МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное

учреждение высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт кибернетики

Кафедра информационной безопасности

КУРСОВАЯ РАБОТА

по дисциплине

«Методы программирования»

Тема курсовой работы

«Реализация шифратора на основе алгоритма Threefish в режиме сцепления блоков шифротекста»

Студентка группы ККСО-03-19: Никишина А.А. _____

Руководитель курсовой работы: Кирюхин В.А. _____

Работа представлена к защите «____» _____ 2021 г.

Допущен к защите «____» _____ 2021 г.

Москва 2021

Содержание

1. Введение

В данной работе реализован шифратор Threefish в режиме CBC.
В качестве реализации данного алгоритма шифрования предложен код на ЯП C (см. Приложение 1). Алгоритм работы шифратора и математическая модель также будут описаны ниже.

Threefish — в криптографии симметричный блочный криптоалгоритм, разработанный группой специалистов во главе с автором Blowfish и Twofish, американским криптографом Брюсом Шнайером в 2008 году для использования в хеш-функции Skein и в качестве универсальной замены существующим блочным шифрам. Основными принципами разработки шифра были: минимальное использование памяти, необходимая для использования в хеш-функции устойчивость к атакам, простота реализации и оптимизация под 64-разрядные процессоры.

Особенности данного алгоритма шифрования:

1) Разрабатывался как часть хэш-функции Skein.

2) В отличие от Blowfish и Twofish, не является шифром сети Фейстеля. Реализован в виде подстановочно-перестановочной сети (SP-сети) на обратимых операциях.

3) Предусматривает использование tweak-значения, использующегося как дополнительный ключ, позволяя изменять выходной текст, без изменения ключа и шифр-текста.

4) Имеет 3 различных вариации с различающимися размерами блока (256, 512 и 1024 бит).

5) Имеет нетрадиционно большое число раундов – 72 или 80, в зависимости от выбранного размера блока.

6) Показывает хорошие скоростные характеристики на 64-разрядной архитектуре и значительно более плохие результаты на 32-разрядной.

2. Теоретическая часть

2.1. Структура алгоритма

Threefish имеет очень простую структуру и может быть использован для замены алгоритмов блочного шифрования, будучи быстрым и гибким шифром, работающим в произвольном режиме шифрования. Threefish S-блоки не использует, основан на комбинации инструкций исключающего или, сложения и циклического сдвига.

Как и AES, шифр реализован в виде подстановочно-перестановочной сети на обратимых операциях, не являясь шифром сети Фейстеля.

Алгоритм предусматривает использование tweak-значения, своего рода вектора инициализации, позволяя изменять таким образом значение выхода, без изменения ключа, что имеет положительный эффект как для реализации новых режимов шифрования, так и на криптостойкости алгоритма.

Как результат мнения авторов, что несколько сложных раундов зачастую хуже применения большого числа простых раундов, алгоритм имеет нетрадиционно большое число раундов — 72 или 80 при ключе 1024 бит, однако, по заявлению создателей, его скоростные характеристики опережают AES примерно в два раза. Стоит заметить, что по причине 64-битной структуры шифра, данное заявление имеет место лишь на 64-разрядной архитектуре. Поэтому Threefish, как и Skein, основанный на нём, на 32-разрядных процессорах показывает значительно худшие результаты, чем на «родном» оборудовании.

Ядром шифра является простая функция «MIX», преобразующая два 64-битных беззнаковых числа, в процессе которой происходит сложение, циклический сдвиг (ROL/ROR), и сложение по модулю 2 (XOR).

2.2. Определения

Threefish является блочным алгоритмом симметричного шифрования с дополнительным параметром настройки (tweak-значение). Размер блока дан-

ных, с которым работает алгоритм — 256, 512 или 1024 бит. Длина ключа равна выбранному размеру блока. Размер tweak-значения для любого из размеров блока — 128 бит.

Определим функцию шифрования $E(K, T, P)$ где:

- K — ключ шифрования, строка длиной 32, 64, или 128 байт (256, 512, или 1024 бит).
- T — tweak-значение, строка длиной 16 байт (128 бит).
- P — открытый текст для шифрования, строка длиной равной размеру блока.

Для обработки блока данные представляются в виде массива 64-битных слов (целых чисел от 0 до $2^{64}-1$). Определим N_w как число 64-битных слов в ключе (и в блоке), тогда:

- Ключ шифрования $K \Rightarrow (k_0, k_1, \ldots, k_{N_w-1})$
- Tweak-значение $T \Rightarrow (t_0, t_1)$
- Открытый текст $P \Rightarrow (p_0, p_1, \ldots, p_{N_w-1})$

Количество раундов, N_r для алгоритма Threefish определяется следующим образом:

Длина ключа/блока	N_w	N_r
256 бит	4	72
512 бит	8	72
1024 бит	16	80

2.3. Расписание ключей

Алгоритм использует $\frac{N_r}{4}+1$ раундовых ключей. Дополним основной ключ и tweak-значение двумя 64-битными словами: $k_{N_w} = C_{240} \oplus k_0 \oplus k_1 \oplus \ldots \oplus k_{N_w-1}$, где C_{240} = 0x1bd11bdaa9fc1a22

$t_2 = t_0 \oplus t_1$

Определим подключ s как $(k_{s,0}, k_{s,1}, \ldots, k_{s,N_w-1})$.

Все операции сложения выполняются по модулю 2^{64}.

$$k_{s,i} = \begin{cases} k_{(s+i)mod(N_w+1)} & i = 0, \ldots, N_w - 4 \\ k_{(s+i)mod(N_w+1)} + t_{smod(3)} & i = N_w - 3 \\ k_{(s+i)mod(N_w+1)} + t_{(s+1)mod(3)} & i = N_w - 2 \\ k_{(s+i)mod(N_w+1)} + s & i = N_w - 1 \end{cases}$$

Где $s = 0, 1, \ldots, \frac{N_r}{4}$.

2.4. Функция MIX

Нелинейная функция смешивания и перестановки $\mathrm{MIX}_{d,j}$ принимает на вход два аргумента (x_0, x_1) и возвращает (y_0, y_1):

$y_0 = (x_0 + x_1) \mod 2^{64}$

$y_1 = (x_1 \lll R_{(d \bmod 8),j}) \oplus y_0$

Где $\lll$ — оператор побитового циклического сдвига влево, а константа $R_{d,j}$ определяется из таблицы:

N_w	4		8				16							
$d \backslash j$	0	1	0	1	2	3	0	1	2	3	4	5	6	7
0	14	16	46	36	19	37	24	13	8	47	8	17	22	37
1	52	57	33	27	14	42	38	19	10	55	49	18	23	52
2	23	40	17	49	36	39	33	4	51	13	34	41	59	17
3	5	37	44	9	54	56	5	20	48	41	47	28	16	25
4	25	33	39	30	34	24	41	9	37	31	12	47	44	30
5	46	12	13	50	10	17	16	34	56	51	4	53	42	41
6	58	22	25	29	39	43	31	44	47	46	19	42	44	25
7	32	32	8	35	56	22	9	48	35	52	23	31	37	20

2.5. Зашифрование

Обозначим $V_{d,i}$, $i = 0, 1, \ldots, N_{w-1}$ внутреннее состояние алгоритма для раунда $d = 0, 1, \ldots, N_r - 1$.
Начальное внутреннее состояние $V_{0,i} \Rightarrow (p_0, p_1, \ldots, N_w - 1)$.
Каждый раунд состоит из нескольких этапов. На первом этапе раунда к внутреннему состоянию добавляется раундовый ключ $k_{s,i}$ как указанно ниже:
$e_{d,i} = (V_{d,i} + k_{d/4,i}) \mod 2^{64}$, если $d \mod 4 = 0$
$e_{d,i} = V_{d,i}$, если $d \mod 4 \neq 0$

На следующем этапе раунда используется нелинейная функция $\mathrm{MIX}_{d,j}$:
$(f_{d,2j}, f_{d,2j+1}) = \mathrm{MIX}_{d,j}(e_{d,2j}, e_{d,2j+1})$, для $j = 0, 1, \ldots, \frac{N_w}{2} - 1$
Следующее внутреннее состояние определяется как:
$V_{d+1,i} = f_{d,p(i)}$, для $i = 0, 1, \ldots, N_w - 1$
Функция перестановки 64-битных слов $p(i)$ определена в таблице ниже:

$N_w \backslash i$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	0	3	2	1												
8	2	1	4	7	6	5	0	3								
16	0	9	2	13	6	11	4	15	10	7	12	3	14	5	8	1

После выполнения всех раундов выходом алгоритма является шифротекст c_i:
$c_i \Rightarrow (V_{N_r,i} + k_{N_r/4,i}) \mod 2^{64}$, для $i = 0, 1, \ldots, N_w - 1$.

2.6. Расшифрование

Для алгоритма Threefish процедура расшифрования обратна процедуре шифрования. Раундовые ключи используются в обратном порядке, и каждый раунд состоит из обратных операций. Вместо функции $\mathrm{MIX}_{d,j}$ используется функция $\mathrm{deMIX}_{d,j}$, которая выполняет операции вычитания по модулю 2^{64} и побитовый циклический сдвиг вправо. Этапы каждого раунда расшифрования также выполняются в обратном порядке.

2.7. Безопасность

По заявлению авторов, алгоритм имеет более высокий уровень безопасности, чем AES. Существует атака на 25 из 72 раундов Threefish, в то время как для AES — на 6 из 10. Threefish имеет показатель фактора безопасности 2,9, в свою очередь, AES - всего 1,7[3]

Для достижения полной диффузии шифру Threefish-256 достаточно 9 раундов, Threefish-512 — 10 раундов и Threefish-1024 — 11 раундов. Исходя из этого, 72 и 80 раундов, соответственно, в среднем, обеспечат лучшие результаты, чем существующие шифры.[4]

В то же время алгоритм имеет гораздо более простую структуру и функцию преобразования, однако выполнение 72-80 раундов, по мнению исследователей, обеспечивает необходимую стойкость. Применяемый размер ключа от 256 до 1024 бит сводит на нет возможность полного перебора паролей при так называемой атаке грубой силой (brute force attack) на современном оборудовании.

2.8. Режим шифрования CBC

Режим сцепления блоков шифротекста (англ. Cipher Block Chaining, CBC) — один из режимов шифрования для симметричного блочного шифра с использованием механизма обратной связи. Каждый блок открытого текста (кроме первого) побитово складывается по модулю 2 (операция XOR) с предыдущим результатом шифрования.

Шифрование может быть описано следующим образом:

$C_0 = IV$

$C_i = E_k\left(P_i \oplus C_{i-1}\right)$

где i — номера блоков, IV — вектор инициализации (синхропосылка), C_i и P_i — блоки зашифрованного и открытого текстов соответственно, а E_k — функция блочного шифрования.

Расшифровка:

$P_i = C_{i-1} \oplus D_k\left(C_i\right)$

Особенностями данного режима шифрования являются следующие пункты:

1) Наличие механизма распространения ошибки: если при передаче произойдёт изменение одного бита шифротекста, данная ошибка распространится и на следующий блок. Однако на последующие блоки (через один) ошибка не распространится, поэтому режим CBC также называют самовосстанавливающимся.

2) Неустойчив к ошибкам, связанным с потерей или вставкой битов, если не используется дополнительный механизм выравнивания блоков.

3) Злоумышленник имеет возможность добавить блоки к концу зашифрованного сообщения, дополняя тем самым открытый текст (однако без ключа получается мусор. А с использованием Hash функции для подтверждения целостности сообщения, сводит этот тип атаки на нет.)

4) Для очень крупных сообщений (32 Гбайта при длине блока 64 бита) всё-таки возможно применение атак, основанных на структурных особенностях открытого текста (следствие парадокса дней рождения).

5) Требует дополнения сообщений до длины кратной длине блока.

6) Уязвим для атаки оракула дополнений (англ: Padding Oracle). В случае, если атакующий может отправлять зашифрованные сообщения на расшифрование неограниченное количество раз, он может, изменяя определенные байты, угадать корректное дополнение. Это приводит к возможности расшифровать сообщение за исключением первого блока, создать корректное произвольное зашифрованное сообщение за исключением первого блока без знания ключа.

3. Практическая часть

Программная реализация алгоритма Threefish на языке C состоит из файла Threefish.c (см. Приложение 1). При запуске программа принимает на вход имя файла, ключ в шестнадцатиричном формате и режим работы (c - шифрование файла; d - расшифрование файла; cd - шифрование, а затем и расшифрование файла).

```
gcc Threefish.c
./a.out filename.txt
01020304abcdef05abcdef0102030405abcdef01efcdab0101deadbeef010101 cd
```

(Пример компиляции и запуска программы)

В функции main происходит проверка длинны ключа и определение режима шифрования (256-битный/512-битный/1024-битный). В соответствии с этим определяются значения N_w и N_r.Затем считывается ключ и в функции setup вычисляются раундовые ключи (см. пункт 2.3). Tweak-значение задаётся в коде. В качестве tweak-значения выбраны первые шестнадцатеричные цифры числа π, вычисленные через формулу Бейли-Боруэйна-Плаффа. Значения перестановки (p_256, p_512, p_1024), а также таблицы констант вращения (r_256, r_512, r_1024) взяты из официальной спецификации алгоритмов Skein и Threefish.

После получения раундовых ключей имя файла передаётся в функцию E или D согласно выбранному режиму работы программы. В данных функциях создаётся итоговый файл, а исходный файл дополняется до размера кратного размеру ключа и шифруется/расшифровывается по алгоритму Threefish.

В функции E каждый считанный блок предварительно ксорится с предыдущим шифрованным блоком. Первый блок ксорится с IV - значением, задающимся рандомно (см. пункт 2.8). IV-значение записывается в конец файла по окончании шифрования.

В функции D напротив - сначала происходит считывание IV-значения, затем расшифровка блока, а уже потом ксор с предыдущим считанным шифрованным блоком (Первый блок как и в функции E ксорится с IV-значением).

В функции crypt_threefish происходит шифрование блока данных по алгоритму подробно описанному в теоретической части данной работы (см. пункт 2.5). В функции decrypt_threefish происходит расшифровка шифр-текста по обратному алгориму.

Функция _mix отвечает за нелинейное смешивание и перестановку. Там происходит сложение по модулю 2^{64} побитовый циклический сдвиг влево на значение из таблицы констант вращения, а также сложение по модулю 2 суммы переданных параметров и циклически сдвинутого первого из них. (см. пункт 2.4)

Обратная ей функция _demix производи сначала операцию ксора, а затем побитовый циклический сдвиг вправо и вычитание.

Таким образом для шифрования используются функции setup, E, crypt_threefish, _mix, а для расшифровки setup, D, decrypt_threefish, _demix.

4. Заключение

В данной курсовой работе был подробно рассмотрен и программно реализован алгоритм шифрования Threefish (см. Приложение 1). Были изучены тонкости работы с битовыми операциями на C, а также все этапы шифрования алгоритма Threefish и режим шифрования CBC в частности. Весь программный код был реализован без использования криптографических библиотек.

Также в процессе изучения спецификации алгоритма Threefish я в общих чертах ознакомилась с алгоритмом хеширования переменной разрядности Skein.

Помимо реализации алгоритма шифрования я написала программу (см. Приложение 2) для сравнения побайтовой энтропии исходного файла и шифрованного, а также построила цепь Маркова и посчитала энтропию биграмм. В результате проведённой работы для файла с английским текстом размера 101,9 кБ - энтропия исходного файла $H_1 = 4.515899$ и энтропия цепи Маркова $H(E_1) = 0.461129$. А для зашифрованного файла $H_2 = 7.998098$ и энтропия цепи Маркова из биграмм $H(E_2) = 0.060570$.

Учитывая, что мы использовали алфавит мощностью 256 символов. Можем рассчитать абсолютную энтропию для данного алфавита $H_{max} = \log_2(N) = \log_2(256) = 8$.

Для английского языка максимальная энтропия приблизительно равна ≈ 4.7. Делая поправку на то что мы использовали алфавит больше, чем английский можем сделать вывод, что для исходного текста значения близки к правдоподобным. А учитывая близость энтропии зашифрованного текста к максимальной для алфавита в 256 символов можем сделать вывод, что данный алгоритм шифрования достаточно надёжный.

Данное утверждение подтверждается и энтропией цепи Маркова. Которая говорит нам о том, что символы в шифрованном тексте попарно почти независимы, в отличии от исходного текста имеющего такую зависимость.

Исходя из проверки энтропийных характеристик шифра можем сделать вывод, что шифр Threefish в режиме CBC является надёжной защитой данных.

5. Список литературы

1) «The Skein Hash Function Family» - Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, Jesse Walker.
(Официальная спецификация алгоритмов Skein и Threefish)

2) https://www.schneier.com/academic/skein/

3) http://www.pgpru.com/novosti/2008/heshfunkcijaskeiniblochnyjjshifrthreefish

4) https://github.com/wernerd/Skein3Fish

5) https://lhs-blog.info/programming/dlang/realizacziya-blochnogo-kriptoalgoritma-threefish/

6) https://lhs-blog.info/programming/ shifrovanie-v-rezhime-cbc-s-pomoshhyu-threefish512/

7) https://ru.wikipedia.org/wiki/Threefish

8) https://xakep.ru/2016/06/30/crypto-part4/#toc09.

9) https://habr.com/ru/post/531140/

6. Приложения

Приложение 1: Программная реализация алгоритма шифрования Threefish в режиме сцепления блоков шифротекста на языке программирования C.

```
     1 #include <fcntl.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <sys/types.h>
6 #include <unistd.h>
7 #include <time.h>
8 #include <limits.h>
9
10 uint Nw;
11 uint Nr;
12 uint Ns;
13
14 uint p_256[4] = {0, 3, 2, 1};
15 uint p_512[8] = {2, 1, 4, 7, 6, 5, 0, 3};
16 uint p_1024[16] = {0, 9, 2, 13, 6, 11, 4, 15,
   10, 7, 12, 3, 14, 5, 8, 1};
17
18 uint p_1_256[4] = {0, 3, 2, 1};
19 uint p_1_512[8] = {6, 1, 0, 7, 2, 5, 4, 3};
20 uint p_1_1024[16] = {0, 15, 2, 11, 6, 13, 4, 9,
   14, 1, 8, 5, 10, 3, 12, 7};
21
22 uint r_256[8][2] = {
23     {14, 16},
24     {52, 57},
25     {23, 40},
26     {5, 37},
```

```
27     {25, 33},
28     {46, 12},
29     {58, 22},
30     {32, 32}
31 };
32
33 uint r_512[8][4] = {
34     {46, 36, 19, 37},
35     {33, 27, 14, 42},
36     {17, 49, 36, 39},
37     {44, 9, 54, 56},
38     {39, 30, 34, 24},
39     {13, 50, 10, 17},
40     {25, 29, 39, 43},
41     {8, 35, 56, 22},
42 };
43
44 uint r_1024[8][8] = {
45     {24, 13, 8, 47, 8, 17, 22, 37},
46     {38, 19, 10, 55, 49, 18, 23, 52},
47     {33, 4, 51, 13, 34, 41, 59, 17},
48     {5, 20, 48, 41, 47, 28, 16, 25},
49     {41, 9, 37, 31, 12, 47, 44, 30},
50     {16, 34, 56, 51, 4, 53, 42, 41},
51     {31, 44, 47, 46, 19, 42, 44, 25},
52     {9, 48, 35, 52, 23, 31, 37, 20}
53 };
54
55 ulong t[3];
56
57 ulong _mix(ulong y[2], ulong r, ulong x[2]){
58     y[0] = x[0] + x[1];
```

```
59      y[1] = (x[1] << r) | (x[1] >> (64 - r));
60      y[1] ^= y[0];
61  }
62
63  ulong _demix(ulong y[2], ulong r, ulong x[2]){
64      y[1] ^= y[0];
65      x[1] = (y[1] << (64 - r)) | (y[1] >> r);
66      x[0] = y[0] - x[1];
67  }
68
69  void crypt_threefish(ulong* plain, ulong* c, ulong** subKeys){
70      uint round;
71      ulong f[Nw];
72      ulong e[Nw];
73      ulong* y = (ulong*)malloc(2*sizeof(ulong));
74      ulong* x = (ulong*)malloc(2*sizeof(ulong));
75      ulong v[Nw];
76      uint s;
77      uint i;
78
79      memcpy (&v[0], plain, Nw*8);
80
81
82      for (round = 0; round < Nr; round++)
83      {
84          if (round % 4 == 0)
85          {
86              s = round / 4;
87              for (i = 0; i < Nw; i++)
88              {
89                  e[i] = v[i] + subKeys[s][i];
90              }
```

```
91          }
92          else
93          {
94              for (i = 0; i < Nw; i++)
95              {
96                  e[i] = v[i];
97              }
98          }
99
100          for (i = 0; i < Nw / 2; i++)
101          {
102              x[0] = e[i * 2];
103              x[1] = e[i * 2 + 1];
104
105              if(Nw == 4) //key 256 bit
106                  _mix(y, r_256[round % 8][i], x);
107              else if (Nw == 8) //key 512 bit
108                  _mix(y, r_512[round % 8][i], x);
109              else if (Nw == 16) //key 1024 bit
110                  _mix(y, r_1024[round % 8][i], x);
111              else{
112                  printf("error crypt_threefish");
113                  return;
114              }
115
116              f[i * 2] = y[0];
117              f[i * 2 + 1] = y[1];
118          }
119
120          for (i = 0; i < Nw; i++)
121          {
122              if(Nw == 4) //key 256 bit
```

```
123                 v[i] = f[p_256[i]];
124             else if (Nw == 8) //key 512 bit
125                 v[i] = f[p_512[i]];
126             else if (Nw == 16) //key 1024 bit
127                 v[i] = f[p_1024[i]];
128             else{
129                 printf("error crypt_threefish");
130                 return;
131             }
132         }
133     }
134
135     for (i = 0; i < Nw; i++)
136     {
137         c[i] = v[i] + subKeys[Nr / 4][i];
138     }
139
140     free(x);
141     free(y);
142 }
143
144 void decrypt_threefish(ulong* plain, ulong* c, ulong** subKeys)
145 {
146     uint round;
147     ulong f[Nw];
148     ulong e[Nw];
149     ulong* y = (ulong*)malloc(2*sizeof(ulong));
150     ulong* x = (ulong*)malloc(2*sizeof(ulong));
151     ulong v[Nw];
152     uint s;
153     uint i;
154
```

```
155     memcpy(&v[0], plain, Nw*8);
156
157     for (round = Nr; round > 0; round--)
158     {
159         if (round % 4 == 0)
160         {
161             s = round / 4;
162             for (i = 0; i < Nw; i++)
163             {
164                 f[i] = v[i] - subKeys[s][i];
165             }
166         }
167         else
168         {
169             for (i = 0; i < Nw; i++)
170             {
171                 f[i] = v[i];
172             }
173         }
174
175         for (i = 0; i < Nw; i++)
176         {
177             if(Nw == 4) //key 256 bit
178                 e[i] = f[p_1_256[i]];
179             else if (Nw == 8) //key 512 bit
180                 e[i] = f[p_1_512[i]];
181             else if (Nw == 16) //key 1024 bit
182                 e[i] = f[p_1_1024[i]];
183             else{
184                 printf("error decrypt_threefish");
185                 return;
186             }
```

```
187         }
188
189         for (i = 0; i < Nw / 2; i++)
190         {
191             y[0] = e[i * 2];
192             y[1] = e[i * 2 + 1];
193
194             if(Nw == 4) //key 256 bit
195                 _demix(y, r_256[(round - 1) % 8][i], x);
196             else if (Nw == 8)  //key 512 bit
197                 _demix(y, r_512[(round - 1) % 8][i], x);
198             else if (Nw == 16) //key 1024 bit
199                 _demix(y, r_1024[(round - 1) % 8][i], x);
200             else{
201                 printf("error decrypt_threefish");
202                 return;
203             }
204
205             v[i * 2] = x[0];
206             v[i * 2 + 1] = x[1];
207         }
208     }
209
210     for (i = 0; i < Nw; i++)
211     {
212         c[i] = v[i] - subKeys[0][i];
213     }
214     free(x);
215     free(y);
216 }
217
218 void setup(ulong* keyData, ulong* tweakData, ulong** subKeys)
```

```
219 {
220     uint i, round;
221     ulong K[Nw];
222     ulong T[2];
223     ulong key[Nw+1];
224 
225     ulong kNw = 0x1bd11bdaa9fc1a22;
226 
227     memcpy(&K[0], &keyData[0], Nw*8);
228     memcpy(&T[0], &tweakData[0], 16);
229 
230     for (i = 0; i < Nw; i++)
231     {
232         kNw ^= K[i];
233         key[i] = K[i];
234     }
235 
236     key[Nw] = kNw;
237 
238     t[0] = T[0];
239     t[1] = T[1];
240     t[2] = T[0] ^ T[1];
241 
242     for (round = 0; round <= (Nr / 4); round++)
243     {
244         for (i = 0; i < Nw; i++)
245         {
246             subKeys[round][i] = key[(round + i) % (Nw + 1)];
247 
248             if (i == (Nw - 3)){
249                 subKeys[round][i] += t[round % 3];
250             }
```

```
251             else if (i == (Nw - 2)){
252                 subKeys[round][i] += t[(round + 1) % 3];
253             }
254             else if (i == (Nw - 1)){
255                 subKeys[round][i] += round;
256             }
257         }
258     }
259 }
260
261 void D(char* file_name, ulong* keyData, ulong* tweakValue,
    ulong** subKeys){
262     FILE* fd_crypt;
263     char file_crypt[strlen(file_name)+12];
264     strcpy(file_crypt, file_name);
265     strcat(file_crypt, "_decrypt.txt");
266
267     if ((fd_crypt = fopen(file_crypt, "w+")) == NULL){
268         printf("Не удалось открыть файл %s\n", file_crypt);
269         return;
270     }
271
272
273     int fp;
274     if ((fp = open(file_name, O_RDWR)) == -1){
275         printf("Не удалось открыть файл %s\n", file_name);
276         return;
277     }
278     ulong size_file = lseek(fp, 0, SEEK_END) - Nw*sizeof(ulong);
        //узнали длинну файла
279
280     lseek(fp, Nw*sizeof(ulong), SEEK_END);
```

```
281     close(fp);
282
283     if(size_file%(64*Nw)!=0){// Проверяем размер
        зашифрованного файла
284         printf("Данный файл не был зашифрован этой программой.
            Размерность файла не кратна размеру ключа\n");
285         return;
286     }
287
288     FILE* fd_original;
289     if ((fd_original = fopen(file_name, "rt")) == NULL){
290         printf("Не удалось открыть файл %s\n", file_name);
291         return;
292     }
293
294     if(fseek(fd_original, size_file, SEEK_SET) != 0){
295         printf("Ошибка каретки\n");
296         return;
297     }
298     ulong* IV = (ulong*)malloc(Nw*sizeof(ulong));
299     if(fread(IV, sizeof(ulong), Nw, fd_original) != Nw){
300         printf("Не удалось считать символы из файла %s в массив
            IV\n", file_name);
301         return;
302     }
303     lseek(fp, 0, SEEK_SET);
304
305
306     ulong* plainData = (ulong*)malloc(Nw*sizeof(ulong));
307     ulong* result = (ulong*)malloc(Nw*sizeof(ulong));
308     ulong* before_block = (ulong*)malloc(Nw*sizeof(ulong));
309
```

```
310     for (int count = 0; count < (size_file/(Nw*8)); count++) {
311         if(fread(plainData, sizeof(ulong), Nw, fd_original) != Nw){
312             printf("Не удалось считать символы из файла %s\n",
                file_name);
313             return;
314         }
315
316         for (int i = 0; i < Nw; i++){
317             result[i] = 0;
318         }
319         decrypt_threefish(plainData, result, subKeys);
320
321         if(count!=0){
322             for (int i = 0; i < Nw; i++){
323                 result[i] ^= before_block[i];
324             }
325         }
326         else {
327             for (int i = 0; i < Nw; i++){
328                 result[i] ^= IV[i];
329             }
330         }
331
332
333         if(fwrite(result, sizeof(ulong), Nw, fd_crypt) != Nw){
334             printf("Не удалось записать символы в файл %s\n",
                file_crypt);
335             return;
336         }
337
338         for (int i = 0; i < Nw; i++){
339             before_block[i] = plainData[i];
```

```
340         }
341     }
342 
343     printf("Вжух! Расшифрование готово\n");
344 
345     fclose(fd_original);
346     fclose(fd_crypt);
347 
348     free(plainData);
349     free(result);
350     free(before_block);
351     free(IV);
352 }
353 
354 
355 void E(char* file_name, ulong* keyData, ulong* tweakValue,
    ulong** subKeys){
356     FILE* fd_crypt;
357     char file_crypt[strlen(file_name)+10];
358     strcpy(file_crypt, file_name);
359     strcat(file_crypt, "_crypt.txt");
360 
361     if ((fd_crypt = fopen(file_crypt, "w+")) == NULL){
362         printf("Не удалось открыть файл %s\n", file_crypt);
363         return;
364     }
365 
366 
367     int fp;
368     if ((fp = open(file_name, O_RDWR)) == -1){
369         printf("Не удалось открыть файл %s\n", file_name);
370         return;
```

```
371     }
372     ulong size_file = lseek(fp, 0, SEEK_END);
        //узнали длинну файла
373
374     if(size_file%(64*Nw)!=0){// если длинна не кратна размеру
        ключа, то дописываем символы
375         printf ("+\n");
376         char str_null[(64*Nw)-(size_file%(64*Nw))];
377         printf ("Дополняем файл...\n");
378         for(int i=0; i<(64*Nw)-(size_file%(64*Nw)); i++)
379             str_null[i]=0;
380         write (fp, str_null, (64*Nw)-(size_file%(64*Nw)));
381
382         lseek(fp, 0, SEEK_SET);
383         size_file = lseek(fp, 0, SEEK_END);
384         printf ("+\n\n");
385     }
386     close(fp);
387     FILE* fd_original;
388     if ((fd_original = fopen(file_name, "rt")) == NULL){
389         printf("Не удалось открыть файл %s\n", file_name);
390         return;
391     }
392
393
394     ulong* plainData = (ulong*)malloc(Nw*sizeof(ulong));
395     ulong* result = (ulong*)malloc(Nw*sizeof(ulong));
396     ulong* before_block = (ulong*)malloc(Nw*sizeof(ulong));
397     ulong* IV = (ulong*)malloc(Nw*sizeof(ulong));
398
399     srand(time(NULL));
400
```

```
401     for (int count = 0; count < (size_file/(Nw*8)); count++){
402         if(fread(plainData, sizeof(ulong), Nw, fd_original)
        != Nw){
403             printf("Не удалось считать символы из файла %s\n",
            file_name);
404             return;
405         }
406
407         if(count!=0){
408             for (int i = 0; i < Nw; i++){
409                 plainData[i] ^= before_block[i];
410             }
411         }
412         else {
413             for (int i = 0; i < Nw; i++){
414                 IV[i] = rand()%ULONG_MAX;
415                 plainData[i] ^= IV[i];
416             }
417         }
418
419         for (int i = 0; i < Nw; i++){
420             result[i] = 0;
421         }
422         crypt_threefish(plainData, result, subKeys);
423
424         if(fwrite(result, sizeof(ulong), Nw, fd_crypt) != Nw){
425             printf("Не удалось записать символы в файл %s\n",
            file_crypt);
426             return;
427         }
428
429         for (int i = 0; i < Nw; i++){
```

```
430             before_block[i] = result[i];
431         }
432     }
433     if(fwrite(IV, sizeof(ulong), Nw, fd_crypt) != Nw){
        // записываем IV в конец файла
434         printf("Не удалось записать символы в файл %s\n",
            file_crypt);
435         return;
436     }
437
438     printf("Вжух! Шифрование готово\n");
439
440     fclose(fd_original);
441     fclose(fd_crypt);
442
443     free(plainData);
444     free(result);
445     free(before_block);
446     free(IV);
447 }
448
449
450 int    main(int argc, char *argv[]){// ./a.out filename key mode
451     if(strlen(argv[2]) == 4*16){ //key 256 bit
452         Nw = 4;
453         Nr = 72;
454     }
455     else if(strlen(argv[2]) == 8*16){ //key 512 bit
456         Nw = 8;
457         Nr = 72;
458     }
459     else if(strlen(argv[2]) == 16*16){ //key 1024 bit
```

```
460         Nw = 16;
461         Nr = 80;
462     }
463     else{
464         printf("Некорректная длинна ключа %ld\n", strlen(argv[2]));
465         return 0;
466     }
467 
468     Ns = (Nr / 4) + 1;
469 
470     ulong** subKeys = (ulong**)malloc(Ns*sizeof(ulong*));
471     for (int i = 0; i < Ns; i++){
472         subKeys[i] = (ulong*)malloc(8*sizeof(ulong));
473     }
474 
475     ulong tweakValue[2] = {0x243f6a8885a308d3, 0x13198a2e03707344};
476 
477     ulong* keyData = (ulong*)malloc(Nw*sizeof(ulong));
478 
479     for (int i = 0; i < Nw; i++){
480         keyData[i] = 0;
481         for (int j = 0; j < 4; j++){
482             keyData[i]<<=4;
483             if(argv[2][j+i*16]>='0' && argv[2][j+i*16]<='9')
484                 keyData[i]+=argv[2][j+i*16]-'0';
485             else if (argv[2][j+i*16]>='a' && argv[2][j+i*16]<='f')
486                 keyData[i]+=argv[2][j+i*16]-'a'+10;
487             else if (argv[2][j+i*16]>='A' && argv[2][j+i*16]<='F')
488                 keyData[i]+=argv[2][j+i*16]-'A'+10;
489             else{
490                 printf("Некорректный ключ.Введите значения ключа
                    в шестнадцатиричном формате\n");
```

```
491                 return 0;
492             }
493         }
494     }
495
496     printf("\nШифрование будет производиться с размером блока
        %u бит за %u раунда(ов)\n\n", Nw*64, Nr);
497
498     setup(keyData, tweakValue, subKeys); //считаем раундовые ключи
499
500     if (argv[3][0] == 'c' && argv[3][1] == 'd'){
501         E(argv[1], keyData, tweakValue, subKeys);
502
503         char file_crypt[strlen(argv[1])+10];
504         strcpy(file_crypt, argv[1]);
505         strcat(file_crypt, "_crypt.txt");
506         D(file_crypt, keyData, tweakValue, subKeys);
507     }
508     else if (argv[3][0] == 'c'){
509         E(argv[1], keyData, tweakValue, subKeys);
510     }
511     else if (argv[3][0] == 'd'){
512         D(argv[1], keyData, tweakValue, subKeys);
513     }
514     else {
515         printf("Введите режим работы\nc - шифрование\n
            d - расшифрование\ncd - шифрование + расшифрование
            файла\n");
516     }
517
518     for (int i = 0; i < Ns; i++){
519         free(subKeys[i]);
```

```
520     }
521     free(subKeys);
522     free(keyData);
523
524     return 0;
525 }
```

Приложение 2: Программная реализация подсчёта энтропии открытого текста и шифрованного с помощью алгоритма шифрования Threefish в режиме сцепления блоков шифротекста на языке программирования C.

```
1 #include <fcntl.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <stdint.h>
5 #include <string.h>
6 #include <math.h>
7
8
9 void entropy(char* buf_file_name){
10     printf("\n\n\t\ttype = %s\n\n\n", buf_file_name);
11     long double arr_symbols[256];
12     long double bigram[256][256];
13     u_int64_t count = 0;
14     for (int i = 0; i < 256; i++){
15         arr_symbols[i]=0;
16         for (int j = 0; j < 256; j++){
17             bigram[i][j]=0;
18         }
19
20     }
```

```
21
22      int fd_count;
23      fd_count = open(buf_file_name, O_RDONLY);
24      if (fd_count == -1)
25          perror("open failed on input file");
26      int size = lseek(fd_count, 0, SEEK_END);
27      close(fd_count);
28
29      FILE *fd;
30      if((fd = fopen(buf_file_name, "r"))==NULL){
31          perror("open failed on input file");
32      }
33      int ch1, ch;
34      for (int j=0; j<size; j++) {
35          ch = fgetc(fd);
36          arr_symbols[ch]++;
37          if(count > 0){
38              bigram[ch1][ch]++;
39          }
40          ch1 = ch;
41          count++;
42      }
43
44      fclose(fd);
45      printf("number of bytes = %d\n", size);
46
47
48      long double H = 0, H_bigram = 0;
49      for (int i = 0; i < 256; i++){
50          if(arr_symbols[i]!=0){
51              arr_symbols[i] /= (double)count;
52              H -= arr_symbols[i]*log2(arr_symbols[i]);
```

```
53          }
54
55          long double buf_H = 0;
56          for (int j = 0; j < 256; j++){
57              if (bigram[i][j]!=0){
58                  bigram[i][j] /= (double)count;
59                  buf_H -= bigram[i][j]*log2(bigram[i][j]);
60              }
61          }
62          H_bigram += buf_H*arr_symbols[i];
63      }
64      printf("H = %Lf\nH(E) = %Lf\n", H, H_bigram);
65
66  }
67
68  int main(int argc, const char** argv) {
69      entropy(argv[1]);
70      entropy(argv[2]);
71      return 0;
72  }
```