**Алгоритм хэширования Tiger**

**Tiger** — это криптографический хэш-алгоритм, разработанный для обеспечения высокой скорости и безопасности. Он используется для создания 192-битных хэш-значений и широко применяется в различных приложениях, включая системы контроля целостности и цифровые подписи.\

**История и преимущества:**

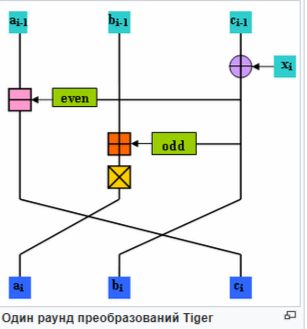
Алгоритм был разработан в 1995 году Россом Андерсоном и Эли Бихамом. То время характеризовалось тем, что для популярных хеш-функций [MD4](https://ru.wikipedia.org/wiki/MD4) и [Snefru](https://ru.wikipedia.org/wiki/Snefru) были уже найдены коллизии. Последнее, по мнению авторов, ставило под вопрос и надежность их производных, таких как [MD5](https://ru.wikipedia.org/wiki/MD5) и [Snefru-8](https://ru.wikipedia.org/wiki/Snefru). Основными целями при разработке Tiger были:

* [безопасность](https://ru.wikipedia.org/wiki/N-Hash#%D0%91%D0%B5%D0%B7%D0%BE%D0%BF%D0%B0%D1%81%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D1%85%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B9);
* быстрая работа на новых 64-битных процессорах при разумной скорости на 32-битных;
* замена по возможности серий MD и SHA.

**Алгоритм**

* Количество используемых S-box’ов — 4. S-box выполняет преобразование 8 бит в 64 бита. То есть в каждом из них 256 64-битных слов и общий размер памяти, требуемой для хранения S-box’ов 4\*256\*8 = 8192 = 8 Кбайт. Для этого хватает кэша большинства процессоров, хотя могут быть сложности при реализации на микроконтроллерах.
* Как и в семействе [MD4](https://ru.wikipedia.org/wiki/MD4), к сообщению добавляется бит «1», за которым следуют нули. Входные данные делятся на n блоков по 512 бит.
* Выбираем первый 512-битный блок. Этот блок делится на восемь 64-битных слов x0, x1, …, x7. [Порядок байтов](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%80%D1%8F%D0%B4%D0%BE%D0%BA_%D0%B1%D0%B0%D0%B9%D1%82%D0%BE%D0%B2) — [little-endian](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%80%D1%8F%D0%B4%D0%BE%D0%BA_%D0%B1%D0%B0%D0%B9%D1%82%D0%BE%D0%B2" \l "%D0%9F%D0%BE%D1%80%D1%8F%D0%B4%D0%BE%D0%BA_%D0%BE%D1%82_%D0%BC%D0%BB%D0%B0%D0%B4%D1%88%D0%B5%D0%B3%D0%BE_%D0%BA_%D1%81%D1%82%D0%B0%D1%80%D1%88%D0%B5%D0%BC%D1%83" \o "Порядок байтов).
* Берутся три 64-битных регистра с начальными значениями (значение хеша **h0**):

Схема:

**Подробный разбор алгоритма:**

**Основные шаги алгоритма Tiger**

1. **Инициализация состояния:**
   * Tiger использует три 64-битных регистра состояния, которые инициализируются следующими значениями:
     + a=0x0123456789ABCDEF
     + b=0xFEDCBA9876543210
     + c=0xF096A5B4C3B2E187
2. **Обработка сообщения:**

* Сообщение разбивается на блоки по 512 бит (64 байта).
* Если последний блок меньше 512 бит, он дополняется битом 1, а затем достаточным количеством нулей, чтобы общий размер был кратен 512 битам. Последние 64 бита представляют длину исходного сообщения в битах.

3. **Функция сжатия (Compression function):**

* Для каждого 512-битного блока алгоритм использует сжимающую функцию, которая изменяет состояние регистров a, b и c.

4.**Расширение блока (Key schedule):**

* 512-битный блок сообщения разбивается на восемь 64-битных слов X0,X1,…,X7X\_0, X\_1, \ldots, X\_7X0​,X1​,…,X7​. Эти слова затем модифицируются на каждом шаге сжимающей функции.

5. **Функция сжатия (Compression function):**

* Выполняется 24 раунда обработки, разбитых на три этапа по 8 раундов. Каждый этап использует определенные нелинейные преобразования и операции с использованием таблицы подстановки (S-box).

6. **Финальный хеш:**

* После обработки всех блоков сообщения регистры a, b и c содержат финальные значения, которые вместе составляют итоговый хеш (192 бита).

Вот детальное описание алгоритма хэширования Tiger, включая его основные шаги и подробности одного раунда:

### Подробный шаг 1-го раунда Tiger

Каждый раунд Tiger состоит из следующих шагов, использующих модифицированный Feistel:

1. **Операция сложения (Modulo Addition):**
   * В первом раунде Tiger регистр a складывается с одним из входных слов Xi​ и одним из элементов таблицы подстановки S-Box. Результат записывается обратно в регистр a.

a=(a+Xi+S[index])mod  2^

**XOR с константой:**

* + После сложения результат XOR-ится с одним из значений из регистров состояния:

b=b⊕a

1. **Побитовый сдвиг:**
   * Результат сдвигается вправо на фиксированное количество бит (обычно 5 или 7 бит в зависимости от раунда):

a=(a>>k)∣(a<<(64−k))(где k — количество сдвига)

**S-Box Lookup:**

* + Регистр a используется для поиска значения в S-Box (таблице подстановок), что добавляет нелинейность в процесс.

1. **Дополнительные операции:**
   * Повторяются операции сложения, XOR и сдвига с использованием других регистров и значений, таких как b и c.
2. **Swap регистров:**
   * В конце каждого раунда Tiger происходит обмен значениями между регистрами a, b и c для создания диффузии:

(a,b,c)=(b,c,a)

Листинг кода:

from sbox import \*  
A = 0x0123456789ABCDEF  
B = 0xFEDCBA9876543210  
C = 0xF0E1D2C3B4A59687  
  
def preprocess\_message(message):  
 # Преобразуем сообщение в байты  
 message\_bytes = message.encode('utf-8')  
 original\_length = len(message\_bytes) \* 8 # Длина сообщения в битах  
  
 # Добавляем бит '1'  
 message\_bytes += b'\x80'  
  
 # Дополняем нулями до 448 бит (56 байт)  
 while (len(message\_bytes) % 64) != 56:  
 message\_bytes += b'\x00'  
  
 # Добавляем длину сообщения (64 бита)  
 message\_bytes += original\_length.to\_bytes(8, byteorder='big')  
 print((message\_bytes))  
 # Разбиваем на блоки по 64 байта (512 бит)  
 blocks = [message\_bytes[i:i + 64] for i in range(0, len(message\_bytes), 64)]  
  
 print(f"Processed message blocks: {[block.hex() for block in blocks]}") #Вывод результата  
 return blocks  
  
def compress\_block(block, a, b, c, sbox1, sbox2, sbox3, sbox4):  
 # Разделение блока на 8 64-битных частей  
 x = [int.from\_bytes(block[i:i+8], 'little') for i in range(0, 64, 8)]  
  
 # Начальное сжатие, используя каждый подблок  
 for i in range(8):  
 # Определяем S-box для текущего блока  
 sbox = sbox1 if i < 2 else sbox2 if i < 4 else sbox3 if i < 6 else sbox4  
  
 # Применение подстановок с использованием таблицы S-box  
 a += x[i] + sbox[i % len(sbox)] # Пример использования S-box  
 a &= (1 << 64) - 1 # Ограничение до 64 бит  
  
 b ^= a  
 c = (c + b) % (1 << 64) # 64-битное ограничение  
  
 # Выполнение начального преобразования регистров  
 a, b, c = c, a, b  
  
 print(f"Compressed block results: A={a:X}, B={b:X}, C={c:X}") # Вывод результатов  
 return a, b, c  
  
def expand\_block(block):  
 # Разделяем 512-битный блок на 8 64-битных слов  
 x = [int.from\_bytes(block[i:i + 8], 'little') for i in range(0, 64, 8)]  
 return x  
  
def to\_binary\_string(value):  
 *"""Преобразует число в строку двоичного представления фиксированной длины (64 бита)."""* return format(value, '064b') # 64 бита  
  
def compress\_function(block, a, b, c, sbox1, sbox2, sbox3, sbox4):  
 print("1 round of compression----")  
 # Расширение блока  
 x = expand\_block(block)  
  
 # Первый раунд  
 round = 0  
 # Определяем текущий S-box в зависимости от номера раунда  
 sbox = sbox1 # Используем первый S-box  
  
 print("Initial values:")  
 print(f"a = {to\_binary\_string(a)} (decimal: {a})")  
 print(f"b = {to\_binary\_string(b)} (decimal: {b})")  
 print(f"c = {to\_binary\_string(c)} (decimal: {c})")  
 print(f"x[0] = {to\_binary\_string(x[0])} (decimal: {x[0]})")  
 print(f"sbox[0] = {to\_binary\_string(sbox[0])} (decimal: {sbox[0]})")  
  
 # Выполнение операций для первого раунда  
 print(f"\nRound {round + 1} calculations:")  
  
 # Расчет a  
 a\_prev = a # Сохраняем предыдущее значение a для вывода  
 a = (a + x[round % 8] + sbox[round % 256]) & ((1 << 64) - 1) # Ограничение до 64 бит  
 print(f"a (before) = {to\_binary\_string(a\_prev)} (decimal: {a\_prev})")  
 print(f"x[round % 8] = {to\_binary\_string(x[round % 8])} (decimal: {x[round % 8]})")  
 print(f"sbox[round % 256] = {to\_binary\_string(sbox[round % 256])} (decimal: {sbox[round % 256]})")  
 print(f"a (after) = {to\_binary\_string(a)} (decimal: {a})")  
  
 # Расчет b  
 b\_prev = b # Сохраняем предыдущее значение b для вывода  
 b = (b ^ a) & ((1 << 64) - 1) # Ограничение до 64 бит  
 print(f"b (before) = {to\_binary\_string(b\_prev)} (decimal: {b\_prev})")  
 print(f"b (after) = {to\_binary\_string(b)} (decimal: {b})")  
  
 # Расчет c  
 c\_prev = c # Сохраняем предыдущее значение c для вывода  
 c = (c + b) & ((1 << 64) - 1) # 64-битное ограничение  
 print(f"c (before) = {to\_binary\_string(c\_prev)} (decimal: {c\_prev})")  
 print(f"c (after) = {to\_binary\_string(c)} (decimal: {c})")  
  
 # Побитовые сдвиги  
 a\_prev = a # Сохраняем предыдущее значение a для вывода  
 a = (a >> 7) | (a << (64 - 7)) # Сдвиг вправо на 7 бит  
 print(f"Right shift a by 7 bits (before) = {to\_binary\_string(a\_prev)} (decimal: {a\_prev})")  
 a = a & ((1 << 64) - 1) # Ограничение до 64 бит  
  
 b = b & ((1 << 64) - 1) # Ограничение до 64 бит  
 c = c & ((1 << 64) - 1) # Ограничение до 64 бит  
  
 # Обмен регистров  
 a, b, c = b, c, a  
  
 print(f"\nValues after first round: A={to\_binary\_string(a)}, B={to\_binary\_string(b)}, C={to\_binary\_string(c)}")  
 return a, b, c  
  
  
def final\_compression(a, b, c):  
 # Финальное значение хэша  
 final\_hash = (a.to\_bytes(8, 'big') +  
 b.to\_bytes(8, 'big') +  
 c.to\_bytes(8, 'big'))  
  
 return final\_hash  
  
# Пример использования  
message = "Hello, world!"  
blocks = preprocess\_message(message)  
  
# Печать полученных блоков  
for i, block in enumerate(blocks):  
 print(f"Block {i}: {block.hex()}")  
  
print(compress\_block(block,A,B,C,sbox1,sbox2,sbox3,sbox4))  
  
  
  
# Использование первого блока  
block = blocks[0] # Используем первый блок из обработанных данных  
new\_a, new\_b, new\_c = compress\_function(block, A, B, C, sbox1, sbox2, sbox3, sbox4)  
  
# Финальная компрессия  
final\_hash = final\_compression(new\_a, new\_b, new\_c)  
print(f"Final hash: {final\_hash.hex().upper()}") # Вывод финального хэша

результат:

Message: Hello, Tiger!

b'Hello, Tiger!\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00h'

Processed message blocks: ['48656c6c6f2c20546967657221800000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000068']

Block 0:

48656c6c6f2c20546967657221800000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000068

Compressed block results: A=9F9D49DFFF6693D3, B=96CDCFFA37FC6435, C=EA48B4EACD0E170E

(11501430249785562067, 10866570149502936117, 16881942123753903886)

1 round of compression----

Initial values:

a = 0000000100100011010001010110011110001001101010111100110111101111 (decimal: 81985529216486895)

b = 1111111011011100101110101001100001110110010101000011001000010000 (decimal: 18364758544493064720)

c = 1111000011100001110100101100001110110100101001011001011010000111 (decimal: 17357386176853808775)

x[0] = 0101010000100000001011000110111101101100011011000110010101001000 (decimal: 6061893955512722760)

sbox[0] = 0000000000000000000000000000000011110111111010010000110001011110 (decimal: 4159245406)

Round 1 calculations:

a (before) = 0000000100100011010001010110011110001001101010111100110111101111 (decimal: 81985529216486895)

x[round % 8] = 0101010000100000001011000110111101101100011011000110010101001000 (decimal: 6061893955512722760)

sbox[round % 256] = 0000000000000000000000000000000011110111111010010000110001011110 (decimal: 4159245406)

a (after) = 0101010101000011011100011101011111101110000000010011111110010101 (decimal: 6143879488888455061)

b (before) = 1111111011011100101110101001100001110110010101000011001000010000 (decimal: 18364758544493064720)

b (after) = 1010101110011111110010110100111110011000010101010000110110000101 (decimal: 12366826644501237125)

c (before) = 1111000011100001110100101100001110110100101001011001011010000111 (decimal: 17357386176853808775)

c (after) = 1001110010000001100111100001001101001100111110101010010000001100 (decimal: 11277468747645494284)

Right shift a by 7 bits (before) = 0101010101000011011100011101011111101110000000010011111110010101 (decimal: 6143879488888455061)

Values after first round: A=1010101110011111110010110100111110011000010101010000110110000101, B=1001110010000001100111100001001101001100111110101010010000001100, C=0010101010101010100001101110001110101111110111000000001001111111

Final hash (after 24 rounds): AB9FCB4F98550D859C819E134CFAA40C2AAA86E3AFDC027F

### Заключение

Алгоритм Tiger — это мощный и быстрый хэш-алгоритм, который эффективно обрабатывает данные и создает безопасные хэш-значения. Он сочетает в себе простоту и скорость выполнения, что делает его подходящим для широкого спектра криптографических приложений. Подробное понимание каждого этапа алгоритма, особенно его раундов, является ключом к его правильной реализации и анализу безопасности.

Ассиметричные алгоритмы шифрования

**Асимметричное шифрование** — это метод шифрования данных, предполагающий использование двух ключей — открытого и закрытого.

**Открытый (публичный) ключ** применяется для шифрования информации и может передаваться по незащищённым каналам.

**Закрытый (приватный) ключ** применяется для расшифровки данных, зашифрованных открытым ключом.

**Некоторые распространённые алгоритмы асимметричного шифрования**:

* **RSA** (аббревиатура от Rivest, Shamir и Adelman, фамилий создателей алгоритма) — алгоритм, в основе которого лежит вычислительная сложность факторизации (разложения на множители) больших чисел. Применяется в защищенных
* протоколах SSL и TLS, стандартах шифрования, например в PGP и S/MIME.
* **DSA** (Digital Signature Algorithm, «алгоритм цифровой подписи») — алгоритм, основанный на сложности вычисления дискретных логарифмов. Используется для генерации цифровых подписей.
* **Схема Эль-Гамаля** — алгоритм, основанный на сложности вычисления дискретных логарифмов. Лежит в основе DSA и устаревшего российского стандарта ГОСТ 34.10–94. Применяется как для шифрования, так и для создания цифровых подписей.
* **ECDSA** (Elliptic Curve Digital Signature Algorithm) — алгоритм, основанный на сложности вычисления дискретного логарифма в группе точек эллиптической кривой. Применяется для генерации цифровых подписей, в частности для подтверждения транзакций в криптовалюте Ripple.