**Введение**

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.

Преимущества HMAC:

* возможность использования хэш-функций, уже имеющихся в программном продукте;
* отсутствие необходимости внесения изменений в реализации существующих хэш-функции (внесение изменений может привести к ухудшению производительности и криптостойкости);
* возможность замены хэш-функции в случае появления более безопасной или более быстрой хэш-функции.

В рамках лабораторной работы необходимо реализовать программные средства контроля целостности сообщений с помощью вычисления хэш функций 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.

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**

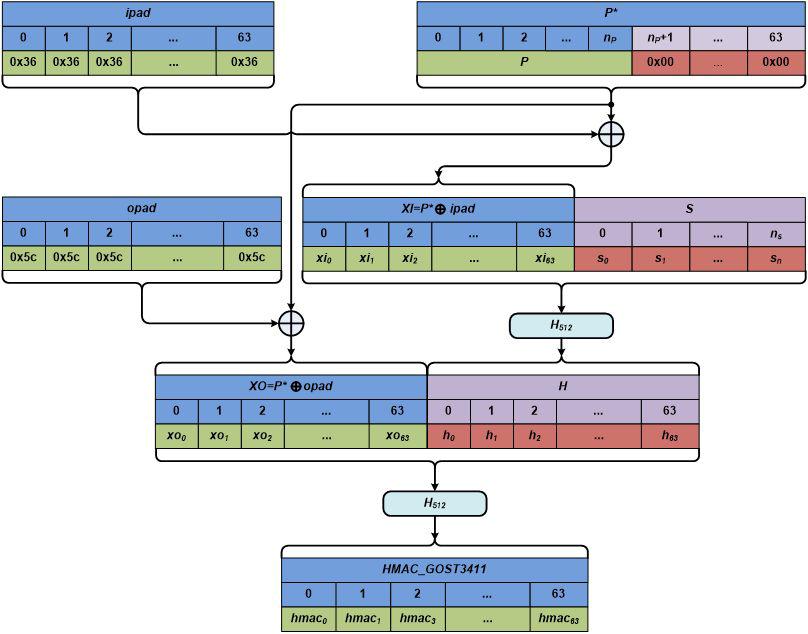


Рис.1. Схема алгоритма

**Пример работы программы**

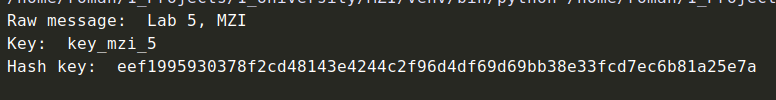


Рис.2. Пример работы

**Код программы**

**K = [**

0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,

0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,

0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,

0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,

0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,

0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,

0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,

0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2

]

*def* rotate\_right(*num*, *shift*, *size*=32):

*return* (*num* >> *shift*) | (*num* << *size* - *shift*)

*def* sha256(*sha256\_text*):

sha256\_text = bytearray(*sha256\_text*)

length = len(*sha256\_text*) \* 8

*sha256\_text*.append(0x80)

*while* (len(*sha256\_text*) \* 8 + 64) % 512 != 0:

*sha256\_text*.append(0x00)

*sha256\_text* += length.to\_bytes(8, 'big')

blocks = []

*for* i *in* range(0, len(*sha256\_text*), 64):

blocks.append(*sha256\_text*[i:i + 64])

h0 = 0x6a09e667

h1 = 0xbb67ae85

h2 = 0x3c6ef372

h3 = 0xa54ff53a

h5 = 0x9b05688c

h4 = 0x510e527f

h6 = 0x1f83d9ab

h7 = 0x5be0cd19

*for* message\_block *in* blocks:

message\_schedule = []

*for* t *in* range(0, 64):

*if* t <= 15:

message\_schedule.append(bytes(message\_block[t \* 4:(t \* 4) + 4]))

*else*:

term1 = (rotate\_right((int.from\_bytes(message\_schedule[t - 2], 'big')), 17)

^ rotate\_right((int.from\_bytes(message\_schedule[t - 2], 'big')), 19)

^ ((int.from\_bytes(message\_schedule[t - 2], 'big')) >> 10))

term2 = int.from\_bytes(message\_schedule[t - 7], 'big')

term3 = (rotate\_right((int.from\_bytes(message\_schedule[t - 15], 'big')), 7)

^ rotate\_right((int.from\_bytes(message\_schedule[t - 15], 'big')), 18)

^ ((int.from\_bytes(message\_schedule[t - 15], 'big')) >> 3))

term4 = int.from\_bytes(message\_schedule[t - 16], 'big')

schedule = ((term1 + term2 + term3 + term4) % 2 \*\* 32).to\_bytes(4, 'big')

message\_schedule.append(schedule)

a = h0

b = h1

c = h2

d = h3

e = h4

f = h5

g = h6

h = h7

*for* t *in* range(64):

t1 = ((h + (rotate\_right(e, 6) ^ rotate\_right(e, 11) ^ rotate\_right(e, 25))

+ ((e & f) ^ (~e & g)) + K[t] +

int.from\_bytes(message\_schedule[t], 'big')) % 2 \*\* 32)

t2 = ((rotate\_right(a, 2) ^ rotate\_right(a, 13) ^ rotate\_right(a, 22))

+ ((a & b) ^ (a & c) ^ (b & c))) % 2 \*\* 32

h = g

g = f

f = e

e = (d + t1) % 2 \*\* 32

d = c

c = b

b = a

a = (t1 + t2) % 2 \*\* 32

h0 = (h0 + a) % 2 \*\* 32

h1 = (h1 + b) % 2 \*\* 32

h2 = (h2 + c) % 2 \*\* 32

h3 = (h3 + d) % 2 \*\* 32

h4 = (h4 + e) % 2 \*\* 32

h5 = (h5 + f) % 2 \*\* 32

h6 = (h6 + g) % 2 \*\* 32

h7 = (h7 + h) % 2 \*\* 32

*return* ((h0).to\_bytes(4, 'big') + (h1).to\_bytes(4, 'big') +

(h2).to\_bytes(4, 'big') + (h3).to\_bytes(4, 'big') +

(h4).to\_bytes(4, 'big') + (h5).to\_bytes(4, 'big') +

(h6).to\_bytes(4, 'big') + (h7).to\_bytes(4, 'big'))

*def* hmac(*hmac\_key*, *hmac\_text*):

i\_key = bytearray()

o\_key = bytearray()

hmac\_key = *hmac\_key*.encode()

hmac\_text = *hmac\_text*.encode()

block\_size = 64

*if* len(*hmac\_key*) > block\_size:

hmac\_key = bytearray(sha256(*hmac\_key*))

*elif* len(*hmac\_key*) < block\_size:

i = len(*hmac\_key*)

*while* i < block\_size:

*hmac\_key* += b"\x00"

i += 1

*for* i *in* range(block\_size):

i\_key.append(0x36 ^ *hmac\_key*[i])

o\_key.append(0x5C ^ *hmac\_key*[i])

hmac\_text = bytes(o\_key) + sha256(bytes(i\_key) + *hmac\_text*)

*return* sha256(*hmac\_text*).hex()

*if* \_\_name\_\_ == "\_\_main\_\_":

key = "key\_mzi\_5"

text = "Lab 5, MZI"

print("Raw message: ", text)

print("Key: ", key)

h\_key = hmac(key, text)

print("Hash key: ", h\_key)

**Вывод**

Безопасность любой функции MAC на основе встроенных хеш-функций зависит от криптостойкости базовой хеш-функции. Привлекательность HMAC — в том, что его создатели смогли доказать точное соотношение между стойкостью встроенных хеш-функций и стойкостью HMAC.

* ходе написания лабораторной работы были изучен алгоритм хэширования HMAC, а также написана его программная реализация.