Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Кафедра информатики

Отчёт

по лабораторной работе №5

**Хэш-функции**

|  |  |
| --- | --- |
| Выполнил:  студент группы 653502  Куликов А.Д. | Проверил:  Артемьев В.С. |

Минск 2019

# Постановка задачи

Реализовать программные средства контроля целостности сообщений с помощью вычисления хэш-функции и алгоритма HMAC.

# Теоретическая справка

Хеш-функция, или функция свёртки – функция, осуществляющая преобразование массива входных данных произвольной длины в (выходную) битовую строку установленной длины, выполняемое определённым алгоритмом. Преобразование, производимое хеш-функцией, называется хешированием. Исходные данные называются входным массивом, «ключом» или «сообщением». Результат преобразования (выходные данные) называется «хешем», «хеш-кодом», «хеш-суммой», «сводкой сообщения».

Хеш-функции применяются в следующих случаях:

* при построении ассоциативных массивов;
* при поиске дубликатов в сериях наборов данных;
* при построении уникальных идентификаторов для наборов данных;
* при вычислении контрольных сумм от данных (сигнала) для последующего обнаружения в них ошибок (возникших случайно или внесённых намеренно), возникающих при хранении и/или передаче данных;
* при сохранении паролей в системах защиты в виде хеш-кода (для восстановления пароля по хеш-коду требуется функция, являющаяся обратной по отношению к использованной хеш-функции);
* при выработке электронной подписи (на практике часто подписывается не само сообщение, а его «хеш-образ»);

и др.

В общем случае (согласно принципу Дирихле) нет однозначного соответствия между хеш-кодом (выходными данными) и исходными (входными) данными. Возвращаемые хеш-функцией значения (выходные данные) менее разнообразны, чем значения входного массива (входные данные). Случай, при котором хеш-функция преобразует более чем один массив входных данных в одинаковые сводки, называется «коллизией». Вероятность возникновения коллизий используется для оценки качества хеш-функций.

Существует множество алгоритмов хеширования, отличающихся различными свойствами. Примеры свойств:

* разрядность;
* вычислительная сложность;
* криптостойкость.

Выбор той или иной хеш-функции определяется спецификой решаемой задачи. Простейшим примером хеш-функции может служить «обрамление» данных циклическим избыточным кодом (англ. CRC, cyclic redundancy code).

MD5 (англ. Message Digest 5) — 128-битный алгоритм хеширования, разработанный профессором Рональдом Л. Ривестом из Массачусетского технологического института (Massachusetts Institute of Technology, MIT) в 1991 году. Предназначен для создания «отпечатков» или дайджестов сообщения произвольной длины и последующей проверки их подлинности. Широко применялся для проверки целостности информации и хранения хешей паролей.

A close up of text on a white background

Description automatically generated

HMAC (сокращение от англ. hash-based message authentication code, код аутентификации (проверки подлинности) сообщений, использующий хеш-функции) – в информатике (криптографии), один из механизмов проверки целостности информации, позволяющий гарантировать то, что данные, передаваемые или хранящиеся в ненадёжной среде, не были изменены посторонними лицами (см. человек посередине). Механизм HMAC использует MAC, описан в RFC 2104, в стандартах организаций ANSI, IETF, ISO и NIST. MAC – стандарт, описывающий способ обмена данными и способ проверки целостности передаваемых данных с использованием секретного ключа. Два клиента, использующие HMAC, как правило, разделяют общий секретный ключ. HMAC – надстройка над MAC; механизм обмена данными с использованием секретного ключа (как в MAC) и хеш-функций. В зависимости от используемой хеш-функции выделяют HMAC-MD5, HMAC-SHA1, HMAC-RIPEMD128, HMAC-RIPEMD160 и т. п.

Было замечено, что скорость работы хеш-функций (например, MD5, SHA-1, RIPEMD128, RIPEMD-160), обычно, выше скорости работы симметричных блочных шифров (например, DES). Возникло желание использовать хеш-функции в MAC, а наличие готовых библиотек с реализациями различных хеш-функций только подтолкнуло эту идею.

Но использовать некоторые хеш-функции в MAC было невозможно. Например, хеш-функция MD5 не может применяться в MAC, так как принимает только один аргумент — данные (строку, последовательность байт).

В HMAC данные «смешивались» с ключом и хеш-функция применялась дважды.

Были предложены и другие механизмы, позволяющие одновременно использовать данные и секретный ключ в существующих алгоритмах хеширования, но HMAC получил наибольшую поддержку.

Преимущества HMAC:

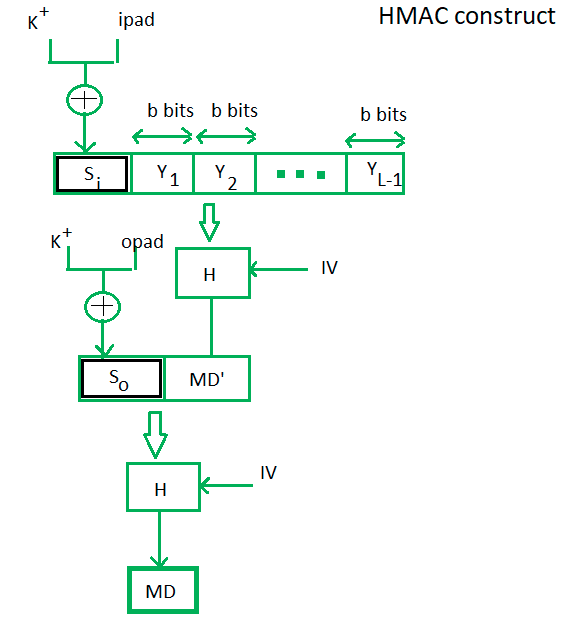
– возможность использования хеш-функций, уже имеющихся в программном продукте;

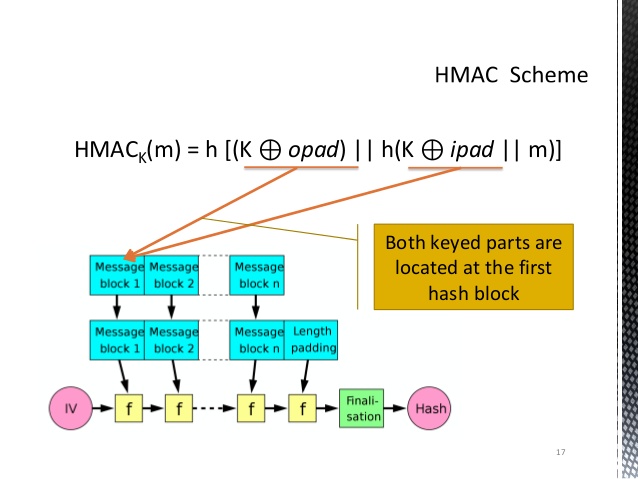
– отсутствие необходимости внесения изменений в реализации существующих хеш-функции (внесение изменений может привести к ухудшению производительности и криптостойкости);

– возможность замены хеш-функции в случае появления более безопасной или более быстрой хеш-функции.

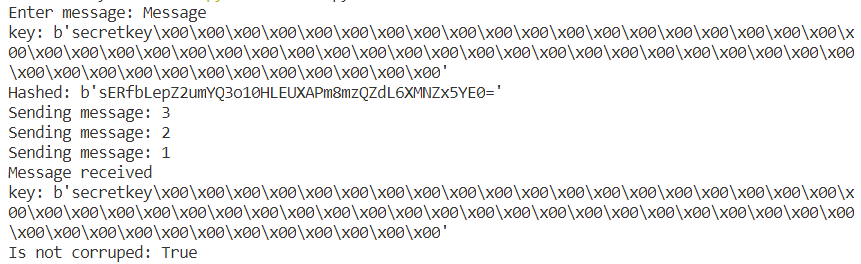
Механизм HMAC был описан в стандартах организаций ANSI, IETF, ISO и NIST. Реализация HMAC является для протокола IPsec. HMAC используется и в других протоколах интернета, например, TLS. Ожидается, что TLS вскоре заменит SSL и SET

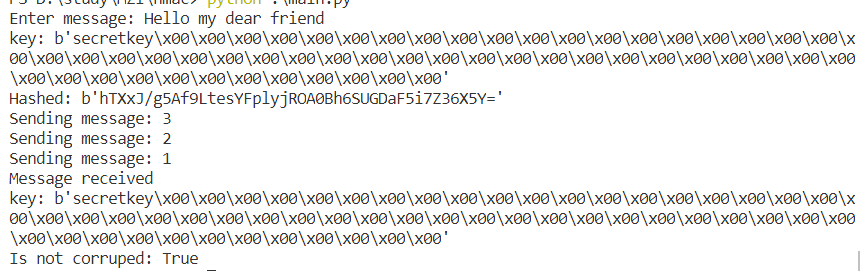
# Блок-схема алгоритма





# Результат выполнения программы





# Вывод

В результате выполнения данной работы были изучены алгоритмы хеширования и проверки целостности сообщений, а так же реализована программа для проверки целостности сообщения с помощью алгоритма hmac-rsa256.

# Приложение

**hmac.py**

from sha256 import sha256

trans\_5C = bytes((x ^ 0x5C) for x in range(256))

trans\_36 = bytes((x ^ 0x36) for x in range(256))

class HMAC:

    blocksize = 64

    def \_\_init\_\_(self, key, msg=None):

        self.outer = sha256('')

        self.inner = sha256('')

        self.digest\_size = self.inner.digest\_size

        self.block\_size = self.inner.block\_size

        if len(key) > self.block\_size:

            key = sha256(key).digest()

        key = key.ljust(self.block\_size, b'\0')

        print('key:', key)

        self.outer.update(key.translate(trans\_5C))

        self.inner.update(key.translate(trans\_36))

        self.update(msg)

    def update(self, msg):

        self.inner.update(msg)

    def \_current(self):

        h = self.outer.copy()

        h.update(self.inner.digest())

        return h

    def digest(self):

        h = self.\_current()

        return h.digest()

    def hexdigest(self):

        h = self.\_current()

        return h.hexdigest()

**sha256.py**

def sha\_final(sha\_info):

    lo\_bit\_count = sha\_info['count\_lo']

    hi\_bit\_count = sha\_info['count\_hi']

    count = (lo\_bit\_count >> 3) & 0x3f

    sha\_info['data'][count] = 0x80

    count += 1

    if count > SHA\_BLOCKSIZE - 8:

        # zero the bytes in data after the count

        sha\_info['data'] = sha\_info['data'][:count] + \

            ([0] \* (SHA\_BLOCKSIZE - count))

        sha\_transform(sha\_info)

        # zero bytes in data

        sha\_info['data'] = [0] \* SHA\_BLOCKSIZE

    else:

        sha\_info['data'] = sha\_info['data'][:count] + \

            ([0] \* (SHA\_BLOCKSIZE - count))

    sha\_info['data'][56] = (hi\_bit\_count >> 24) & 0xff

    sha\_info['data'][57] = (hi\_bit\_count >> 16) & 0xff

    sha\_info['data'][58] = (hi\_bit\_count >> 8) & 0xff

    sha\_info['data'][59] = (hi\_bit\_count >> 0) & 0xff

    sha\_info['data'][60] = (lo\_bit\_count >> 24) & 0xff

    sha\_info['data'][61] = (lo\_bit\_count >> 16) & 0xff

    sha\_info['data'][62] = (lo\_bit\_count >> 8) & 0xff

    sha\_info['data'][63] = (lo\_bit\_count >> 0) & 0xff

    sha\_transform(sha\_info)

    dig = []

    for i in sha\_info['digest']:

        dig.extend([((i >> 24) & 0xff), ((i >> 16) & 0xff),

                    ((i >> 8) & 0xff), (i & 0xff)])

    return bytes(dig)

class sha256(object):

    digest\_size = digestsize = SHA\_DIGESTSIZE

    block\_size = SHA\_BLOCKSIZE

    def \_\_init\_\_(self, s=None):

        self.name = 'sha256'

        self.\_sha = sha\_init()

        if s:

            sha\_update(self.\_sha, s)

    def update(self, s):

        sha\_update(self.\_sha, s)

    def digest(self):

        return sha\_final(self.\_sha.copy())[:self.\_sha['digestsize']]

    def hexdigest(self):

        return ''.join(['%.2x' % i for i in self.digest()])

    def copy(self):

        new = sha256()

        new.\_sha = self.\_sha.copy()

        return new