МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра информационной безопасности

ОТЧЕТ
по лабораторной работе №6
по дисциплине «Криптографические методы защиты информации»
ТЕМА: Изучение хеш-функций

Студентка гр. 9363 _____ Труханова В.А.

Преподаватель _____ Племянников А.К.

Санкт-Петербург
2023

Цель работы

Исследование хеш-функций MD5, SHA-256, SHA-512, SHA-3, кода аутентификации HMAC для контроля целостности и анализ атак дополнительной коллизии на хеш-функцию. Получить практические навыки работы с хеш-функциями и алгоритмом атаки дополнительной коллизии, в том числе с использованием приложения CrypTool 1 и 2.

MD5, SHA-1, SHA-256, SHA-512

6.1. Задание

1. Открыть текст не менее 1000 знаков. Добавить ваши ФИО последней строкой. Перейти к утилите Indiv.Procedures –> Hash –> Hash Demonstration.
2. Задать хеш-функцию, подлежащую исследованию: MD5, SHA-1, SHA-256, SHA-512.
3. Для каждой хеш-функции повторить следующие действия:

а) изменить (добавлением, заменой, удалением символа) исходный файл;

б) зафиксировать количество измененных битов в дайджесте модифицированного сообщения;

в) вернуть сообщение в исходное состояние.

4. Выполнить процедуру 3 раза (добавлением, заменой, удалением символа) и подсчитать среднее количество измененных бит дайджеста. Зафиксировать результаты в таблице.

6.1.1. Основные параметры и обобщенная схема хеш-функций MD5, SHA-1.

Алгоритм вычисления значений хеш-функций MD5, SHA-1 основаны на схеме Меркеля–Дамгарда (рисунок 1).

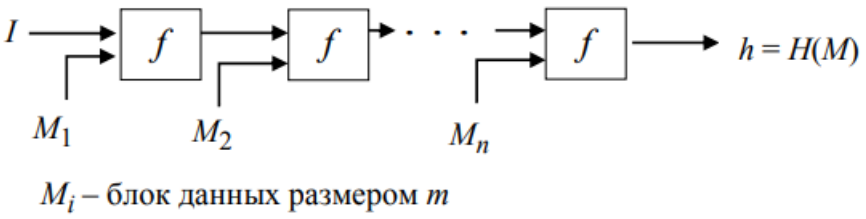


Рисунок 1 – Схема Меркеля-Дамгарда

Хэш-функция MD5 принимает на вход произвольное сообщение фиксированной или переменной длины и возвращает 128-битный хеш-код.

Алгоритм MD5 состоит из следующих шагов:

1. Исходное сообщение дополняется битами до длины, кратной 512 битам. Дополнение производится таким образом, чтобы последние 64 бита содержали длину исходного сообщения в битах.
2. Исходное сообщение разбивается на блоки по 512 бит каждый.
3. Начальное значение буфера итерации (buffer state) устанавливается в определенную константу.
4. Для каждого блока сообщения выполняются четыре раунда обработки, каждый из которых состоит из 16 операций. Каждый цикл переопределяет значение буфера.
5. Результатом обработки всех блоков сообщения является 128-битный хеш-код.

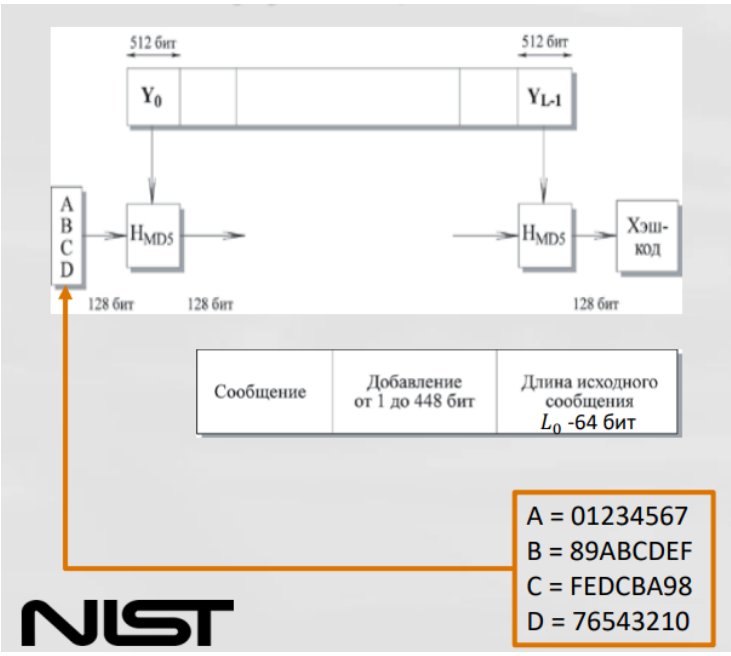


Рисунок 2 – Обобщенная схема хеш-функции MD5

Функция H_{MD5} принимает на вход 128-битный буфер и 512-битный блок данных и возвращает обновленный 128-битный буфер.

H_{MD5} состоит из четырех раундов (Rounds), каждый из которых состоит из 16 операций (Operations).

В каждой операции используется одна из четырех нелинейных функций F, G, H и I. Функция F используется в первом раунде, функция G - во втором раунде, функция H - в третьем раунде и функция I - в четвертом раунде.

Каждый 512-битный блок проходит 4 этапа вычислений по 16 раундов. Для этого блок представляется в виде массива X из 16 слов по 32 бита. Все раунды однотипны и имеют вид: $[abcd\ k\ s\ i]$, определяемы как:

$$a = b + ((a + f(b, c, d) + X[k] + T[i]) \lll s)$$

где k – номер 32-битного слова из текущего 512-битного блока сообщения, и – циклический сдвиг влево на s бит полученного 32-битного аргумента;

$T[1 \dots 64]$ – 64-элементная таблица данных, построенная следующим образом: $T[n] = int(2^{32} \cdot |\sin n|$;

f – одна из элементарных функций:

- Для функции F: F(X,Y,Z) = (X AND Y) OR (NOT X AND Z)
- Для функции G: G(X,Y,Z) = (X AND Z) OR (Y AND NOT Z)
- Для функции H: H(X,Y,Z) = X XOR Y XOR Z
- Для функции I: I(X,Y,Z) = Y XOR (X OR NOT Z)

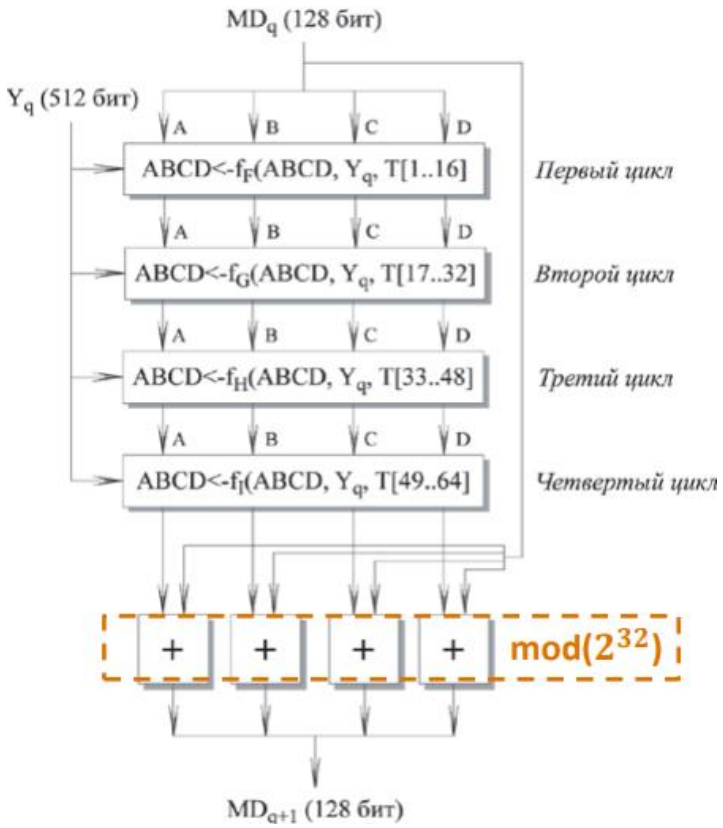


Рисунок 3 – Схема функции сжатия H_{MD5}

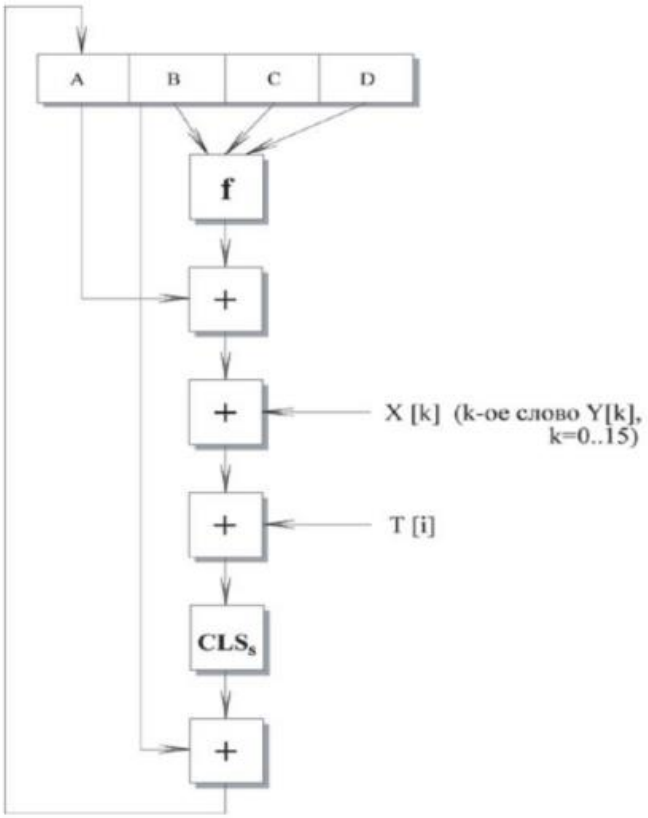
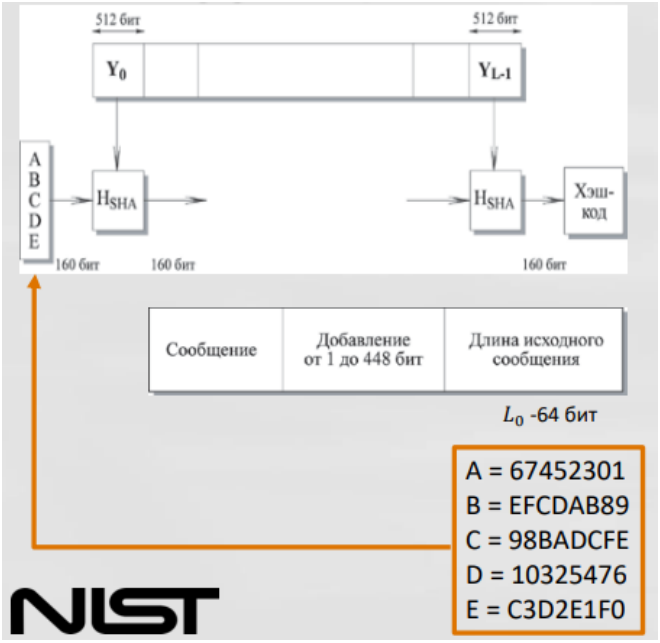


Рисунок 4 – Цикл сжатия H_{MD5}

- Хеш-фукция SHA-1:

Функция сжатия SHA-1 основана на блочном шифре и использует 80-битный ключ. Она принимает на вход 160-битный буфер и 512-битный блок данных и возвращает обновленный 160-битный буфер. Функция сжатия SHA-1 выполняет следующие шаги:

1. Исходное сообщение дополняется битами до длины, кратной 512 битам. Дополнение производится таким образом, чтобы последние 64 бита содержали длину исходного сообщения в битах.
2. Разбивается 512-битный блок данных на 16 слов по 32 бита.
3. Начальное значение буфера итерации (buffer state) устанавливается в определенную константу.
4. Выполняется 80 итераций (Rounds), в каждом из которых текущее значение буфера обрабатывается с помощью операции, зависящих от номера итерации. В каждой итерации используется дополнительная константа K_t принимающая 4 различных значения.
5. Операция включает в себя циклические сдвиги, побитовые операции И, ИЛИ и ИСКЛЮЧАЮЩЕЕ ИЛИ, а также сложение по модулю 2^{32}.
6. После 80 итераций, текущее значение буфера добавляется к начальному значению, чтобы получить итоговый 160-битный хеш-код.



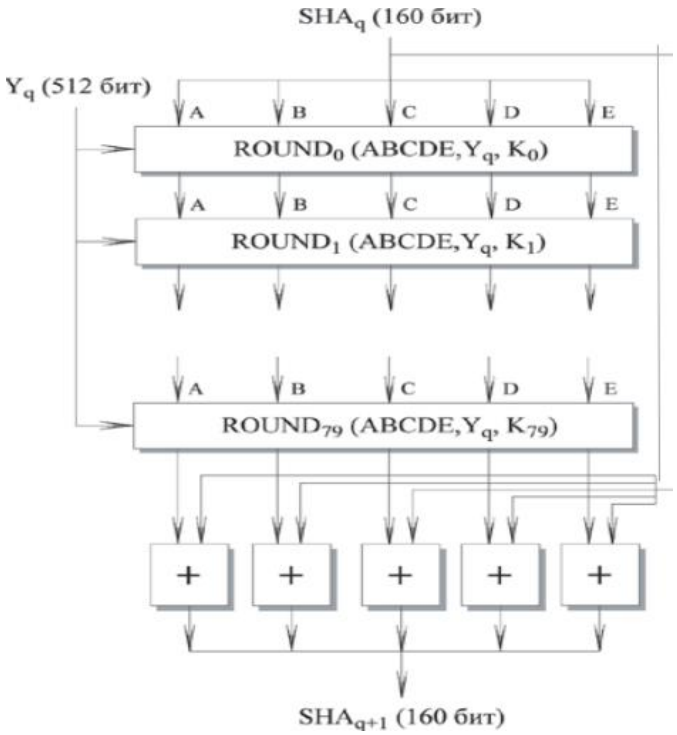


Рисунок 5 – Схема функции сжатия H_{SHA-1}

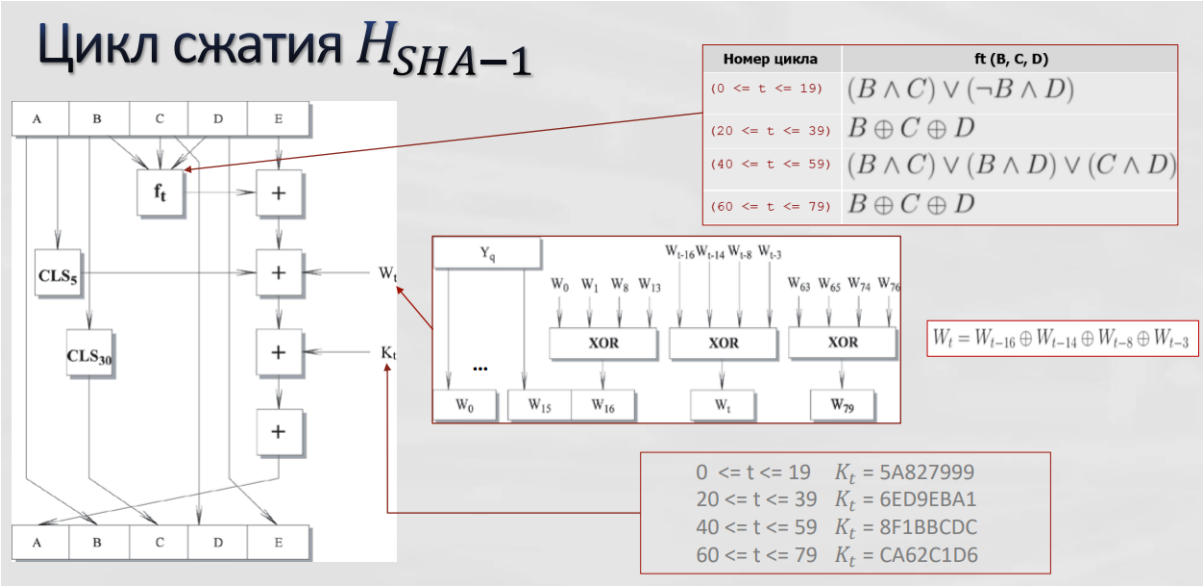


Рисунок 6 – Цикл сжатия H_{SHA-1}

6.1.2. Ход выполнения задания

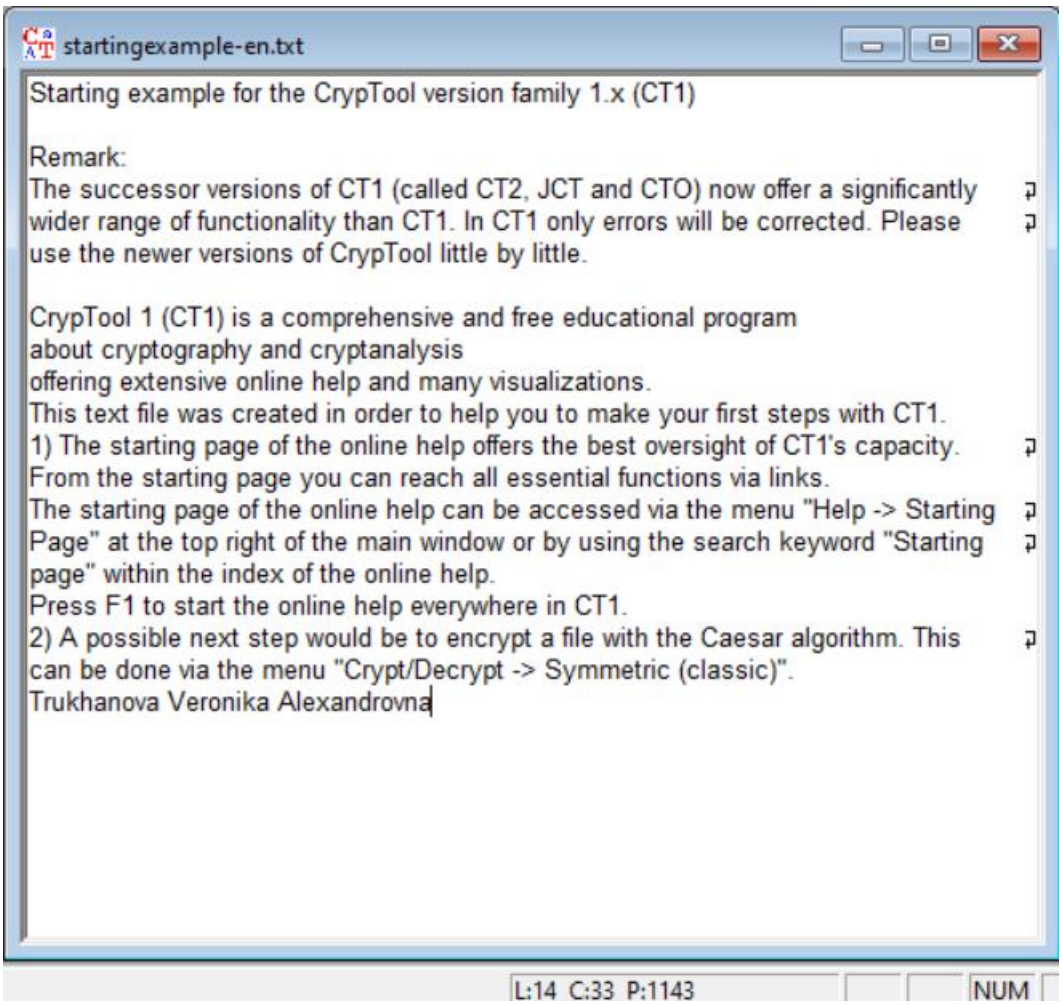


Рисунок 7 – Исходный файл длинной 1143 символа

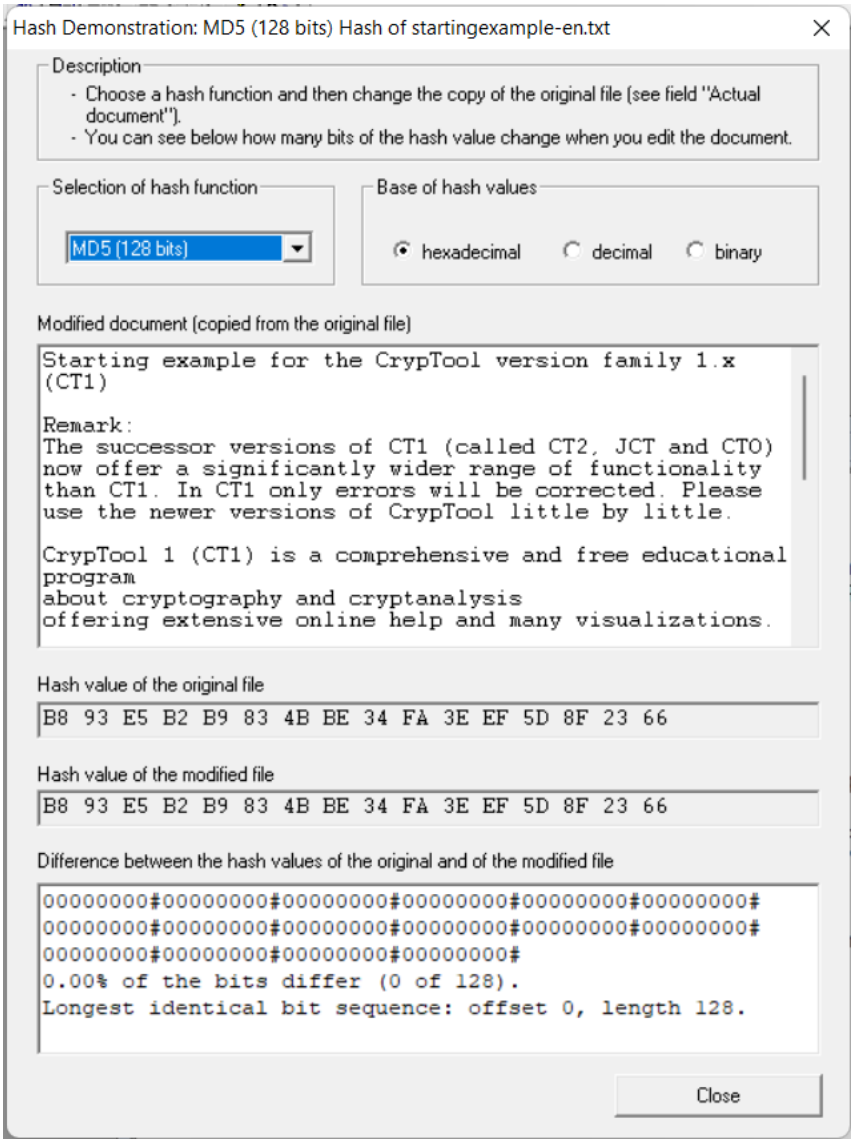


Рисунок 8 – Окно Hash Demonstration

Для каждой хеш-функции повторим следующие действия:

а) изменить (добавлением, заменой, удалением символа) исходный файл;

б) зафиксировать количество измененных битов в дайджесте модифицированного сообщения;

в) вернуть сообщение в исходное состояние.

Выполним процедуру 3 раза (добавлением, заменой, удалением символа) и подсчитаем среднее количество измененных бит дайджеста. Зафиксируем результаты в таблице 1.

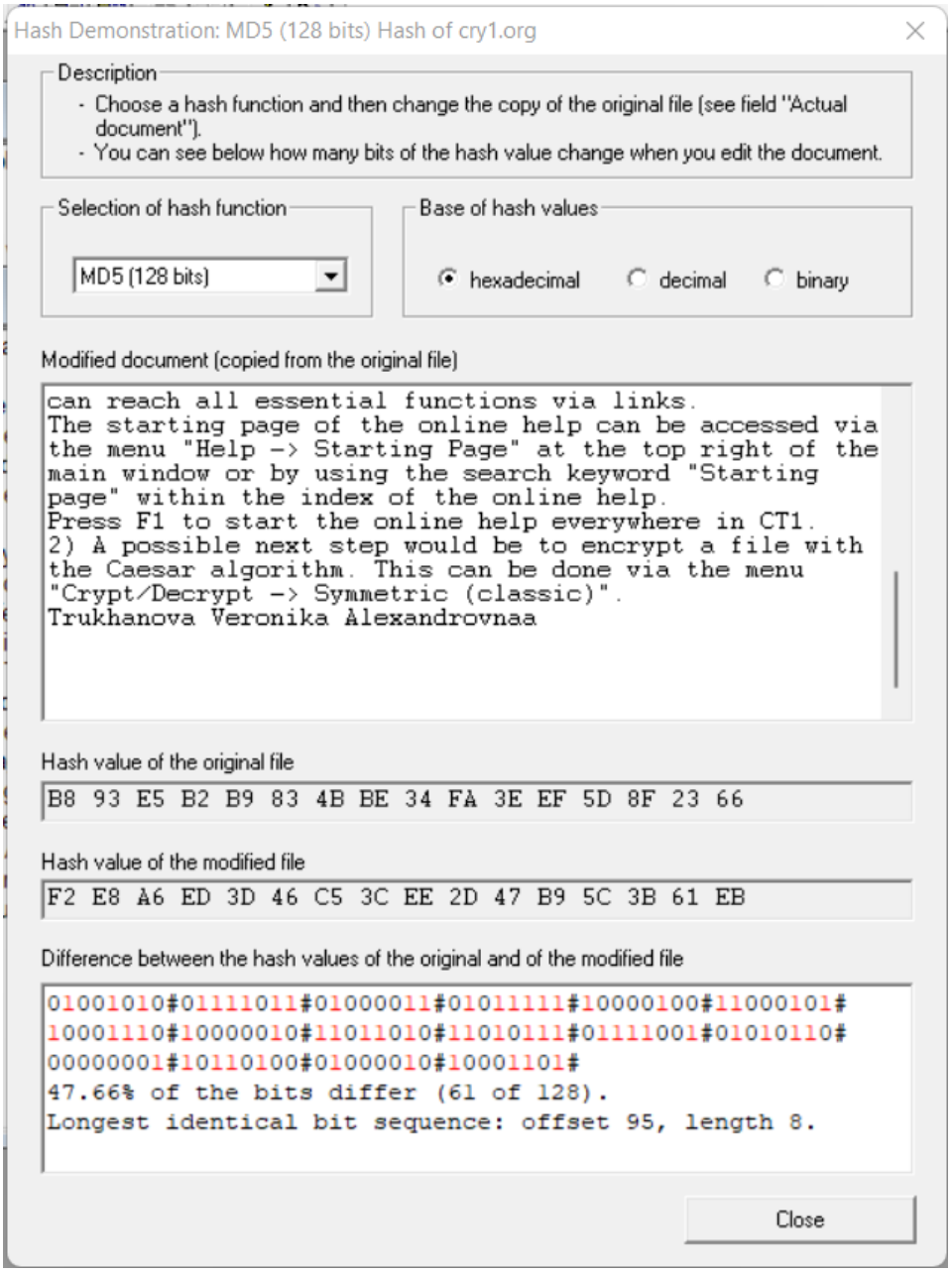


Рисунок 9 – Пример добавления символа

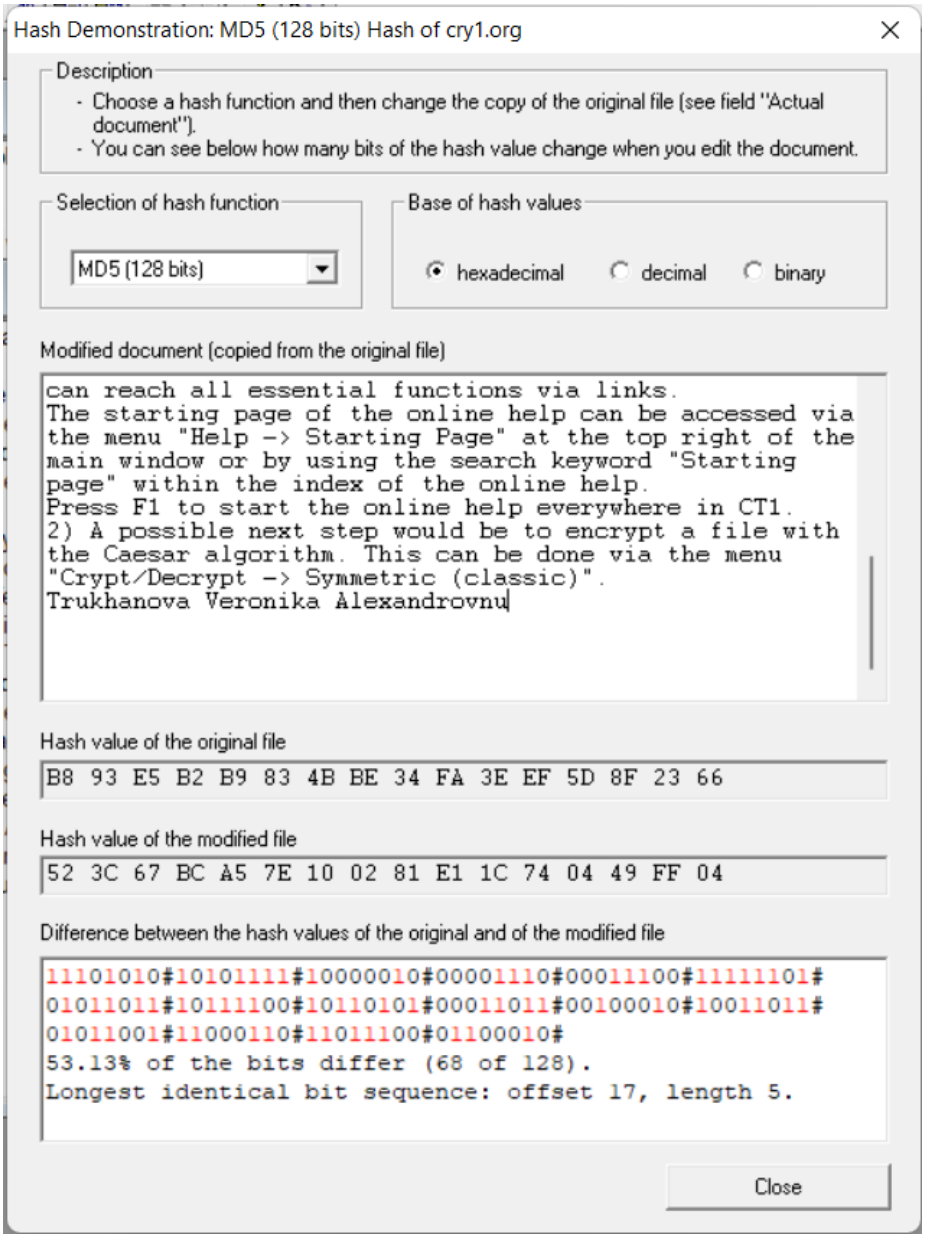


Рисунок 10 – Пример замены символа

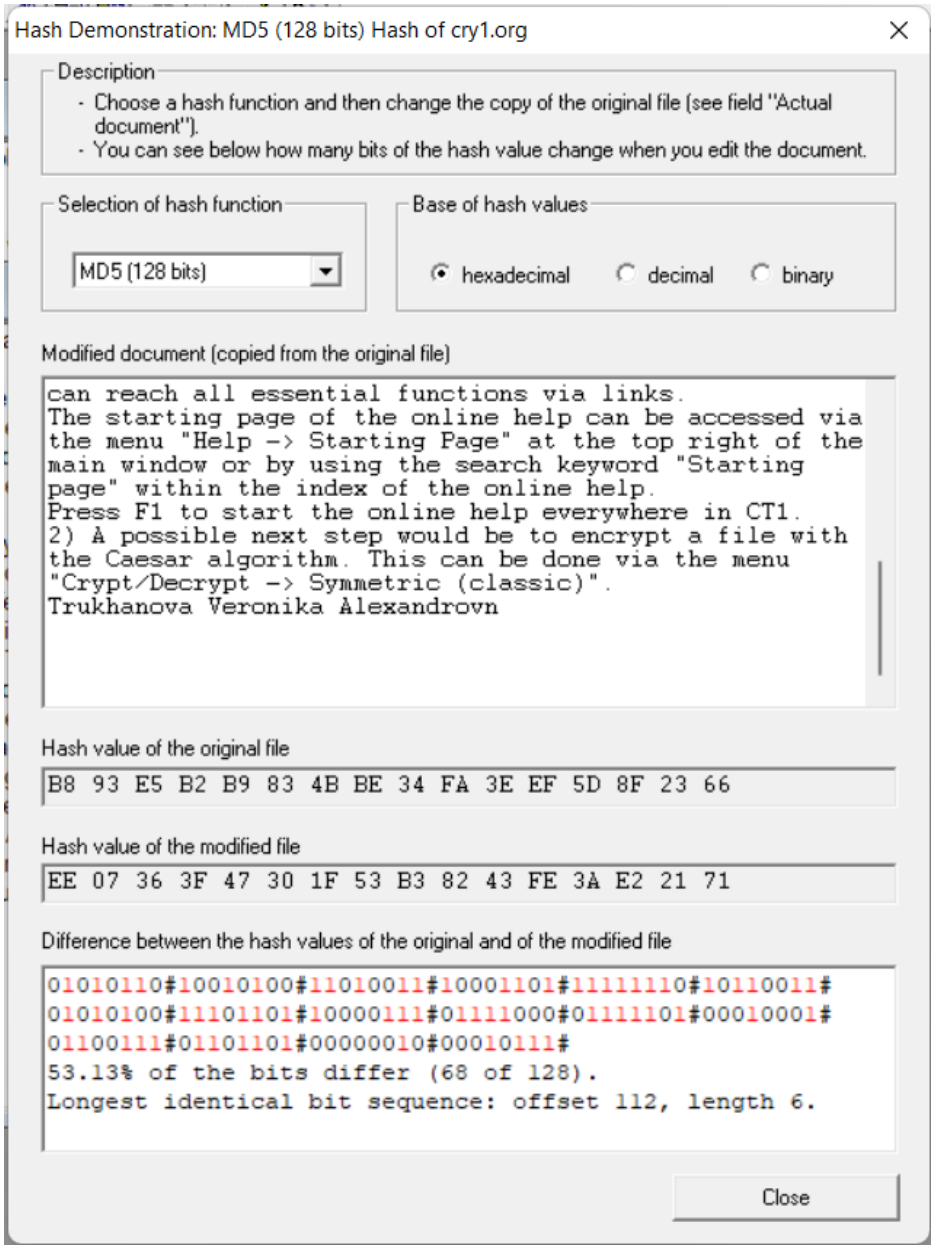


Рисунок 11 – Пример удаления символа

Таблица 1 Количество измененных бит

	MD5 (128)	SHA-1(160)	SHA-256	SHA-512
Добавление символа	61	83	133	245
Замена символа	68	79	129	258
Удаление символа	68	70	123	249
Усредненное	68	79	128	251

SHA-3

6.2. Задание

- Открыть шаблон Keccak Hash (SHA-3) в Cryptool 2.
- В модуле Keccak сделать следующие настройки:
  a. Adjust manually=ON;
  b. Keccak version= SHA3-512;
- Загрузить файл из предыдущего задания.
- Запустить проигрывание шаблона в режиме ручного управления:
  a. Сохранить скриншоты преобразований первого раунда;
  b. Сохранить скриншот заключительной фазы;
  c. Сохранить значение дайджеста.
- Вычислить значения дайджеста для модифицированных текстов из предыдущего задания.
- Подсчитать лавинный эффект с помощью самостоятельно разработанной автоматизированной процедуры.

6.2.1. Основные параметры, обобщенная схема и перестановки хеш-функции Keccak (SHA-3)

В основе Keccak (SHA-3) лежит конструкция под названием Sponge – губка. Сам алгоритм состоит из 2-х этапов:

1. Впитывание – Absording. На каждом шаге очередной блок сообщения pi длиной r подмешивается к части внутреннего состояния S, которая затем целиком модифицируется функцией f многораундовой бесключевой псевдослучайной перестановкой.

2. Отжатие – Squeezing. Чтобы получить хэш, функция f многократно применяется к состоянию, и на каждом шаге сохраняется кусок размера r до тех пор, пока не получим выход Z необходимой длины (путем конкатенации).

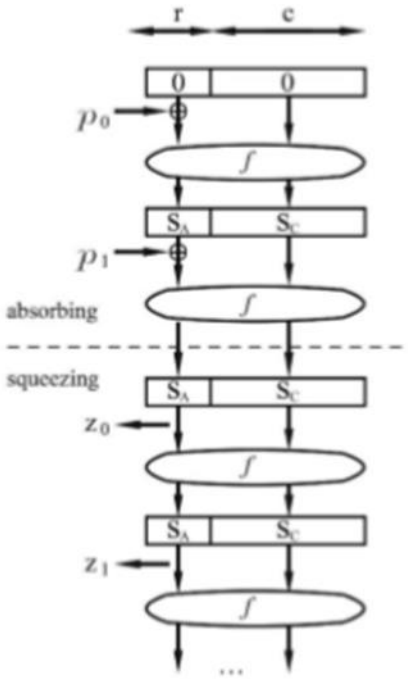


Рисунок 12 – Общая схема работы SHA-3

6.2.2. Ход выполнения задания

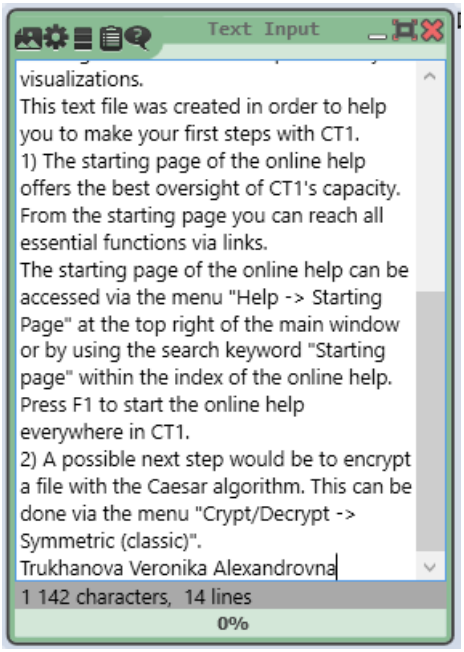


Рисунок 13 – Исходный файл

Состояние конструкции губки инициализировано. Каждый из 1600 бит состояния инициализируется 0. Состояние разделено на две части: емкость (1024 бит) и скорость передачи (576 бит). Вы можете свободно настраивать, если выберете версию Keccak "Keccak" в настройках.

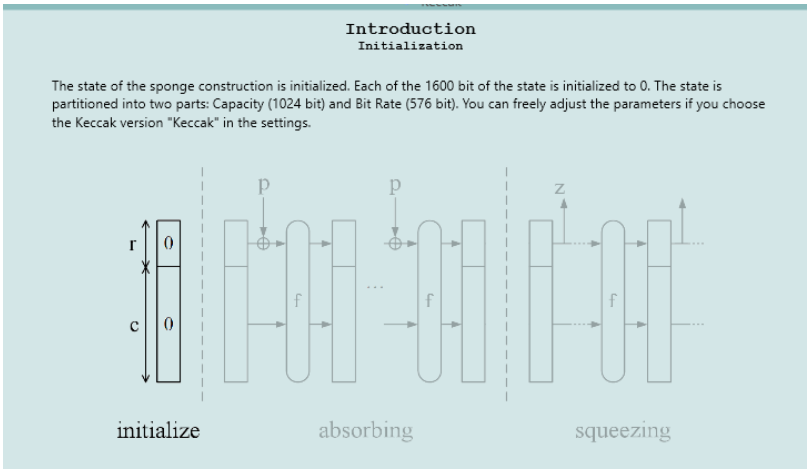


Рисунок 14 – Инициализация состояния

Каждый блок p дополненного ввода поглощается состоянием губки с операцией XOR. Входные блоки воздействуют на 576 бит r-битной части. Часть c-бита не затрагивается. После поглощения выполняется перестановка Keccak (Keccak-f) для рассеивания битов состояния.

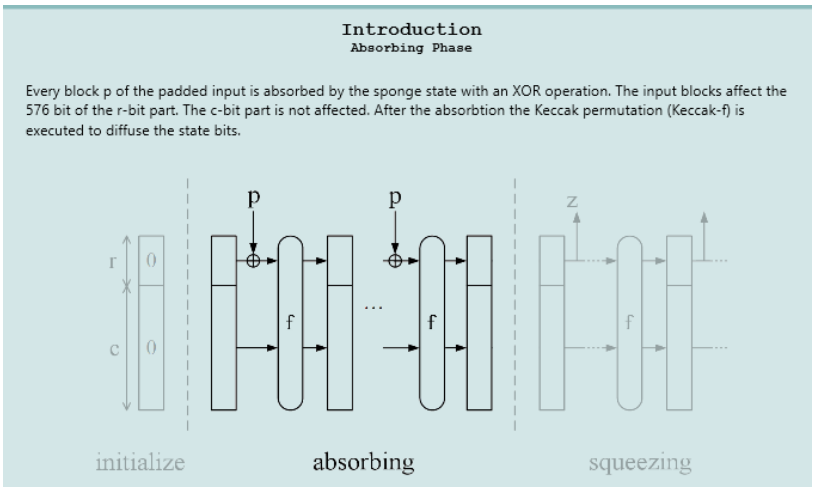


Рисунок 15 – Схема работы впитывания

Хеш-значение извлекается из r-битной части состояния (S). Если размер запрошенного вывода больше r, состояние многократно переставляется, и между перестановками извлекается больше вывода, пока не будет получена требуемая длина вывода.

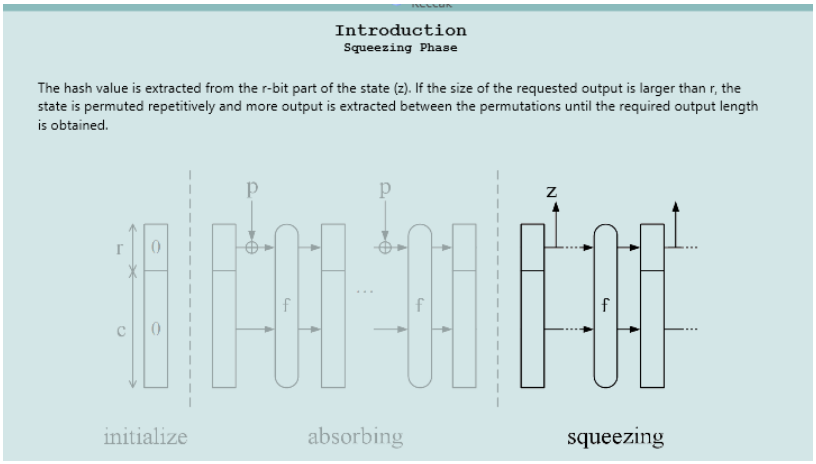


Рисунок 16 – Схема работы выжимания

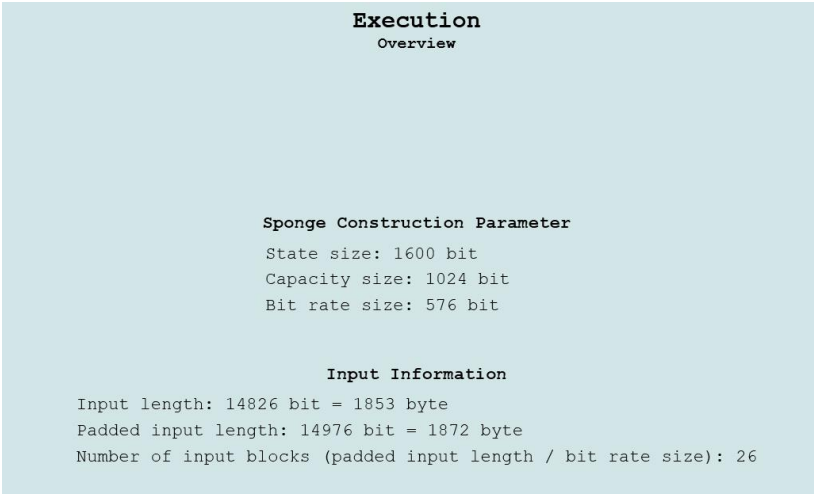


Рисунок 17 – Характеристики входного открытого текста

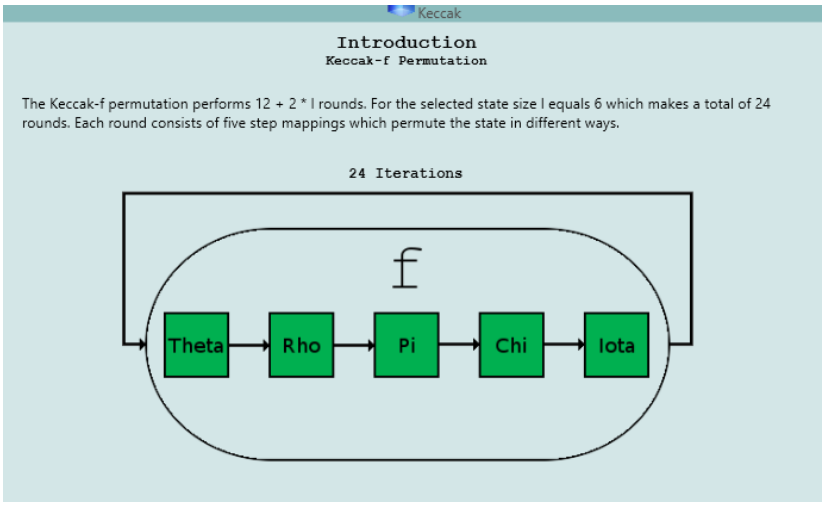


Рисунок 18 – Цикл работы функции Keccak-f

Перестановка Keccak-f выполняет 12+2*l раундов. Для выбранного размера состояния l равно 6, что в сумме дает 24 раунда. Каждый раунд состоит из пяти сопоставлений шагов, которые меняют состояние по-разному.

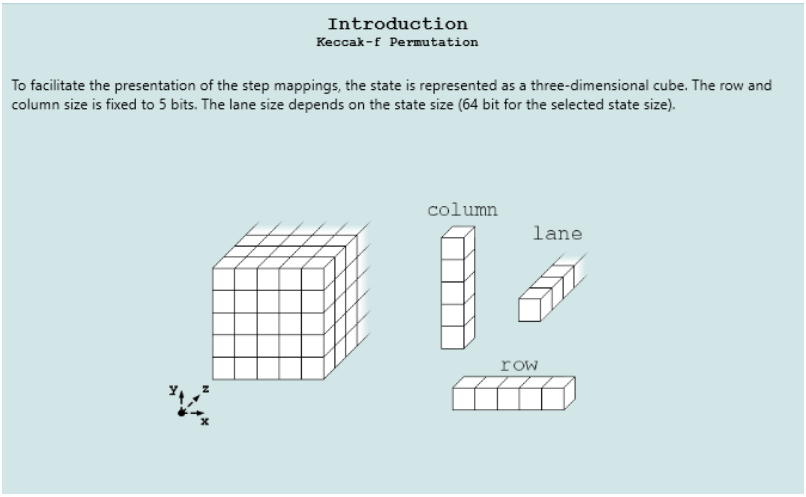


Рисунок 19 – Схема работы операции Theta

Для облегчения представления пошагового отображения состояние представлено в виде трехмерного куба. Размер строки и столбца фиксирован и равен 5 битам. Шестая дорожка зависит от размера состояния (64 бита для выбранного размера состояния).

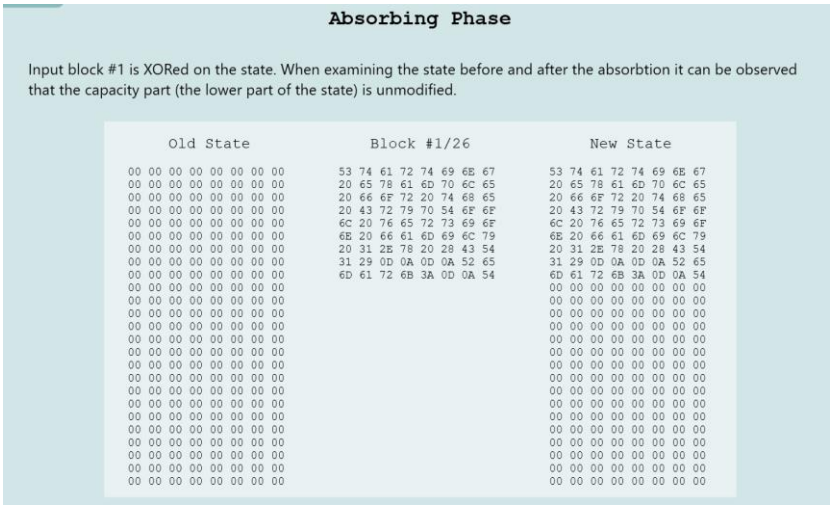


Рисунок 20 – Первое впитывание

Входной блок №1 подвергается операции XOR над состоянием. При исследовании состояния до и после поглощения можно заметить, что емкостная часть (нижняя часть состояния) не изменена.

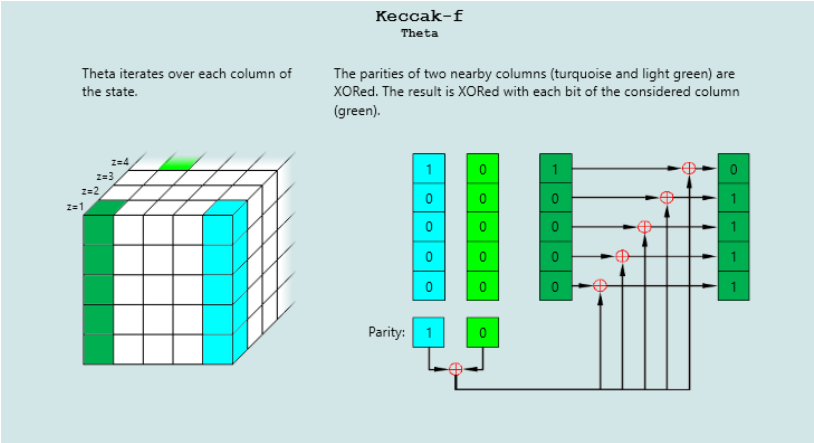


Рисунок 21 – Пример реализации операции Theta

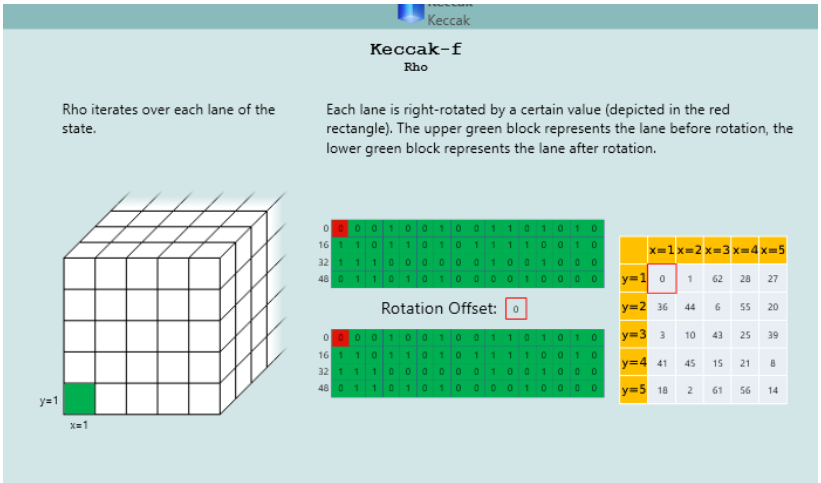


Рисунок 22 – Пример реализации операции Rho

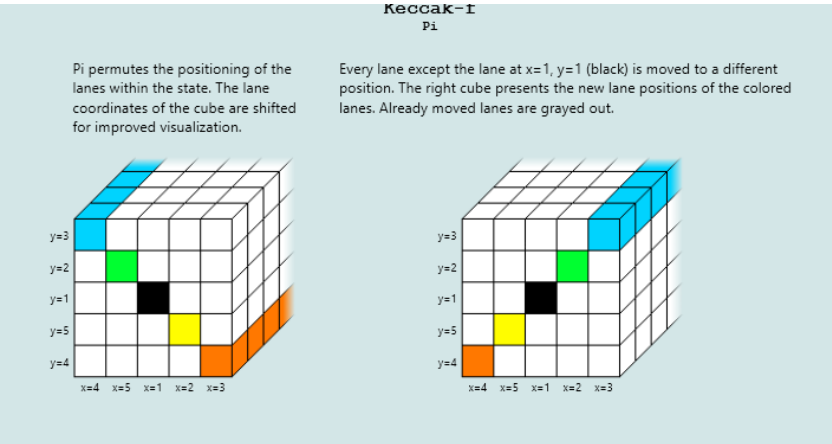


Рисунок 23 – Пример реализации операции Pi

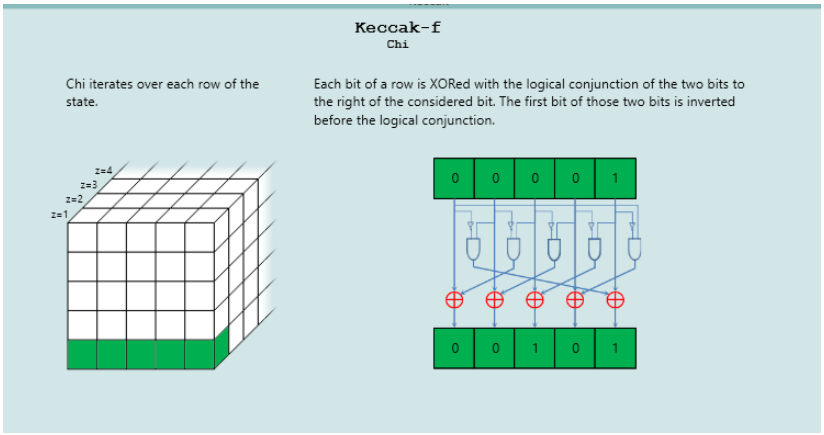


Рисунок 24 – Пример реализации операции Chi

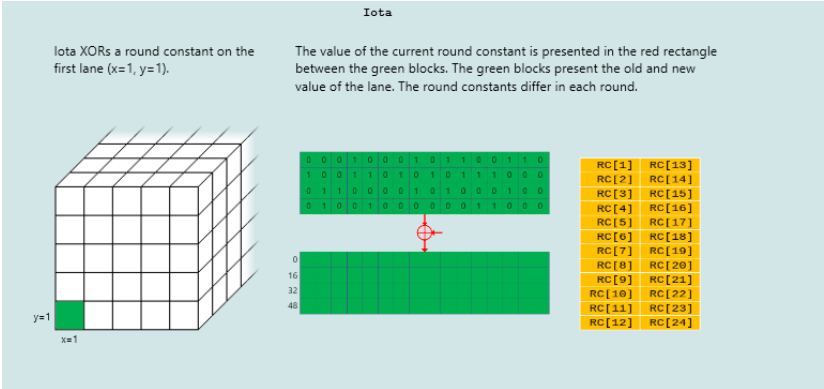


Рисунок 25 – Пример реализации операции Iota

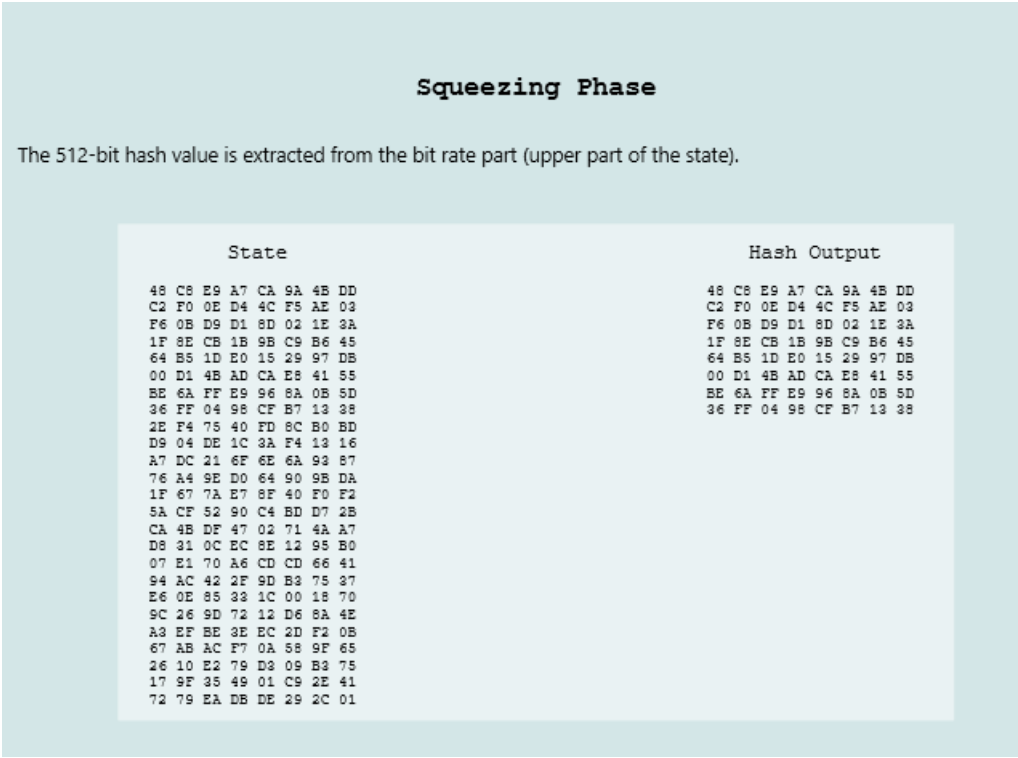


Рисунок 26 – Результат

Полученный хэш:

48 C8 E9 A7 CA 9A 4B DD C2 F0 0E D4 4C F5 AE 03 F6 0B D9 D1 8D 02 1E 3A 1F 8E CB 1B 9B C9 B6 45 64 B5 1D E0 15 29 97 DB 00 D1 4B AD CA E8 41 55 BE 6A FF E9 96 8A 0B 5D 36 FF 04 98 CF B7 13 38

Хэш после добавления в исходный текст символа:

1E FE 7F BD 07 F1 61 34 7B 1B EF F0 31 5E 4A 3B 29 89 10 07 67 D3 AB CA CF 99 A5 AF 20 A0 B9 AE 3B 95 5D C9 A4 19 96 DD 0A 5F 5C A7 86 91 FA 51 5B 6E 45 0D DD CD EE 56 7A F1 87 39 DD 6B 48 49

Хэш после замены символа в исходном тексте:

E8 B4 4F F1 5C A3 DF 64 3F 68 19 04 09 EE 4E F5 CF 41 05 B5 1F 27 D7 65 3F 8C DB 5E 6B 79 DA 49 47 E6 28 27 3A 35 F8 07 C6 F5 54 E0 34 E2 6E 11 AA A7 2B 98 F8 7D C9 03 98 3C B3 79 D3 E2 C2 74

Хэш после удаления символа в исходном тексте:

80 E4 9A 54 CC 6C F6 83 B3 2B 29 68 1B B5 5D 4F BB 6B CA 22 04 80 D5 58 D8 77 79 66 64 31 34 3B 50 00 DE DC B3 7F 34 F8 4C 29 65 76 B1 C6 95 92 10 BE EC A8 A6 9C E5 E7 0D E5 11 7B 00 90 50 98

6.2.3. Лавинный эффект

Для подсчета лавинного эффекта была написана программа, которая считает количество измененных бит (код в Приложении А).

Таблица 2. Количество измененных бит

	Keccak(SHA-3-512)
Добавление символа	266
Замена символа	264
Удаление символа	243
Среднее	258

Контроль целостности по коду HMAC

6.3. Задание

- Выбрать текст на английском языке (не менее 1000 знаков), добавить собственное ФИО и сохранить в файле формата .TXT

- Придумать пароль и сгенерировать секретный ключ утилитой Indiv.Procedures->Hash-> Key Generation из Cryptool 1. Сохранить ключ в файле формата .TXT. Прочитать Help к этой утилите.
- Сгенерировать HMAC для имеющегося текста и ключа с помощью утилиты Indiv.Procedures->Hash-> Generation of HMACs. Сохранить HMAC в файле формата .TXT. Прочитать Help к этой утилите.
- Передать пароль, HMAC (и его характеристики), исходный текст и модифицированный текст коллеге, не раскрывая, какой текст является корректным. Попросите коллегу определить это самостоятельно.

6.3.1. Выбранная схема генерации ключа и ее параметры

Пароль: veronika

Хеш-функция: SHA-1

Полученный ключ: EC B1 18 A3 12 86 10 39 DA C0 6F BD 1C DB 3F 59 DF DC 95 27

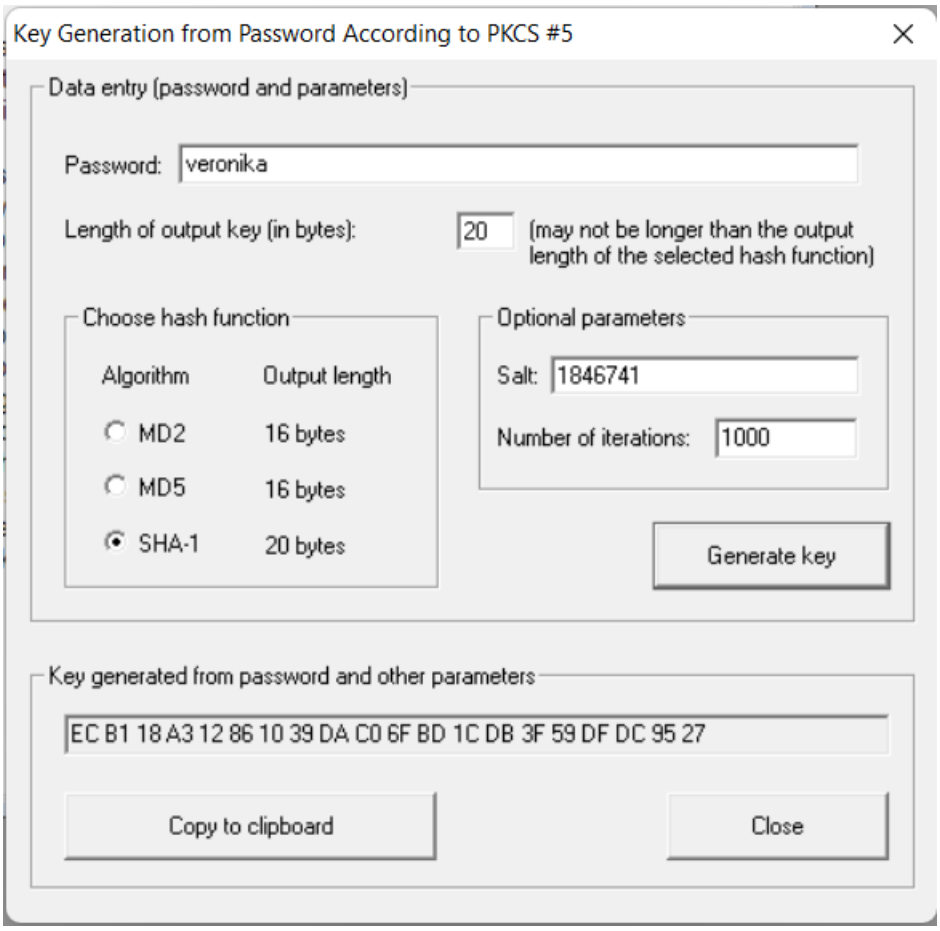


Рисунок 27 – Генерация секретного ключа

6.3.2. Выбранная схема создания HMAC

Схема: H(k, m)

Хеш-функция: SHA-1

Полученный HMAC: 2A E6 CA 85 FD EB E0 B5 99 03 61 7D 2A D6 76 C5 E7 25 45 F1

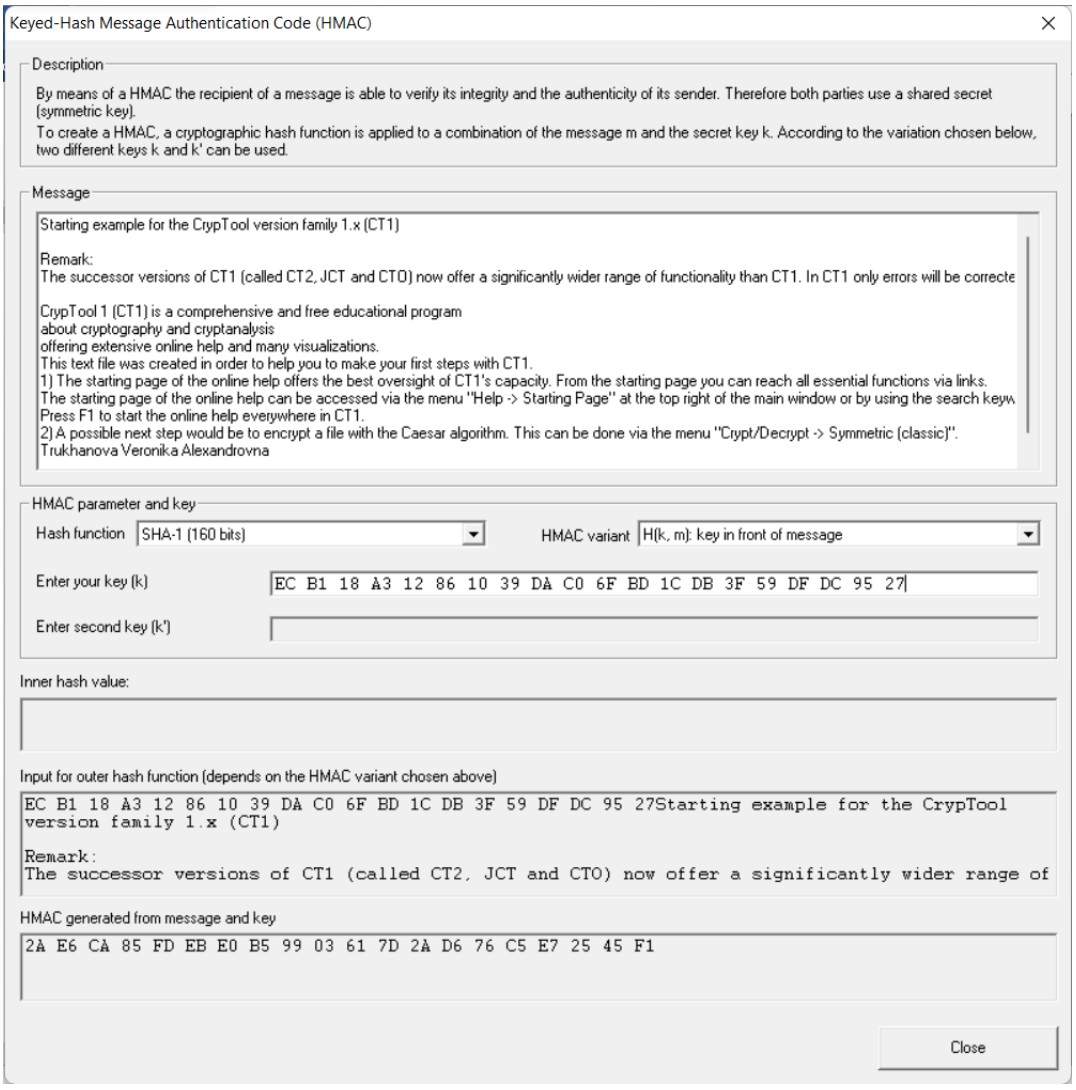


Рисунок 28 – Генерация HMAC

6.3.3. Описание действия передающей стороны

- Выбрать текст на английском языке (не менее 1000 знаков);
- Придумать пароль и сгенерировать секретный ключ;
- Сгенерировать HMAC для имеющегося текста и ключа;
- Модифицировать начальный текст;

- Передать пароль, HMAC (и его характеристики), исходный и модифицированный тексты, не раскрывая, какой текст корректен.

6.3.4. Описание действий принимающей стороны

Полученные файлы содержат следующие данные:

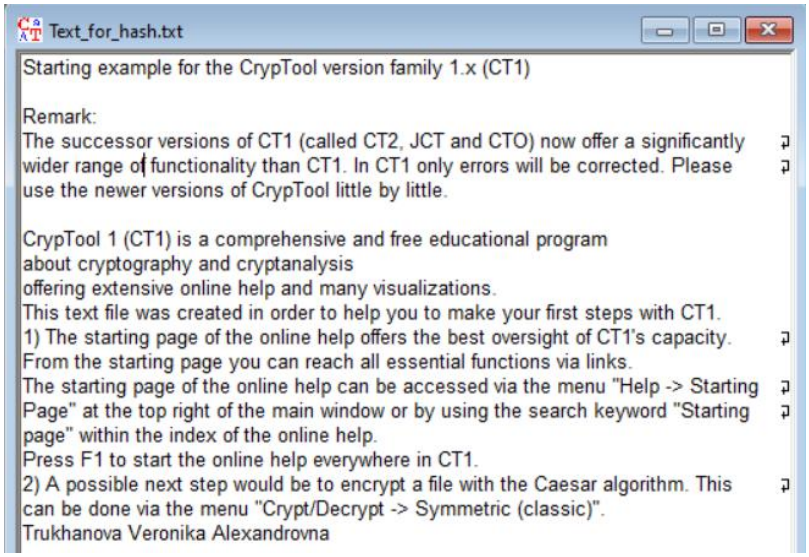


Рисунок 29 – Первый полученный текст

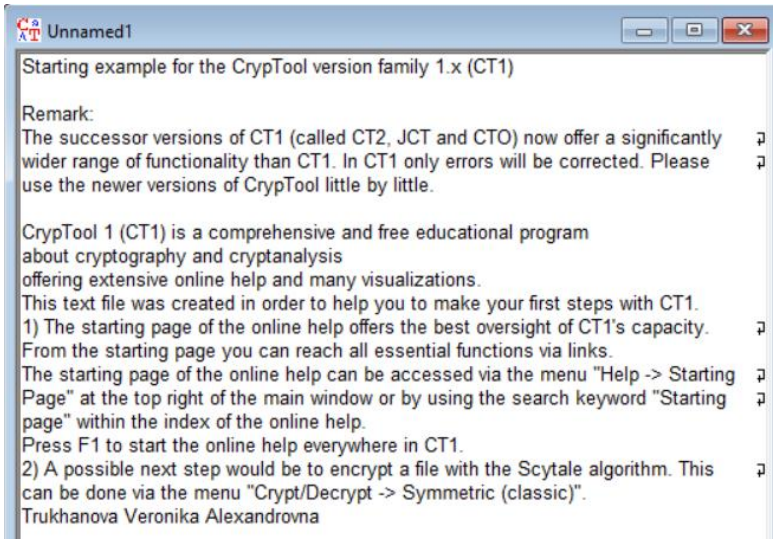


Рисунок 30 – Второй полученный текст

HMAC: 2A E6 CA 85 FD EB E0 B5 99 03 61 7D 2A D6 76 C5 E7 25 45 F1 при H(k,m).

Ключ: Veronika (соль = 1846741, SHA-1)

Для того, чтобы проверить, какой текст является корректным, получатель генерирует ключ, затем хэширует каждое из полученных сообщений. То сообщение, HMAC которого оказался идентичен полученному является корректным.

HMAC 1-го сообщения: 2A E6 CA 85 FD EB E0 B5 99 03 61 7D 2A D6 76 C5 E7 25 45

HMAC 2-го сообщения: C0 59 B3 FC ED 7C 38 34 23 33 83 2C FE E3 3F 70 C2 64 33 18

Следовательно, 1-е сообщение корректно.

Атака дополнительной коллизии на хеш-функцию

6.4. Задание

- Сформировать два текста на английском языке - один истинный, а другой фальсифицированный. Сохранить тексты в файлах формата *.txt
- Утилитой *Analysis-> Attack on the hash value...* произвести модификацию сообщений для получения одинакового дайджеста. В качестве метода модификации выбрать *Attach characters-> Printable characters.*
- Проверить, что дайджесты сообщений действительно совпадают с заданной точностью.
- Сохранить исходные тексты, итоговые тексты и статистику атаки для отчета.
- Зафиксировать временную сложность атаки для 8, 16, 32,40, 48, … бит совпадающих частей дайджестов.

6.4.1. Описание атаки в терминах парадокса «дней рождения»

Рассмотрим 4 парадокса дней рождения:

- Парадокс 1: Каково минимальное число k- студентов в аудитории, такое, что с некоторой вероятностью по крайней мере один студент имеет заранее заданный день рождения?
- Парадокс 2: Каково минимальное число k - студентов в аудитории, такое, что с некоторой вероятностью по крайней мере один студент имеет тот же самый день рождения, как и студент, выбранный преподавателем?
- Парадокс 3: Каково минимальное число k - студентов в аудитории, такое, что с заданной вероятностью по крайней мере два студента имеют тот же самый день рождения?

- Парадокс 4: Имеются две аудитории, каждая с k студентами. Каково минимальное значение k, такое, чтобы по крайней мере один студент из первой аудитории с некоторой вероятностью имел тот же самый день рождения, что и студент из второй аудитории?

На основе парадоксов дней рождения могут построить несколько типов атак на хэш-функции, включая:

- Атака прообраза: Злоумышленник перехватил дайджест D; он хочет найти любое сообщение M', такое, что $D = h(\mathrm{M}')$. Злоумышленник может создать список k сообщений и выполнить поиск. Вероятность успеха, как в парадоксе 1, асимптотическая сложность атаки $O(2^n)$.
- Атака второго прообраза: Злоумышленник перехватил сообщение M и хочет найти другое сообщение M', такое, чтобы что $h(M) = h(\mathrm{M}')$. Злоумышленник может создать список из k – 1 сообщений и выполнить поиск. Вероятность успеха как в парадоксе 2, асимптотическая сложность атаки $O(2^n)$.
- Атака коллизии: Злоумышленник должен найти два сообщения, M и M', такие, что $h(M) = h(\mathrm{M}')$. Он может создать список из k сообщений и выполнить эту проверку для всех пар. Вероятность успеха как в парадоксе 3, асимптотическая сложность атаки $O(2^{n/2})$.
- Дополнительная атака коллизии: Злоумышленник создает k различных вариантов $\mathrm{M}(M_1, M_2, \ldots, M_k)$ и k различных вариантов $\mathrm{M}'(M_1', M_2', \ldots, M_k')$. Оба набора значимы по содержанию. Затем злоумышленник выполняет проверку для всех пар из наборов. Для создания таких пар сообщений, злоумышленник сначала находит коллизию между двумя сообщениями, затем добавляет к этим сообщениям произвольную последовательность битов и находит новые сообщения, дающие тот же хеш-код, что и исходные сообщения. Таким образом, злоумышленник получает две пары сообщений, которые имеют различную длину и содержат различную информацию, но дают одинаковый хеш-код. Вероятность успеха как в парадоксе 4, асимптотическая сложность атаки $O(2^{n/2})$.

6.4.2. Представление результатов атаки

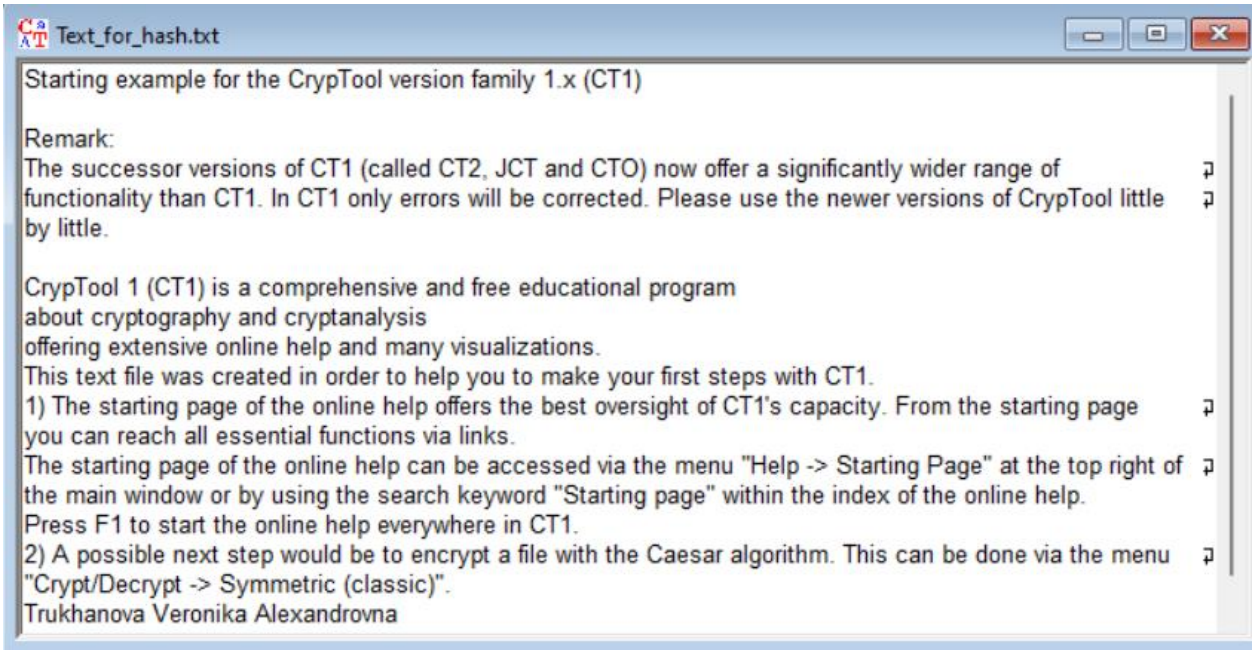


Рисунок 31 – Истинный текст

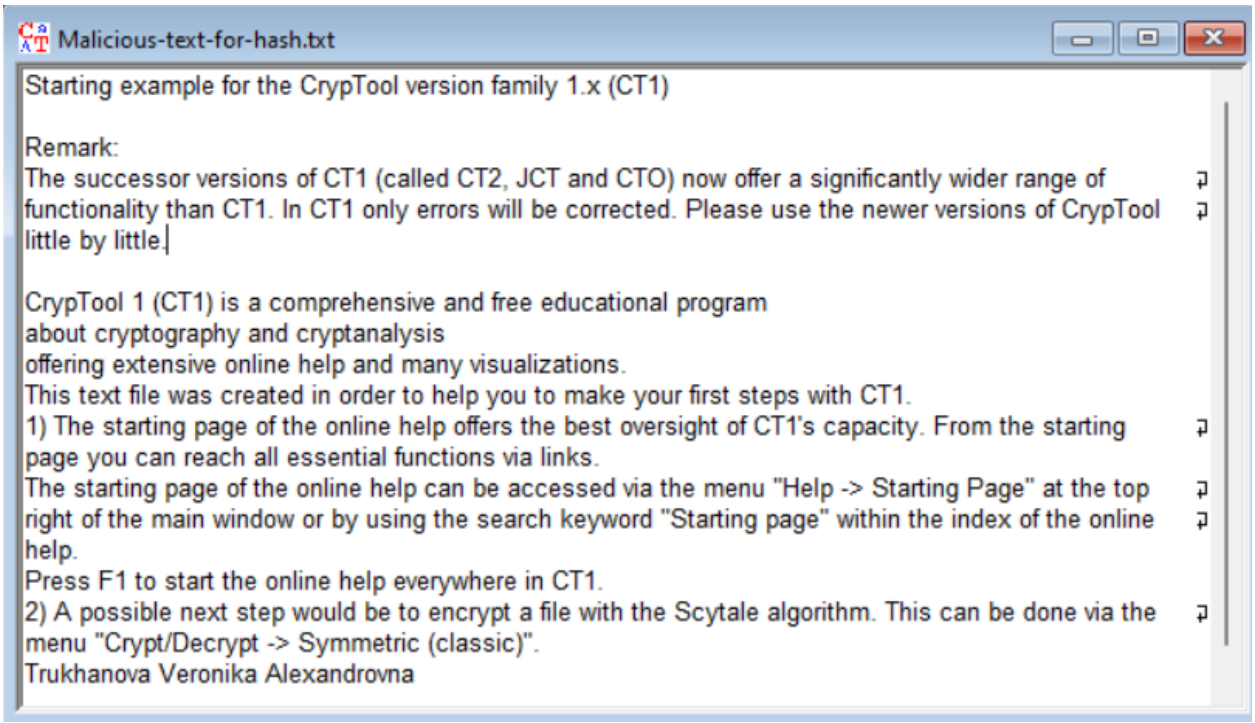


Рисунок 32 – Фальсифицированный текст

С помощью утилиты «Attack on the hash value of the digital signature» произведем модификацию сообщений для получения одинакового дайджеста.

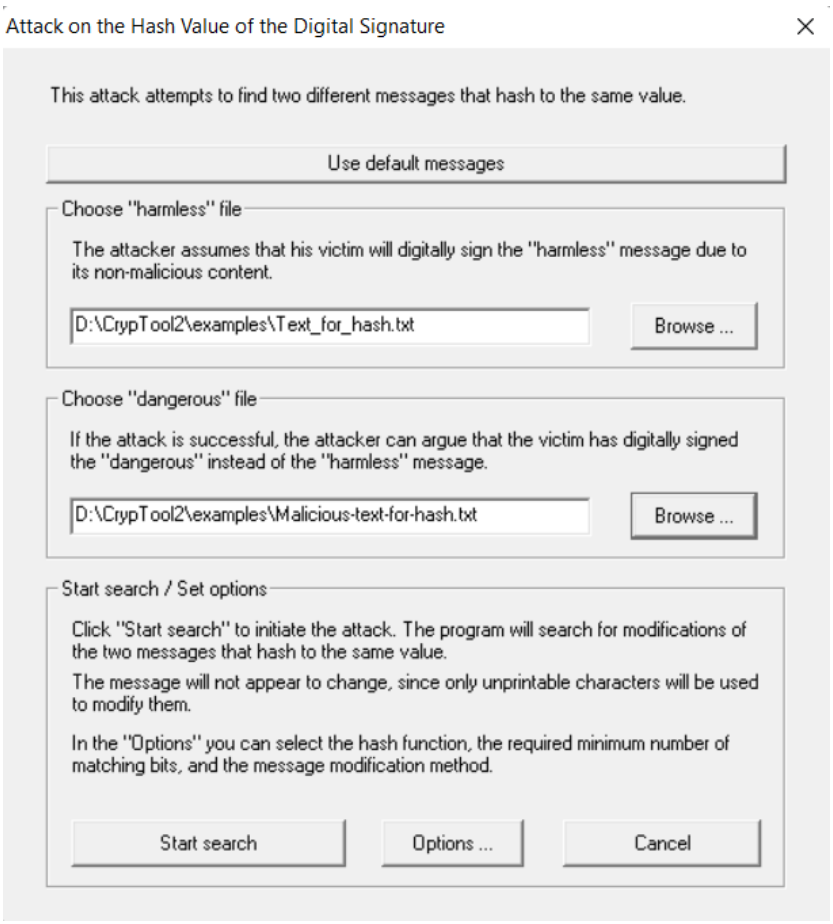


Рисунок 31 – Реализация атаки

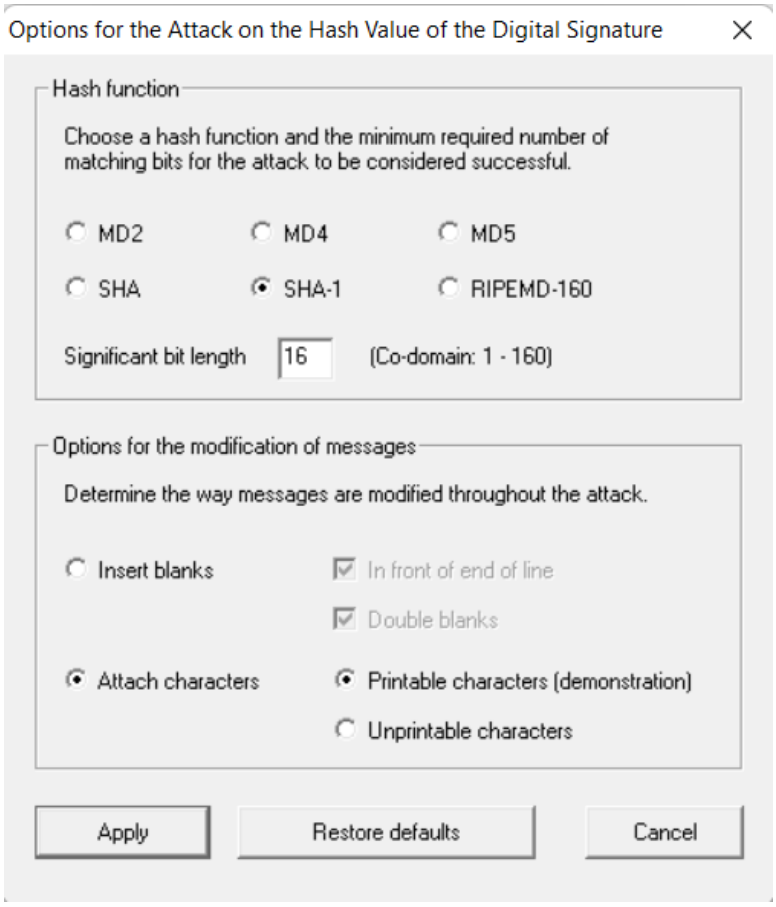


Рисунок 32 – Выбор параметров

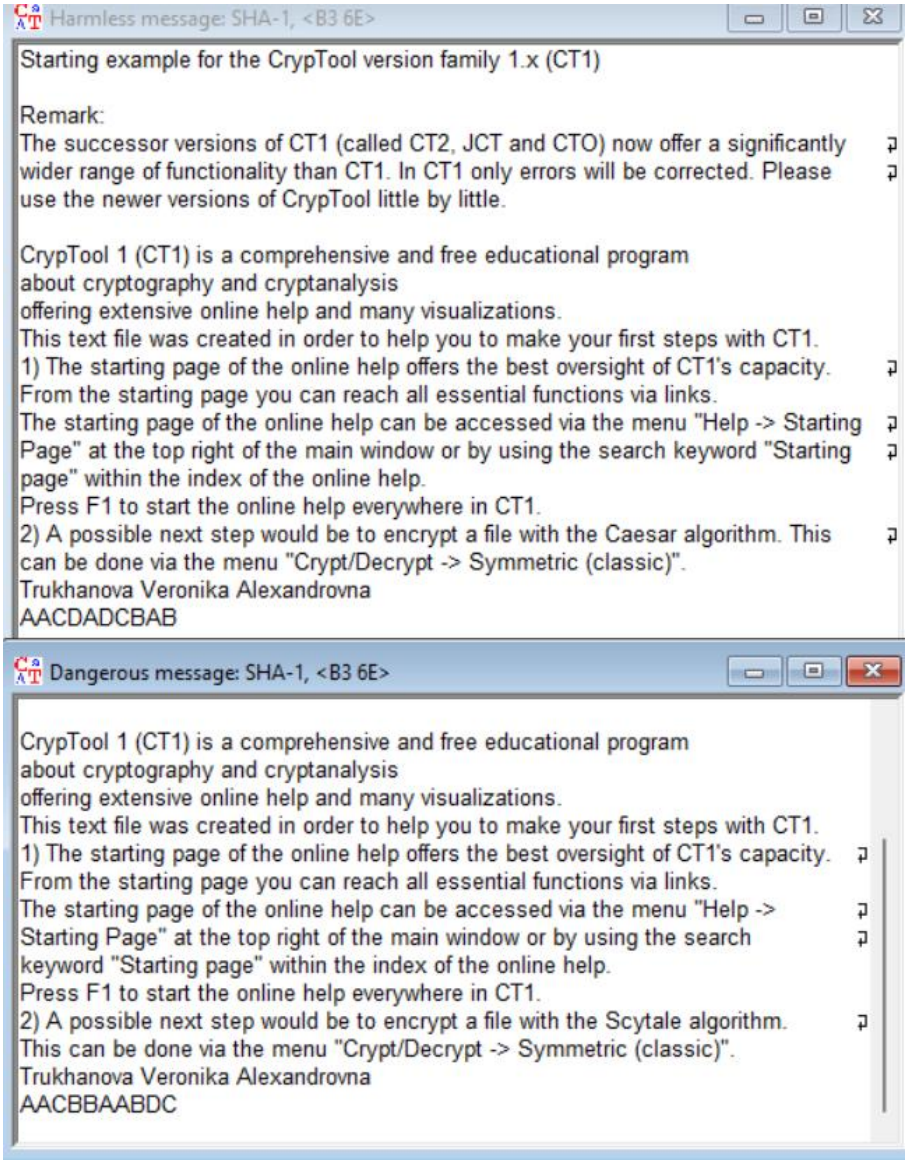


Рисунок 33 – Итоговые сообщения после атаки

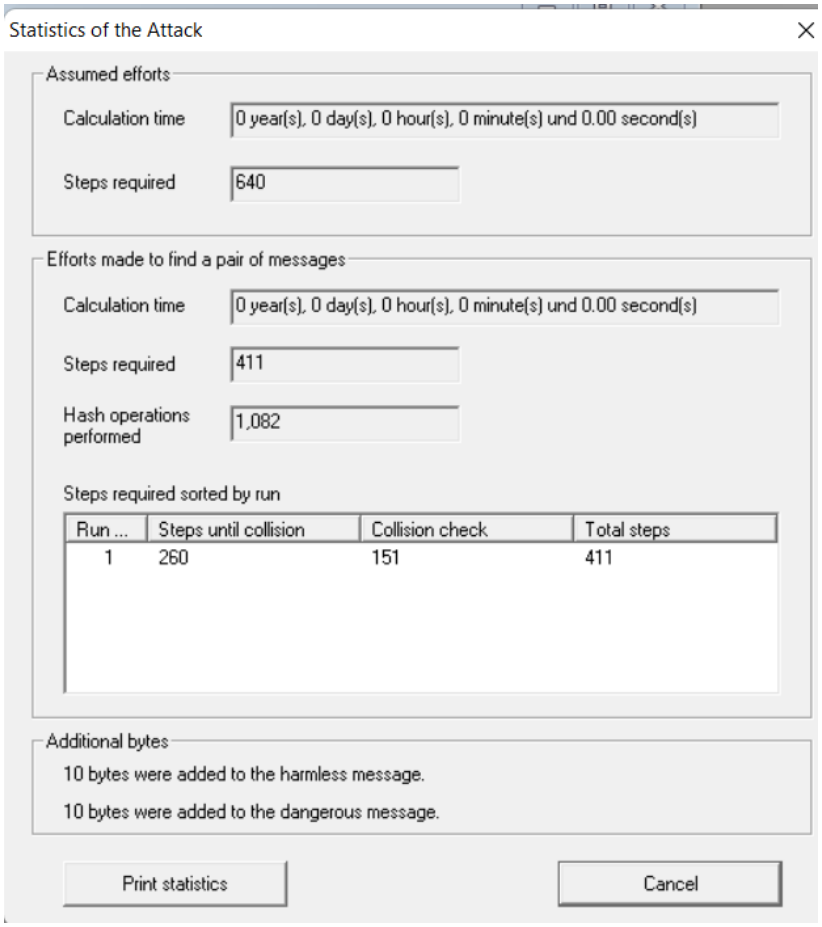


Рисунок 34 – Статистика атаки

SHA-1 хэш измененного истинного текста: B3 6E 53 0C 9A DD 0D B3 60 25 FB 48 13 08 49 69 4C AD 5E 68

SHA-1 хэш измененного фальсифицированного текста: B3 6E 37 B8 86 CB 06 D8 E2 CE F0 3B D2 FC 8F DC A9 9F 06 A8

После проведенной атаки первые 16 бит совпадают

6.4.3. Таблица с оценками временной сложности

Таблица 2. Временная сложность атаки

Количество совпадающих частей дайджестов	Время
8	0 с
16	0 с
32	1.72 с
40	27.56 с
48	5 минут
56	1 час

64	22 часа
72	15.3 дня
80	247 дней

Заключение

Название	**Размер дайджеста**	**Размер входного блока**	**Лавинный эффект**
SHA-1	160 бит	512 бит	49%
SHA-256	256 бит	512 бит	50%
SHA-512	512 бит	1024 бит	49%
MD5	128 бит	512 бит	53%
SHA-3 (Keccak)	256 бит 512 бит	25 50 100 200 400 800 1600	Для SHA-512(f-1600) 50%

По итогу выполнения данной работы были сделаны следующие выводы:

1. Среди хеш-функций MD5, SHA-1, SHA-256 и SHA-512 самым высоким лавинным эффектом (53% в среднем) обладает функция MD5.
2. SHA-3 также обладает высоким лавинным эффектом равным в среднем 50%. Для оценки лавинного эффекта была написана программа на языке Go.
3. Для проверки целостности и подлинности сообщения можно использовать HMAC – код, основанный на хешировании. Для того чтобы проверить является корректным полученный текст, получатель генерирует ключ из пароля, и потом генерирует HMAC. Если сгенерированный HMAC полученному, то текст является корректным, иначе текст был модифицирован.
4. Атака дополнительной коллизией является не эффективной, так как за время подбора дополнения, исходный перехваченный файл успеет утратить свою ценность.

ПРИЛОЖЕНИЕ А

Исходный код процедуры расчёта лавинного эффекта, написанный на языке Go:

```
package main

import (
	"fmt"
	"strconv"
	"strings"
	"bufio"
	"os"
)

//функция перевода строки из шестнадцатеричной системы в
//двочиную
func conv(hex_str1, hex_str2 [] string) ([64]string,[64]string){
	var bin1, bin2 [64]string
	for i , v := range hex_str1 {
		j, _:= strconv.ParseInt(v, 16, 64)
		bin1[i] = strconv.FormatInt(j, 2)
	}

	for i , v := range hex_str2 {
		j, _:= strconv.ParseInt(v, 16, 64)
		bin2[i] = strconv.FormatInt(j, 2)
	}

	return bin1, bin2

}

//функция сравнения битов двух строк
func comp(bin1, bin2 [64]string) int {
	count:= 0

	for i := range bin1{
		b1 := 8 - len(bin1[i])
		if b1 > 0 {
			for j:=0; j < b1; j++ {
				bin1[i] = "0" + bin1[i]
			}
		}
	}

	for i := 0; i < 64; i++{
		b2 := 8 - len(bin2[i])
		if b2 > 0 {
			for j:=0; j < b2; j++ {
				bin2[i] = "0" + bin2[i]
			}
		}
```

```
     }


     for i := range bin1 {
          for j := 0; j < 8; j++{
               if bin2[i][j] == bin1[i][j]{
                    count++
               }
          }
     }
     return count
}


func main() {
     var hex_str1, hex_str2 string
     var bin1, bin2 [64] string

     fmt.Println("Введите оригинальный хэш")
     hex_str1, _ = bufio.NewReader(os.Stdin).ReadString('\n')
     fmt.Println("Введите измененный хэш")
     hex_str2, _ = bufio.NewReader(os.Stdin).ReadString('\n')

     bin1, bin2 = conv(strings.Fields(hex_str1),
strings.Fields(hex_str2))

     fmt.Print (comp(bin1, bin2))
}
```