**МОЛДАВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет Математики и Информатики**

**Департамент Информатики**

Аттестация №1

Криптография

Сети фейстеля, хэш алгоритм tiger.

Проверил: профессор Cerbu Olga

Выполнил: Mamaliga Artur grupa I2302-1

Кишинев, 2024

1. Введение  
   1.1 Цель работы

Описать сети Фейстеля и создать программу для хэширования с использованием алгоритма Tiger. Показать на примере созданной программы один раунд хэширования указывая побитово изменения в изначальном сообщении.

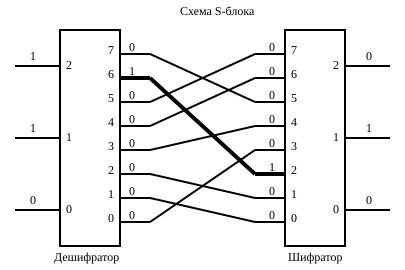
* 1. Структура отчета  
     Цель работы  
     Что такое сеть Фейстеля?  
     Примеры сетей Фейстеля  
     Описание алгоритма tiger  
     Процесс создания хэша  
     Задача хэша  
     Структура программы  
     Тестирование программы  
     Вывод по проделанной работе  
     Приложения

1. Теоретическая часть

2.1 Что такое сеть Фейстеля

|  |  |
| --- | --- |
| Сеть Фейстеля является методом для создания блочных шифров. Она представляет собой систему, состоящую из ячеек, через которые проходят данные и ключ. Каждая ячейка принимает на вход определенный блок данных и ключ, а на выходе выдает измененные данные и модифицированный ключ. Все ячейки имеют одинаковую структуру, и работа сети основана на многократном повторении этих операций. Ключи меняются от одной ячейки к другой в зависимости от алгоритма шифрования или расшифрования. При этом сам процесс шифрования и расшифрования идентичен, отличие заключается лишь в порядке применения ключей. Благодаря простоте выполняемых операций сеть Фейстеля легко реализуется как на уровне программного обеспечения, так и аппаратно. Этот подход используется в ряде известных блочных шифров, таких как DES. | Conor Deegan | Feistel Network in Python |

2.1.1 Основные принципы работы сетей Фейстеля  
Основными принципами работы сетей Фейстеля являются действия выполненные поэтапно, начинается все с разбиения данных на блоки, входные данные разделяются на два равных блока так называемые “левый” и “правый” блок. Далее этап итеративного преобразования, процесс шифрования повторяемый несколько раз, в котором соблюдается одинаковая последовательность операций, на каждом раунде один из блоков проходит изменение при помощи подключа а другой передается в следующую итерацию без изменений. На каждом раунде применяется сложение по модулю 2 (XOR) подблока с подключем. Блоки меняются местами и последовательность операций повторяется, это происходит на каждом раунде(итерации), что обеспечивает криптостойкость. Для расшифровки используется тот же процесс, однако подключи подаются в обратном порядке.

2.1.2 Функции используемые в сетях Фейстеля  
В оригинальной работе автора описываются два блока преобразований:  
 блок подстановок s-block  
 блок перестановок p-block  
s-блок реализуем как аппаратно с использованием специальных микросхем:  


Так и программно при помощи таблиц. Процесс подстановки заключается в использовании части шифруемых данных для получения других данных используя нелинейные преобразования

Пример таблицы замены для приведенного 3-разрядного s-блока  
№ комбинации 0 1 2 3 4 5 6 7

Вход 000 001 010 011 100 101 110 111

Выход 011 000 001 100 110 111 010 101

|  |  |
| --- | --- |
| p-блок всего лишь изменяет положение битов и является линейным устройством. Для реализации p-блока необходимо выполнить циклический сдвиг |  |

2.2 Пример использования сетей Фейстеля на алгоритме DES

|  |  |  |
| --- | --- | --- |
| s-block 6b | s-block res 4b | s-block res joined into 32b |
| [+] S block: 1 00011000  [+] S block: 2 11000000  [+] S block: 3 00110000  [+] S block: 4 01001100  [+] S block: 5 11110000  [+] S block: 6 11110000  [+] S block: 7 11001100  [+] S block: 8 01001000 | raw:0 col:3 res:1 [+] Sblock 4b 1: 00000001  raw:2 col:8 res:5 [+] Sblock 4b 2: 00000101  raw:0 col:6 res:15 [+] Sblock 4b 3: 00001111  raw:1 col:9 res:7 [+] Sblock 4b 4: 00000111  raw:2 col:14 res:0 [+] Sblock 4b 5: 00000000  raw:2 col:14 res:11 [+] Sblock 4b 6: 00001011  raw:3 col:9 res:5 [+] Sblock 4b 7: 00000101  raw:0 col:9 res:9 [+] Sblock 4b 8: 00001001 | [+] Sblock joined to 32b: 0001 0101 1111 0111 0000 1011 0101 1001Function S-box S1 (in the DES algorithm) represented as a table and its...  | Download Scientific Diagram |

1. Алгоритм хэширования Tiger

3.1 Введение в хэширование  
Хэширование — это процесс преобразования данных произвольного размера в строку фиксированной длины, которая называется хэш-кодом или дайджестом. Алгоритмы хэширования широко используются в различных областях, таких как обеспечение целостности данных, криптография, хранение паролей, создание цифровых подписей, а также для быстрых операций поиска и индексации в базах данных.

Основная цель хэширования — получить уникальный результат (хэш-код) для каждого набора данных.

3.2.1 Описание алгоритма tiger

Tiger является криптографической хэш-функцией, которая преобразует входные данные произвольного размера в фиксированный 192-битный (24 байта) хэш-код. Основными особенностями алгоритма являются простота реализации, высокая скорость выполнения и устойчивость к коллизиям, что делает его подходящим для использования в цифровых подписях, криптографических протоколах и системах контроля целостности данных.

3.2.2 Принцип работы алгоритма   
Подготовка данных: Входные данные разбиваются на блоки по 512 бит, и если последний блок меньше 512 бит, он дополняется до необходимого размера с использованием стандартного подхода дополнения (например, добавление единичного бита, а затем нулей).  
Начальные значения: Алгоритм использует три 64-битных регистра (A, B, C) для хранения промежуточных значений, которые инициализируются заранее определенными константами.  
Основной цикл: Каждый блок данных проходит через основной цикл преобразований, который включает использование нелинейных замен (S-блоков), побитовых операций (например, XOR), сдвигов и сложений по модулю 2. Алгоритм Tiger включает три прохода по данным, каждый из которых усиливает стойкость хэш-функции к криптоаналитическим атакам.  
Финальные вычисления: После обработки всех блоков данные из регистров (A, B, C) объединяются, формируя итоговый 192-битный хэш-код.

|  |
| --- |
|  |
| Диаграмма раунда преобразования tiger |

3.2.3 Основные преимущества tiger  
Высокая производительность: Tiger был разработан с учетом архитектуры 64-битных процессоров, что обеспечивает его быструю работу на таких системах.  
Криптостойкость: Алгоритм устойчив к атакам на коллизии, что делает его надежным инструментом для обеспечения целостности данных.

Эффективность: Tiger показывает хорошие результаты в условиях ограниченных вычислительных ресурсов, что делает его подходящим для встраиваемых систем.

1. Разработка алгоритма Tiger с примерами

4.1.1 Постановка задачи

Создать алгорит хэширования при помощи языка программирования C++ без использования криптографических библиотек.

4.1.2 Описание алгоритма одного раунда хэширования  
Выбираем первый 512 битный блок и делим на 8 слов по 64 бита длинной

X0,X1,…,X7

Далее берутся три 64 битных регистра и инициализируются в определенных значениях

Допустим это значения h0

\_a = 0x0123456789ABCDEF;

\_b = 0xFEDCBA9876543210;

\_c = 0xF096A5B4C3B2E187;

для выполнения одного раунда хэширования выполняются следующие операции  
save\_abc  
pass(a, b, c,5)  
key\_schedule  
pass(c, a, b,7)  
key\_schedule  
pass(b, c, a,9)  
feedforward

В операции save\_abc происходит сохранение значения регистров хэша в буффер

aa = \_a;  
bb = \_b;  
cc = \_c;

операция pass(a, b, c, multiplier) выполняет следующие действия над каждым блоком из входных данных  
round(a, b, c, 0, mul);

round(b, c, a, 1, mul);

round(c, a, b, 2, mul);

round(a, b, c, 3, mul);

round(b, c, a, 4, mul);

round(c, a, b, 5, mul);

round(a, b, c, 6, mul);

round(b, c, a, 7, mul);

где в свою очередь операция round выполняет следующие действия

c ^= \_x[index];

a -= sboxes::t1[c\_arr[0]] ^ sboxes::t2[c\_arr[2]] ^ sboxes::t3[c\_arr[4]] ^ sboxes::t4[c\_arr[6]];

b -= sboxes::t4[c\_arr[1]] ^ sboxes::t3[c\_arr[3]] ^ sboxes::t2[c\_arr[5]] ^ sboxes::t1[c\_arr[7]];

b \*= mul;

где c\_arr[i] означает i-й байт входного С длинной в 64 бита.

Операция key\_schedule генерирует ключ, необходимая для того чтобы даже изменение небольшого числав бит сообщения x вызывало изменения большего числа бит на следующем выполнении pass (лавинный эффект).

\_x[0] -= \_x[7] ^ 0xA5A5A5A5A5A5A5A5;

\_x[1] ^= \_x[0];

\_x[2] += \_x[1];

\_x[3] -= \_x[2] ^ ((~\_x[1]) << 19);

\_x[4] ^= \_x[3];

\_x[5] += \_x[4];

\_x[6] -= \_x[5] ^ ((~\_x[4]) >> 23);

\_x[7] ^= \_x[6];

\_x[0] += \_x[7];

\_x[1] -= \_x[0] ^ ((~\_x[7]) << 19);

\_x[2] ^= \_x[1];

\_x[3] += \_x[2];

\_x[4] -= \_x[3] ^ ((~\_x[2]) >> 23);

\_x[5] ^= \_x[4];

\_x[6] += \_x[5];

\_x[7] -= \_x[6] ^ 0x0123456789ABCDEF;

Операция feedforward сохраняет результат раунда в изначальные регистры для использования полученных значений в следующей итерации хэширования

\_a ^= aa;

\_b -= bb;

\_c += cc;

Всего мы получает 24 раунда, конкатенация полученных значений в регистрах \_a \_b \_c дает результат хэш функции и представляет собой 192битное значение.

* 1. Пример работы хэш алгоритма

Эта часть объясняет один конкретный раунд алгоритма хеширования Tiger на основе вывода программы. Алгоритм Tiger делит данные на блоки и обрабатывает их через несколько раундов, применяя криптографические операции на каждом этапе. Ниже рассмотрены изменения данных на каждом шаге первого раунда, сосредоточив внимание на значениях регистров, операциях и том, как изменяются биты.

Шаг 1: Начальные значения

В начале процесса хеширования три основных регистра `\_a`, `\_b` и `\_c` инициализируются фиксированными значениями. Эти значения используются в качестве отправной точки для преобразования входных данных:

**\_a: 0000000100100011010001010110011110001001101010111100110111101111**

**\_b: 1111111011011100101110101001100001110110010101000011001000010000**

**\_c: 1111000010010110101001011011010011000011101100101110000110000111**

Шаг 2: Обработка блока данных

Входная строка «AM» преобразуется в 512-битный блок. Первый блок \_x[0] содержит двоичное представление символов «A» и «M», тогда как оставшиеся блоки, с \_x[1] до \_x[7], заполняются нулями.

**\_x[0]: 0000000000000000000000000000000000000000000000000100000101001101**

**\_x[7]: 0000000000000000000000000000000000000000000000000000000000000000**

Шаг 3: Сохранение промежуточных значений

Перед началом первого раунда текущие значения регистров `\_a`, `\_b` и `\_c` сохраняются для дальнейшего использования в процессе обратной связи (feedforward).

**Saved \_a: 0000000100100011010001010110011110001001101010111100110111101111**

**Saved \_b: 1111111011011100101110101001100001110110010101000011001000010000**

**Saved \_c: 1111000010010110101001011011010011000011101100101110000110000111**

Шаг 4: Извлечение байтов из регистра `\_c`

Далее значение регистра `\_c` разбивается на массив из восьми байтов. Каждый байт представлен в двоичной форме:

**c\_arr[0]: 0000000000000000000000000000000000000000000000000000000010000111**

**c\_arr[1]: 0000000000000000000000000000000000000000000000000000000011100001**

**c\_arr[2]: 0000000000000000000000000000000000000000000000000000000010110010**

**c\_arr[3]: 0000000000000000000000000000000000000000000000000000000011000011**

**c\_arr[4]: 0000000000000000000000000000000000000000000000000000000010110100**

**c\_arr[5]: 0000000000000000000000000000000000000000000000000000000010100101**

**c\_arr[6]: 0000000000000000000000000000000000000000000000000000000010010110**

**c\_arr[7]: 0000000000000000000000000000000000000000000000000000000011110000**

Шаг 5: Вычисления первого раунда

В этом раунде к регистрам `\_a`, `\_b` и `\_c` применяются серия криптографических операций, включая XOR, вычитание и обращения к таблицам S-боксов. Множитель для этого раунда равен 5, а индекс используемого блока — `\_x[0]`. Перед началом раунда значения регистров `\_a`, `\_b` и `\_c` следующие:

**Before round \_a: 0000000100100011010001010110011110001001101010111100110111101111**

**Before round \_b: 1111111011011100101110101001100001110110010101000011001000010000**

**Before round \_c: 1111000010010110101001011011010011000011101100101110000110000111**

Операции на регистре \_c:

**c ^= \_x[index];**

Описание: Это операция побитового XOR между регистром \_c и первым блоком данных \_x[0] (в котором хранится двоичное представление строки 'AM'). Это служит начальным изменением данных.

Результат: значение \_c изменится после выполнения этой операции, но полное новое значение не приводится до завершения всех шагов.

Извлечение байтов из \_c:

Значение регистра \_c разбивается на массив из 8 байтов. Это необходимо для работы с таблицами S-боксов:

\_c[0] — первый байт,

\_c[1] — второй байт и т.д.

Операции на регистре \_a:

**a -= sboxes::t1[c\_arr[0]] ^ sboxes::t2[c\_arr[2]] ^ sboxes::t3[c\_arr[4]] ^ sboxes::t4[c\_arr[6]];**

Описание: Значение регистра \_a изменяется с использованием 4 разных S-боксов. Каждый S-бокс — это заранее определенная таблица значений, которые выбираются в зависимости от байтов массива c\_arr, полученных из регистра \_c. Здесь происходит:

Побитовое XOR-сложение данных из разных S-боксов с использованием значений байтов c\_arr[0], c\_arr[2], c\_arr[4], и c\_arr[6].

Результат XOR-операций вычитается из регистра \_a.

Результат: новое значение \_a становится:

**0100001111011000110100100111000000100100011010010100000100010117.**

Операции на регистре \_b:

**b -= sboxes::t4[c\_arr[1]] ^ sboxes::t3[c\_arr[3]] ^ sboxes::t2[c\_arr[5]] ^ sboxes::t1[c\_arr[7]];**

**b \*= mul;**

Описание: Регистру \_b также выполняется несколько операций:

Побитовое XOR-сложение данных из разных S-боксов с использованием значений байтов c\_arr[1], c\_arr[3], c\_arr[5], и c\_arr[7].

Результат этих XOR-операций вычитается из регистра \_b.

После этого значение \_b умножается на множитель mul, который равен 5 в этом раунде.

Результат: новое значение \_b становится:

**1001010010101000111110001111000000010010101100100110111010001101.**

После применения криптографических операций значения регистров \_a, \_b и \_c обновляются следующим образом:

**\_a: 0100001111011000110100100111000000100100011010010100000100010111**

(Этот регистр претерпел изменения в результате вычитания XOR-значений из S-боксов, основанных на байтах массива c\_arr.)

**\_b: 1001010010101000111110001111000000010010101100100110111010001101**

(Значение регистра было изменено аналогичным образом и дополнительно умножено на множитель 5.)

**\_c: 1111000010010110101001011011010011000011101100101010000011001010**

(Регистру \_c был применён XOR с первым блоком данных \_x[0], который повлиял на его новое значение.)

Функция feedforward() выполняет три операции, обновляя значения регистров \_a, \_b и \_c с помощью переменных aa, bb и cc, которые, вероятно, были сохранены ранее. Вот объяснение каждой операции:

**\_a ^= aa;**

Операция XOR (исключающее ИЛИ) применяется между текущим значением регистра \_a и значением переменной aa. Это изменяет биты регистра \_a, добавляя нелинейность в процесс хеширования.

**\_b -= bb;**

Из регистра \_b вычитается значение переменной bb. Это уменьшает значение \_b, что также влияет на конечное состояние данных после обработки.

**\_c += cc;**

К регистру \_c добавляется значение переменной cc. Операция сложения изменяет значение регистра \_c.

Шаг 6. Эти операции обеспечивают механизм обратной связи (feedforward), который добавляет больше сложности и стойкости хеш-а **\_a: 0x42FB9717ADC28CF8**

**\_a: 0x42FB9717ADC28CF8**

(Это результат операции XOR между значением \_a после раунда и aa.)

**\_b: -0x6A33C1A863A1C383**

(Это результат вычитания bb из \_b после раунда. Поскольку результат отрицательный, он обозначается с минусом.)

**\_c: 0x1E12D4B6987658251**

(Это результат сложения cc к \_c после раунда.)

Таким образом, после применения операций функция feedforward() обновляет значения регистров в соответствии с заданными переменными.

По прошествии 24 раундов мы получим на выходе результат:  
**[+]Your hashed text**

**\\_\_\_\_[93609b06266aeba41e4aedef914113edf1ce46c3457383f2]**

Что является конкатенацией результата помещенного в регистры \_a \_b \_c

1. Заключение

5.1 Выводы по проделанной работе

В ходе выполнения работы было подробно изучено и проанализировано два ключевых аспекта криптографии: сети Фейстеля и алгоритм Tiger.

Сети Фейстеля являются важным компонентом в современных криптографических систем. Они представляют собой архитектурный подход, позволяющий обеспечить высокий уровень безопасности данных через многократное применение простых операций.

Основная идея заключается в делении блока данных на две части и выполнении сложных преобразований, что способствует увеличению стойкости к атакам. Функциональность сетей Фейстеля лежит в основе многих известных шифров, таких как DES и AES, что подтверждает их значимость в криптографии.

Исследование показало, что правильный выбор функций раундов и параметров, таких как ключи и количество раундов, критически важен для обеспечения надежности и безопасности шифрования.

Алгоритм Tiger, разработанный для хеширования, отличается высокой скоростью обработки и эффективностью в вычислительных ресурсах, что делает его подходящим для использования в различных приложениях, включая цифровые подписи и проверку целостности данных.

Хеш-функция Tiger обеспечивает защиту от коллизий и позволяет генерировать уникальные хеши для различных наборов данных. Это свойство делает алгоритм надежным инструментом для обеспечения безопасности информации.

В процессе реализации программы, создающей хеш для одного раунда Tiger, было продемонстрировано, как основные принципы алгоритма могут быть реализованы на практике, что углубило понимание его внутренней структуры и работы.

В заключение, изучение сетей Фейстеля и алгоритма Tiger показало, что они играют ключевую роль в обеспечении безопасности данных. Эти методы продолжают оставаться актуальными и важными в области криптографии, предоставляя необходимые инструменты для защиты информации в современных цифровых системах.

1. Приложения

6.1 Текст программы

Tigerhash.cpp

#include "outputs.h"

#include "tigerhash.h"

#include "sboxes.h"

#include <string>

#include <iostream>

using namespace std;

int main()

{

string input;

info("input string to hash for example your initials:\n");

getline(cin, input);

string hash\_in\_hex = tiger::hash(input);

okay("Your hashed text\n");

printf("\t\\\_\_\_\_[%s]", hash\_in\_hex.data());

return EXIT\_SUCCESS;

}

Tigerhash.h

#pragma once

#ifndef TIGERHASH\_TIGERHASH\_H

#define TIGERHASH\_TIGERHASH\_H

#include <sstream>

#include <iostream>

#include <stdint.h>

#include "sboxes.h"

#include "outputs.h"

namespace tiger {

// 64 bits words for intermediate hash values

uint64\_t \_a;

uint64\_t \_b;

uint64\_t \_c;

// Buf values

uint64\_t aa;

uint64\_t bb;

uint64\_t cc;

// Global array for storing block for hash computation

uint64\_t \_x[8];

// Save intermediate hash in buf values

void save\_abc() {

aa = \_a;

bb = \_b;

cc = \_c;

info("Saving intermediate values: \n\t\_a=%s, \n\t\_b=%s, \n\t\_c=%s\n",

std::bitset<64>(\_a).to\_string().c\_str(),

std::bitset<64>(\_b).to\_string().c\_str(),

std::bitset<64>(\_c).to\_string().c\_str());

}

// Function for computing one round of hash

void round(uint64\_t& a, uint64\_t& b, uint64\_t& c, unsigned int index, char mul) {

// Convert c 64-bit value to array of bytes

std::string c\_str = std::bitset<64>(c).to\_string();

unsigned char c\_arr[8]{ 0 };

for (std::size\_t i = 0, j = 7; i < c\_str.length(); i += 8, --j) {

c\_arr[j] = (unsigned char)std::bitset<8>(c\_str.substr(i, 8)).to\_ulong();

info("c\_arr[%d] = \n\t%s\n", j, std::bitset<64>(c\_arr[j]).to\_string().c\_str());

}

info("Before round: \n\ta=%s, \n\tb=%s, \n\tc=%s\n",

std::bitset<64>(a).to\_string().c\_str(),

std::bitset<64>(b).to\_string().c\_str(),

std::bitset<64>(c).to\_string().c\_str());

c ^= \_x[index];

a -= sboxes::t1[c\_arr[0]] ^ sboxes::t2[c\_arr[2]] ^ sboxes::t3[c\_arr[4]] ^ sboxes::t4[c\_arr[6]];

b -= sboxes::t4[c\_arr[1]] ^ sboxes::t3[c\_arr[3]] ^ sboxes::t2[c\_arr[5]] ^ sboxes::t1[c\_arr[7]];

b \*= mul;

info("After round: \n\ta=%s, \n\tb=%s, \n\tc=%s, \n\tmul=%d, \n\tindex=%u\n",

std::bitset<64>(a).to\_string().c\_str(),

std::bitset<64>(b).to\_string().c\_str(),

std::bitset<64>(c).to\_string().c\_str(),

mul, index);

}

// Function for initiating round computation

void pass(uint64\_t& a, uint64\_t& b, uint64\_t& c, char mul) {

round(a, b, c, 0, mul);

round(b, c, a, 1, mul);

round(c, a, b, 2, mul);

round(a, b, c, 3, mul);

round(b, c, a, 4, mul);

round(c, a, b, 5, mul);

round(a, b, c, 6, mul);

round(b, c, a, 7, mul);

}

// Key schedule for modifying \_x array

void key\_schedule() {

for (int i = 0; i < 8; i++){

info("Before key schedule: \_x[%d]=\n\t%s\n",

i,

std::bitset<64>(\_x[i]).to\_string().c\_str());

}

\_x[0] -= \_x[7] ^ 0xA5A5A5A5A5A5A5A5;

\_x[1] ^= \_x[0];

\_x[2] += \_x[1];

\_x[3] -= \_x[2] ^ ((~\_x[1]) << 19);

\_x[4] ^= \_x[3];

\_x[5] += \_x[4];

\_x[6] -= \_x[5] ^ ((~\_x[4]) >> 23);

\_x[7] ^= \_x[6];

\_x[0] += \_x[7];

\_x[1] -= \_x[0] ^ ((~\_x[7]) << 19);

\_x[2] ^= \_x[1];

\_x[3] += \_x[2];

\_x[4] -= \_x[3] ^ ((~\_x[2]) >> 23);

\_x[5] ^= \_x[4];

\_x[6] += \_x[5];

\_x[7] -= \_x[6] ^ 0x0123456789ABCDEF;

for (int i = 0; i < 8; i++) {

info("After key schedule: \_x[%d]=\n\t%s\n",

i,

std::bitset<64>(\_x[i]).to\_string().c\_str());

}

}

// Feedforward step

void feedforward() {

info("Before feedforward: \_a=%s, \_b=%s, \_c=%s\n",

std::bitset<64>(\_a).to\_string().c\_str(),

std::bitset<64>(\_b).to\_string().c\_str(),

std::bitset<64>(\_c).to\_string().c\_str());

\_a ^= aa;

\_b -= bb;

\_c += cc;

info("After feedforward: \_a=%s, \_b=%s, \_c=%s\n",

std::bitset<64>(\_a).to\_string().c\_str(),

std::bitset<64>(\_b).to\_string().c\_str(),

std::bitset<64>(\_c).to\_string().c\_str());

}

// Convert 64-bit value to hex string

std::string to\_hex(uint64\_t value) {

std::stringstream stream;

stream << std::hex << value;

std::string result(stream.str());

while (result.length() != 16) {

result = '0' + result;

}

return result;

}

// Compute Tiger hash from input string

std::string hash(std::string value) {

// Basic hash initialization

\_a = 0x0123456789ABCDEF;

\_b = 0xFEDCBA9876543210;

\_c = 0xF096A5B4C3B2E187;

okay("Initialized hash values: \_a=%s, \_b=%s, \_c=%s\n",

std::bitset<64>(\_a).to\_string().c\_str(),

std::bitset<64>(\_b).to\_string().c\_str(),

std::bitset<64>(\_c).to\_string().c\_str());

// Basic buffer initialization

aa = 0;

bb = 0;

cc = 0;

// Pad input message to 512-bit blocks

while (value.length() % 64 != 0) {

value = char(0) + value;

}

// Process each 512-bit block

for (std::size\_t k = 0; k < value.length(); k += 64) {

std::string buf\_str = value.substr(k, 64);

// Divide the 512-bit block into eight 64-bit blocks

for (std::size\_t i = 0, j = 7; i < buf\_str.length(); i += 8, --j) {

std::string bit\_str;

std::string str\_word = buf\_str.substr(i, 8);

for (int n = 0; n < 8; ++n) {

bit\_str += std::bitset<8>(str\_word[n]).to\_string();

}

\_x[j] = std::bitset<64>(bit\_str).to\_ullong();

}

info("Processing block %zu: \_x[0]=%s, \_x[7]=%s\n", k / 64,

std::bitset<64>(\_x[0]).to\_string().c\_str(),

std::bitset<64>(\_x[7]).to\_string().c\_str());

// Compute intermediate hash values for the block

info("round [%d]\n", k % 64);

save\_abc();

pass(\_a, \_b, \_c, 5);

key\_schedule();

pass(\_c, \_a, \_b, 7);

key\_schedule();

pass(\_b, \_c, \_a, 9);

feedforward();

}

// Return the final hash result in hexadecimal form

return to\_hex(\_a) + to\_hex(\_b) + to\_hex(\_c);

}

}

#endif //TIGERHASH\_TIGERHASH\_H

* 1. Пример работы программы сокращенный

[i]input string to hash for example your initials:

try to hash me if u can

[+]Initialized hash values: \_a=0000000100100011010001010110011110001001101010111100110111101111, \_b=1111111011011100101110101001100001110110010101000011001000010000, \_c=1111000010010110101001011011010011000011101100101110000110000111

[i]Processing block 0: \_x[0]=0110100101100110001000000111010100100000011000110110000101101110, \_x[7]=0000000000000000000000000000000000000000000000000000000000000000

[i]round [0]

[i]Saving intermediate values:

\_a=0000000100100011010001010110011110001001101010111100110111101111,

\_b=1111111011011100101110101001100001110110010101000011001000010000,

\_c=1111000010010110101001011011010011000011101100101110000110000111

…..

[i]c\_arr[7] =

0000000000000000000000000000000000000000000000000000000010101110

[i]c\_arr[6] =

0000000000000000000000000000000000000000000000000000000011101011

[i]c\_arr[5] =

0000000000000000000000000000000000000000000000000000000011110111

[i]c\_arr[4] =

0000000000000000000000000000000000000000000000000000000000101101

[i]c\_arr[3] =

0000000000000000000000000000000000000000000000000000000001011110

[i]c\_arr[2] =

0000000000000000000000000000000000000000000000000000000000101100

[i]c\_arr[1] =

0000000000000000000000000000000000000000000000000000000001010100

[i]c\_arr[0] =

0000000000000000000000000000000000000000000000000000000011000000

[i]Before round:

a=1111001000001011001001010100101100101111011010011001101001100000,

b=1010110111100111101110100010001000111100111101001111001110010100,

c=1010111011101011111101110010110101011110001011000101010011000000

[i]After round:

a=1010000000111100010011001001101000101000010111000110000001011010,

b=1010011010011110111011011001011111111110110001011111100100001001,

c=1000110011011000000111111110001100001100000111101111110101111001,

mul=9,

index=7

[i]Before feedforward: \_a=1010011010011110111011011001011111111110110001011111100100001001, \_b=1000110011011000000111111110001100001100000111101111110101111001, \_c=1010000000111100010011001001101000101000010111000110000001011010

[i]After feedforward: \_a=1010011110111101101010001111000001110111011011100011010011100110, \_b=1000110111111011011001010100101010010101110010101100101101101001, \_c=1001000011010010111100100100111011101100000011110100000111100001

[+]Your hashed text

\\_\_\_\_[a7bda8f0776e34e68dfb654a95cacb6990d2f24eec0f41e1]

* 1. Список литературы

1. <https://ru.wikipedia.org/wiki/Tiger_(%D1%85%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F)>
2. <https://iacr.org/archive/asiacrypt2007/48330539/48330539.pdf>
3. <https://en.wikipedia.org/wiki/Tiger_(hash_function)>
4. <https://www.ixopay.com/en/news/what-is-a-feistel-cipher>
5. <https://en.wikipedia.org/wiki/Feistel_cipher>
6. https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%82%D1%8C\_%D0%A4%D0%B5%D0%B9%D1%81%D1%82%D0%B5%D0%BB%D1%8F