МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет информационных технологий

Кафедра информационных систем и технологий

Специальность Информационные системы и технологии

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №9 НА ТЕМУ:

Исследование криптографических хеш-функций

Выполнила студентка 3 курса 1 группы

Пригодич Вера Валерьевна

Минск 2023

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

**Теоретические сведения**

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.*

Хеширование – это преобразование входного массива данных определенного типа и произвольной длины (практически) в выходную битовую строку фиксированной длины.

К основным свойствам хеш-функций можно отнести следующие:

1. Детерминированность.
2. Скорость вычисления хеша *h*.
3. Сложность обратного вычисления.
4. Даже минимальные изменения в хешируемых данных (*М* ≠ *М*') должны изменять хеш: *Н*(*M*) ≠ *Н*(*М*').
5. Коллизионная устойчивость (стойкость)

Мерой криптостойкости хеш-функции считается вычислительная сложность нахождения коллизии. Для хеш-функций одним из основных средств поиска коллизий является метод, основанный на известной статистической задаче – «парадоксе дня рождения». В более общем случае: для того чтобы хеш-функция *H*(*M*) считалась криптографически стойкой, она должна удовлетворять трем основным требованиям: необратимостью вычислений (свойство 3), устойчивостью к коллизиям первого рода и устойчивостью к коллизиям второго рода.

Основной постулат парадокса «дней рождения» гласит: в группе минимум из 23 человек с вероятностью более 0,5 день рождения одинаков хотя бы у двух членов группы. Парадоксом является высокая (как кажется на первый взгляд) вероятность наступления указанного события. При этом предполагается, что:

• в этой группе нет близнецов;

• люди рождаются независимо друг от друга, т. е. дата (день) рождения любого человека не влияет на дату рождения другого;

• люди рождаются равномерно и случайно, т. е. люди с равной вероятностью могут рождаться в любой день года; с формальной точки зрения это означает, что вероятность *р*1 рождения отдельно выбранного члена группы (как и любого человека) в любой выбранный день равна *р*1 = 1 / 365 (хотя известно, что в реальности рождение людей не совсем соответствует такому предположению).

При атаке «дней рождения» злоумышленник будет случайным образом подбирать *Мi*и *Мj*и сохранять пары их хешей, пока не найдет двух значений, при которых *h*(*Мi*) = *h*(*Мj*).

Иначе говоря, стоит задача отыскания наименьшего *n*, при котором хеши двух значений *m* будут «одним днем рождения».

Устойчивость хеш-функции к коллизиям имеет первостепенное значение в технологиях электронной цифровой подписи и криптовалютных технологиях. В настоящее время длина хеша *l* = 64 не относится к числу криптостойких.

Алгоритмы семейства MD-x (2/4/5/6) являются творениями Р. Ривеста; MD – Message Digest. Алгоритм MD6, в отличие от предыдущих версий алгоритма этого семейства, не стандартизован. Алгоритмы семейства SHA (SHA – Secure Hash Algorithm) являются в настоящее время широко распространенными. По существу, во многих случаях завершился переход от SHA-1 к стандартам версии SHA-2. SHA-2 – собирательное название алгоритмов SHA-224, SHA-256, SHA-384 и SHA-512. SHA-224 и SHA-384 являются, по сути, аналогами SHA-256 и SHA-512 соответственно.

5 августа 2015 г. утвержден и опубликован в качестве действующего стандарта (FIPS 202) алгоритм SHA-3, одной из отличительных особенностей которого является использование конструкции «криптографической губки». В этой конструкции реализован итеративный подход для создания функции с произвольной длиной на входе и произвольной длиной на выходе на основе определенного преобразования.

Алгоритмы семейства MD входные сообщения максимальной длины 264 – 1 битов (в общем случае – *L* битов) преобразуют в хеш длиной *l* = 128 битов. Исключением является последняя (6-я) из версий алгоритма, где длина результирующего хеша может изменяться от 1 до 512 бит.

Максимальный объем хешируемых сообщений для алгоритмов SHA-1, SHA-256, SHA-224 такой же, как и для алгоритмов MD. Однако длина хешей разная: в SHA-1 – 160 битов; в алгоритмах, относящихся к семейству SHA-2, – соответствует числу, дополняющему через дефис название алгоритма. Максимальная же длина входных сообщений в алгоритмах SHA-512, SHA-384, SHA-512/256, SHA-512/224 составляет 2128 – 1 битов.

Базовые алгоритмы обоих рассматриваемых семейств (MD и SHA) условно можно разделить на 5 стадий:

• расширение входного сообщения;

• разбивка расширенного сообщения на блоки;

• инициализация начальных констант;

• обработка сообщения поблочно (основная процедура алгоритма хеширования);

• вывод результата

**Практическое задание**

1. Разработать оконное приложение, реализующее один из алгоритмов хеширования из указанного преподавателем семейства (MD или SHA; или иного). При этом можно воспользоваться доступными готовыми библиотеками. Язык программирования – на свой выбор. Приложение должно обрабатывать входные сообщения, длина которых определяется спецификацией на реализуемый алгоритм.

Был реализован алгоритм MD5.

Предположим, что хешируемое сообщение буква «v». Переведем ее в ASCII. Получим 01110110.

Шаг 1. Сначала к концу потока дописывают единичный бит.

Получаем 011101101.

Затем добавляют некоторое число нулевых бит такое, чтобы новая длина потока стала сравнима с 448 по модулю 512. Выравнивание происходит в любом случае, даже если длина исходного потока уже сравнима с 448.

Получаем 0111011010000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000.

Программно это реализовано следующим образом:

    def \_step\_1(cls):

        # Заметка: входные байты представлены строкой из бит,

        # причем первый бит — старший (big-endian).

        bit\_array = bitarray(endian="big")

        bit\_array.frombytes(cls.\_string.encode("utf-8"))

        # Подготовка: добавляем бит "1" в конец сообщения.

        bit\_array.append(1)

        # Подготовка: дописываем нулевые биты, пока длина сообщения не станет сравнима с 448 по модулю 512

        while len(bit\_array) % 512 != 448:

            bit\_array.append(0)

        # (Результат в формате little-endian)

        return bitarray(bit\_array, endian="little")

Листинг 1 – Шаг 1

Шаг 2. В конец сообщения дописывают 64-битное представление длины данных (количество бит в сообщении) до выравнивания. Сначала записывают младшие 4 байта, затем старшие. Если длина превосходит 264−1то дописывают только младшие биты (эквивалентно взятию по модулю 264). После этого длина потока станет кратной 512. Вычисления будут основываться на представлении этого потока данных в виде массива слов по 512 бит.

Получаем:

0111011010000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000**1**000000000000000000000000000000000000000000000000000000000000.

Программно это реализовано следующим образом:

    def \_step\_2(cls, step\_1\_result):

        # Дописываем остаток от деления изначальной длины сообщения на 2^64

        length = (len(cls.\_string) \* 8) % pow(2, 64)

        length\_bit\_array = bitarray(endian="little")

        length\_bit\_array.frombytes(struct.pack("<Q", length))

        result = step\_1\_result.copy()

        result.extend(length\_bit\_array)

        print(result)

        return result

Листинг 2 – Шаг 2

Шаг 3. Для вычислений инициализируются четыре переменные размером по 32 бита, начальные значения которых задаются шестнадцатеричными числами (порядок байтов little-endian):

class MD5Buffer(Enum):

    A = 0x67452301

    B = 0xEFCDAB89

    C = 0x98BADCFE

    D = 0x10325476

class MD5(object):

    \_string = None

    \_buffers = {

        MD5Buffer.A: None,

        MD5Buffer.B: None,

        MD5Buffer.C: None,

        MD5Buffer.D: None,

    }

    def \_step\_3(cls):

        for buffer\_type in cls.\_buffers.keys():

            cls.\_buffers[buffer\_type] = buffer\_type.value

Листинг 3 – Шаг 3

В этих переменных будут храниться результаты промежуточных вычислений. Начальное состояние ABCD называется инициализирующим вектором.

Шаг 4. Определение функций и констант. Для каждого раунда потребуется своя функция. Определим таблицу констант T [1…64] — 64-элементная таблица данных. Каждый 512-битный блок проходит 4 этапа вычислений по 16 раундов. Для этого блок представляется в виде массива X из 16 слов по 32 бит. Заносим в блок данных элемент n из массива 512-битных блоков. Сохраняются значения A, B, C и D, оставшиеся после операций над предыдущими блоками (или их начальные значения, если блок первый).

def \_step\_4(cls, step\_2\_result):

        F = lambda x, y, z: (x & y) | (~x & z)

        G = lambda x, y, z: (x & z) | (y & ~z)

        H = lambda x, y, z: x ^ y ^ z

        I = lambda x, y, z: y ^ (x | ~z)

        rotate\_left = lambda x, n: (x << n) | (x >> (32 - n))

        modular\_add = lambda a, b: (a + b) % pow(2, 32)

        T = [floor(pow(2, 32) \* abs(sin(i + 1))) for i in range(64)]

        N = len(step\_2\_result) // 32

        # Разбиваем подготовленное сообщение на 512-битные "куски":

        for chunk\_index in range(N // 16):

            # разбиваем "кусок" на 16 блоков по 32 бита

            start = chunk\_index \* 512

            X = [step\_2\_result[start + (x \* 32) : start + (x \* 32) + 32] for x in range(16)]

            # Инициализируем переменные для текущего куска:

            A = cls.\_buffers[MD5Buffer.A]

            B = cls.\_buffers[MD5Buffer.B]

            C = cls.\_buffers[MD5Buffer.C]

            D = cls.\_buffers[MD5Buffer.D]

            # 4 раунда по 16 операций

            for i in range(4 \* 16):

                if 0 <= i <= 15:

                    k = i

                    s = [7, 12, 17, 22]

                    temp = F(B, C, D)

                elif 16 <= i <= 31:

                    k = ((5 \* i) + 1) % 16

                    s = [5, 9, 14, 20]

                    temp = G(B, C, D)

                elif 32 <= i <= 47:

                    k = ((3 \* i) + 5) % 16

                    s = [4, 11, 16, 23]

                    temp = H(B, C, D)

                elif 48 <= i <= 63:

                    k = (7 \* i) % 16

                    s = [6, 10, 15, 21]

                    temp = I(B, C, D)

                temp = modular\_add(temp, X[k])

                temp = modular\_add(temp, T[i])

                temp = modular\_add(temp, A)

                temp = rotate\_left(temp, s[i % 4])

                temp = modular\_add(temp, B)

                A = D

                D = C

                C = B

                B = temp

            # Прибавляем результат текущего "куска" к общему результату

            cls.\_buffers[MD5Buffer.A] = modular\_add(cls.\_buffers[MD5Buffer.A], A)

            cls.\_buffers[MD5Buffer.B] = modular\_add(cls.\_buffers[MD5Buffer.B], B)

            cls.\_buffers[MD5Buffer.C] = modular\_add(cls.\_buffers[MD5Buffer.C], C)

            cls.\_buffers[MD5Buffer.D] = modular\_add(cls.\_buffers[MD5Buffer.D], D)

Листинг 4 – Шаг 4

Шаг 5. Результат вычислений находится в буфере ABCD, это и есть хеш. Если выводить побайтово, начиная с младшего байта A и заканчивая старшим байтом D, то мы получим MD5-хеш.

    def \_step\_5(cls):

        A = struct.unpack("<I", struct.pack(">I", cls.\_buffers[MD5Buffer.A]))[0]

        B = struct.unpack("<I", struct.pack(">I", cls.\_buffers[MD5Buffer.B]))[0]

        C = struct.unpack("<I", struct.pack(">I", cls.\_buffers[MD5Buffer.C]))[0]

        D = struct.unpack("<I", struct.pack(">I", cls.\_buffers[MD5Buffer.D]))[0]

        return f"{format(A, '08x')}{format(B, '08x')}{format(C, '08x')}{format(D, '08x')}"

Листинг 5 – Шаг 5

В результате получаем захешрованное сообщение 4980d0be862848516678813555ef334a.

1. Оценить быстродействие выбранного алгоритма хеширования.

Быстродействие алгоритма хеширования является его важным свойством, так как если процесс вычисления *h* недостаточно быстрый, система просто не будет эффективной.

Для реализованного алгоритма время хеширования 1 буквы составило 0.001013994216918945 с.

Хеширование 200 слов (1069 символов) из «Превращение» Франца Кафки заняло 0.013066768646240234с.

Таким образом можно говорить, что алгоритм обладает высокой скоростью вычисления.

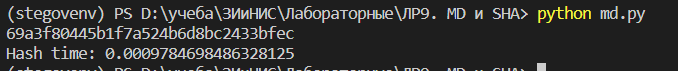


Рисунок 1 – Результат работы программы

Вывод: были изучены алгоритмы хеширования и приобретены практические навыки их реализации и использования в криптографии. В частности был реализован алгоритм MD5 и оценено его быстродействие.

**Контрольные вопросы**

1. Дать определение хеш-функции.

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.*

2. Что такое «однонаправленность» хеш-функций и какова роль этого свойства хеш-функций в криптографии?

Это свойство хеш-функции, заключающееся в сложности обратного вычисления: для известного *H*(*М*) невозможно (практически) определить *М*.

3. Что такое «коллизия»? Типы коллизий хеш-функций.

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*').

4. Сформулировать в общем виде парадокс «дней рождений».

Описанная задача может быть сформулирована в общем виде:

* даны случайные числа;
* случайные числа распределены равномерно в диапазоне от 1 до *d*;
* *n* — количество случайных чисел;
* определить *p* (*n*; *d*) — вероятность того, что совпадут хотя бы два числа из заданных.

5. Как парадокс «дней рождений» используется в криптографии?

Для выполнения анализа атаки на основе парадокса «дней рождения» будем использовать те же принципы, которые мы применяли для вероятностной оценки дней рождения. В атаке «дней рождения» m соответствует количеству календарных дней в году, а *М* – множеству людей, составляющих группу. Люди «хешируются» в их дни рождения, которые могут быть одним из значений *m*.

1. Сколько попыток нужно сделать, чтобы с вероятностью более 0,5 (0,7; 0,8; 0,9) обнаружить коллизию при длине хеша (*l*) 64 (128; 256; 512) битов?

Коллизия – это ситуация, когда два разных входных сообщения дают одинаковый хеш-код. Вероятность обнаружить коллизию зависит от длины хеша и количества входных сообщений, которые необходимо обработать.

По формуле:



где *N* – количество попыток, которое необходимо сделать, *l* – длина хеша в битах, *p* – вероятность обнаружения коллизии

Для вероятностей 0,5, 0,7, 0,8 и 0,9 будем иметь:

1. *l* = 64:

*P* = 0,5 -> *N* = 232 ≈ 4,3 млрд

*P* = 0,7 -> *N* = 5,9 млрд

*P* = 0,8 -> *N* = 6,8 млрд

*P* = 0,9 -> *N* = 8,2 млрд

1. *l* = 128:

*P* = 0,5 -> *N* = 264 ≈ 1,8 \* 1019

*P* = 0,7 -> *N* = 2,5 \* 1019

*P* = 0,8 -> *N* = 2,9 \* 1019

*P* = 0,9 -> *N* = 3,5 \* 1019

1. *l* = 256:

*P* = 0,5 -> *N* = 2^128 ≈ 3,4 \* 1038

*P* = 0,7 -> *N* = 4,7 \* 1038

*P* = 0,8 -> *N* = 5,4 \* 1038

*P* = 0,9 -> *N* = 6,5 \* 1038

1. *l* = 512:

*P* = 0,5 -> *N* = 2^256 ≈ 1,2 \* 1077

*P* = 0,7 -> *N* = 1,6 \* 1077

*P* = 0,8 -> *N* = 1,9 \* 1077

*P* = 0,9 -> *N* = 2,3 \* 1077

7. Дать общую характеристику алгоритмам хеширования семейств MD и SHA. Из каких основных стадий состоит алгоритм хеширования сообщения?

Алгоритмы семейства MD-x (2/4/5/6) являются творениями Р. Ривеста; MD – Message Digest. Алгоритм MD6, в отличие от предыдущих версий алгоритма этого семейства, не стандартизован. Алгоритмы семейства SHA (SHA – Secure Hash Algorithm) являются в настоящее время широко распространенными. По существу, во многих случаях завершился переход от SHA-1 к стандартам версии SHA-2. SHA-2 – собирательное название алгоритмов SHA224, SHA-256, SHA-384 и SHA-512. SHA-224 и SHA-384 являются, по сути, аналогами SHA-256 и SHA-512 соответственно

Базовые алгоритмы обоих рассматриваемых семейств (MD и SHA) условно можно разделить на 5 стадий:

• расширение входного сообщения;

• разбивка расширенного сообщения на блоки;

• инициализация начальных констант;

• обработка сообщения поблочно (основная процедура алгоритма хеширования);

• вывод результата

8. Рассчитать общую длину (L') хешируемого сообщения после предварительной стадии на основе алгоритма MD, если объем (L) исходного сообщения составлял: 0; 484; 512; 1000; 2000; 16 000 битов. Какова в каждом случае будет длина хеша?

Для алгоритма MD длина хеша составляет 128 бит (16 байт).

Для рассчета общей длины хешируемого сообщения после предварительной стадии на основе алгоритма MD нужно выполнить следующие шаги:

1. Добавить в исходное сообщение бит "1" и необходимое количество битов "0", чтобы длина сообщения стала кратной 512 битам.
2. Добавить 64-битное число, представляющее длину исходного сообщения в битах (также кратное 512 битам).

Таким образом, для каждого из заданных объемов (L) исходного сообщения, мы можем рассчитать общую длину (L') хешируемого сообщения после предварительной стадии на основе алгоритма MD:

1. L = 0 битов:

L' = 512 битов (бит "1" и 511 бит "0")

1. L = 484 бита:

L' = 960 битов (бит "1", 483 бит "0" и 64-битное число)

1. L = 512 битов:

L' = 1024 бита (бит "1" и 64-битное число)

1. L = 1000 битов:

L' = 1536 битов (бит "1", 511 бит "0" и 64-битное число)

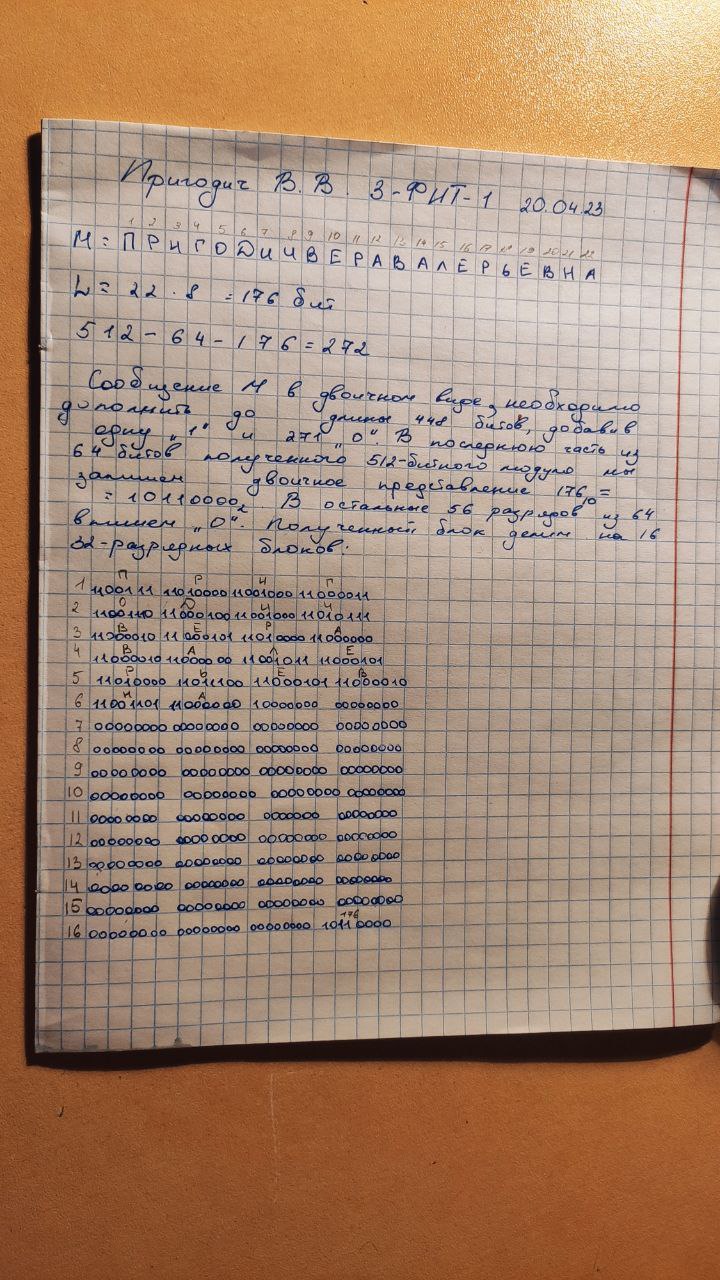
1. L = 2000 битов:

L' = 2560 битов (бит "1", 1535 бит "0" и 64-битное число)

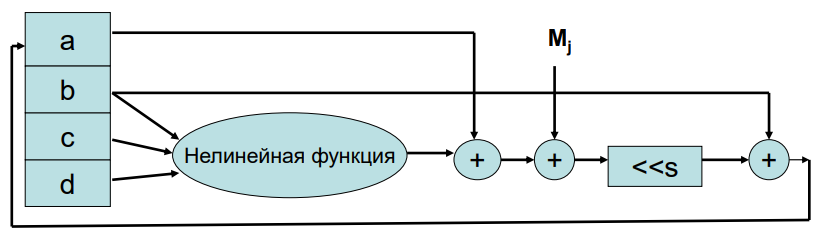
1. L = 16000 битов:

L' = 16128 битов (бит "1", 15359 бит "0" и 64-битное число)

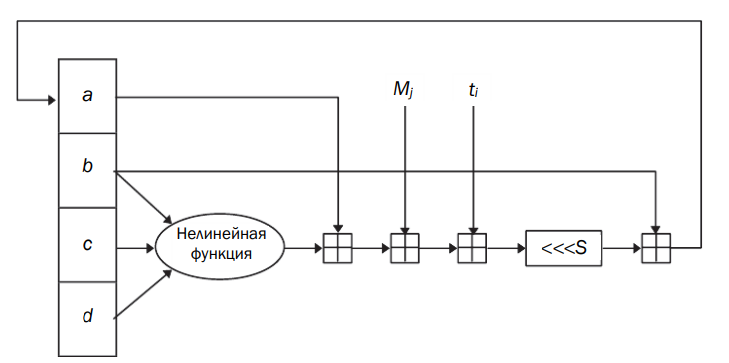
9. Входное сообщение (прообраз) состоит; а) из вашего имени; б) из ваших фамилии\_имени\_отчества (алфавит – на свой выбор). Используя представление сообщения в кодах ASCII, представить в табличной форме (как выше в примере 5) содержание каждого 32-битного подблока расширенного входного сообщения.



10. Представить и охарактеризовать структурную схему одного раунда алгоритмов хеширования на основе MD4; MD5; SHA-1.



Структурная схема одной операции алгоритма MD-4



Структурная схема одной операции алгоритма MD-5

Главный модуль состоит из четырех похожих этапов. На каждом этапе 16 раз используются различные операции. Каждая операция представляет собой нелинейную функцию над тремя из *a*, *b*, *c* и *d*. Затем она добавляет этот результат к четвертой переменной, подблоку текста *Мj* и константе *ti*. Далее результат циклически сдвигается вправо на переменное число s битов и добавляет результат к одной из переменных *a*, *b*, *c* и *d*. Наконец, результат заменяет одну из этих переменных.

Результатом хеширования *h* является конкатенация последних значений указанных переменных, т. е. 32 · 4 = 128 битов.

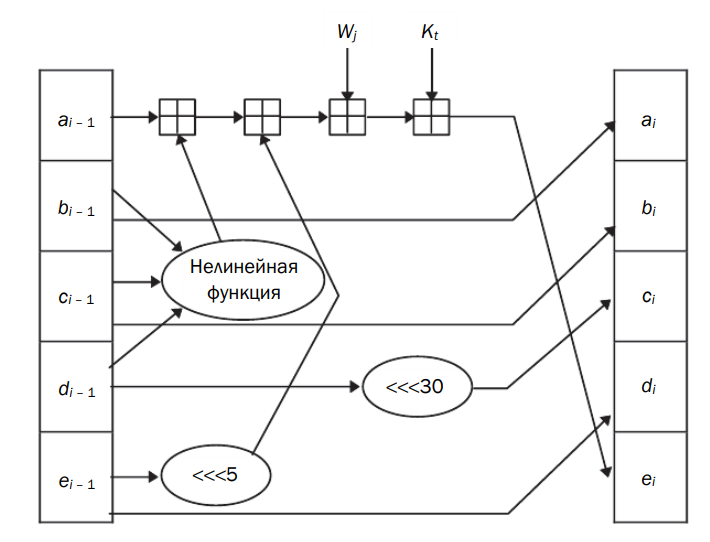


Схема выполнения одной операции в алгоритме SHA-1

Цикл состоит из четырех этапов по 20 операций в каждом (в MD5 – 4 этапа по 16 операций в каждом). Каждая операция представляет собой нелинейную функцию над тремя из пяти: *a*, *b*, *c*, *d*, *e.* Сдвиг и сложение – аналогично MD5.

11. На чем основан «лавинный эффект» в алгоритмах хеширования? В чем состоит цель его реализации?

«Лавинный эффект» – это свойство алгоритмов хеширования, которое гласит, что если входные данные изменились на даже небольшую величину, то результат хеширования должен измениться существенным образом. Это свойство основано на использовании нелинейных и сложных математических операций в алгоритмах хеширования.

Цель реализации лавинного эффекта в алгоритмах хеширования заключается в увеличении криптографической стойкости хешей, то есть сделать невозможным восстановление исходного сообщения по его хешу. Если входные данные изменятся, то изменится и хеш, что затрудняет возможность создания коллизий (разных сообщений с одинаковым хешем), что является одним из способов атак на алгоритмы хеширования.

12. В чем состоят основные структурные и функциональные особенности алгоритма хеширования SHA-3?

Основными структурными и функциональными особенностями алгоритма хеширования SHA-3 являются:

* Конструкция «спонж-функции» (sponge function) - SHA-3 является первым алгоритмом хеширования, основанным на этой конструкции. Она позволяет создавать хеш-функции произвольной длины путем преобразования блоков данных в хеш-значения, что делает алгоритм более гибким и универсальным.
* Использование принципа «маскировки» (masking) - SHA-3 использует специальный метод маскировки, который позволяет защитить алгоритм от возможных атак посредством анализа побочных эффектов при выполнении операций.
* Операции на основе битовых операций - SHA-3 использует множество операций, основанных на битовых операциях (XOR, NOT, AND, OR), что делает алгоритм более эффективным и быстрым.
* Высокая скорость работы – SHA-3 быстрее своих предшественников, таких как SHA-1 и SHA-2, благодаря новым методам и оптимизациям.
* Высокая стойкость к атакам – SHA-3 имеет более высокую криптографическую стойкость, чем предыдущие версии алгоритмов SHA. Он защищен от известных атак, таких как атаки дневных лучей, дифференциального криптоанализа и других.
* Поддержка параллелизма – SHA-3 может быть эффективно использован в многопоточных системах благодаря своей способности обрабатывать данные параллельно.

13. Охарактеризовать структурные, функциональные особенности и криптостойкость белорусского государственного стандарта хеширования (СТБ 34.101.77–2016).

Основные структурные и функциональные особенности СТБ 34.101.77–2016:

* Основан на конструкции Меркла-Дамгарда - это классическая конструкция хеш-функций, в которой данные разбиваются на блоки и каждый блок обрабатывается отдельно.
* Использует 12 раундов - каждый раунд состоит из нескольких шагов, включающих в себя преобразования и подстановки.
* Размер хеш-значения составляет 256 бит - это позволяет обеспечить достаточную стойкость к атакам.
* Поддерживает параллельную обработку данных - это позволяет повысить скорость работы алгоритма на многопроцессорных системах.
* Обладает высокой скоростью работы - благодаря оптимизациям и упрощенной структуре, СТБ 34.101.77–2016 работает быстрее некоторых других хеш-функций.
* Поддерживает входные данные произвольной длины - благодаря конструкции Меркла-Дамгарда, СТБ 34.101.77–2016 может обрабатывать данные произвольной длины.
* Имеет открытый и свободный код - это позволяет независимым экспертам проверять его безопасность и надежность.

Криптостойкость СТБ 34.101.77–2016 была проверена на различных атаках, включая атаки дневных лучей, дифференциальный криптоанализ, линейный криптоанализ и другие. Алгоритм показал высокую стойкость к атакам и считается криптографически надежным.