Учреждение образования

«Белорусский государственный технологический университет»

**Отчет по лабораторной работе №9**

по дисциплине: «Защита информации и надежность информационных систем»

Исполнитель:

Воликов Д. А., ФИТ 4-7

Руководитель:

Асс. Савельева М. Г.

Минск 2024

# Лабораторная работа №9. Исследование криптографических хеш-функций

**Цель**: изучение алгоритмов хеширования и приобретение практических навыков их реализации и использования в криптографии.

**Задачи**:

1. Закрепить теоретические знания по алгебраическому описанию и алгоритмам реализации операций вычисления однонаправленных хеш-функций.
2. Освоить методику оценки криптостойкости хеш-преобразований на основе «парадокса дня рождения».
3. Разработать приложение для реализации заданного алгоритма хеширования (из семейств MD и SHA).
4. Оценить скорость вычисления кодов хеш-функций.
5. Результаты выполнения лабораторной работы оформить в виде описания разработанного приложения, методики выполнения экспериментов с использованием приложения и результатов эксперимента.

## Теоретические сведения

Определение 1. Хеш-функция – математическая или иная функция *h* = *H*(*М*), которая принимает на входе строку символов *М*, называемую также прообразом, переменной длины *n* и преобразует ее в выходную строку фиксированной (обычно – меньшей) длины *l*.

Определение 2. Хеширование (или хэширование, англ. hashing) – это преобразование входного массива данных определенного типа и произвольной длины (практически) в выходную битовую строку фиксированной длины.

Преобразования называются хеш-функциями, или функциями свертки, а их результаты называют хешем, хеш-кодом, хеш-таблицей или дайджестом сообщения (англ. message digest).

Определение 3. Криптографическая хеш-функция – это специальный класс хеш-функций, который имеет различные свойства, необходимые для решения задач в области криптографии.

Основные задачи, решаемые с помощью хеш-функций:

* аутентификация (хранение паролей);
* проверка целостности данных;
* защита файлов;
* обнаружение зловредного ПО;
* криптовалютные технологии.

К основным свойствам хеш-функций можно отнести следующие.

**Свойство 1**. Детерминированность: независимо от того, сколько раз вычисляется *H*(*M*), *M* – const, при использовании одинакового алгоритма код хеш-преобразования h всегда должен быть одинаковым.

**Свойство 2**. Скорость вычисления хеша *h*: если процесс вычисления *h* недостаточно быстрый, система просто не будет эффективной.

**Свойство 3**. Сложность обратного вычисления: для известного *H*(*М*) невозможно (практически) определить *М*. Это важнейшее свойство хеш-функции для криптографических применений – свойство односторонности преобразования.

Это означает, что по хеш-коду должно быть практически невозможно восстановление входной строки *М*.

Свойство 4. Даже минимальные изменения в хешируемых данных (*М* ≠ *М*') должны изменять хеш: *Н*(*M*) ≠ *Н*(*М*').

**Определение 4**. Коллизией хеш-функции *Н* называют ситуацию, при которой различным входам (в общем случае – *х* и *у* или *М* ≠ *М*') соответствует одинаковый хеш-код: *H*(x) = *H*(y) или *H*(*М*) = *H*(*М*').

**Свойство 5**. Коллизионная устойчивость (стойкость).

Зная *М*, трудно найти такое *М*' (*М* ≠ *М*'), для которого *H*(*М*) = *H*(*М*').

Если последнее равенство выполняется, то говорят о коллизии 1-го рода.

Если случайным образом выбраны два сообщения (*М* и *М*'), для которых *H*(*М*) = *H*(*М*'), говорят о коллизии 2-го рода.

Мерой криптостойкости хеш-функции считается вычислительная сложность нахождения коллизии.

## Практическое задание

1. Разработать оконное приложение, реализующее алгоритм хеширования SHA-1. При этом можно воспользоваться доступными готовыми библ
2. иотеками. Язык программирования – *C*#.

Как и любого другого алгоритма хэширования, следует выделить основные операции, которые задействованы в процессе вычисления хэш-значения.

Во-первых, сообщение следует расширить, если длина битов сообщения не кратна 512. Это делается для того, чтобы точно разделить сообщение на блоки по 512 бит.

Процесс расширения сообщения. Сначала к исходному сообщению добавляется 1 бит (в виде байта «0*х*80»), чтобы начать процесс выравнивания. Далее добавляются нулевые биты (в виде байта «0*x*00»), чтобы длина сообщения имела остаток 448 при делении на 512. Это обеспечивает, что оставшиеся 64 бита можно будет использовать для хранения длины исходного сообщения. В конце добавляется 64-битное представление длины исходного сообщения (в битах), записанное в формате big-endian. Это позволяет алгоритму знать, сколько бит было в исходном сообщении.

Таким образом, блок имеет структуру, которая описана на рисунке ниже.

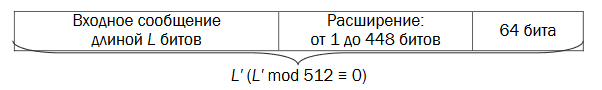


Рисунок 1 – Состав и размеры дополненного хэшируемого сообщения

Основная операция хэширования заключается в циклической (пораундовой или поэтапной) обработке 512-битных блоков. Таких циклов в алгоритме SHA-1 ровно 4. В каждом цикле используется своя нелинейная функция (обычно обозначаемая по порядку *F*, *G*, *H*, *I*), которые меняют начальные значения переменных *A*, *B*, *C*, *D* и *E*. После всех расчётов значения этих переменных конкатенируются и возвращаются, как результат хэширования.

Подробный ход операций одного раунда можно увидеть на рисунке ниже.

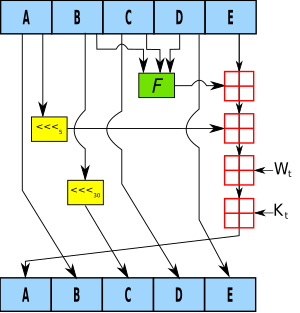


Рисунок 2 – Общая схема раунда SHA-1

Теперь посмотрим, как вычисляются значения и правильно ли вообще.

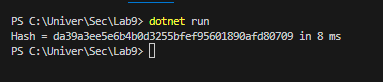


Рисунок 3 – Хэш-значение пустой строки программы

Как можем заметить, программа вернула значение результата расcчитанного хэша в 16-ой системе счисления, которое равно «da39a3ee5e6b4b0d3255bfef95601890afd80709».

Теперь сравним с онлайн решением для вычисления хэша.

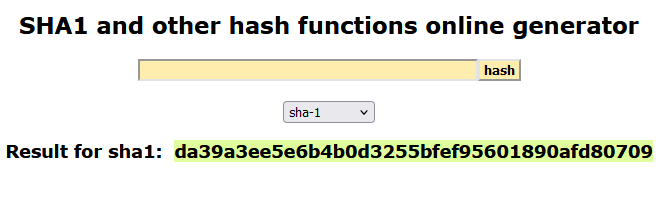


Рисунок 4 – Хэш-значение пустой строки онлайн сервиса

Как можно заметить, хэш значения абсолютно одинаковы. Протестируем на другом тексте. Пусть *M* = «hello». Результаты вычислений хэш-значений для нового сообщения можно увидеть на рисунке ниже.



Рисунок 5 – Сравнение хэш-значений для нового сообщения

Как можно заметить, результаты снова одинаковые, что подтверждает правильность реализации алгоритма хэширования SHA-1.

1. Оценить быстродействие выбранного алгоритма хеширования.

График оценки скорости алгоритма хэширования SHA-1 можно увидеть на рисунке 6.

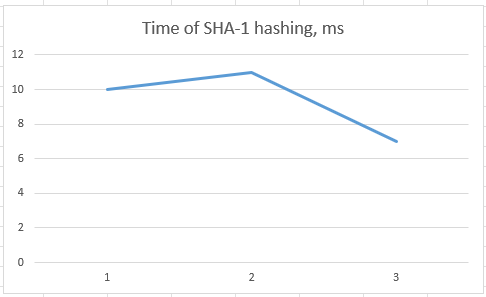


Рисунок 6 – График скорости алгоритма в мс

Алгоритм работает довольно быстро, что говорит об эффективности использования данного решения в каких-либо проектах.

Вывод: в ходе этой лабораторной работы я закрепил знания об алгоритмах хэширования, полученных на лекциях. На основе этого я разработал приложение, в котором реализован алгоритм хэширования SHA-1. Оценил скорость хэширования сообщений, а также правильность результатов, сравнивая с онлайн сервисами для хэширования.

# Листинг «Лабораторная работа №9»

using System.Text;

namespace Lab9.Hash

{

public class SHA1

{

private static uint H0 = 0x67452301;

private static uint H1 = 0xEFCDAB89;

private static uint H2 = 0x98BADCFE;

private static uint H3 = 0x10325476;

private static uint H4 = 0xC3D2E1F0;

private static readonly uint[] K = [

0x5A827999,

0x6ED9EBA1,

0x8F1BBCDC,

0xCA62C1D6

];

public static string Hash(string message)

{

ResetHash();

byte[] paddedMessage = GetPaddedMessage(message);

for (int i = 0; i < paddedMessage.Length; i += 64)

{

HashBlock([.. paddedMessage.Skip(i).Take(64)]);

}

string hash = $"{H0:x8}{H1:x8}{H2:x8}{H3:x8}{H4:x8}";

ResetHash();

return hash;

}

private static byte[] GetPaddedMessage(string message)

{

List<byte> result = [.. Encoding.UTF8.GetBytes(message)];

uint messageLength = (uint)(result.Count \* 8);

int lastBits = result.Count \* 8 % 512;

if (lastBits != 0 || message.Length == 0)

{

result.Add(0x80);

int paddingLength = (448 - result.Count \* 8 % 512) / 8;

if (paddingLength < 0)

{

paddingLength += 64;

}

result.AddRange(Enumerable.Repeat((byte)0x0, paddingLength));

byte[] lengthBytes = BitConverter.GetBytes(messageLength);

Array.Reverse(lengthBytes);

result.AddRange(new byte[8 - lengthBytes.Length]);

result.AddRange(lengthBytes);

}

return [.. result];

}

private static void HashBlock(byte[] block)

{

uint A = H0;

uint B = H1;

uint C = H2;

uint D = H3;

uint E = H4;

uint[] W = new uint[80];

for (int t = 0; t < 16; t++)

{

W[t] = (uint)block[t \* 4] << 24;

W[t] |= (uint)block[t \* 4 + 1] << 16;

W[t] |= (uint)block[t \* 4 + 2] << 8;

W[t] |= block[t \* 4 + 3];

}

for (int t = 16; t < 80; t++)

{

W[t] = SHA1CircularShift(1, W[t - 3] ^ W[t - 8] ^ W[t - 14] ^ W[t - 16]);

}

for(int t = 0; t < 20; t++)

{

var temp = SHA1CircularShift(5, A) + ((B & C) | ((~B) & D)) + E + W[t] + K[0];

E = D;

D = C;

C = SHA1CircularShift(30, B);

B = A;

A = temp;

}

for(int t = 20; t < 40; t++)

{

var temp = SHA1CircularShift(5, A) + (B ^ C ^ D) + E + W[t] + K[1];

E = D;

D = C;

C = SHA1CircularShift(30, B);

B = A;

A = temp;

}

for(int t = 40; t < 60; t++)

{

var temp = SHA1CircularShift(5, A) + ((B & C) | (B & D) | (C & D)) + E + W[t] + K[2];

E = D;

D = C;

C = SHA1CircularShift(30, B);

B = A;

A = temp;

}

for(int t = 60; t < 80; t++)

{

var temp = SHA1CircularShift(5, A) + (B ^ C ^ D) + E + W[t] + K[3];

E = D;

D = C;

C = SHA1CircularShift(30, B);

B = A;

A = temp;

}

H0 += A;

H1 += B;

H2 += C;

H3 += D;

H4 += E;

}

private static uint SHA1CircularShift(int bits, uint word) => (word << bits) | (word >> (32 - bits));

private static void ResetHash()

{

H0 = 0x67452301;

H1 = 0xEFCDAB89;

H2 = 0x98BADCFE;

H3 = 0x10325476;

H4 = 0xC3D2E1F0;

}

}

}

Листинг 1 – Код алгоритма хэширования SHA-1