МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ

Кафедра дифференциальных уравнений и системного анализа

КРИПТОСИСТЕМА RC4

Курсовая работа

Петроченко Виктории
Андреевны

студента 3 курса,
специальность 1-31 03 09
Компьютерная математика
и системный анализ

Научный руководитель:
кандидат физ.-мат. наук,
доцент Д. Н. Чергинец

Минск, 2019

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

С древнейших времен криптография использовалась для защиты военной и дипломатической связи. Необходимость защиты правительственной связи вполне очевидна, и до недавнего времени широкое применение криптографии было почти исключительно правом государства. В настоящее время большинство правительств контролирует или сами исследования в этой области, или, по крайней мере, производство криптографического оборудования и программного обеспечения. В Республике Беларусь деятельность по технической и (или) криптографической защите информации является лицензируемой. С началом информационного века возникла срочная необходимость использования криптографии в частном секторе. Сегодня огромное количество конфиденциальной информации (такой, например, как персональные данные, истории болезней, юридические документы, данные о финансовых операциях) передается между ЭВМ по обычным линиям связи. Поэтому возникает необходимость обеспечения секретности и подлинности подобной информации. Криптология (происходит от греческих корней, означающих "тайный" и "слово") – наука о шифровании и дешифровании. Шифрование – метод, используемый для преобразования исходных данных в зашифрованный текст (криптограмму) для того, чтобы они могли быть прочитаны только пользователем, обладающим соответствующим ключом шифрования для расшифрования содержимого. Криптология делится на две части: криптографию (шифрование) и криптоанализ. Криптография занимается разработкой методов обеспечения секретности и (или) аутентичности (подлинности) сообщений. Криптоанализ предназначен для решения обратной задачи – раскрытия (взлома) шифра с целью получения возможности несанкционированного чтения зашифрованного сообщения или осмысленной подделки такого сообщения. Кроме того, криптоанализ применяют при исследовании шифров с целью улучшения их

свойств, например, криптографической стойкости. В современных информационных системах ключ – это двоичная строка, которая может быть интерпретирована набором символом другого алфавита. Цель криптографической системы, чаще всего, заключается в том, чтобы зашифровать осмысленный исходный текст, получив в результате зашифрованный текст бессмысленный с точки зрения постороннего наблюдателя. Получатель, которому он предназначен, должен быть способен расшифровать этот криптотекст, восстановив, таким образом, соответствующий ей открытый текст. При этом противник (криптоаналитик) должен быть неспособен раскрыть исходный текст.

Рассматриваемая в работе криптосистема RC4 относится к множеству симметричных синхронных поточных систем шифрования.

Целью настоящей работы является анализ криптосистемы RC4 с последующей реализацией алгоритмических преобразований шифрования и дешифрования в виде динамической библиотеки классов на языке C#.

ГЛАВА 1
КЛАССИФИКАЦИЯ КРИПТОГРАФИЧЕСКИХ АЛГОРИТМОВ. АБСОЛЮТНО СТОЙКИЙ ШИФР, ГАММИРОВАНИЕ. ПОТОЧНОЕ ШИФРОВАНИЕ

1.1 Классификация криптографических алгоритмов

Криптографические алгоритмы можно разделить на три категории в зависимости от числа ключей, которые они используют:

- бесключевые алгоритмы – алгоритмы, которые не используют какие-либо ключи;
- одноключевые алгориты – алгоритмы, которые работают с одним секретным ключом;
- двухключевые алгоритмы – алгоритмы, которые на различных этапах применяют два вида ключей: секретный и открытый.

Каждая категория алгоритмов также имеет свою классификацию, одна из которых показана на рисунке 1.1.

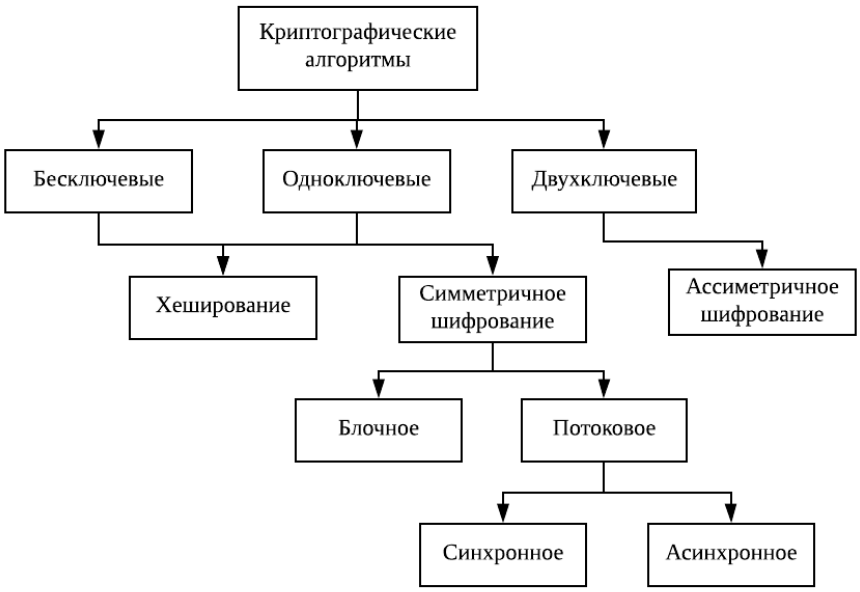

Рисунок 1.1. Классификация криптографических алгоритмов

Хеширование – это сворачивание данных переменной длины в последовательность фиксированного размера. Иными словами, это контрольное суммирование данных.

Симметричное шифрование – это вид шифрования, который использует один и тот же ключ для зашифровывания и расшифровывания данных. Симметричное шифрование можно разделить на две категории: блочное и потоковое.

Основной идеей *блочного шифрования* является разбиение информации на блоки фиксированной длины. Далее ко всем блокам применяются функции шифрования, таким образом данные зашифровываются. Текущее поколение блочных шифров работает с блоками текста, имеющие длину 128 бит: на входе шифр принимает 128-битовый открытый текст и на выходе выдает 128-битовый шифрованный текст. Блочные шифры являются обратимыми, т.е. существует функция дешифрования, которая получает на входе из зашифрованный текст и на выходе выдает открытый.

Потоковое шифрование применяют тогда, когда информацию невозможно разбить на блоки, например, есть какой-то поток данных, каждый символ которого необходимо зашифровать, не дожидаясь остальных данных, чтобы сформировать блок. Потоковое шифрование шифрует данные по одному биту за такт шифрования.

Ассиметричное шифрование – это вид шифрования, который использует два вида ключей для зашифровывания и расшифровывания данных. В качестве ключа для зашифровывания информации берется открытый ключ, а для расшифровывания – закрытый.

1.2 Абсолютно стойкий шифр. Гаммирование

Популярность поточных шифров можно связать с работой К. Шеннона, которая была посвящена анализу схемы однократного использования (рисунок 1.2), которую изобрел Г.С. Вернам.

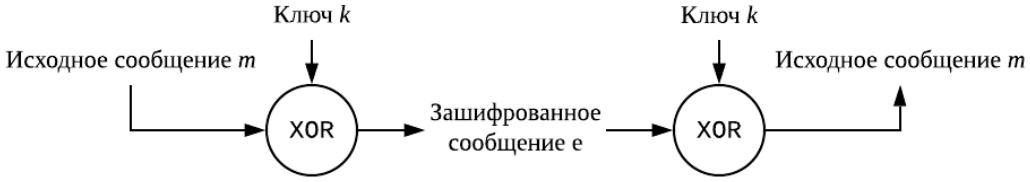

Рисунок 1.2. Схема однократного использования Вернама

Пусть n – число битов открытого текста. Необходимо зашифровать сообщение $m = m_1 m_2 ... m_n$. Формируется ключ $k = k_1 k_2 ... k_n$ – n-разрядная случайная двоичная последовательность, которая известна только отправителю и получателю сообщения. Тогда зашифрованное сообщение $e = e_1 e_2 ... e_n$ получается по формуле:

$$e_i = m_i \oplus k_i,$$

где $i = \overline{1,n}$ и $\oplus$ обозначает сложение по модулю 2.

Так как сложение и вычитание по модулю 2 совпадают, то легко увидеть, что процесс дешифрования осуществляется по формуле:

$$m_i = e_i \oplus k_i, i = \overline{1,n}.$$

Пример 1.1. Пусть $m = 01001,\ k = 11010$. Тогда получаем:

$$e = m \oplus k = (01001) \oplus (11010) = 10011,$$

$$m = e \oplus k = (10011) \oplus (11010) = 01001.$$

К. Шенноном доказано, что если

- ключ образован из независимых и равновероятных символов,
- длина ключа равна длине исходного сообщения,
- ключ используется только один раз, после чего уничтожается,

то такой шифр является абсолютно стойким, его невозможно раскрыть, даже если криптоаналитик располагает неограниченным запасом времени и неограниченным набором вычислительных ресурсов. Действительно, противнику известно только зашифрованное сообщение e, при этом все различные ключевые последовательности k возможны и равновероятны, а значит возможны и любые сообщения m, т.е. криптоалгоритм не дает никакой информации об открытом тексте. Таким образом шифр Вернама является абсолютно стойким шифром.

Абсолютная стойкость рассматриваемой схемы очень затруднительна. Основной ее недостаток – необходимость генерировать случайные последовательности очень большого объема и передавать их по закрытым каналам связи. Применение схемы оправдано лишь в нечасто используемых каналах связи для шифрования исключительно важных сообщений. Таким образом, построить эффективный алгоритм можно, отказавшись от абсолютной стойкости. Была высказана идея использовать в качестве ключа не случайные последовательности, а порожденные при помощи генераторов псевдослучайных чисел. Данный результат может быть достигнут при использовании гаммирования, схема которого изображена на рисунке 1.3.

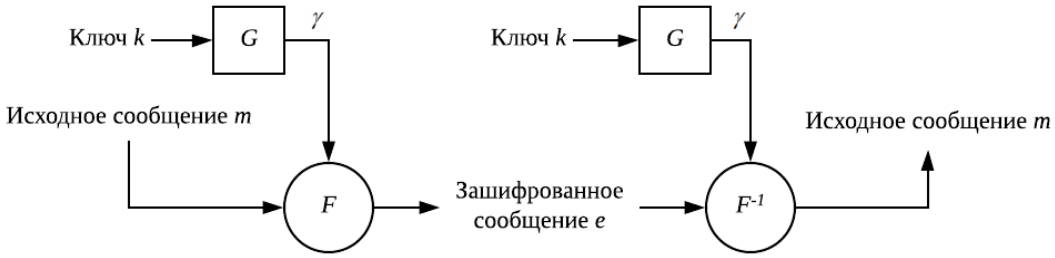

Рисунок 1.3. Шифрование информации методом гаммирования.

G – генератор ПСП, F – функция гаммирования,

F^{-1} - функция, обратная F , γ - гамма шифра

Гаммирование – процедура наложения с помощью некоторой функции F на входные данные псевдослучайную последовательность, называемую гаммой шифра. Если символы входной информационной последовательности и гаммы представлены в двоичном виде, то в качестве функции F берется функция поразрядного сложения по модулю 2.

Поточный шифр по своей сути пытается имитировать концепцию абсолютно стойкого шифра, используя короткий ключ для генерации псевдослучайной последовательности.

1.3 Поточное шифрование

Поточные шифры являются разновидностью гаммирования и преобразуют открытый текст последовательно по 1 биту за такт шифрования.

Поточное шифрование выглядит следующим образом:

- Перед началом шифрования отправитель и получатель должны обладать общим секретным ключом.

- Секретный ключ используется для генерации инициализирующей последовательности генератора гаммы.
- Генераторы отправителя и получателя используются для получения одинаковой гаммы $k_1 k_2 ... k_n$. Последовательности будут одинаковыми, если для их получения использовались одинаковые ГПСЧ, которые были проинициализированный одной и той же инициализирующей последовательностью.
- Символы открытого текста $m_1 m_2 ... m_n$ на стороне отправителя складываются по модулю 2 с символами гаммы:

$$e_i = m_i \oplus k_i,\ i = \overline{1,n}.$$

Полученное таким образом зашифрованное сообщение передается по каналу связи.

- На стороне легального получателя с символами зашифрованного сообщения и гаммы выполняется обратная операция для получения открытого текста. Для сложения по модулю 2 обратной является эта же операция:

$$m_i = e_i \oplus k_i = m_i \oplus k_i \oplus k_i = m_i,\ i = \overline{1,n}.$$

Криптостойкость поточных шифров целиком зависит от внутренней структуры генерации ключевой последовательности. Необходимые атрибуты используемых генераторов:

- большой размер инициализирующей последовательности,
- длинный период,
- большая линейная сложность.

Генератор гаммы выдает поток битов, который выглядит случайным, но является детерминированным. Чем больше генерируемый поток похож на случайный, тем больше потребуется от криптоаналитика для взлома шифра.

1.3.1 Синхронное поточное шифрование

Потоковый шифр называется синхронным, если выходные значения генератора гаммы не зависят от открытого или шифрованного текстов. Основная сложность в данном подходе заключается в синхронизации генераторов передающей и приемной сторонах. При синхронном шифровании зашифровывающее и расшифровывающее устройства должны работать строго согласованно, так как дешифрование будет неудачным до тех пор, пока шифрующие последовательности для шифрования и дешифрования не засинхронизируются. Если в процессе передачи произошло выпадение или вставка хотя бы одного бита, то получатель обнаружит бессмысленные данные с места, где сбилась синхронизация. Восстановление синхронизации происходит с помощью поиска возможных сдвигов между тактами работы устройств отправителя и получателя. Обычно синхронизация достигается вставкой в передаваемое сообщение специальных “маркеров”, с их помощью бит шифротекста, пропущенный в процессе передачи, приводит к неправильному дешифрованию лишь до тех пор, пока не будет принят один из маркеров.

Синхронные поточные шифры имеют свойство не распространять ошибки. Дешифрование одного искаженного бита шифротекста приведет к искажению только одного соответствующего бита открытого текста. Однако такие шифры уязвимы к изменению отдельных бит. Если злоумышленник знает исходный текст, то он сможет изменять биты в потоке шифрованного текста так, что текст будет расшифровываться таким образом, как необходимо злоумышленнику.

Синхронные шифры также защищают от вставок и выбрасываний отрезков шифрованного текста из потока, так как такие действие приведут к нарушению синхронизации, а это сражу же будет обнаружено на приемной стороне.

1.3.2 Асинхронное поточное шифрование

Потоковый шифр называется асинхронным (самосинхронизирующимся), если внутреннее состояние генератора гаммы является функцией фиксированного числа предшествующих битов шифрованного текста. Так как внутреннее состояние зависит только от *n* бит шифрованного текста, то на приемной стороне генератор войдет в синхронизм с передающей стороной после получения *n* бит. Каждое сообщение предваряется случайным заголовком длины *n* бит. Этот заголовок шифруется и передается получателю. На приемной стороне заголовок расшифровывается. Результат расшифровки будет неверным, однако после обработки *n* бит заголовка состояние дешифратора на приемной стороне становится идентичным состоянию шифратора, таким образом синхронизация установлена.

В отличие от синхронного поточного шифрования, асинхронное шифрование имеет свойство распространять ошибки. При дешифровании одного искаженного бита генератор выдаст *n* неверных бит ключевой последовательности, пока искаженный бит не будет вытолкнут из памяти, что приведет к неверному дешифрованию *n* бит исходного текста. После приема *n* последовательных неискаженных бит поточный дешифратор опять способен расшифровывать правильно.

ГЛАВА 2
ХАРАКТЕРИСТИКА КРИПТОСИСТЕМА RC4

Криптосистема RC4 – это класс алгоритмов поточного шифрования с переменной длиной ключа. Шифр был разработан в 1987 году Рональдом Райвистом для компании RSA Data Security. В течении 7 лет RC4 лицензировался компанией только на условиях неразглашения, однако в 1994 году он был анонимно опубликован в интернете. С тех пор шифр RC4 стал доступен для независимого анализа. В настоящее время алгоритм RC4 широко применяется в различных системах защиты информации, в компьютерных сетях (например, в протоколе *SSL*, для шифрования информации в операционных системах семейства *Windows*, и др.).

RC4 стал популярен благодаря простоте его аппаратной и программной реализации, а также благодаря высокой скорости работы алгоритма в обоих случаях.

2.1 Описание алгоритмов RC4

RC4 – поточный шифр, следовательно шифрование и дешифрование будет происходить следующим образом:

- шифрование – $e_i = m_i \oplus k_i$,
- дешифрование – $m_i = e_i \oplus k_i$,

где m_i, e_i – i-ое слово открытого и закрытого текста соответственно, а k_i – i-ое слово псевдослучайной последовательности, которая была получена при помощи двух алгоритмов шифра RC4: Алгоритма Ключевого Расписания (Key Scheduling Algorithm, KSA) и Алгоритма Псевдослучайной Генерации (Pseudo Random

Generator Algorithm, PRGA). В обоих алгоритмах используется *S*-блок – массив из 2^n бит, где n – количество бит в одном слове для алгоритма. Параметр n может быть любым, однако обычно используют $n = 8$.

2.1.1 Алгоритм ключевого расписания (KSA – key scheduling algorithm)

Алгоритм ключевого расписания так же называют алгоритмом инициализации генератора гаммы или алгоритмом инициализации *S*-блока. RC4 использует *l*-словарный ключ, которым заполняется массив *K*, такого же размера как массив *S*. Если $l < 2^n$, то ключ повторяется нужное число раз, пока массив *K* полностью не заполнится. Затем выполняется алгоритм ключевого расписания (KSA): инициализация начинается с заполнения массива *S*, далее этот массив перемешивается путем перестановок, определяемых ключом.

Вход: Массив K. Выход: Инициализированный S-блок.
1. $j \leftarrow 0$, $S \leftarrow (0,1,...,2^n - 1)$; 2. Для $i = 0,1,...,2^n - 1$: 2.1. $j \leftarrow (j + S[i] + K[i]) \bmod 2^n$; 2.2. $S[i] \leftrightarrow S[j]$.

Алгоритм 2.1. Алгоритм KSA

В результате *S*-блок оказывается инициализированным при помощи перемешивания значений его начального заполнения ($S \leftarrow (0,1,...,2^n - 1)$) в зависимости от секретного ключа. После этого генератор гаммы готов к работе.

2.1.2 Алгоритм псевдослучайной генерации (PRGA)

Теперь требуется получить псевдослучайное слово k генератора гаммы (для $n=8$ длина слова равна 1 байту), чтобы потом применить процедуру шифрования либо дешифрования. Получение псевдослучайного слова осуществляется алгоритмом псевдослучайной генерации (PRGA): генератор гаммы RC4 переставляет значения, которые хранятся в массиве *S*, таким образом в одном цикле определяется одно *n*-битное слово *k* из ключевого потока.

Вход: i, j. Выход: Слово генератора гаммы k_i.
1. $i \leftarrow (i+1) \bmod 2^n$; 2. $j \leftarrow (j + S[i]) \bmod 2^n$; 3. $S[i] \leftrightarrow S[j]$; 4. $t \leftarrow (S[i] + S[j]) \bmod 2^n$; 5. $k_r \leftarrow S[t]$.

Алгоритм 2.2. Алгоритм PRGA

Для получения нулевого *n*-битного слова генератора гаммы первоначальные значения переменных i и j должны равняться нулю. Для получения следующих *n*-битных слов алгоритм выполняется повторно. Таким образом получаем k_r, $r = \overline{0, p}$ псевдослучайных слов, где p - количество бит в открытом (закрытом) тексте.

2.2 Пример работы алгоритмов RC4

Как уже говорилось, количество бит в одном слове для алгоритма задает параметр n, который может принимать любые значения, обычно $n=8$. Однако для

демонстрации примера работы шифра RC4 примем $n = 4$. Тогда массивы S и K будут иметь длину $2^n = 2^4 = 16$ бит.

Пусть секретный ключ состоит из шести 4-битовых значений (приведем их в десятичном виде): 1, 2, 3, 4, 5, 6. Тогда массив K заполнится следующим образом:

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

Таблица 2.1. Заполнение массива K

Покажем этапы работы алгоритма KSA.

1. $j = 0$, S-блок имеет вид:

Номер элемента S-блока	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Значение	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Таблица 2.2. Начальное заполнение S-блока

2. $i = 0$,

   $j = 0$,

   $j = (j + S[i] + K[i]) \bmod 16 = (0 + 0 + 1) \bmod 16 = 1 \bmod 16 = 1$,

   $S[0] \leftrightarrow S[1]$:

Номер элемента S-блока	**0**	**1**	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Значение	1	0	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Таблица 2.3. Заполнение S-блока для $i = 0$

3. $i = 1$,

$j = 1$,

$j = (j + S[i] + K[i]) \bmod 16 = (1 + 0 + 2) \bmod 16 = 3 \bmod 16 = 3$,

$S[1] \leftrightarrow S[3]$:

Номер элемента *S*-блока	0	**1**	2	**3**	4	5	6	7	8	9	10	11	12	13	14	15
Значение	1	3	2	0	4	5	6	7	8	9	10	11	12	13	14	15

Таблица 2.4. Заполнение *S*-блока для $i = 1$

4. $i = 2$,

$j = 3$,

$j = (j + S[i] + K[i]) \bmod 16 = (3 + 2 + 3) \bmod 16 = 8 \bmod 16 = 8$,

$S[2] \leftrightarrow S[8]$:

Номер элемента *S*-блока	0	1	**2**	3	4	5	6	7	**8**	9	10	11	12	13	14	15
Значение	1	3	8	0	4	5	6	7	2	9	10	11	12	13	14	15

Таблица 2.5. Заполнение *S*-блока для $i = 2$

[шаги для $i = \overline{3,13}$ опущены]

16. $i = 14$,

$j = 1$,

$j = (j + S[i] + K[i]) \bmod 16 = (1 + 14 + 3) \bmod 16 = 18 \bmod 16 = 2$,

$S[14] \leftrightarrow S[2]$:

Номер элемента *S*-блока	0	1	**2**	3	4	5	6	7	8	9	10	11	12	13	**14**	15
Значение	10	13	14	12	2	15	6	4	5	3	1	9	8	11	0	7

Таблица 2.6. Заполнение *S*-блока для $i = 14$

17. $i = 15,$

$j = 2,$

$j = (j + S[i] + K[i]) \bmod 16 = (2 + 7 + 4) \bmod 16 = 13 \bmod 16 = 13,$

$S[15] \leftrightarrow S[13]:$

Номер элемента *S*-блока	0	1	2	3	4	5	6	7	8	9	10	11	12	**13**	14	**15**
Значение	10	13	14	12	2	15	6	4	5	3	1	9	8	7	0	11

Таблица 2.7. Заполнение *S*-блока для $i = 15$

Таким образом, после завершения работы алгоритма KSA S-блок будет заполнен так, как показано в таблице 2.7.

Теперь продемонстрируем этапы работы алгоритма PRGA для получения первых трех слов гаммы.

1. Для получения нулевого слова оператора гамы установим $i = 0,\ j = 0.$

1.1. $i = (i + 1) \bmod 16 = (0 + 1) \bmod 16 = 1 \bmod 16 = 1.$

1.2. $j = (j + S[i]) \bmod 16 = (0 + 13) \bmod 16 = 13 \bmod 16 = 13.$

1.3. $S[1] \leftrightarrow S[13]:$

Номер элемента *S*-блока	0	**1**	2	3	4	5	6	7	8	9	10	11	12	**13**	14	15
Значение	10	7	14	12	2	15	6	4	5	3	1	9	8	13	0	11

Таблица 2.8. *S*-блока для получения нулевого слова

1.4. $t = (S[i] + S[j]) \bmod 16 = (7 + 13) \bmod 16 = 20 \bmod 16 = 4$.

1.5. $k_0 = S[t] = S[4] = 2$.

2. $i = 1,\ j = 13$.

2.1. $i = (i + 1) \bmod 16 = (1 + 1) \bmod 16 = 2 \bmod 16 = 2$.

2.2. $j = (j + S[i]) \bmod 16 = (13 + 14) \bmod 16 = 27 \bmod 16 = 11$.

2.3. $S[2] \leftrightarrow S[11]$:

Номер элемента *S*-блока	0	1	**2**	3	4	5	6	7	8	9	10	**11**	12	13	14	15
Значение	10	7	9	12	2	15	6	4	5	3	1	14	8	13	0	11

Таблица 2.9. *S*-блока для получения первого слова

2.4. $t = (S[i] + S[j]) \bmod 16 = (9 + 14) \bmod 16 = 23 \bmod 16 = 7$.

2.5. $k_1 = S[t] = S[7] = 4$.

3. $i = 2,\ j = 11$.

3.1. $i = (i + 1) \bmod 16 = (2 + 1) \bmod 16 = 3 \bmod 16 = 3$.

3.2. $j = (j + S[i]) \bmod 16 = (11 + 12) \bmod 16 = 23 \bmod 16 = 7$.

3.3. $S[3] \leftrightarrow S[7]$:

Номер элемента *S*-блока	0	1	2	**3**	4	5	6	**7**	8	9	10	11	12	13	14	15
Значение	10	7	9	4	2	15	6	12	5	3	1	14	8	13	0	11

Таблица 2.10. *S*-блока для получения второго слова

3.4. $t = (S[i] + S[j]) \bmod 16 = (4+12) \bmod 16 = 16 \bmod 16 = 0$.

3.5. $k_2 = S[t] = S[0] = 10$.

Таким образом получили первые 3 слова гаммы: 2, 4, 10.

Допустим первые три слова исходного сообщения m были 3, 6, 13, тогда первые три слова зашифрованного сообщения $e_i = m_i \oplus k_i$ получаются следующим образом:

$$e_0 = m_0 \oplus k_0 = 3_{10} \oplus 2_{10} = 0011_2 \oplus 0010_2 = 0001_2 = 1_{10},$$

$$e_1 = m_1 \oplus k_1 = 6_{10} \oplus 4_{10} = 0110_2 \oplus 0100_2 = 0010_2 = 2_{10},$$

$$e_2 = m_2 \oplus k_2 = 13_{10} \oplus 10_{10} = 1101_2 \oplus 1010_2 = 0111_2 = 7_{10}.$$

Таким образом первыми тремя словами зашифрованного сообщения e являются 1, 2, 7. Покажем процедуру дешифрования для этих слов:

$$m_0 = e_0 \oplus k_0 = 1_{10} \oplus 2_{10} = 0001_2 \oplus 0010_2 = 0011_2 = 3_{10},$$

$$m_1 = e_1 \oplus k_1 = 2_{10} \oplus 4_{10} = 0010_2 \oplus 0100_2 = 0110_2 = 6_{10},$$

$$m_2 = e_2 \oplus k_2 = 7_{10} \oplus 10_{10} = 0111_2 \oplus 1010_2 = 1101_2 = 13_{10}.$$

Исходный код динамической библиотеки классов на языке C# представлен в приложении А.

ГЛАВА 3
АТАКА ФЛУРЕРА, МАНТИНА И ШАМИРА. ПРОТОКОЛ WEP

Атака Флурера, Мантина и Шамира, или FMS-атака, – это атака, которая применяется против протокола WEP. Она основана на слабости алгоритма ключевого расписания (KSA) в RC4 и использовании инициализирующих векторов *IV*.

3.1 Протокол WEP

Wired Equivalent Privacy (WEP) — алгоритм для обеспечения безопасности сетей Wi-Fi. Используется для обеспечения конфиденциальности и защиты передаваемых данных авторизированных пользователей беспроводной сети. Существует две разновидности WEP: WEP-40 и WEP-104, различающиеся только длиной ключа. Рассмотрим процесс шифрования и дешифрования в протоколе WEP.

Шифрование в WEP выполняется отдельно для каждого текстового сообщения, обозначим его буквой *M*. Процесс шифрования можно разделить на следующие части.

- Вычисление контрольной суммы сообщения *M*, для того, чтобы потом можно было проверить его целостность. Контрольная сумма считается при помощи 32-разрядного циклического избыточного сложения. Полученная контрольная сумма дописывается в конце сообщения *M*. Таким образом получается открытый текст *P*.

Рисунок 3.1. Открытый текст *P*

- Шифрование в WEP осуществляется с помощью криптосистемы RC4. Ключом *key* для данного шифра является WEP-ключ, к которому впереди приписывается вектор инициализации *IV*. Вектор инициализации состоит из 24 бит, которые генерируются для каждого сообщения. Таким образом ключ *key* имеет следующую структуру (рисунок 3.2).

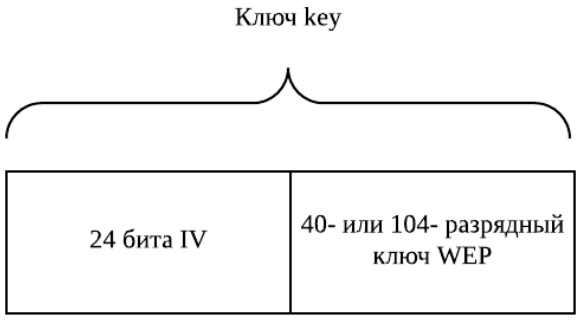

Рисунок 3.2. Ключ *key*

После этого ключ *key* подается на вход RC4 в результате чего генерируется ключевой поток. Этот ключевой поток поразрядно складывается с открытым текстом *P*, в результате получается зашифрованное сообщение *E*. Вектор *IV* дописывается спереди к шифрованному тексту, все это получает заголовок и отправляется получателю.

Получатель, приняв данное сообщение, производит процесс, обратный шифрованию.

- Получатель извлекает из сообщения инициализирующий вектор *IV* и присоединяет к нему свой WEP-ключ, создавая таким образом ключ *key*. Для такого, чтобы значения *key* у отправителя и у получателя были одинаковыми, необходимо, чтобы были одинаковыми их WEP-ключи. Получившийся ключ *key* подается на вход RC4, который сгенерирует тот же ключевой поток, который был у отправителя сообщения. Ключевой поток складывают с оставшейся частью зашифрованного сообщения, таким образом получается исходное открытый текст *P*. С помощью 32-разрядного циклического избыточного сложения получатель вычисляет контрольную сумму получившегося сообщения *M* и сравнивает ее с принятым значением контрольной суммы. Если контрольные суммы совпали, то сообщение расшифровано верно, в противном случае произошли какие-то ошибки при передаче сообщения либо не совпадают WEP-ключи отправителя и получателя.

Существуют слабые значения векторов инициализации *IV* из-за которых информация о секретном ключе попадает в нулевое слово генератора гаммы. Так как один и тот же ключ многократно используется с разными векторами *IV*, то, собрав большое количество сообщений и зная нулевое слово гаммы ключа, можно определить ключ. Это реализовывает атака Флурера, Мантина и Шамира.

3.2 Атака Флурера, Мантина и Шамира

Атака Флурера, Мантина и Шамира — атака на протокол WEP, которая позволяет, зная первое слово ключевого потока, вектор инициализации *IV* и первые *m* слов ключа, злоумышленнику получить *(m+1)*-е слово ключа благодаря слабости

генерации псевдослучайных чисел, которые используются для генерации ключевой последовательности. Существуют слабые значения *IV*, из-за которых информация о секретном ключе попадает в первое слово ключевого потока. Так как один и тот же ключ многократно используется с разными *IV*, собрав большое количество данных со слабыми *IV* и зная нулевое слово генератора гаммы, можно определить ключ.

Слабые *IV* имеют вид $(A+3, N-1, X)$, где A – номер взламываемого *n*-битного слова ключа, $N = 2^n$, X – любое число от 0 до $2^n - 1$. Таким образом, для взлома нулевого слова ключа, то есть $A = 0$, используются 2^n слабых *IV* вида $(3, N-1, X)$. Слова ключа необходимо взламывать по порядку, то есть первое слово нельзя взломать, пока не будет известно нулевое.

3.2.1 Описание алгоритма FMS-атаки

1. Создаются все возможные вектора инициализации $IV = (A+3, N-1, X)$. Они используются как первые три элемента массива *K*. Массив *K* заполняется при помощи конкатенации *IV* и ключа.
2. Для каждого из *IV* для i от 0 до $(A+3)-1$ выполняется алгоритм развертывания ключа KSA. Этот шаг можно осуществить, не зная самого ключа, так как *IV* занимает первые три *n*-битных слова массива *K*.
    - Если на $(A+3)$-м шаге алгоритма KSA $j < 2$, то атака с данным вектором прекращается.
    - Если на $(A+3)$-м шаге алгоритма KSA $j \geq 2$, то выполняем $predictedKey_X \leftarrow (k_A - j - S[A+3]) \bmod 2^n$, где $predictedKey_X$ - возможное *A*-е слово ключа, k_A - *A*-е слово генератора гаммы. Таким образом для каждого вектора получаем свой $predictedKey_X$, который может принимать значения от 0 до $2^n - 1$.

3. Выбираем то значение $predictedKey_X$, которое встречается наибольшее число раз. Оно и является *A*-м словом ключа.

3.2.2 Пример работы алгоритма атаки Флурера, Мантина и Шамира

Для демонстрации примера работы алгоритма атаки Флурера, Мантина и Шамира примем $n = 4$. Тогда массивы *S* и *K* будут иметь длину $2^n = 2^4 = 16$ бит.

Пусть ключ равен (1,2,3,4,5) и взламывается нулевое слово ключа, то есть $A = 0$, и пусть $X = 2$, тогда получим вектор инициализации $IV = (3,15,2)$. Таким образом начальное значение нашего ключа, если бы мы знали ключ, имело бы вид $(3,15,2,1,2,3,4,5)$. Исходя из этого значения нулевым словом оператора гаммы является $k_0 = 9$.

Имеем:

$A = 0$

$k_0 = 9$,

$IV = (3,15,2)$,

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

Таблица 3.1. Заполнение массива *K* (Z – неизвестные значения ключа)

$j = 0$,

Номер элемента *S*-блока	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Значение	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Таблица 3.2. Начальное заполнение *S*-блока

Покажем этапы работы алгоритма KSA:

1. $i = 0$,

   $j = (j + S[i] + K[i]) \bmod 16 = (0 + 3 + 0) \bmod 16 = 3 \bmod 16 = 3$,

   $S[0] \leftrightarrow S[3]$

Номер элемента *S*-блока	**0**	1	2	**3**	4	5	6	7	8	9	10	11	12	13	14	15
Значение	3	1	2	0	4	5	6	7	8	9	10	11	12	13	14	15

Таблица 3.3. Заполнение *S*-блока для $i = 0$

2. $i = 1$,

   $j = (j + S[i] + K[i]) \bmod 16 = (3 + 1 + 15) \bmod 16 = 19 \bmod 16 = 3$,

   $S[1] \leftrightarrow S[3]$

Номер элемента *S*-блока	0	**1**	2	**3**	4	5	6	7	8	9	10	11	12	13	14	15
Значение	3	0	2	1	4	5	6	7	8	9	10	11	12	13	14	15

Таблица 3.4. Заполнение *S*-блока для $i = 1$

3. $i = 2$,

   $j = (j + S[i] + K[i]) \bmod 16 = (3 + 2 + 2) \bmod 16 = 7 \bmod 16 = 7$,

   $S[2] \leftrightarrow S[7]$

Номер элемента *S*-блока	0	1	**2**	3	4	5	6	**7**	8	9	10	11	12	13	14	15
Значение	3	0	7	1	4	5	6	2	8	9	10	11	12	13	14	15

Таблица 3.5. Заполнение *S*-блока для $i = 2$

Получаем $j = 7 > 2$, следовательно возможное нулевое слово ключа равно 1:

$$predictedKey_2 = (k_A - j - S[A+3]) \bmod 2^n = (k_0 - j - S[3]) \bmod 16 = (9 - 7 - 1) \bmod 16 = 1 \bmod 16 = 1.$$

Таким образом находятся все возможные нулевые слова $predictedKey_X$, после чего выбирается то слово, которое встречается наибольшее число раз.

Далее, для нахождения первого слова, выбирается $A = 1$, для каждого вектора *IV* инициализируются массивы *K* и *S* и алгоритм разворачивания ключа KSA повторяется для *i* от 0 до 3. Для каждого вектора находят $predictedKey_X$ и выбирают то слово, которое встретилось наибольшее количество раз. Алгоритм повторяется до тех пор, пока весь ключ не будет восстановлен.

ЗАКЛЮЧЕНИЕ

В результате выполненной работы:

- рассмотрен общий принцип функционирования поточных криптосистем;
- выполнен анализ алгоритмов криптосистемы RC4 с целью определения структур данных и методов их обработки при программной реализации;
- исследована работа протокола WEP;
- исследована работа алгоритма атаки Флурера, Мантина и Шамира;
- разработана и верифицирована динамическая библиотека классов на языке C# для выполнения процедур шифрования и дешифрования по алгоритмам криптосистемы RC4.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Мао, Венбо Современная криптография = Modern Cryptography : теория и практика / Венбо Мао ; [пер. с англ. и ред. Д. А. Клюшина]. - Москва; Санкт-Петербург; Киев: Вильямс, 2005. - 764с.
2. Фергюсон, Нильс Практическая криптография = Practical Cryptography / Нильс Фергюсон, Брюс Шнайер ; [пер. с англ. Н. Н. Селиной ; под ред. А. В. Журавлева]. - Москва; Санкт-Петербург; Киев: Диалектика, 2005. - 422с.
3. Смарт, Н. Криптография / Н. Смарт; пер. с англ. С. А. Кулешова под ред. С. К. Ландо. - Москва: Техносфера, 2006. - 525 с.
4. Scott R. Fluhrer, Itsik Mantin, Adi Shamir Weaknesses in the key scheduling algorithm of RC4 // Lecture notes in computer science. – 2001. – V. 2259. – P. 1—24.
5. Эриксон Дж. Хакинг: искусство эксплойта. 2-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2010. – 512 с.
6. Баричев С.Г. , Серов Р.Е. Основы современной криптографии. Учебное пособие. — М.: Горячая Линия — Телеком, 2006. — 152 с.

ПРИЛОЖЕНИЕ А.

Исходный код динамической библиотеки классов на языке C#

```
namespace RC4
{
    static class SwapElem
    {
        // Функция, которая меняет местами i1-й и i2-й элементы массива
array
        public static void Swap<T>(this T[] array, int i1, int i2)
        {
            T temp = array[i1];
            array[i1] = array[i2];
            array[i2] = temp;
        }
    }
    public class CRC4
    {
        private byte[] S; //S-блок - массив
        private int length; //Размер S-блока - размер массива
        private int ii = 0; //параметр для метода PRGA
        private int jj = 0; //параметр для метода PRGA

        // Конструктор с параметром для создания массива из 2^n элеиентов
        public CRC4(int n)
        {
```

```
            length = 1 << n;
            S = new byte[length];
        }

        // Метод, который реализует заполнение S-блока начальными значениями
        private void FillS()
        {
            for (int i = 0; i < length; i++)
            {
                S[i] = (byte)i;
            }
        }

        // Метод, который реализует алгоритм KSA
        private void KSA(byte[] key)
        {
            FillS();
            int keyLength = key.Length;
            int j = 0;
            for (int i = 0; i < length; i++)
            {
                j = (j + S[i] + key[i % keyLength]) % length;
                S.Swap(i, j);
            }
        }
```

```
// Метод, который реализует алгоритм PRGA
private byte PRGA()
{
    ii = (ii + 1) % length;
    jj = (jj + S[ii]) % length;
    S.Swap(ii, jj);
    int t = (S[ii] + S[jj]) % length;
    return S[t];
}

// Метод зашифровать
public byte[] Encode(byte[] openT, byte[] key)
{
    KSA(key);
    byte[] closeT = new byte[openT.Length];
    for (int m = 0; m < openT.Length; m++)
    {
        closeT[m] = (byte)(openT[m] ^ PRGA());
    }
    return closeT;
}

// Метод дешифровать
public byte[] Decode(byte[] closeT, byte[] key)
{
    KSA(key);
    byte[] openT = new byte[closeT.Length];
```

```
            for (int m = 0; m < openT.Length; m++)
            {
                openT[m] = (byte)(closeT[m] ^ PRGA());
            }
            return openT;
        }
    }
}
```