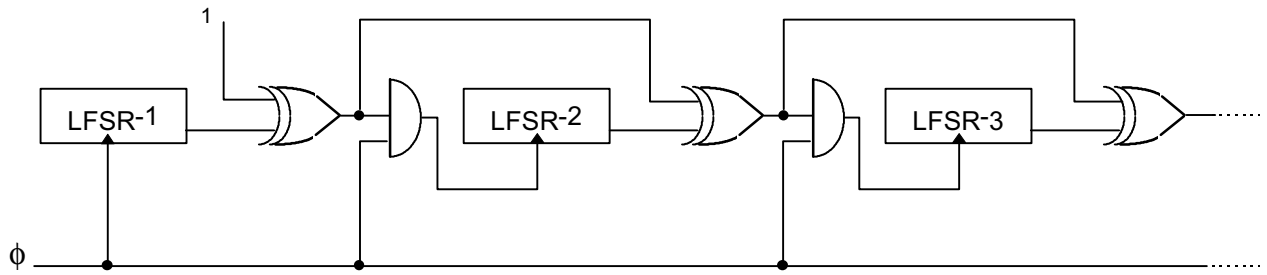


Рис. 16-16. Каскад Голлманна.

Это дерзкая идея: концептуально они очень просты и могут быть использованы для генерации последовательностей с огромными периодами, огромными линейными сложностями и хорошими статистическими свойствами. Они чувствительны к вскрытию, называемому **запиранием** (lock-in) [640] и представляющему метод, с помощью которого сначала криптоаналитик восстанавливает вход последнего сдвигового регистра в каскаде, а затем взламывает весь каскад, регистр за регистром. В некоторых случаях это представляет собой серьезную проблему и уменьшает эффективную длину ключа алгоритма, но для минимизации возможности такого вскрытия можно предпринять ряд определенных мер.

Дальнейший анализ показал, что с ростом *k* последовательность приближается к случайной [637, 638, 642, 639]. На основании недавних вскрытий коротких каскадов Голлманна [1063], я советую использовать *k* не меньше 15. Лучше использовать больше коротких LFSR, чем меньше длинных LFSR.

Прореживаемый генератор

Прореживаемый (shrinking) генератор [378] использует другую форму управления тактированием. Возьмем два LFSR: LFSR-l и LFSR -2. Подадим тактовый импульс на оба регистра. Если выходом LFSR-l является 1, то выходом генератора является выход LFSR-2. Если выход LFSR-l равен 0, оба бита сбрасываются, LFSR тактируются заново и все повторяется.

Идея проста, достаточно эффективна и кажется безопасной. Если многочлены обратной связи прорежены, генератор чувствителен к вскрытию, но других проблем обнаружено не было. Хотя этот тип генератора достаточно нов. Одна из проблем реализации состоит в том, что скорость выдачи результата не постоянна, если LFSR-l генерирует длинную последовательность нулей, то на выходе генератора ничего нет. Для решения этой проблемы авторы предлагают использовать буферизацию [378]. Практическая реализация прореживаемого генератора рассматривается в [901].

Самопрореживаемый генератор

Самопрореживаемый (self-shrinking) генератор [1050] является вариантом прореживаемого генератора. Вместо двух LFSR используется пара битов одного LFSR. Протактируйте LFSR дважды. Если первым битом пары будет 1, то второй бит будет выходом генератора. Если первый бит - 0, сбросьте оба бита и попробуйте снова. Хотя для самопрореживаемого генератора нужно примерно в два раза меньше памяти, чем для прореживаемо-

го, он работает в два раза медленнее.

Хотя самопрореживаемый генератор также кажется безопасным, он может вести себя непредсказуемым образом и обладать неизвестными свойствами. Это очень новый генератор, дайте ему немного времени.

16.5 A5

A5 - это потоковый шифр, используемый для шифрования GSM (Group Special Mobile). Это европейский стандарт для цифровых сотовых мобильных телефонов. Он используется для шифрования канала "телефон-базовая станция". Оставшаяся часть канала не шифруется, телефонная компания может легко сделать что-нибудь с вашими разговорами.

Вокруг этого протокола ведутся странные политические игры. Первоначально предполагалось, что криптография GSM позволит запретить экспорт телефонов в некоторые страны. Теперь ряд чиновников обсуждает, не повредит ли A5 экспортным продажам несмотря на то, что он так слаб, что вряд ли сможет служить препятствием. По слухам в середине 80-х различные секретные службы НАТО сцепились по вопросу, должно ли шифрование GSM быть сильным или слабым. Немцам была нужна сильная криптография, так как рядом с ними находился Советский Союз. Взяла верх другая точка зрения, и A5 представляет собой французскую разработку.

Большинство деталей нам известно. Британская телефонная компания передала всю документацию Брэдфордскому университету (Bradford University), не заставив подписать соглашение о неразглашении. Информация где-то просочилась и наконец была опубликована в Internet. A5 описывается в [1622], также в конце этой книги приведен код этого протокола.

A5 состоит из трех LFSR длиной 19, 22 и 23, все многочлены обратной связи - прорежены. Выходом является XOR трех LFSR. В A5 используется изменяемое управление тактированием. Каждый регистр тактируется в зависимости от своего среднего бита, затем выполняется XOR с обратной пороговой функцией средних битов всех трех регистров. Обычно на каждом этапе тактируется два LFSR.

Существует тривиальное вскрытие, требующее 2^{40} шифрований: предположите содержание первых двух LFSR и попытайтесь определить третий LFSR по потоку ключей. (Действительно ли такой способ вскрытия возможен, остается под вопросом, который скоро будет разрешен разрабатываемой машиной для аппаратного поиска ключей [45].)

Тем не менее, становится ясно, что идеи, лежащие в основе A5, неплохи. Алгоритм очень эффективен. Он удовлетворяет всем известным статистическим тестам, единственной его слабостью является то, что его регистры слишком коротки, чтобы предотвратить поиск ключа перебором. Варианты A5 с более длинными сдвиговыми регистрами и более плотными многочленами обратной связи должны быть безопасны.

16.6 Hughes XPD/KPD

Этот алгоритм был предложен Hughes Aircraft Corp. Эта фирма встроила его в армейские тактические рации и оборудование поиска направления для продажи за границу. Алгоритм был разработан в 1986 году и получил название XPD, сокращение от Exportable Protection Device - Экспортируемое устройство защиты. Позднее он был переименован в KPD - Устройство кинетической защиты - и рассекречен [1037, 1036].

Алгоритм использует 61-битовый LFSR. Существует 2^{10} различных примитивных многочлена обратной связи, одобренных NSA. Ключ выбирает один из этих многочленов (хранящихся где-то в ПЗУ), а также начальное состояние LFSR.

В алгоритме восемь различных нелинейных фильтров, каждый из которых использует шесть отводов LFSR, выдавая один бит. Объединяясь, эти биты образуют байт, который и применяется для шифрования или дешифрования потока данных.

Этот алгоритм выглядит очень привлекательно, но у меня есть определенные сомнения. NSA разрешило его экспорт, следовательно должен быть способ вскрытия порядка, не большего чем 2^{40}. Но какой?

16.7 Nanoteq

Nanoteq - это южноафриканская электронная компания. Именно этот алгоритм используется южноафриканской полицией при шифровании передачи факсов, а возможно и для прочих нужд.

Более или менее этот алгоритм описан в [902, 903]. Он использует 127-битовый LFSR с фиксированным многочленом обратной связи, ключ представляет собой начальное состояние регистра. При помощи 25 элементарных ячеек 127 битов регистра превращаются в один бит потока ключей. У каждой ячейки 5 входов и один выход:

$$f(x_1, x_2, x_3, x_4, x_5) = x_1 + x_2 + (x_1 + x_3)(x_2 + x_4 + x_5) + (x_1 + x_4)(x_2 + x_3) + x_5$$

Каждый выход функции подвергается операции XOR с некоторым битом ключа. Кроме того, существует секретная перестановка, зависящая от конкретной реализации и не описанная в статьях подробно. Этот алгоритм доступен только в аппаратном виде.

Безопасен ли он? Я не уверен. Ряд интересных факсов, передаваемых между полицейскими участками, ин огда появлялся в либеральных газетах. Это вполне могло быть результатом американской, английской или сове тской разведывательной деятельности. Росс Андерсон (Ross Anderson) предпринял ряд первых шагов, крипто анализируя этот алгоритм в [46], я думаю, что скоро появятся новые результаты.

16.8 Rambutan

Rambutan - это английский алгоритм, разработанный Communications Electronics Security Croup (Группа по безопасности электронных коммуникаций, одно из объединений, использованное CCHQ). Он продается только в виде аппаратного модуля и одобрен для защиты документов вплоть до грифа "Конфиденциально". Сам алгоритм засекречен, и микросхема не предназначена для широкой коммерческой продажи.

Rambutan использует 112-битовый ключ (плюс биты четности) и может работать трех режимах: ECB, CBC, и 8-битовый CFB. Это сильный аргумент в пользу того, что этот алгоритм - блочный, но слухи утверждают иное. Предположительно это потоковый шифр с LFSR. У него пять приблизительно 80-битовых сдвиговых р егистров различной длины. Полиномы обратной связи значительно прорежены, в каждом из них всего лишь 10 отводов. Каждый сдвиговый регистр обеспечивает четыре входа для очень большой и сложной нелинейной функции, которая и выдает единственный бит.

Почему Rambutan? Возможно из-за фрукта, который колючий и неприступный снаружи, но мягкий и не жный внутри. Но с другой стороны никакой причины может и не быть.

16.9 Аддитивные генераторы

Аддитивные генераторы (иногда называемые запаздывающими генераторами Фиббоначи) очень эффективны, так как их результатом являются случайные слова, а не случайные биты [863]. Сами по себе они не безопасны, но их можно использовать в качестве составных блоков для безопасных генераторов.

Начальное состояние генератора представляет собой массив n-битовых слов: 8-битовых слов, 16-битовых слов, 32-битовых слов, и т.д.: X_1, X_2, X_3, ..., X_m. Это первоначальное состояние и является ключом. i-ое слово генератора получается как

$$X_i = (X_{i-a} + X_{i-b} + X_{i-c} + + X_{i-m}) \bmod 2^n$$

При правильном выборе коэффициентов a, b, c, . . . , m период этого генератора не меньше 2^n-1. Одним из требований к коэффициентам является то, что младший значащий бит образует LFSR максимальной длины.

Например, (55,24,0) - это примитивный многочлен mod 2 из 14-й. Это означает, что длина следующего адд итивного генератора максимальна.

$$X_i = (X_{i-55} + X_{i-24}) \bmod 2^n$$

Это работает, так как у примитивного многочлена три коэффициента. Если бы их было больше, для получ ения максимальной длины потребовались бы дополнительные условия. Подробности можно найти в [249].

Fish

Fish - это аддитивный генератор, основанный на методах, используемых в прореживаемом генераторе [190]. Он выдает поток 32-битовых слов, которые могут быть использованы (с помощью XOR) с потоком открытого текста для получения шифротекста или с потоком шифротекста для получения открытого текста. Название алгоритма представляет собой сокращение от Fibonacci shrinking generator - прореживаемый генератор Фиббоначи.

Во первых используйте два следующих аддитивных генератора. Ключом является начальные состояния этих генераторов.

$$A_i = (A_{i-55} + A_{i-24}) \bmod 2^{32}$$

$$B_i = (B_{i-52} + B_{i-19}) \bmod 2^{32}$$

Эти последовательности прореживаются попарно в зависимости от младшего значащего бита B_i: если его значение равно 1, то пара используется, если 0 - игнорируется. C_j - это последовательность используемых слов Ai, а D_j - это последовательность используемых слов B_i. Для генерации двух 32-битовых слов-результатов K_{2j} и K_{2j+1} эти слова используются пар ами - C_{2j}, C_{2j+1}, D_{2j}, D_{2j+1}.

$$E_{2j} = C_{2j} \oplus (D_{2j}, \wedge D_{2j+1})$$

$F_{2j} = D_{2j+1} \wedge (E_j, \wedge C_{2j+1})$

$K_{2j} = E_{2j} \oplus F_{2j}$

$K_{2j+1} = C_{2j+1} \oplus F_{2j}$

Этот алгоритм быстр. на процессоре i486/33 реализация Fish на языке C шифрует данные со скоростью 15-Мбит/с. К сожалению он также не безопасен, порядок вскрытия составляет около 2^{40} [45].

Pike

Pike - это обедненная и урезанная версия Fish, предложенная Россом Андерсоном, тем, кто взломал Fish [45]. Он использует три аддитивных генератора. Например:

$A_i = (A_{i-55} + A_{i-24}) \bmod 2^{32}$

$B_i = (B_{i-57} + B_{i-7}) \bmod 2^{32}$

$C_i = (C_{i-58} + C_{i-19}) \bmod 2^{32}$

Для генерации слова потока ключей взгляните на биты переноса при сложении . Если все три одинаковы (все нули или все единицы), то тактируются все три генератора. Если нет, то тактируются только два совпадающих генератора. Сохраните биты переноса для следующего раза. Окончательным выходом является XOR выходов трех генераторов.

Pike быстрее Fish, так как в среднем для получения результата нужно 2.75 действия, а не 3. Он также слишком нов, чтобы ему доверять, но выглядит очень неплохо.

Mush

Mush представляет собой взаимно прореживающий генератор. Его работу объяснить легко [1590]. Возьмем два аддитивных генератора: A и B. Если бит переноса A установлен, тактируется B. Если бит переноса B установлен, тактируется A. Тактируем A и при переполнении устанавливаем бит переноса. Тактируем B и при переполнении устанавливаем бит переноса. Окончательным выходом является XOR выходов A и B. Проще всего использовать те же генераторы, что и в Fish:

$A_i = (A_{i-55} + A_{i-24}) \bmod 2^{32}$

$B_i = (B_{i-52} + B_{i-19}) \bmod 2^{32}$

В среднем для генерации одного выходного слова нужно три итерации генератора . И если коэффициенты аддитивного генератора выбраны правильно и являются взаимно простыми, длина выходной последовательности будет максимальна. Мне неизвестно об успешных вскрытиях, но не забывайте, что этот алгоритм очень нов .

16.10 Gifford

Дэвид Джиффорд (David Gifford) изобрел потоковый шифр и использовал его для шифрования сводок новостей в районе Бостона с 1984 по 1988 год [608, 607, 609]. Алгоритм использует единственный 8-байтовый регистр: $b_0, b_1, \ldots, b_7$. Ключом является начальное состояние регистра. Алгоритм работает в режиме OFB, открытый текст абсолютно не влияет на работу алгоритма . (См. -1-й).

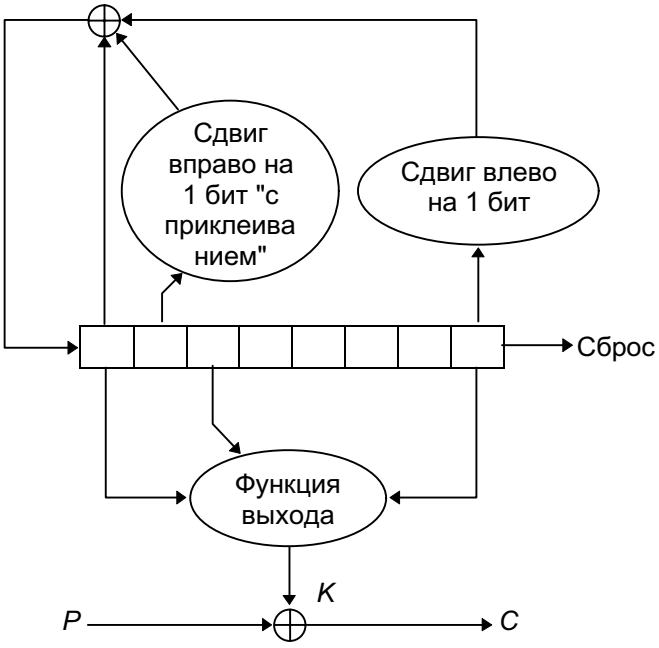


Рис. 16-17. Gifford.

Для генерации байта ключа k_i объединим b_0 и b_1, а также объединим b_4 и b_7. Перемножим полученные числа, получая 32-битовое число. Третьим слева байтом и будет k_i.

Для обновления регистра возьмем b_1 и сдвинем вправо "с приклеиванием" на 1 бит следующим образом: крайний левый бит одновременно и сдвигается, и остается на месте. Возьмем b_7 и сдвинем его на один бит влево, в крайней правой позиции должен появиться 0. Выполним XOR измененного b_1, измененного b_7 и b_0. Сдвинем первоначальный байт регистра на 1 бит вправо и поместим этот байт в крайнюю левую позицию.

В течение всего времени использования этот алгоритм оставался безопасным, но он был взломан в 1994 году [287]. Оказалось, что многочлен обратной связи не был примитивным и, таким образом, мог быть вскрыт.

16.11 Алгоритм M

Это название дано Кнутом [863]. Алгоритм представляет собой способ объединить несколько псевдослучайных потоков, увеличивая их безопасность. Выход одного генератора используется для выбора отстающего выхода другого генератора [996, 1003]. На языке C:

```
#define ARR_SIZE (8192) /* например - чем больше, тем лучше */
static unsigned char delay[ ARRSIZE ] ;
unsigned char prngA( void )
long prngB( void ) ;
void init_algM( void ) {
     long i ;
     for ( i = 0 ; i < ARR_SIZE ; i++ )
            delay[i] = prngA() ;
} /* lnlt_algM */
unsigned char alglM( void ) {
     long j,v ;
     j = prngB() % ARR_SIZE ; /* получить индекс delay[]*/
     v = delay[j] ; /* получить возвращаемое значение */
     delay[j] = prngA() ; /* заменить его */
     return ( v ) ;
     } /* algM */
```

Смысл состоит в том, что если prngA - действительно случайно, невозможно ничего узнать о prngB (и, следовательно, невозможно выполнить криптоанализ). Если prngA имеет такой вид, что его криптоанализ может быть выполнен только, если его выход доступен в свою очередь (т.е., только если сначала был выполнен кри п-

тоанализ prngB), а в противном случае оно по сути действительно случайно, то эта комбинация должна быть безопасной.

16.12 PKZIP

Алгоритм шифрования, встроенный в программу сжатия данных PKZIP, был разработан Роджером Щлафлы (Roger Schlafly). Это потоковый шифр, шифрующий данные побайтно. По крайней мере этот алгоритм используется в версии 2.04g. Я не могу ничего сказать о более поздних версиях, но если не было сделано никаких заявлений об обратном, можно считать с большой вероятностью, что алгоритм не изменился. Алгоритм использует три 32-битовых переменных, инициализированных следующим образом :

K_0 = 305419896

K_1 = 591751049

K_2 = 878082192

Используется 8-битовый ключ K_3, полученный из K_2. Вот этот алгоритм (в стандартной нотации C):

$C_i = P_i \wedge K_3$

K_0= crc32 (K_0, P_i)

$K_1 = K_1 + (K_0 \,\&\, 0x000000ff)$

$K_1 = K_1 * 134775813 + 1$

K_2 = crc32 (K_2, $K_1 >> 24$)

$K_3 = ((K_2 \mid 2) * ((K_2 \mid 2) \wedge 1)) >> 8$

Функция crc32 берет свое предыдущее значение и байт, выполняет их XOR и вычисляет следующее значение с помощью многочлена CRC, определенного 0xedb88320. На практике 256-элементная таблица может быть рассчитана заранее, и вычисление crc32 превращается в:

crc32 (a, b) = ($a >> 8$) ^ table [(a & 0xff) $\oplus$ b]

Таблица рассчитывается в соответствии с первоначальным определением crc32:

table [i] = crc32 (i, 0)

Для шифрования потока открытого текста сначала для обновления ключей зациклим байты ключа в алгоритме шифрования. Полученный шифротекст на этом этапе игнорируется. Затем побайтно зашифруем открытый текст. Открытому тексту предшествуют двенадцать случайных байтов, но это на самом деле неважно. Дешифрирование похоже на шифрование за исключением того, что во втором действии алгоритма вместо P_i используется C_i.

Безопасность PKZIP

К сожалению она не слишком велика. Для вскрытия нужно от 40 до 2000 байтов известного открытого текста, временная сложность вскрытия составит около 2^{27} [166]. На вашем персональном компьютере это можно сделать за несколько часов. Если в сжатом файле используются какие-нибудь стандартные заголовки, получение известного открытого текста не представляет собой проблемы. Не используйте встроенное в PKZIP шифрование.

Глава 17
Другие потоковые шифры и генераторы настоящих случайных последовательностей

17.1 RC4

RC4 - это потоковый шифр с переменным размером ключа, разработанный в 1987 году Роном Ривестом для RSA Data Security, Inc. В течение семи лет он находился в частной собственности, и подробное описание алгоритма предоставлялось только после подписания соглашения о неразглашении.

В сентябре 1994 кто-то анонимно опубликовал исходный код в списке рассылки "Киберпанки" (Cypherpunks). Он быстро распространился в телеконференнции Usenet sci.crypt и через Internet по различным ftp-серверам во всем мире. Обладатели легальных копий RC4 достоверность этого кода. RSA Data Security, Inc. попыталась загнать джинна обратно в бутылку, утверждая, что несмотря на опубликование алгоритм остается торговым секретом, было слишком поздно. С тех пор алгоритм обсуждался и изучался в Usenet, распространялся на конференциях и служил в качестве учебного пособия на курсах по криптографии.

Описывать RC4 просто. Алгоритм работает в режиме OFB: поток ключей не зависит от открытого текста. Используется S-блок размером 8*8: S_0, S_1, . . . , S_{255}. Элементы представляют собой перестановку чисел от 0 до 255, а перестановка является функцией ключа переменной длины. В алгоритме применяются два счетчика, i и j, с нулевыми начальными значениями.

Для генерации случайного байта выполняется следующее:

$i = (i + 1) \bmod 256$

$j = (j + S_i) \bmod 256$

поменять местами S_i и Sj

$t = (S_i + S_j) \bmod 256$

$K = S_t$

Байт K используется в операции XOR с открытым текстом для получения шифротекста или в операции XOR с шифротекстом для получения открытого текста. Шифрование выполняется примерно в 10 раз быстрее, чем DES.

Также несложна и инициализация S-блока. Сначала заполним его линейно: $S_0 = 0$, $S_1 = 1$, . . . , $S_{255} = 255$. Затем заполним ключом другой 256-байтовый массив, при необходимости для заполнения всего массива повторяя ключ: K_0, K_1, . . . , K_{255}. Установим значение индекса j равным 0. Затем:

for $i = 0$ to 255:

$j = (j + S_i + K_i) \bmod 256$

поменять местами S_i и S_j

И это все. RSADSI утверждает, что алгоритм устойчив к дифференциальному и линейному криптоанализу, что, по-видимому, в нем нет никаких коротких циклов, и что он в высокой степени нелинеен. (Опубликованных криптоаналических результатов нет. RC4 может находиться в примерно 2^{1700} ($256! * 256^2$) возможных состояний: невероятное число.) S-блок медленно изменяется при использовании: i обеспечивает изменение каждого элемента, а j - что элементы изменяются случайным образом. Алгоритм настолько несложен, что большинство программистов могут закодировать его просто по памяти.

Эту идею можно обобщить на S-блоки и слова больших размеров. Выше была описана 8-битовая версия RC4. Нет причин, по которым нельзя бы было определить 16-битовый RC4 с 16*16 S-блоком (100 K памяти) и 16-битовым словом. Начальная итерация займет намного больше времени - для сохранения приведенной схемы нужно заполнить 65536-элементный массив - но получившийся алгоритм должен быть быстрее.

RC4 с ключом длиной не более 40 битов обладает специальным экспортным статусом (см. раздел 13.8). Этот специальный статус никак не влияет на безопасность алгоритма, хотя в течение многих лет RSA Data Security, Inc. намекало на обратное. Название алгоритма является торговой маркой, поэтому каждый, кто напишет собственный код, должен назвать его как-то иначе. Различные внутренние документы RSA Data Security, Inc. до сих пор не были опубликованы [1320, 1337].

Итак, какова же ситуация вокруг алгоритма RC4? Он больше не является торговым секретом, поэтому кто угодно имеет возможность воспользоваться им. Однако RSA Data Security, Inc. почти наверняка возбудит дело против каждого, кто применит нелицензированный RC4 в коммерческом продукте. Возможно им и не удастся

выиграть процесс, но почти наверняка для другой компании дешевле купить лицензию, чем судиться .

RC4 входит в десятки коммерческих продуктов, включая Lotus Notes, AOCE компании Apple Computer и and Oracle Secure SQL. Этот алгоритм также является частью спецификации Сотовой цифровой пакетной передачи данных (Cellular Digital Packet Data) [37].

17.2 SEAL

SEAL - это программно эффективный потоковый шифр, разработанный в IBM Филом Рогэвэем (Phil Rogaway) и Доном Копперсмитом (Don Coppersmith) [1340]. Алгоритм оптимизирован для 32-битовых процессоров: Для нормальной работы ему нужно восемь 32-битовых регистров и кэш-память на несколько килобайт . Чтобы избежать влияния использования медленных операций SEAL выполняет ряд предварительных действий с ключом, сохраняя результаты в нескольких таблицах. Эти таблицы используются для ускорения шифрования и дешифрирования.

Семейство псевдо случайных функций

Особенностью SEAL является то, что он в действительности является не традиционным потоковым шифром, а представляет собой **семейство псевдослучайных функций**. При 160-битовом ключе k и 32-битовом n SEAL растягивает n в L-битовую строку $k(n)$. L может принимать любое значение, меньшее 64 Кбайт. SEAL, по видимому, использует следующее свойство: если k выбирается случайным образом, то $k(n)$ должно быть вычислительно неотличимо от случайной L-битовой функции n.

Практический эффект того, что SEAL является семейством псевдослучайных функций, состоит в том, что он удобен в ряде приложений, где неприменимы традиционные потоковые шифры . Используя большинство потоковых шифров, вы создаете однонаправленную последовательность битов: единственным способом определить i-ый бит, зная ключ и позицию i, является генерирование всех битов вплоть до i-ого. Отличие семейства псевдослучайных функций состоит в том, что вы можете легко получить доступ к любой позиции потока ключей . Это очень полезно.

Представим себе, что вам нужно "закрыть" жесткий диск. Вы хотите зашифровать каждый 512-байтовый сектор. Используя семейство псевдослучайных функций, подобное SEAL, содержимое сектора n можно зашифровать, выполнив его XOR с $k(n)$. Это то же самое, как если бы была выполнена операция XOR всего диска с длинной псевдослучайной функцией, и любая часть этой длинной строки может быть независимо вычислена без всяких проблем.

Семейство псевдослучайных функций также упрощает проблему синхронизации, встречающуюся в стандартных потоковых шифрах. Предположим, что вы посылаете шифрованные сообщения по каналу, в котором данные иногда теряются. С помощью семейства псевдослучайных функций можно зашифровать ключом k n-ое передаваемое сообщение, x_n, выполнив XOR x_n and $k(n)$. Получателю не нужно хранить состояние шифра для восстановления x_n, ему не приходится беспокоиться и о потерянных сообщениях, влияющих на процесс дешифрирования.

Описание SEAL

Внутренний цикл SEAL показан на 16th. Алгоритм управляется тремя полученными из ключа таблицами: R, S и T. Предварительная обработка отображает ключ k на эти таблицы с помощью процедуры, основанной на SHA (см. раздел 18.7). 2-килобайтная таблица T представляет собой S-блок 9*32 битов.

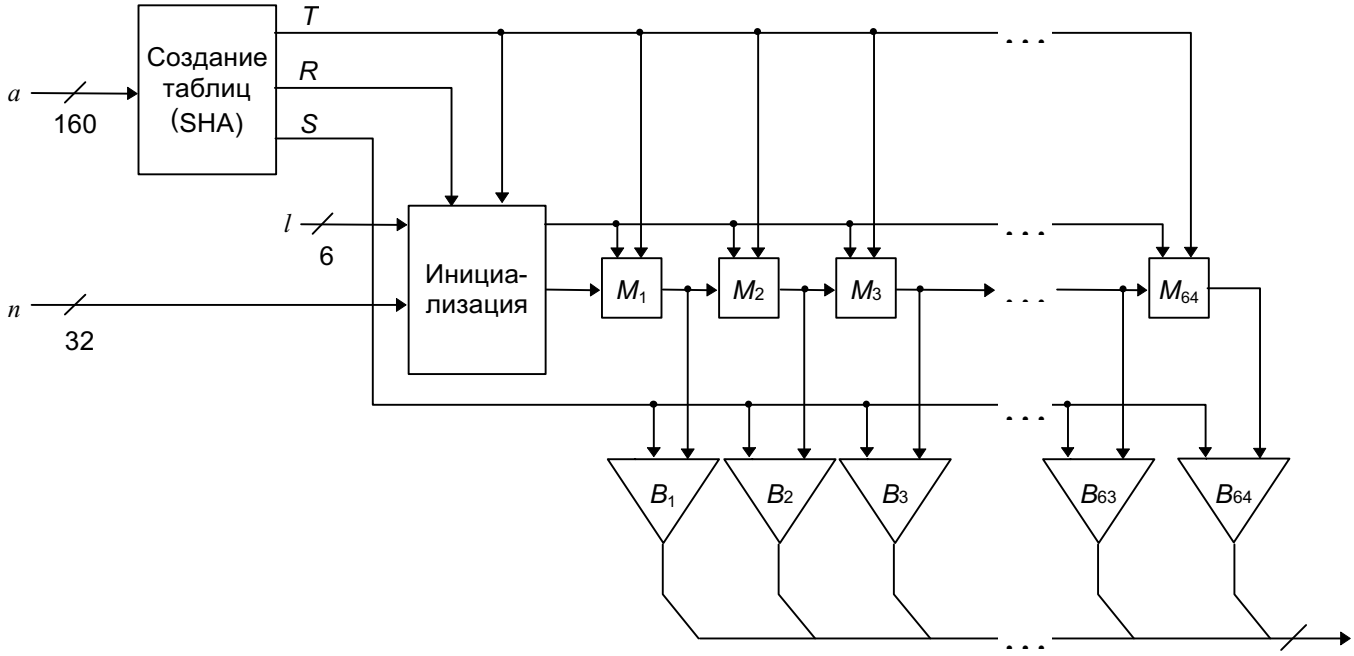


Рис. 17-1. Внутренний цикл SEAL.

SEAL также использует четыре 32-битовых регистра, *A*, *B*, *C* и *D*, начальные значения которых определяются *n* и полученными по *k* таблицами *R* и *T*. Эти регистры изменяются в ходе итераций, каждая из которых состоит из восьми этапов. На каждом этапе 9 битов первого регистра (все равно *A*, *B*, *C* или *D*) используются в качестве индекса таблицы *T*. Затем выбранное из *T* значение складывается со вторым регистром (снова одному из *A*, *B*, *C* или *D*) или объединяется с его содержимым с помощью XOR. Потом первый регистр циклически сдвигается на 9 позиций. На некоторых этапах второй регистр далее модифицируется с помощью сложения или XOR с содержимым первого регистра (уже сдвинутым). После 8 таких этапов *A*, *B*, *C* и *D* добавляются к потоку ключей, при этом каждый из них маскируется сложением или XOR с определенным словом из *S*. Итерация завершается прибавлением к *A* и *C* дополнительных значений, зависящих от *n*, n_1, n_2, n_3, n_4, выбор конкретного значения определяется четностью номера итерации. По видимому, при разработке этой схемы главными были следующие идеи:

1. Использование большого, секретного, получаемого из ключа S-блока (*T*).
2. Чередующиеся некоммутируемые арифметические операции (сложение и XOR).
3. Использование внутреннего состояния, поддерживаемого шифром, которое не проявляется явно в потоке данных (значения *ni*, которые модифицируют *A* и *C* в конце каждой итерации).
4. Изменение функции этапа в соответствии с номером этапа и изменение функции итерации в соответствии с номером итерации.

Для шифрования каждого байта текста SEAL требует около пяти элементарных операций. На 50-мегагерцовом процессоре i486 он работает со скоростью 58 Мбит/с. SEAL возможно является самым быстрым из описанных в этой книге.

С другой стороны SEAL должен выполнить предварительную обработку, заполняя внутренние таблицы. Размер этих таблиц составляет примерно 3 Кбайт, а для их расчета нужно примерно 200 вычислений SHA. Таким образом, SEAL не подходит для тех случаев, когда не хватает времени для обработки ключа или памяти для хранения таблиц.

Безопасность SEAL

SEAL достаточно новый алгоритм, ему еще предстоит пройти через горнило открытого криптоанализа. Это вызывает определенную настороженность. Однако SEAL кажется хорошо продуманным алгоритмом. Его особенности, в конечном счете, наполнены смыслом. К тому же Дон Копперсмит считается лучшим криптоаналитиком в мире.

Патенты и лицензии

SEAL запатентован [380]. По поводу лицензирования нужно обращаться к Управляющему по лицензиям IBM (Director of Licenses, IBM Corporation, 500 Columbus Ave., Thurnwood, NY, 10594).

17.3 WAKE

WAKE - сокращение от Word Auto Key Encryption (Автоматическое шифрование слов ключом)- это алгоритм, придуманный Дэвидом Уилером (David Wheeler) [1589]. Он выдает поток 32-битовых слов, которые с помощью XOR могут быть использованы для получения шифротекста из открытого текста или открытого текста из шифротекста. Это быстрый алгоритм.

WAKE работает в режиме CFB, для генерации следующего слова ключа используется предыдущее слово шифротекста. Алгоритм также использует S-блок из 256 32-битовых значений. Этот S-блок обладает одним особым свойством: Старший байт всех элементов представляет собой перестановку всех возможных байтов, а 3 младших байта случайны.

Сначала по ключу сгенерируем элементы S-блока, S_i. Затем проинициализируем четыре регистра с использованием того же или иного ключа: a_0, b_0, c_0 и d_0. Для генерации 32-битового слова потока ключей K_i.

$K_i = d_i$

Слово шифротекста C_i представляет собой XOR слова открытого текста P_i с K_i. Затем обновим четыре регистра:

$a_{i+1} = M(a_i,d_i)$

$b_{i+1} = M(b_i,a_{i+1})$

$c_{i+1} = M(c_i,b_{i+1})$

$d_{i+1} = M(d_i,c_{i+1})$

Функция M представляет собой

$M(x,y) = (x + y) >> 8 \oplus S_{(x+y)\text{^}255}$

Схема алгоритма показана на 15-й. Знак >> обозначает обычный, не циклический сдвиг вправо. Младшие 8 битов $x+y$ являются входом S-блока. Уилер приводит процедуру генерации S-блока, но на самом деле она неполна. Будет работать любой алгоритм генерации случайных байтов и случайной перестановки .

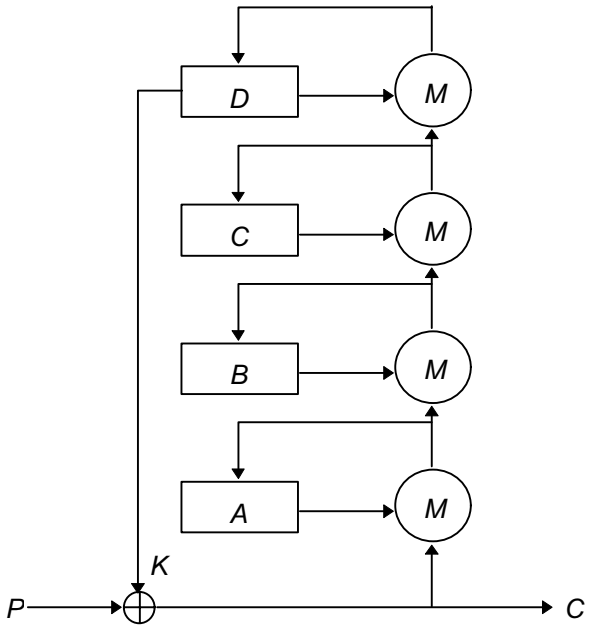


Рис. 17-2. WAKE.

Самым ценным качеством WAKE является его скорость. Однако он чувствителен к вскрытию с выбранным открытым текстом или выбранным шифротекстом. Этот алгоритм использовался в предыдущей версии антивирусной программы д-ра Соломона.

17.4 Сдвиговые регистры с обратной связью по переносу

Сдвиговый регистр с обратной связью по переносу, или FCSR (feedback with carry shift register), похож на LFSR. В обоих есть сдвиговый регистр и функция обратной связи, разница в том, что в FCSR есть также регистр переноса (см. 14-й). Вместо выполнения XOR над всеми битами отводной последовательности эти биты скла-

дываются друг с другом и с содержимым регистра переноса. Результат mod 2 и становится новым битом. Результат, деленный на 2, становится новым содержимым регистра переноса .

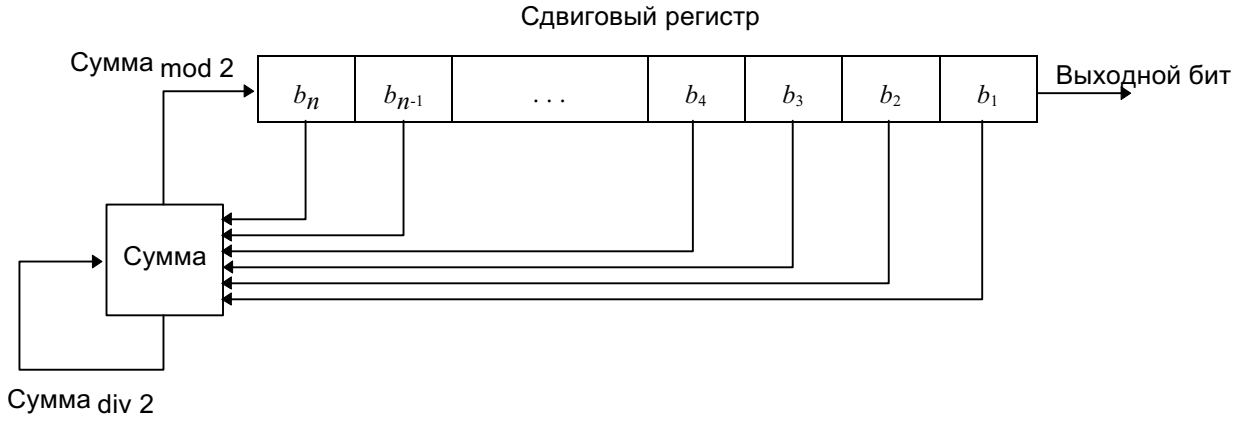


Рис. 17-3. Сдвиговый регистр с обратной связью по переносу.

На 13-й приведен пример 3-битового FCSR с ответвлениями в первой и второй позициях. Пусть его начальное значение 001, а начальное содержимое регистра переноса равно 0. Выходом будет является крайний правый бит сдвигового регистра.

Сдвиговый регистр	Регистр переноса
0 0 1	0
1 0 0	0
0 1 0	0
1 0 1	0
1 1 0	0
1 1 1	0
0 1 1	1
1 0 1	1
0 1 0	1
0 0 1	1
0 0 0	1
1 0 0	0

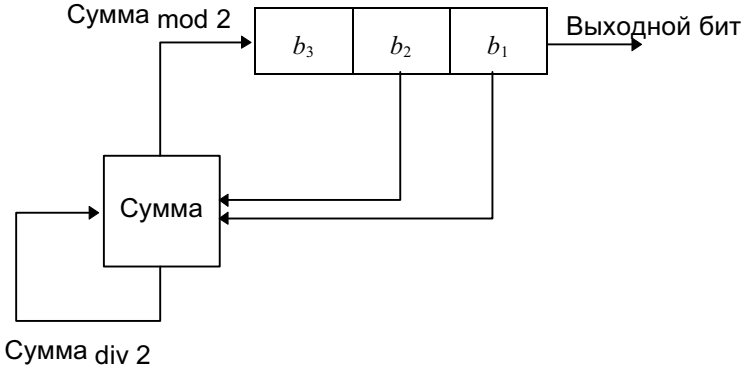


Рис. 17-4. 3-битовый FCSR.

Заметим, что конечное внутреннее состояние (включая содержимое регистра переноса) совпадает со вторым внутренним состоянием. С этого момента последовательность циклически повторяется с периодом, равным 10 .

Стоит упомянуть и еще о нескольких моментах. Во первых, регистр переноса является не битом, а числом. Размер регистра переноса должен быть не меньше $\log_2 t$, где t - это число ответвлений. В предыдущем примере

только три ответвления, поэтому регистр переноса однобитовый. Если бы было четыре ответвления, то регистр переноса состоял бы из двух битов и мог принимать значения 0, 1, 2 или 3.

Во вторых, существует начальная задержка прежде, чем FCSR перейдет в циклический режим. В предыдущем примере никогда не повторяется только одно состояние. Для больших и более сложных FCSR задержка может быть больше.

В третьих, максимальный период FCSR не 2^{n-1}, где n - длина сдвигового регистра. Максимальный период равен q-1, где q - это **целое число связи**. Это число задает ответвления и определяется как:

$q = 2q_1 + 2^2q_2 + 2^3q_3 + \ldots + 2^nq_n$-1

(Да, q_i отсчитываются слева направо.) И даже хуже, q должно быть простым числом, для которого 2 является примитивным корнем. В дальнейшем предполагается, что q удовлетворяет этому условию.

В приведенном примере q = 2*0 + 4*1 + 8*1 - 1 == 11. 11 - это простое число, примитивным корнем которого является 2. Роэтому максимальный период равен 10.

Не все начальные состояния дают максимальный период. Например, рассмотрим FCSR с начальным значением 101 и регистром переноса, установленным в 4.

Сдвиговый регистр	Регистр переноса
1 0 1	4
1 1 0	2
1 1 1	1
1 1 1	1

С этого момента регистр выплевывает бесконечную последовательность единиц.

Любое начальное состояние приводит к одной из четырех ситуаций. Во первых, оно может быть частью максимального периода. Во вторых, оно может перейти в последовательность максимального периода после начальной задержки. В третьих, после начальной задержки оно может породить бесконечную последовательность нулей. В четвертых, после начальной задержки оно может породить бесконечную последовательность единиц.

Для определения, чем закончится конкретное начальное состояние, существует математическая формула, но намного проще проверить это опытным путем. Запустите на некоторое время FCSR. (Если m - это начальный объем памяти, а t - количество ответвлений, то достаточно $\log_2(t) + \log_2(m)$ +1 тактов.) Если выходной поток вырождается в бесконечную последовательность нулей или единиц за n битов, где n - это длина FCSR, не используйте это начальное состояние. В противном случае его можно использовать. Так как начальное состояние FCSR соответствует ключу потокового шифра, это означает, что ряд ключей генератора на базе FCSR будут слабыми.

В 16-й перечислены все целые числа связи, меньшие 10000, для которых 2 является примитивным корнем. Для всех этих чисел максимальный период равен q-1. Чтобы получить по одному из этих чисел последовательность ответвлений, рассчитаем бинарный состав q+1. Например, 9949 дает последовательность ответвлений в позициях 1, 2, 3, 4, 6, 7, 9, 10 и 13, так как

$9950 = 2^{13} + 2^{10} + 2^9 + 2^7 + 2^6 + 2^4 + 2^3 + 2^2 + 2^1$

В 15-й перечислены *все* отводные последовательности из четырех ответвлений, которые дают FCSR максимальной длины для сдвиговых регистров с длиной 32 бита, 64 бита и 128 битов. q, простое число, примитивным корнем которого является 2, получается объединением всех четырех значений, a, b, c и d.

$q = 2^a + 2^b + 2^c + 2^d$ - 1

Для создания FCSR с периодом q - 1 можно использовать любую из этих последовательностей.

Идея использовать в криптографии FCSR все еще является очень новой, впервые она была выдвинута Энди Клаппером (Andy Klapper) и Марком Горески (Mark Goresky) [844, 845, 654, 843, 846]. Также, как анализ LFSR основан на сложении примитивных многочленов mod 2, анализ FCSR основан на сложении неких чисел, называемых **2-adic**. Соответствующая теория выходит далеко за пределы этой книги, но в мире 2-adic чисел существуют аналоги для всего. Точно также, как определяется линейная сложность, можно определить и 2-adic сложность. Существует 2-adic аналог и для алгоритма Berlekamp-Massey. Это означает, что перечень возможных потоковых шифров по крайней мере удвоился. Все, что можно делать с LFSR, можно делать и с FCSR.

Существуют работы, развивающие эту идею и рассматривающие несколько регистров переноса. Анализ этих генераторов последовательностей основан на сложении разветвленных расширений 2-adic чисел [845, 846].

17.5 Потоковые шифры, использующие FCSR

Потоковые шифры на базе FCSR не описаны в литературе, теория все еще слишком нова. Чтобы как-то "погнать зайца дальше" я предложу здесь несколько вариантов. Я охватываю два направления: предлагаю потоковые шифры на базе FCSR, которые совпадают с ранее предложенными генераторами LFSR, а также предлагаю потоковые шифры, использующие FCSR и LFSR одновременно. Безопасность первого варианта возможно может быть проанализирована с помощью 2-adic чисел, генераторы второго варианта не могут быть проанализированы с использованием алгебраических методов - возможно их анализ может быть выполнен только косвенным образом. В любом случае, важно выбирать LFSR и FCSR с взаимно простыми периодами.

Все придет потом. Сейчас мне неизвестно ни о реализации, ни об анализе ни одной из этих идей. Подождите несколько лет и просматривайте литературу, прежде чем вы поверите в одну из этих идей.

Каскадные генераторы

Существует два способа использовать FCSR в каскадных генераторах:

— Каскад FCSR. Каскад Голлманна с FCSR вместо LFSR.

— Каскад LFSR/FCSR. Каскад Голлманна с генераторами, меняющими LFSR на FCSR и наоборот.

Комбинированные генераторы FCSR

Эти генераторы используют переменное количество LFSR и/или FCSR и множество функций, объединяющих регистры. Операция XOR разрушает алгебраические свойства FCSR, поэтому имеет смысл использовать эту операцию для их объединения. Генератор, показанный на 12th, использует переменное число FCSR. Его выходом является XOR выходов отдельных FCSR.

Другими генераторами, являющимися развитием аналогичных линий, являются:

— Генератор четности FCSR. Все регистры - FCSR, а объединяющая функция - XOR.

— Генератор четности LFSR/FCSR. Используется смесь LFSR и FCSR, объединяемых с помощью XOR.

— Пороговый генератор FCSR. Все регистры - FCSR, а объединяющей функцией является мажорирование.

— Пороговый генератор LFSR/FCSR. Используется смесь LFSR и FCSR, объединяемых с помощью мажорирования.

— Суммирующий генератор FCSR. Все регистры - FCSR, а объединяющая функция - сложение с переносом.

— Суммирующий генератор LFSR/FCSR. Используется смесь LFSR и FCSR, объединяемых с помощью сложения с переносом.

Табл. 17-1.
Целые значения связи для FCSR с максимальным периодом

2	211	587	947
5	227	613	1019
11	269	619	1061
13	293	653	1091
19	317	659	1109
29	347	661	1117
37	349	677	1123
53	373	701	1171
59	379	709	1187
61	389	757	1213
67	419	773	1229
83	421	787	1237
101	443	797	1259
107	461	821	1277
131	467	827	1283
139	491	829	1291
149	509	853	1301
163	523	859	1307
173	541	877	1373
179	547	883	1381
181	557	907	1427
197	563	941	1451

1453
1483
1493
1499
1523
1531
1549
1571
1619
1621
1637
1667
1669
1693
1733
1741
1747
1787
1861
1867
1877
1901
1907
1931
1949
1973
1979
1987
1997
2027
2029
2053
2069
2083
2099
2131
2141
2213
2221
2237
2243
2267
2269
2293
2309
2333
2339
2357
2371
2389
2437
2459
2467
2477
2531
2539
2549
2557
2579
2621
2659
2677
2683
2693
2699
2707
2741
2789
2797
2803
2819
2837
2843
2851
2861
2909
2939
2957
2963
3011
3019
3037
3067
3083
3187
3203
3253
3299
3307
3323
3347
3371
3413
3461
3467
3469
3491
3499
3517
3533
3539
3547
3557
3571
3581
3613
3637
3643
3659
3677
3691
3701
3709
3733
3779
3797
3803
3851
3853
3877
3907
3917
3923
3931
3947
3989
4003
4013
4019
4021
4091
4093
4099
4133
4139
4157
4219
4229
4243
4253
4259
4261
4283
4349
4357
4363
4373
4397
4451
4483
4493
4507
4517
4547
4603
4621
4637
4691
4723
4787
4789
4813
4877
4933
4957
4973
4987
5003
5011
5051
5059
5077
5099
5107
5147
5171
5179
5189
5227
5261
5309
5333
5387
5443
5477
5483
5501
5507
5557
5563
5573
5651
5659
5683
5693
5701
5717
5741
5749
5779
5813
5827
5843
5851
5869
5923
5939
5987
6011
6029
6053
6067
6101
6131
6173
6197
6203
6211
6229
6269
6277
6299
6317
6323
6373
6379
6389
6397
6469
6491
6547
6619
6637
6653
6659
6691
6701
6709
6733
6763
6779
6781
6803
6827
6829
6869
6883
6899

6907	7589	8429	9293
6917	7603	8443	9323
6947	7621	8467	9341
6949	7643	8539	9349
6971	7669	8563	9371
7013	7691	8573	9397
7019	7717	8597	9419
7027	7757	8627	9421
7043	7789	8669	9437
7069	7829	8677	9467
7109	7853	8693	9491
7187	7877	8699	9533
7211	7883	8731	9539
7219	7901	8741	9547
7229	7907	8747	9587
7237	7933	8803	9613
7243	7949	8819	9619
7253	8053	8821	9629
7283	8069	8837	9643
7307	8093	8861	9661
7331	8117	8867	9677
7349	8123	8923	9733
7411	8147	8933	9749
7451	8171	8963	9803
7459	8179	8971	9851
7477	8219	9011	9859
7499	8221	9029	9883
7507	8237	9059	9901
7517	8243	9173	9907
7523	8269	9181	9923
7541	8291	9203	9941
7547	8293	9221	9949
7549	8363	9227	
7573	8387	9283	

Табл. 17-2.
Отводные последовательности для FCSR максимальной длины

(32, 6, 3, 2)	(32, 29, 19, 2)	(64, 27, 22, 2)	(64, 49, 19, 2)
(32, 7, 5, 2)	(32, 29, 20, 2)	(64, 28, 19, 2)	(64, 49, 20, 2)
(32, 8, 3, 2)	(32, 30, 3, 2)	(64, 28, 25, 2)	(64,52,29,2)
(32, 13, 8, 2)	(32, 30, 7, 2)	(64, 29, 16, 2)	(64,53,8,2)
(32, 13, 12, 2)	(32, 31, 5, 2)	(64, 29, 28, 2)	(64, 53, 43, 2)
(32, 15, 6, 2)	(32, 31, 9, 2)	(64, 31, 12, 2)	(64, 56, 39, 2)
(32, 16, 2, 1)	(32, 31, 30, 2)	(64, 32, 21, 2)	(64, 56, 45, 2)
(32, 16, 3, 2)		(64, 35, 29, 2)	(64, 59, 5, 2)
(32, 16, 5, 2)	(64, 3, 2, 1)	(64, 36, 7, 2)	(64, 59, 8, 2)
(32, 17, 5, 2)	(64,14,3,2)	(64, 37, 2, 1)	(64, 59, 28, 2)
(32, 19, 2, 1)	(64,15,8,2)	(64, 37, 1 1, 2)	(64, 59, 38, 2)
(32, 19, 5, 2)	(64, 17, 2, 1)	(64,39,4,2)	(64,59,44,2)
(32, 19, 9, 2)	(64, 17, 9, 2)	(64, 39, 25, 2)	(64, 60, 49, 2)
(32, 19, 12, 2)	(64, 17, 16, 2)	(64, 41, 5, 2)	(64, 61, 51, 2)
(32, 19, 17, 2)	(64, 19, 2, 1)	(64, 41, 1 1, 2)	(64, 63, 8, 2)
(32, 20, 17, 2)	(64, 19, 18, 2)	(64,41,27,2)	(64, 63, 13, 2)
(32, 21, 9, 2)	(64, 24, 19, 2)	(64, 43, 21, 2)	(64, 63, 61, 2)
(32, 21, 15, 2)	(64, 25, 3, 2)	(64, 43, 28, 2)	
(32,23,8,2)	(64,25,4,2)	(64, 45, 28, 2)	(96, 15, 5. 2)
(32, 23, 21, 2)	(64, 25, 1 1, 2)	(64, 45, 41, 2)	(96, 21, 17, 2)
(32, 25, 5, 2)	(64, 25, 19, 2)	(64, 47, 5, 2)	(96, 25, 19, 2)
(32, 25, 12, 2)	(64, 27, 5, 2)	(64, 47, 21, 2)	(96, 25, 20, 2)
(32,27,25,2)	(64, 27, 16, 2)	(64, 47, 30, 2)	(96, 29, 15, 2)

(96, 29, 17, 2)
(96, 30, 3, 2)
(96, 32, 21, 2)
(96, 32, 27, 2)
(96,33,5,2)
(96, 35, 17, 2)
(96, 35, 33, 2)
(96, 39, 21, 2)
(96,40,25,2)
(96, 41, 12, 2)
(96, 41, 27, 2)
(96, 41, 35, 2)
(96, 42, 35, 2)
(96, 43, 14, 2)
(96, 44, 23, 2)
(96, 45, 41, 2)
(96, 47, 36, 2)
(96, 49, 31, 2)
(96,51,30,2)
(96,53,17,2)
(96, 53, 19, 2)
(96, 53, 32, 2)
(96, 53, 48, 2)
(96, 54, 15, 2)
(96, 55, 44, 2)
(96, 55, 53, 2)
(96, 56, 9, 2)
(96,56,51,2)
(96, 57, 3, 2)
(96, 57, 17, 2)
(96, 57, 47, 2)
(96, 58, 35, 2)
(96, 59, 46, 2)
(96, 60, 29, 2)
(96, 60, 41, 2)
(96, 60, 45, 2)
(96, 61, 17, 2)
(96, 63, 20, 2)
(96, 65, 12, 2)
(96, 65, 39, 2)
(96, 65, 51, 2)
(96, 67, 5, 2)
(96, 67, 25, 2)
(96,67,34,2)
(96, 68, 5, 2)
(96, 68, 19, 2)
(96, 69, 17, 2)
(96,69,36,2)
(96, 70, 23, 2)
(96, 71, 6, 2)
(96, 71, 40, 2)
(96, 72, 53, 2)
(96, 73, 32, 2)
(96, 77, 27, 2)
(96, 77, 31, 2)
(96, 77, 32, 2)
(96, 77, 33, 2)
(96,77,71,2)
(96,78,39,2)
(96, 79, 4, 2)
(96, 81, 80, 2)
(96, 83, 14, 2)
(96, 83, 26, 2)
(96, 83, 54, 2)
(96, 83, 60, 2)
(96, 83, 65, 2)
(96, 83, 78, 2)
(96, 84, 65, 2)
(96, 85, 17, 2)
(96, 85, 31, 2)
(96, 85, 76, 2)
(96,85,79,2)
(96,86,39,2)
(96,86,71,2)
(96, 87, 9, 2)
(96, 87, 44, 2)
(96, 87, 45, 2)
(96, 88, 19, 2)
(96, 88, 35, 2)
(96, 88, 43, 2)
(96,88,79,2)
(96, 89, 35, 2)
(96, 89, 51, 2)
(96, 89, 69, 2)
(96, 89, 87, 2)
(96, 92, 51, 2)
(96,92,71,2)
(96, 93, 32, 2)
(96, 93, 39, 2)
(96, 94, 35, 2)
(96, 95, 4, 2)
(96, 95, 16, 2)
(96, 95, 32, 2)
(96, 95, 44, 2)
(96, 95, 45, 2)

(128, 5, 4, 2)
(128, 15, 4, 2)
(128, 21, 19, 2)
(128, 25, 5, 2)
(128, 26, 11, 2)
(128,27,25,2)
(128, 31, 25, 2)
(128, 33, 21, 2)
(128, 35, 22, 2)
(128, 37, 8, 2)
(128, 41, 12, 2)
(128, 42, 35, 2)
(128, 43, 25, 2)
(128,43,42,2)
(128,45,17,2)
(128,45,27,2)
(128, 49, 9, 2)
(128, 51, 9, 2)
(128, 54, 51, 2)
(128, 55, 45, 2)
(128, 56, 15, 2)
(128, 56, 19, 2)
(128,56,55,2)
(128, 57, 21, 2)
(128, 57, 37, 2)
(128, 59, 29, 2)
(128, 59, 49, 2)
(128, 60, 57, 2)
(128,61,9,2)
(128, 61, 23, 2)
(128, 61, 52, 2)
(128, 63, 40, 2)
(128, 63, 62, 2)
(128, 67, 41, 2)
(128, 69, 33, 2)
(128, 71, 53, 2)
(128, 72, 15, 2)
(128,72,41,2)
(128, 73, 5, 2)
(128, 73, 65, 2)
(128, 73, 67, 2)
(128, 75, 13, 2)
(128, 80, 39, 2)
(128,80,53,2)
(128, 81, 55, 2)
(128, 82, 67, 2)
(128, 83, 60, 2)
(128, 83, 61, 2)
(128, 83, 77, 2)
(128, 84, 15, 2)
(128, 84, 43, 2)
(128,85,63,2)
(128,87,57,2)
(128,87,81,2)
(128, 89, 81, 2)
(128, 90, 43, 2)
(128, 91, 9, 2)
(128, 91, 13, 2)
(128, 91, 44, 2)
(128, 92, 35, 2)
(128,95,94,2)
(128, 96, 23, 2)
(128, 96, 61, 2)
(128, 97, 25, 2)
(128, 97, 68, 2)
(128, 97, 72, 2)
(128,97,75,2)
(128, 99, 13, 2)
(128, 99, 14, 2)
(128, 99, 26, 2)
(128, 99, 54, 2)
(128, 99, 56, 2)
(128, 99, 78, 2)
(128, 100, 13, 2)
(128, 100, 39, 2)
(128,101,44,2)
(128, 101, 97, 2)
(128, 103, 46, 2)
(128, 104, 13, 2)
(128, 104, 19, 2)
(128, 104, 35, 2)
(128,105,7,2)
(128, 105, 11, 2)
(128, 105, 31, 2)
(128, 105, 48, 2)
(128, 107, 40, 2)
(128, 107, 62, 2)
(128, 107, 102, 2)
(128, 108, 35, 2)
(128,108,73,2)
(128,108,75,2)
(128,108,89,2)
(128, 109, 1 1, 2)
(128, 109, 108, 2)
(128, 1 10, 23, 2)
(128, Ill, 61, 2)
(128, 113, 59, 2)
(128, 114, 83, 2)
(128,115,73,2)
(128, 117, 105, 2)
(128, 119, 30, 2)
(128, 119, 101, 2)
(128, 120, 9, 2)
(128, 120, 27, 2)
(128,120,37,2)
(128, 120, 41, 2)
(128, 120, 79, 2)
(128, 120, 81, 2)
(128, 121, 5, 2)
(128, 121, 67, 2)
(128, 121, 95, 2)
(128, 121, 96, 2)
(128, 123, 40, 2)
(128,123,78,2)
(128, 124, 41, 2)
(128, 124, 69, 2)
(128, 124, 81, 2)
(128, 125, 33, 2)
(128, 125, 43, 2)
(128,127,121,2)

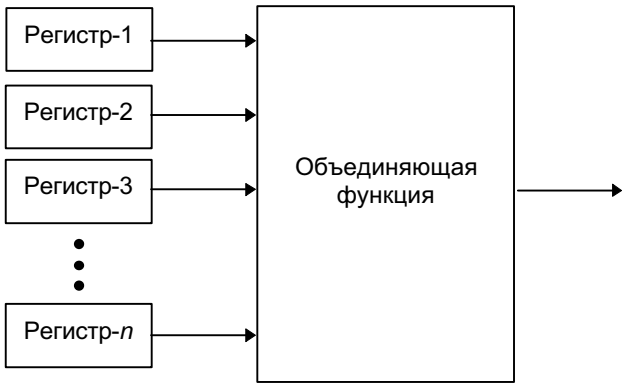


Рис. 17-5. Комбинированные генераторы.

Каскад LFSR/FCSR с суммированием/четностью

По теории сложение с переносом разрушает алгебраические свойства LFSR, а XOR разрушает алгебраические свойства FCSR. Данный генератор объединяет эти идеи, используемые в перечисленных суммирующем генераторе LFSR/FCSR и генераторе четности LFSR/FCSR, с каскадом Голлманна.

Генератор представляет собой последовательность массивов регистров , тактирование каждого массива определяется выходом предыдущего массива. На 11-й показан один этап такого генератора. Тактируется первый массив LFSR, и результаты объединяются сложением с переносом. Если выход функции объединения равен 1, то тактируется следующий массив (из FCSR), и выход этих FCSR объединяется с выходом предыдущей функции объединения с помощью XOR. Если выход первой функции объединения равен 0, то массив FCSR не тактируется, и выход просто складывается с переносом, полученным на предыдущем этапе Если выход этой второй функции объединения равен 1, то тактируется третий массив (из LFSR), и т.д.

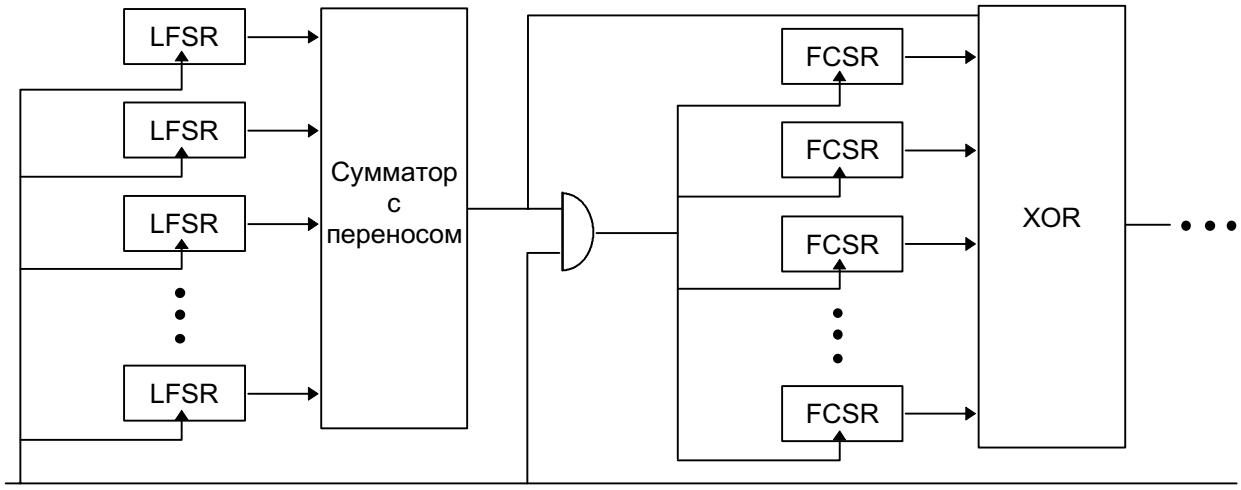


Рис. 17-6. Придуманный генератор.

Генератор использует много регистров: $n*m$, где n - количество этапов, а m - количество регистров на этапе. Я рекомендую $n = 10$ и $m = 5$.

Чередующиеся генераторы "стоп-пошел"

Эти генераторы использую FCSR вместо некоторых LFSR. Кроме того, операция XOR может быть заменена сложением с переносом (см. 10-й).

— Генератор "стоп-пошел" FCSR. Регистр-1, Регистр-2 и Регистр-3 - это FCSR. Объединяющая функция - XOR.

— Генератор "стоп-пошел" FCSR/LFSR. Регистр-1 - FCSR, а Регистр-2 и Регистр-3 - LFSR. Объединяющая функция - сложение с переносом.

— Генератор "стоп-пошел" LFSR/FCSR. Регистр-1 - LFSR, а Регистр-2 и Регистр-3 - FCSR. Объединяющая функция - XOR.

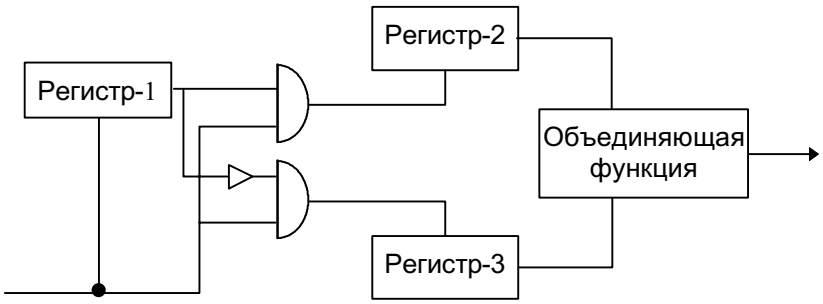


Рис. 17-7. Чередующийся генератор "стоп-пошел"

Прореживаемые генераторы

Существует четыре основных типа генераторов, использующих FCSR:

— Прореживаемый генератор FCSR. Прореживаемый генератор с FCSR вместо LFSR.

— Прореживаемый генератор FCSR/LFSR. Прореживаемый генератор с LFSR, прореживающим FCSR.

— Прореживаемый генератор LFSR/FCSR. Прореживаемый генератор с FCSR, прореживающим LFSR.

— Самопрореживаемый генератор FCSR. Самопрореживаемый генератор с FCSR вместо LFSR.

17.6 Сдвиговые регистры с нелинейной обратной связью

Нетрудно представить более сложную, чем используемая в LFSR или FCSR, последовательность обратной связи. Проблема в том, что не существует математического аппарата, позволяющего провести анализ таких п о-следовательностей. Что-то получится, но кто знает что? Вот некоторые из проблем, связанных со сдвиговыми регистрами с нелинейной обратной связью.

— В выходной последовательности могут быть смещения, например, единиц может быть больше, чем нулей.

— Максимальный период последовательности может быть меньше, чем ожидалось .

— Период последовательности для различных начальных значений может быть различным .

— Последовательность какое-то время может выглядеть как случайная, а потом "скатываться" к единстве н-ному значению. (Это можно легко устранить, выполняя XOR крайнего правого бита с нелинейной фун к-цией.)

Плюсом является то, что из-за отсутствия теории анализа сдвиговых регистров с нелинейной обратной св я-зью существует немного способов криптоанализировать потоковые шифры, основанные на таких регистрах . Использовать сдвиговые регистры с нелинейной обратной связью можно, но очень осторожно .

В сдвиговом регистре с нелинейной обратной связью функция обратной связи может быть произвольной (например, как на).

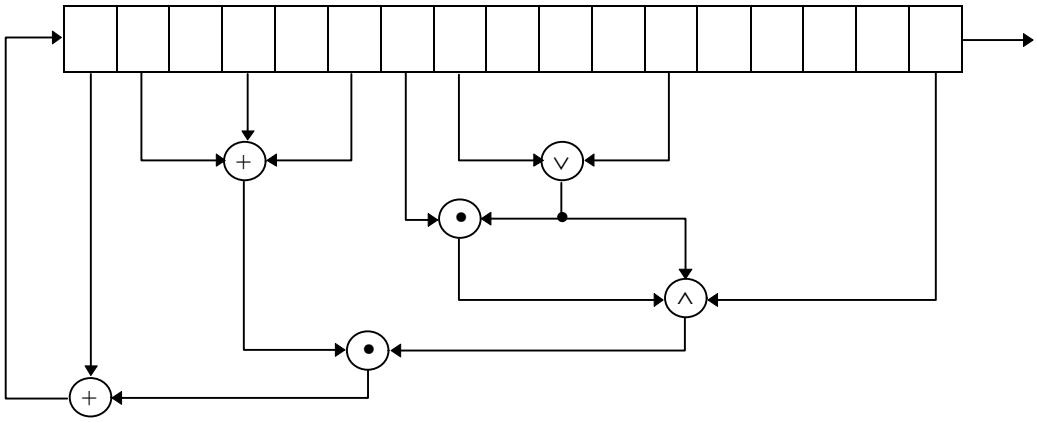


Рис. 17-8. Сдвиговый регистр с нелинейной обратной связью (возможно небезопасный).

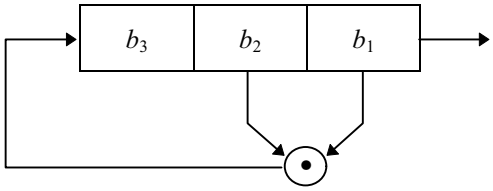


Рис. 17-9. 3-битовый сдвиговый регистр с нелинейной обратной связью.

На 8-й показан 3-битовый генератор со следующей обратной связью: новым битом является произведение первого и второго битов. Если его проинициализировать значением 110, то последовательность внутренних состояний будет следующей:

1 1 0

0 1 1

1 0 1

0 1 0

0 0 1

0 0 0

0 0 0

И так до бесконечности. Выходом является последовательность младших значащих битов:

0 1 1 0 1 0 0 0 0 0 0 0. . . .

Это не слишком полезно.

Может быть и хуже. Если начальное значение 100, то следующими состояниями являются 010, 001, а затем всегда 000. Если начальным значением является 111, то оно будет повторяться всегда и с самого начала.

Была проделана определенная работа по вычислению линейной сложности произведения двух LFSR [1650, 726, 1364, 630, 658, 659]. Конструкция, включающая вычисление LFSR над полем нечетных характеристик [310] не является безопасной [842.].

17.7 Другие потоковые шифры

В литературе описывались и другие потоковые шифры. Вот некоторые из них.

Генератор Плесса (Pless)

Этот генератор использует свойства J-K триггеров [1250]. Восемь LFSR управляют четырьмя J-K триггерами; каждый триггер нелинейно объединяет два LFSR. Чтобы избежать проблемы, что выход триггера определяет и источник, и значение следующего выходного бита, после тактирования четырех триггеров их выходы перемешиваются для получения окончательного потока ключей.

Этот алгоритм был криптоаналитически взломан с помощью вскрытия каждого триггера в отдельности

[1356]. К тому же, объединение J-K триггеров слабо криптографически; генераторы такого типа не устоят перед корреляционным вскрытием [1451].

Генератор на базе клеточного автомата

В [1608, 1609], Стив Вольфрам (Steve Wolfram) предложил использовать в качестве генератора псевдослучайных чисел одномерный клеточный автомат. Рассмотрение клеточного автомата не является предметом этой книги, но генератор Вольврама состоит из одномерного массива битов $a_1, a_2, a_3, \ldots, a_k, \ldots, an$ и функции обновления:

$a_k' = a_{k-1} \oplus (a_k \vee a_{k+1})$

Бит извлекается из одного из значений a_k, реально все равно какого.

Генератор ведет себя как вполне случайный. Однако для этих генераторов существует успешное вскрытие с известным открытым текстом [1052]. Это вскрытие выполнимо на PC со значениями n вплоть до 500 битов. Кроме того, Пол Барделл (Paul Bardell) доказал, что выход клеточного автомата может быть также сгенерирован с помощью сдвигового регистра с линейной обратной связью той же длины и, следовательно, не дает большей безопасности [83].

Генератор 1/p

Этот генератор был предложен и подвергнут криптоанализу в [193]. Если внутреннее состояние генератора в момент времени t равно x_t, то

$x_{t+1} = bx_t \bmod p$

Выходом генератора является младший значащий бит $x_t \operatorname{div} p$, где div - это целочисленное деление с усечением. Для максимального периода константы b и p должны быть выбраны так, что p - простое число, а b - примитивный корень mod p. К сожалению, этот генератор не безопасен. (Заметим, что для $b = 2$ FCSR целыми числами связи выдает последовательность, обратную данной.)

crypt(1)

Оригинальный алгоритм шифрования UNIX, crypt(1), представляет собой потоковый шифр, использующий те же идеи, что и Энигма. Это 256-элементный, одноротorный подстановочный шифр с отражателем. И ротор, и отражатель получаются из ключа. Этот алгоритм намного проще, чем немецкая Энигма времен второй мировой войны, и квалифицированному криптоаналитику несложно его взломать [1576, 1299]. Для вскрытия файлов, зашифрованных crypt(1), можно использовать свободно доступную программу UNIX, называемую Crypt Breakers Workbench (CBW, инструмент взломщика шифров).

Другие схемы

Еще один генератор основан на проблеме рюкзака (см. раздел 19.2) [1363]. CRYPTO-LEGGO небезопасен [301]. Джоан Дэймен (Joan Daemen) разработала SubStream, Jam и StepRightUp [402], но они слишком новы, чтобы их комментировать. Множество других алгоритмов описано в литературе, но еще больше хранится в секрете и встроено в аппаратуру.

17.8 Системно-теоретический подход к проектированию потоковых шифров

На практике, проектирование потокового шифра во многом похоже проектирование блочного шифра. В этом случае используется больше математической теории, но в конце концов криптограф предлагает какую-то схему и затем пытается выполнить ее анализ.

Согласно Райнеру Рюппелу существует четыре различных подхода к проектированию потоковых шифров [1360, 1362]:

— Системно-теоретический подход. Используя ряд фундаментальных критериев и законов проектирования, пытается удостовериться, что каждая схема создает сложную и неизвестную проблему для криптоанализа,.

— Информационно-теоретический подход. Пытается сохранить открытый текст в тайне от криптоаналитика. Независимо от того, как много действий выполнит криптоаналитик, он никогда не получит однозначного решения.

— Сложностно-теоретический подход. Пытается использовать в качестве основания для криптосистемы некоторую известную и сложную проблему, такую как разложение на множители или взятие дискретных логарифмов, или сделать криптосистему эквивалентной этой проблеме.

— Рандомизированный подход. Пытается создать чрезвычайно большую проблему, заставляя криптоанал итика проверить множество бессмысленных данных в ходе попыток криптоанализа .

Эти подходы отличаются предположениями о возможностях и способностях криптоаналитика, определением успеха криптоанализа и пониманием безопасности. Большинство исследований в этой области - теоретические, но среди бесполезных потоковых шифров есть и вполне приличные .

Системно-теоретический подход использовался во всех ранее приведенных потоковых шифрах, результатом его применения являются большинство используемых в реальном мире потоковых шифров . Криптограф разрабатывает генераторы потока ключей, обладающие проверяемыми характеристиками безопасности - периодом, распределением битов, линейной сложностью и т.д. - а не шифры, основанные на математической теории . Криптограф также изучает различные методы криптоанализа этих генераторов и проверяет, устойчивы ли ген ераторы по отношению к этим способам вскрытия.

Со временем этот подход привел к появлению набора критериев проектирования потоковых шифров [1432, 99, 1357, 1249]. Они рассматривались Рюппелом в [1362], где он подробно приводит теоретические основы этих критериев.

— Длинный период без повторений.

— Критерий линейной сложности - большая линейная сложность , линейный профиль сложности, локальная линейная сложность и т.д.

— Статистические критерии, например, идеальные k-мерные распределения.

— Путаница - каждый бит потока ключей должен быть сложным преобразованием всех или большинства битов ключа.

— Диффузия - избыточность в подструктурах должна рассеиваться, приводя к более "размазанной" стат истике.

— Критерии нелинейности для логических функций, такие как отсутствие корреляции m-го порядка, расстояние до линейных функций, лавинный критерий, и т.д.

Этот перечень критериев проектирования не уникален для потоковых шифров, разработанных с помощью системно-теоретического подхода, он справедлив для всех потоковых шифров. Это справедливо и для всех блочных шифров. Особенностью системно-теоретического подхода является то, что потоковые шифры неп осредственно разрабатываются, чтобы удовлетворить этим критериям .

Главной проблемой таких криптосистем является невозможность доказать их безопасность, никогда не было доказано, что эти критерии проектирования необходимы или достаточны для безопасности . Генератор потока ключей может удовлетворять всем правилам разработки, но тем не менее оказаться небезопасным . Другой может оказаться безопасным. Этом процессе все еще остается что-то магическое.

С другой стороны вскрытие любого из этих генераторов потока ключей представляет собой отличную пр облему для криптоаналитика. Если будет разработано достаточно различных генераторов , может оказаться, что криптоаналитик не станет тратить время, взламывая каждый из них . Может, его больше заинтересует возмо жность прославиться, достигнув успеха, разлагая на множители большие числа или вычисляя дискретные лог арифмы.

17.9 Сложностно-теоретический подход к проектированию потоковых шифров

Рюппел также очертил сложностно-теоретический подход к проектированию потоковых шифров . В соответствии с ним криптограф пытается использовать теорию сложности, чтобы доказать его генераторы безопасны . Следовательно, генераторы должны быть как можно больше сложнее, основываясь на тех же трудных пробл емах, что и криптография с открытыми ключами. И, также как алгоритмы с открытыми ключами, они оказыв аются медленными и громоздкими.

Генератор псевдослучайных чисел Шамира

Эди Шамир использовал в качестве генератора псевдослучайных чисел алгоритм RSA [1417]. Хотя Шамир показал, что предсказание выхода генератора псевдослучайных чисел равносильно взлому RSA, потенциальное смещение выхода была продемонстрирована в [1401, 200].

Генератор Blum-Micali

Безопасность этого генератора определяется трудностью вычисления дискретных логарифмов [200]. Пусть g - простое число, а p - еще одно простое число. Ключ x_0 начинает процесс:

$x_{i+1} = g^{x_i} \bmod p$

Выходом генератора является 1, если $x_i < (p - 1)/2$, и 0 в противном случае.

Если p достаточно велико, чтобы вычисление дискретных логарифмов mod p стало физически невозможным, то этот генератор безопасен. Дополнительные теоретические результаты можно найти в [1627, 986, 985, 1237, 896, 799].

RSA

Этот генератор RSA [35, 36] является модификацией [200]. Начальные параметры - модуль N, произведение двух больших простых чисел p и q, и целое число e, относительно простое с $(p-1)(q-1)$, а также стартовое случайное число x_0, меньшее N.

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

Выход генератора представляет собой младший значащий бит x_i. Безопасность этого генератора опирается на сложность вскрытия RSA. Если N достаточно велико, то генератор безопасен. Дополнительная теория приведена в [1569, 1570, 1571, 30, 354].

Blum, Blum, and Shub

Простейший и наиболее эффективный генератор, использующий сложностно-теоретический подход, в честь своих авторов называется Blum, Blum, and Shub. Мы сократим его название до BBS, хотя иногда его называют генератором с квадратичным остатком [193].

Теория генератора BBS использует квадратичные остатки по модулю n (см. раздел 11.3). Вот как он работает.

Сначала найдем два простых числа, p и q, которые конгруэнтны 3 modulo 4. Произведение этих чисел, n, является целым числом Блюма (Blum). Выберем другое случайное целое число x, взаимно простое с n. Вычислим

$x_0 = x^2 \bmod n$

Это стартовое число генератора.

Теперь можно начать вычислять биты. i-ым псевдослучайным битом является младший значащий бит x_i, где

$x_i = x_{i-1}^2 \bmod n$

Самым интригующим свойством этого генератора является то, что для получения i-го бита не нужно вычислять предыдущие i-1 биты. Если вам известны p и q, вы можете вычислить i-ый бит непосредственно.

b_i - это младший значащий бит x_i, где $x_i = x_0^{(2^i) \bmod ((p-1)(q-1))}$

Это свойство означает, что вы можете использовать этот криптографически сильный генератор псевдослучайных чисел в качестве потоковой криптосистемы для файла с произвольным доступом.

Безопасность этой схемы основана на сложности разложения n на множители. Можно опубликовать n, так что кто угодно может генерировать биты с помощью генератора. Однако пока криптоаналитик не сможет разложить n на множители, он никогда не сможет предсказать выход генератора - ни даже утверждать что-нибудь вроде: "Следующий бит с вероятностью 51 процент будет единицей".

Более того, генератор BBS **непредсказуем в левом направлении** и непредсказуем в правом направлении. Это означает, что получив последовательность, выданную генератором, криптоаналитик не сможет предсказать ни следующий, ни предыдущий бит последовательности. Это вызвано не безопасностью, основанной на каком-то никому не понятном сложном генераторе битов, а математикой разложения n на множители.

Этот алгоритм медленен, но есть способы его ускорить. Оказывается, что в качестве псевдослучайных битов можно использовать несколько каждого x_i. В соответствии с [1569, 1570, 1571, 35, 36] если n - длина x_i, можно использовать $\log_2 n$ младших значащих битов x_i. Генератор BBS сравнительно медленный и не подходит для потоковых шифров. Однако для высоконадежных приложений, таких как генерация ключей, этот генератор лучше многих других.

17.10 Другие подходы к проектированию потоковых шифров

При информационно-теоретическом подходе к потоковым шифрам предполагается, что криптоаналитик обладает неограниченными временем и вычислительной мощностью. Единственным практически реализованным потоковым шифром, защищенным от такого противника, является одноразовый блокнот (см. раздел 1.5). Так как писать биты в блокноте не очень удобно, его иногда называют **одноразовой лентой**. На двух магнитных лентах, на одной для шифрования, а на другой для дешифрирования, должен быть записан идентичный поток ключей. Для шифрования просто выполняется XOR открытого текста с битами ленты. Для дешифрирования

выполняется XOR шифротекста с битами другой, идентичной ленты. Один и тот же поток ключей нельзя использовать дважды. Так как биты потока ключей действительно случайны, предсказать поток ключей невозможно. Если сжигать ленты после использования, то безопасность будет абсолютной (при условии, что у кого-то другого нет копии ленты).

Другой информационно-теоретический потоковый шифр, разработанных Клаусом Шнорром (Claus Schnorr) предполагает, что криптоаналитик имеет доступ только к ограниченному числу битов шифротекста [1395]. Результаты являются слишком теоретическими results и не имеют практического значения. Подробности можно найти [1361, 1643,1193].

С помощью рандомизированного потокового шифра криптограф пытается сделать решение проблемы, стоящей перед криптоаналитиком, физически невозможным. Для этого, сохраняя небольшой размер секретного ключа, криптограф значительно увеличивает количество битов, с которыми придется иметь дело криптоанализу. Это может быть сделано за счет использования при шифровании и дешифрировании большой опубликованной случайной строки. Ключ же указывает, какие части строки будут использованы при шифровании и дешифрировании. Криптоаналитику, не знающему ключа, придется перебирать случайные комбинации частей строки. Безопасность такого шифра можно выразить с помощью среднего числа битов, которые должен проверить криптоаналитик прежде, чем вероятностьопределить ключ значительно удучшится по сравнению с вероятностью простого угадывания.

Шифр "Рип ван Винкль"

Джеймс Массей (James Massey) и Ингемар Ингемарсон (Ingemar Ingemarsson) предложили шифр "Рип ван Винкль" [1011], названный так, потому что получатель, чтобы начать дешифрирование, должен получить 2^n битов шифротекста. Алгоритм, показанный на 7-й, прост в реализации, гарантировано безопасен и совершенно непрактичен. Просто выполните XOR открытого текста с потоком ключей и задержите поток ключей на время от 0 до 20 лет - точная задержка является частью ключа. По словам Массея: "Можно легко доказать, что вражескому криптоаналитику для вскрытия шифра понадобятся тысячи лет, если кто-то согласится подождать с чтением открытого текста миллионы лет." Развитие этой идеи можно найти в [1577, 755].

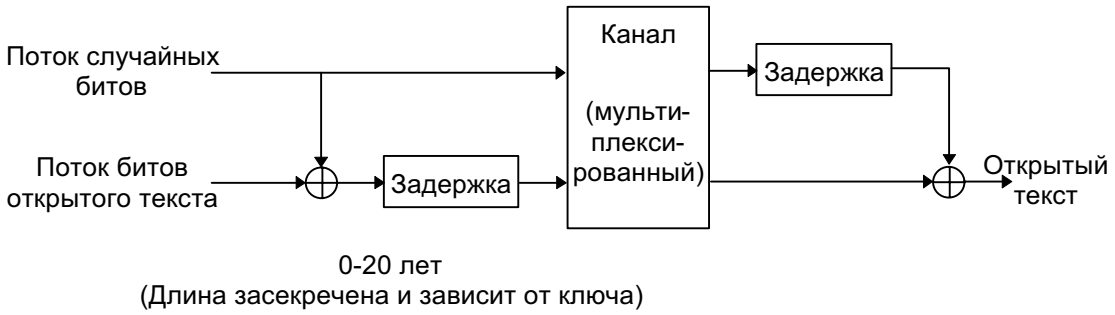


Рис. 17-10. Шифр "Рип ван Винкль".

Рандомизированный потоковый шифр Диффи

Эта схема впервые была предложена Уитфилдом Диффи [1362]. Используется 2^n случайных последовательностей. Ключ представляет собой случайную n-битовую строку. Для шифрования сообщения Алиса использует k-ую случайную строку как одноразовый блокнот. Затем она отправляет шифротекст и 2^n случайных строк по 2^n+1 различным каналам связи.

Боб знает k-, поэтому он может легко выбрать, какой из одноразовых блокнотов использовать для дешифрования сообщения. Еве остается только перебирать случайные последовательности, пока она не найдет правильный одноразовый блокнот. Для вскрытия потребуется проверить некоторое число битов, по порядку равное $O(2^n)$. Рюппел указал, что, если вы отправляете n случайных строк вместо2^n, и если ключ используется для задания линейной комбинации этих случайных строк, безопасность остается на прежнем уровне .

Рандомизированный потоковый шифр Маурера

Уели Маурер (Ueli Maurer) описал схему, основанную на выполнении XOR открытого текста с несколькими большими открытыми последовательностями случайных битов [1034, 1029, 1030]. Ключ является набором стартовых позиций в каждой последовательности. Можно доказать, что такой шифр почти безопасен, с вероятность взлома определяется объемом памяти, имеющейся в распоряжении взломщика, независимо от доступной ему вычислительной мощности. Маурер утверждает, что эта схема становится практичной при 100 различных последовательностях длиной 10^{20} случайных битов каждая. Одним из способов получить рак много битов явля-

ется оцифровка поверхности Луны.

17.11 Шифры с каскадом нескольких потоков

Если производительность не важна, то нет причин выбирать несколько потоковых шифров и объединять их в каскад. Для получения шифротекста просто выполните XOR выхода каждого генератора с открытым текстом. Результат Уели Маурера (см. раздел 15.7) показывает, что если генераторы используют независимые ключи, то безопасность каскада по крайней мере не меньше безопасности самого сильного алгоритма каскада, а скорее всего и намного больше.

Потоковые шифры объединяются теми же способами, что и блоковые (см. главу 15). Потоковые шифры можно объединить в каскад (см. раздел 15.7) с другими потоковыми шифрами или с блочными шифрами.

Ловким трюком является использование одгого алгоритма, потокового или блочного, для частого обновления ключа быстрого потокового алгоритма (которым может быть и блочный алгоритм в режиме OFB). Быстрый алгоритм может быть слабым, так как криптоаналитик никогда не получит достаточно открытого текста, зашифрованного одним ключом.

Существует способ разменять размер внутреннего состояния быстрого алгоритма (который может влиять на безопасность) на частоту смены ключа. Смена ключа должна быть относительно частой, не стоит использовать для этого алгоритмы с длинной процедурой установки ключа. Кроме того, смена ключа не должна зависеть от внутреннего состояния быстрого алгоритма.

17.12 Выбор потокового шифра

Если изучение потоковых шифров и дает какой-либо результат, так это появление с пугающей регулярностью все новых способов вскрытия. Традиционно потоковые шифры опирались на большую математическую теорию. Эту теорию можно было использовать для доказательства положительных качеств шифра, но ее же можно было использовать для поиска новых способов вскрытия шифра. По этой причины любой потоковый шифр, основанный только на LFSR, вызывает мое беспокойство.

Я предпочитаю потоковые шифры, спроектированные подобно блочным шифрам: нелинейные преобразования, большие S-блоки, и т.д. Больше всего мне нравится RC4, а затем SEAL. Мне бы очень хотелось увидеть результаты криптоанализа предложенных мной генераторов, объединяющих LFSR и FCSR. Эта область кажется весьма привлекательной для изучения возможности использования в реальных разработках. Или для получения потокового шифра можно использовать блочный шифр в режиме OFB или CFB.

В 14-й для сравнения приведены временные соотношения для некоторых алгоритмов.

Табл. 17-3.
Скорости шифрования нескольких потоковых шифров на i486SX/33 МГц

Алгоритм	Скорость шифрования (Мбайт/с)
A5	5
PIKE	62
RC4	164
SEAL	381

17.13 Генерация нескольких потоков из одного генератора псевдослучайной последовательности

Если нужно зашифровать несколько каналов связи при помощи одного блока - например, мультиплексора - простым решением является использование для каждого потока своего генератора псевдослучайной последовательности. При этом возникают две следующих проблемы: нужна дополнительная аппаратура, и все генераторы должны быть синхронизированы. Проще было бы использовать один генератор.

Одно из решений - тактировать генератор несколько раз. Если нужно три независимых потока, тактируйте генератор три раза и отправьте по одному биту в каждый поток. Этот метод работает, но могут быть сложности при получении большой частоты. Например, если вы можете тактировать генератор только в три раза быстрее тактирования потока данных, вы сможете создать только три потока. Другим способом является использование одной и той же последовательности для каждого канала, возможно с переменной временной задержкой. Это небезопасно.

Действительно удачная идея [1489], запатентованная NSA, показана на 6-й. Записывайте выход вашего любимого генератора в простой m-битовый сдвиговый регистр. По каждому тактовому импульсу сдвигайте регистр на один бит вправо. Затем для каждого выходного потока выполните AND регистра с другим m-битовым вектором, рассматриваемым как уникальный идентификатор для выбранного выходного потока, затем объедините с помощью XOR все биты, получая выходной бит для этого потока. Если требуется получить параллельно несколько выходных потоков, для каждого выходного потока нужно использовать отдельный вектор и логический массив XOR/AND.

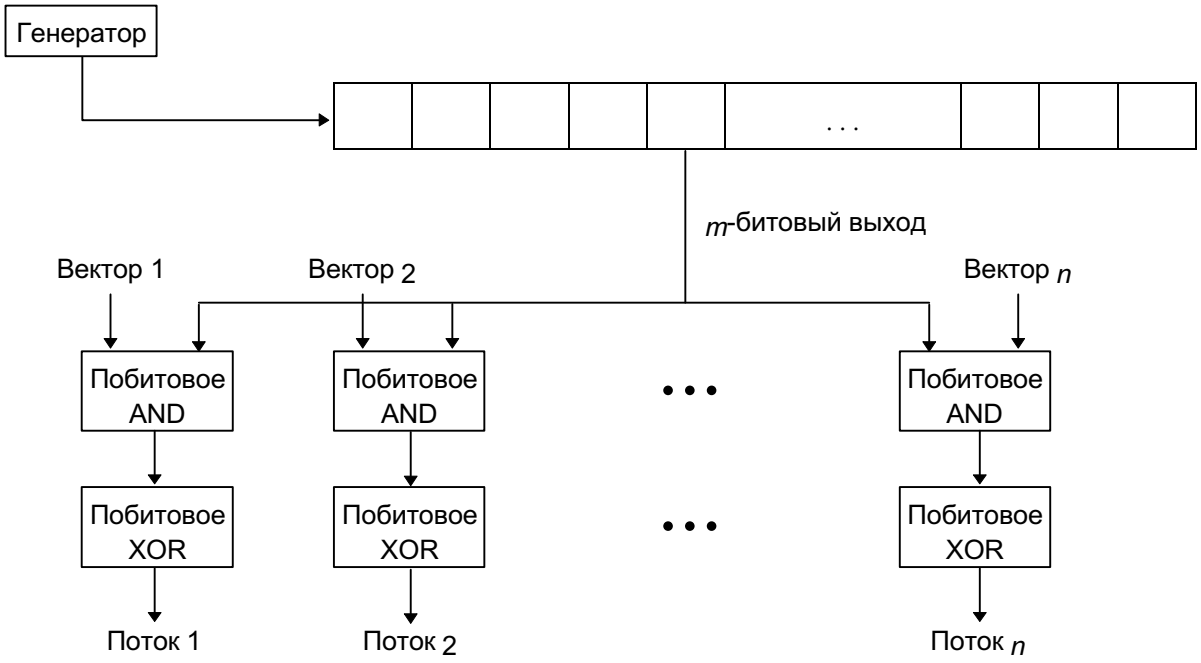


Рис. 17-11. Генератор нескольких битов.

Существует ряд вещей, которые нужно отслеживать. Если любой из этих потоков является линейной комбинацией других потоков, то система может быть взломана. Но если вы достаточно аккуратны, описанный способ является простым и безопасным способом решения проблемы.

17.14 Генераторы реальных случайных последовательностей

Иногда криптографически безопасные псевдослучайные последовательности недостаточно хороши. В криптографии вам могут понадобиться действительно случайные числа. Первое, что приходит в голову - это генерация ключей. Прекрасно можно генерировать случайные криптографические ключи, используя генератор псевдослучайных последовательностей, но если враг добудет копию этого генератора и главный ключ, он сможет создать те же ключи и взломать вашу криптосистему, независимо от надежности ваших алгоритмов. Последовательность, выдаваемую генератором случайных последовательностей, воспроизвести невозможно. Никто, даже вы сами, не сможет воспроизвести последовательность битов, выдаваемую этими генераторами.

Крупной философской проблемой является вопрос о том, дают ли эти методы действительно случайные биты. Я не собираюсь ввязываться в этот спор. Здесь я рассматриваю выдачу битов, которые невозможно воспроизвести, и у которых статистические свойства как у случайных битов.

Для любого генератора действительно случайных последовательностей важным вопросом является его проверка. На эту тему существует множество литературы. Тесты на случайность можно найти в [863, 99]. Маурер показал, что все эти тесты можно получить из попытки сжать последовательность [1031, 1032]. Если случайная последовательность сжимается, то она не является по настоящему случайной.

В любом случае, все, что мы имеем в этой области, во многом относится к черной магии. Главным моментом является генерация последовательности битов, которую не сможет угадать ваш противник. Это гораздо более трудная задача, чем кажется. Я не могу доказать, что любой из описанных методов генерирует случайные биты. Результатом их работы являются последовательности битов, которые невозможно легко воспроизвести. Подробности можно найти в [1375, 1376, 511].

Таблицы RAND

Давным давно, в 1955 году, когда компьютеры все еще были в новинку, Rand Corporation издала книгу, содержавшую миллион случайных цифр [1289]. Их метод описывался так:

> Случайные цифры этой книги были получены при помощи рандомизации основной таблицы, сгенерированной эле к-тронной рулеткой. Вкратце, источник импульсов, выдающий их со случайной частотой в среднем около 100000 импульсов в секунду, открывался раз в секунду импульсом постоянной частоты. Цепи нормализации импульса пропускали импульсы ч е-рез 5-разрядный бинарный счетчик. По сути машина являлась колесом рулетки с 32-позициями, которое в среднем делало около 3000 оборотов за выборку и выдавало одно число в секунду. Использовался двоично-десятичный преобразователь, к о-торый преобразовывал 20 из 32 чисел (оставшиеся двенадцать отбрасываются) и оставлял только последнюю цифру двузна ч-ных чисел. Эти последние цифры попадали в компостер IBM, образуя в конце концов таблицу пробитых карточек случайных цифр.

В книге рассматривались и результаты различных проверок данных на случайность. В ней также предлагался способ, как использовать эту книгу для выбора случайного числа:

> Строки таблицы цифр нумеруются от 00000 до 19999. При использовании таблицы нужно сначала выбрать случайную стартовую позицию. Обычной процедурой для этого является следующее: откройте эту книгу на произвольной странице та б-лицы цифр и, закрыв глаза, выберите пятиразрядное число. Это число после замены первой цифры остатком от деления ее на 2 определяет стартовую строку. Остаток от деления двух цифр справа от первоначально выбранного пятиразрядного числа на 50 задает стартовый столбец в стартовой строке. Чтобы защититься от открытия книги все время на одной странице и естес т-венного стремления выбрать число поближе к центру страницы, каждое использованное для определения стартовой позиции пятиразрядное число должно быть помечено и не должно больше использоваться для этой ц ели.

Главным содержанием этой книги была "Таблица случайных цифр". Цифры приводились пяти разрядными группами - "10097 32533 76520 13586 . . ." - по 50 в строке и по пятьдесят строк на странице. Таблица занимала 400 страниц и, за исключением особенно выдающейся группы на странице 283, выглядевшей как "69696", была достаточно скучным чтивом. В книгу также входила таблица 100000 нормальных отклонений.

Интересным в книге RAND являются не миллионы случайных цифр, а то, что они были созданы до компь ю-терной революции. Во многих криптографических алгоритмах используются произвольные константы - так н а-зываемые "магические числа". Выбор магических чисел из таблиц RAND гарантировал, что они не были в ы-браны специально по каким-то жульническим причинам. Так, например, было сделано в Khafre.

Использование случайного шума

Лучшим способом получить большое количество случайных битов является извлечение их из естественной случайности реального мира. Часто такой метод требует специальной аппаратуры, но этот трюк можно прим е-нить и в компьютерах.

Найдите событие, которое случается регулярно, но случайно: атмосферный шум, преодолевающий какой-то порог, ребенок, падающий, учась ходить. Измерьте интервал между одним подобным событием и событием, следующим за ним. Запишите. Измерьте временной интервал между вторым и третьим событиями. Снова запишите. Если первый временной интервал больше второго, выходным битом будет 1. Если второй интервал больше первого, то выходом события будет 0. Сделайте это снова для следующего соб ытия.

Бросьте стрелу дартс в перечень котировок Нью-Йоркской фондовой бирже в местной газете. Сравните котировку акции, в которую вы попали, с котировкой акции прямо над ней. Если больше та, в которую вы попали, выход равен 0, а если меньше - 1.

Подключите к компьютеру счетчик Гейгера, подсчитайте количество импульсов за фиксированный интервал времени и возьмите младший бит. Или измерьте время между последовательными тиками ticks. (Так как радио-активный источник распадается, среднее время между последовательными тиками непрерывно увеличивается. Чтобы этого избежать, надо выбирать источник с достаточно длинным периодом полураспада - такой как пл у-тоний. Если вы беспокоитесь о своем здоровье, можете внести соответствующие статистические поправки.)

Дж. Б. Эгнью (G. B. Agnew) предложил генератор реально случайных битов, который можно интегрировать в СБИС [21]. Это конденсатор металл-изолятор-полупроводник (metal insulator semiconduction capacitor, MISC). Два таких конденсатора помещаются рядом друг с другом, а случайный бит является функцией разности зар я-дов этих конденсаторов. Другой генератор случайных чисел генерирует поток случайных битов, используя н е-стабильность частоты свободно колеблющегося осциллятора [535]. Коммерческая микросхема от AT&T генери-рует случайные числа, опираясь именно на это явление [67]. М. Гьюд (M. Gude) построил генератор случайных чисел, собирающий случайные биты из физических явлений, например, радиоактивного распада [668, 669]. Манфилд Рихтер (Manfield Richter) разработал генератор случайных чисел на базе температурного шума пол у-проводникового диода [1309].

Предположительно случайны временные интервалы между последовательными 2e4 излучениями света рас-падающегося атома ртути. Используйте. А лучше найдите полупроводниковую фирму, которая изготавливает микросхемы генераторов случайных чисел, их достаточно много.

Существует также генератор случайных чисел, использующий диск компьютера [439]. Он измеряет время, нужное для чтения блока диска, и использует изменения этого времени в качестве источника случайных чисел. Данные фильтруются, чтобы удалить структуру, вызванную квантованием, затем к векторам чисел применяется быстрое преобразование Фурье. Это устраняет смещение и корреляцию. Наконец, в качестве случайных битов используются спектральные углы для частот в диапазоне $(0, \pi)$, нормализованные на единичный интервал.

Большая часть изменений скорости вращения диска вызвана турбулентностью воздуха, которая и является источником случайности в системе. Хотя надо учесть следующее. Если вы выдаете на выход слишком много битов, то вы используете в качестве генератора случайных чисел быстрое преобразование Фурье и рискуете получить определенную предсказуемость. И лучше снова и снова читать один и тот же дисковый блок, чтобы вам не пришлось фильтровать структуру, источником которой является планировщик диска. Реализация такой системы позволяла получать около 100 битов в минуту [439].

Использование таймера компьютера

Если вам нужен один случайный бит (или даже несколько), воспользуйтесь младшим значащим битом любого регистра таймера. В системе UNIX он может быть не слишком случайным из-за различной возможной синхронизации, но на некоторых персональных компьютерах это работает.

Не стоит извлекать таким образом слишком много битов. Выполнение много раз одной и той же процедуры последовательно может легко сместить биты, генерированные этим способом. Например, если выполнение каждой процедуры генерации бита занимает четное число тиков таймера, на выходе вашего генератора будет бесконечная последовательность одинаковых битов. Если выполнение каждой процедуры генерации бита занимает нечетное число тиков таймера, на выходе вашего генератора будет бесконечная последовательность череду ющихся битов. Даже если зависимость не так очевидна, получающийся битовый поток будет далек от случайного. Один генератор случайных чисел работает следующим образом [918]:

> Наш генератор действительно случайных чисел . . . работает, устанавливая будильник и затем быстро инкрементируя регистр счетчика процессора до тех пор, пока не произойдет прерывание. Далее выполняется XOR содержимого регистра и содержимого байта выходного буфера (данные регистра усекаются до 8 битов). После того, как будет заполнен каждый байт выходного буфера, буфер подвергается дальнейшей обработке циклическим сдвигом каждого символа вправо на два бита. Это приводит к эффекту перемещения наиболее активных (и случайных) младших значащих битов в старшие значащие позиции. Затем весь процесс повторяется три раза. Наконец после прерываний два самых случайных бита регистра счетчика повлияют на каждый символ буфера. То есть происходит $4n$ прерываний, где n - число нужных случайных битов.

Этот метод очень чувствителен к случайности системных прерываний и квантованности таймера. При тестировании на реальных UNIX-машинах результат был очень неплох.

Измерение скрытого состояния клавиатуры

Процесс печатания и случаен, и неслучаен. Он достаточно неслучаен, чтобы его можно было использовать для идентификации печатающего человека, но он достаточно случаен, чтобы его можно было использовать для генерации случайных битов. Измерьте время между последовательными нажатиями клавиш, затем воспользуйтесь младшими значащими битами этих измерений. Эти биты оказываются достаточно случайными. Этот метод не работает на UNIX-терминалах, так как нажатия клавиш прежде, чем они будут переданы вашей программе, проходят через фильтры и другие механизмы, но это будет работать на большинстве персональных компьютеров.

В идеале вы должны по каждому нажатию клавиши генерировать только один бит. Использование большего количества битов может сместить результаты в зависимости от навыков машинистки. Однако этот метод имеет ряд ограничений. Хотя нетрудно посадить за клавиатуру человека, печатающего со скоростью 100 слов в минуту или около того, если есть время для генерации ключа, глупо просить машинистку печатать текст из 100000 слов, чтобы использовать результат работы генератора в качестве одноразового блокнота.

Смещения и корреляции

Главной проблемой подобных систем являются возможные закономерности в генерируемой последовательности. Используемые физические процессы могут быть случайны, но между физическим процессом и компьютером находятся различные измерительные инструменты. Эти инструменты могут легко привести к появлению проблем.

Способом устранить **смещение**, или отклонение, является XOR нескольких битов друг с другом. Если случайный бит смещен к 0 на величину e, то вероятность 0 можно записать как:

$P(0) = 0.5 + e$

XOR двух из таких битов дает:

$P(0) = (0.5 + e)^2 + (0.5 - e)^2 = 0.5 + 2e^2$

Те же вычисления для XOR 4 битов дают:

$P(0) = 0.5 + 8e^4$

XOR m битов экспоненциально сходится к равной вероятности 0 и 1. Если известно максимальное смещение, которое допустимо в вашем приложении, вы можете вычислить, сколько битов вам нужно объединить с помощью XOR, чтобы уменьшить смещение до этого значения.

Еще лучше рассматривать биты попарно. Если 2 бита одинаковы отбросьте их и взгляните на следующую пару. Если 2 бита различны, используйте первый бит в качестве выхода генератора. Это полностью устраняет смещение. Другие методы уменьшения смещения используют распределение переходов сжатие и быстрое пр е-образование Фурье [511].

Потенциальной проблемой обоих методов является то, что при наличии **корреляции** между соседними битами эти методы увеличивают смещение. Одним из способов исправить это является использование нескольких случайных источников. Возьмите четыре случайных источника и выполните XOR битов друг с другом или возьмите два случайных источника и взгляните на их биты попарно.

Например, возьмите радиоактивный источник и присоедините счетчик Гейгера к вашему компьютеру. Возьмите пару шумящих диодов и записывайте в качестве события каждое превышение определенного значения. Измерьте атмосферный шум. Извлеките из каждого источника случайный бит и выполните их XOR друг с другом, получая случайный бит. Возможности бесконечны.

Одно то, что генератор случайных чисел смещен не обязательно означает его бесполезность. Это только означает, что он менее безопасен. Например, рассмотрим проблему Алисы, генерирующей 168-битовый ключ для тройного DES. А все, что у нее есть, - это генератор случайных битов со смещением к 0 : с вероятностью 55 процентов он выдает нули и с вероятностью 45 процентов - единицы. Это означает, что энтропия на бит ключа с оставит только 0.99277 (для идеального генератора она равна 1). Мэллори, пытаясь раскрыть ключ, может оптимизировать выполняемое вскрытие грубой силой, проверяя сначала наиболее вероятные ключи (000 . . . 0) и двигаясь к наименее вероятному ключу (111 . . . 1). Из-за смещения Мэллори может ожидать, что ему удастся обнаружить ключ за 2^{109} попыток. При отсутствии смещения Мэллори потребуется 2^{111} попыток. Полученный ключ менее безопасен, но это практически неощутимо.

Извлеченная случайность

В общем случае лучший способ генерировать случайные числа - найти большое количество кажущихся сл у-чайными событий и извлечь случайность из них. Эта случайность может храниться в накопителе и извлекаться при необходимости. .Однонаправленные хэш-функции прекрасно подходят для этого. Они быстры, поэтому вы можете пропускать биты через них, не слишком заботясь о производительности или действительной случайн о-сти каждого наблюдения. Попробуйте хэшировать почти все, что вам кажется хоть чуть-чуть случайным. Н а-пример:

— Копия каждого нажатия на клавиши

— Команды мыши

— Номер сектора, время дня и задержка поиска для каждой дисковой операции

— Действительное положение мыши

— Номер текущей строки развертки монитора

— Содержание действительно выводимого на экран изображения

— Содержание FAT-таблиц, таблиц ядра, и т.д.

— Времена доступа/изменения /dev/tty

— Загрузка процессора

— Времена поступления сетевых пакетов

— Выход микрофона

— /dev/audio без присоединенного микрофона

Если ваша система использует различные кристаллы-осцилляторы для своего процессора и часов, попытайтесь считывать время дня в плотном цикле. В некоторых (но не всех) системах это приведет к случайным кол е-баниям фазы между двумя осцилляторами.

Так как случайность в этих событиях определяется синхронизацией осцилляторов, используйте часы с как можно меньшим квантом времени. В стандартном PC используется микросхема таймера Intel 8254 (или эквивалентная), работающая на тактовой частоте 1.1931818 МГц, поэтому непосредственное считывание регистра счетчика даст разрешение в 838 наносекунд. Чтобы избежать смещения результатов, не используйте в качестве источника событий прерывание таймера. Вот как выглядит этот процесс на языке C с MD5 (см. раздел 18.5) в качестве хэш-функции:

```
char Randpool[16];

/* Часто вызывается для широкого множества случайных или полуслучайных системных событий для to
churn the randomness pool . Точный формат и длина randevent не имеет значения, пока его содержание
```

```
является в некоторой мере чем-то непредсказуемым. */
   void churnrand(char *randevent,unsigned lnt randlen) {
        MD5_CTX md5;
        MD5Init(&md5);
        MD5Update(&md5, Randpool , sizeof(Randpool));
        MD5Update(&md5 , randevent , randlen );
        MD5Final(Randpool,&md5);
   }
```

После достаточных вызовов churnrand() накопления достаточной случайности в Randpool, можно генерировать из этого случайные биты. MD5 снова становится полезной, в этот раз в качестве генератора псевдослуча йного байтового потока, работающего в режиме счетчика.

```
long Randcnt;
void genrand(char *buf,unsigned int buflen) {
     MD5_CTX md5;
     char tmp[16];
     unsigned int n;
     while(buflen != 0) {
             /* Пул хэшируется счетчиком */
             MD5Init(&md5);
             MD5Update(&md5, Randpool, sizeof(Randpool));
             MD5Update(&md5,(unsigned char *)&Randcnt,sizeof(Randcnt));
             MD5Final(tmp,&md5);
             Randcnt++; /* Инкрементируем счетчик */
             /*Копируем 16 или запрошенное число байтов, если оно меньше 16, в буфер
             пользователя*/
             n = (buflen < 16) ? buflen : 16;
             memcpy(buf, tmp, n);
             buf += n ;
             buflen -= n;
     }
}
```

По многим причинам хэш-функция имеет ключевое значение. Во первых она обеспечивает простой способ генерировать произвольное количество псевдослучайных данных, не вызывая всякий раз churnrand(). На деле, когда запас в накопителе подходит к концу, система постепенно переходит от совершенной случайности к пра к-тической. В этом случае становится *теоретически* возможным использовать результат вызова genrand() для определения предыдущего или последующего результата. Но для этого потребуется инвертировать MD5, что вычислительно невозможно.

Это важно, так как процедуре неизвестно, что делается потом со случайными данными, которые она возвр ащает. Один вызов процедуры может генерировать случайное число для протокола, которое посылается в явном виде, возможно в ответ на прямой запрос взломщика. А следующий вызов может генерировать секретный ключ для совсем другого сеанса связи, в суть которого и хочет проникнуть взломщик. Очевидна важность того, чтобы взломщик не смог получить секретный ключ, используя подобную схему действий.

Но остается одна проблема. Прежде, чем в первый раз будет вызвана genrand() в массиве Randpool[] должно быть накоплено достаточно случайных данных. Если система какое-то время работала с локальным пользовате лем, что-то печатающим на клавиатуре, то проблем нет. Но как насчет независимой системы, которая перегр ужается автоматически, не обращая внимания ни на какие данные клавиатуры или мыши ?

Но есть одна трудность. В качестве частичного решения можно потребовать, чтобы после самой первой з агрузки оператор какое-то время поработал на клавиатуре и создал на диске стартовый файл перед выгрузкой операционной системы, чтобы в ходе перезагрузок использовались случайные данные, переданные в Randseed[]. Но не сохраняйте непосредственно сам Randseed[]. Взломщик, которому удастся заполучить этот файл, сможет определить все результаты genrand() после последнего обращения к churnrand() прежде, чем этот файл будет создан.

Решением этой проблемы является хэширование массива Randseed[] перед его сохранением, может даже вызовом genrandO. При перезагрузке системы вы считываете данные из стартового файла, передаете их

churnrand(), а затем немедленно стираете их. К сожалению это не устраняет угрозы того, что злоумышленник добудет файл между перезагрузками и использует его для предсказания будущих значений функции genrand(). Я не вижу иного решения этой проблемы кроме, как подождать накопления достаточного количества случайных событий, случившихся после перезагрузки, прежде, чем позволить genrand() выдавать результаты.