БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Кафедра информатики

Факультет КСиС

Специальность ИиТП

Контрольная работа № 1

по дисциплине «Методы защиты информации»

Выполнил: Реут М.В.

группа 893551

Минск 2022

# **1. Постановка задачи**

1. Изучить алгоритм формирования Хэш-функции ГОСТ 3411.

2. Создать и протестировать алгоритм формирования Хэш-функции

ГОСТ 3411 на языке высокого уровня.

# **2. Схема алгоритма**

Алгоритм ГОСТ 3411 является отечественным стандартом для хэш функций. Длина хэш-кода, создаваемого алгоритмом ГОСТ 3411, равна 256 битам. Алгоритм разбивает сообщение на блоки, длина которых также равна 256 битам. Кроме того, параметром алгоритма является стартовый вектор хэширования Н - произвольное фиксированное значение длиной также 256 бит. Алгоритм обработки одного блока сообщения Сообщение обрабатывается блоками по 256 бит справа налево.

Каждый блок сообщения обрабатывается по следующему алгоритму.

1. Генерация четырех ключей длиной 256 бит каждый.

2. Шифрование 64-битных значений промежуточного хэш-кода H на ключах Ki(i = 1, 2, 3, 4) с использованием алгоритма ГОСТ 28147 в режиме простой замены.

3. Перемешивание результата шифрования.

Для генерации ключей используются следующие данные:

* промежуточное значение хэш-кода Н длиной 256 бит;
* текущий обрабатываемый блок сообщения М длиной 256 бит;
* параметры - три значения С2, С3 и С4 длиной 256 бит следующего вида: С2 и С4 состоят из одних нулей, а С3 равно 1 8 0 8 1 16 0 24 1 16 0 8 (08 1 8 ) 2 1 8 0 8 (08 1 8 ) 4 (18 0 8 ) 4 где степень обозначает количество повторений 0 или 1.

Используются две формулы, определяющие перестановку и сдвиг. Перестановка Р битов определяется следующим образом: каждое 256- битное значение рассматривается как последовательность тридцати двух 8- битных значений.

Перестановка Р элементов 256-битной последовательности выполняется по формуле y = φ(x), где x - порядковый номер 8-битного значения в исходной последовательности; y - порядковый номер 8-битного значения в результирующей последовательности.

φ(i + 1 + 4(k - 1)) = 8i + k ; i = 0 ÷ 3, k = 1 ÷ 8

Сдвиг А определяется по формуле

A (x) = (x1 x2) || x4 || x3 || x2

Где

xi - соответствующие 64 бита 256-битного значения х,

|| обозначает конкатенацию.

Присваиваются следующие начальные значения: i = 1, U = H, V = M. W = U V, K1 = Р (W)

Ключи K2, K3, K4 вычисляются последовательно по следующему алгоритму: 79 U = A(U) Сi, V = A(A(V)), W = U V, Ki = Р(W)

Далее выполняется шифрование 64-битных элементов текущего значения хэш-кода Н с ключами K1, K2, K3 и K4.

При этом хэш-код Н рассматривается как последовательность 64-битных значений: H = h4 || h3 || h2 || h1

Выполняется шифрование алгоритмом ГОСТ 28147:

si = EKi [hi] i = 1, 2, 3, 4 S = s1 || s2 || s3 || s4

Наконец на заключительном этапе обработки очередного блока выполняется перемешивание полученной последовательности. 256-битное значение рассматривается как последовательность шестнадцати 16-битных значений.

Сдвиг обозначается Ψ и определяется следующим образом: η16 || η15 || ... || η1 - исходное значение η1 η2 η3 η4 η13 η16 || η16 || ... || η2 - результирующее значение

Результирующее значение хэш-кода определяется следующим образом: Χ(M, H) = ψ 61 (H ψ (M ψ 12(S))) где H - предыдущее значение хэш-кода, М - текущий обрабатываемый блок, Ψi - i-ая степень преобразования Ψ. Входными параметрами алгоритма являются:

* исходное сообщение М произвольной длины;
* стартовый вектор хэширования Н, длина которого равна 256 битам;
* контрольная сумма Σ, начальное значение которой равно нулю и длина равна 256 битам;
* переменная L, начальное значение которой равно длине сообщения.

Сообщение М делится на блоки длиной 256 бит и обрабатывается справа налево.

Очередной блок i обрабатывается следующим образом:

1. H = Χ(Mi, H)
2. Σ = Σ ' Mi
3. L рассматривается как неотрицательное целое число, к этому числу прибавляется 256 и вычисляется остаток от деления получившегося числа на 2256. Результат присваивается L.

Где ' обозначает следующую операцию: Σ и Mi рассматриваются как неотрицательные целые числа длиной 256 бит. Выполняется обычное сложение этих чисел и находится остаток от деления результата сложения на 2 256. Этот остаток и является результатом операции.

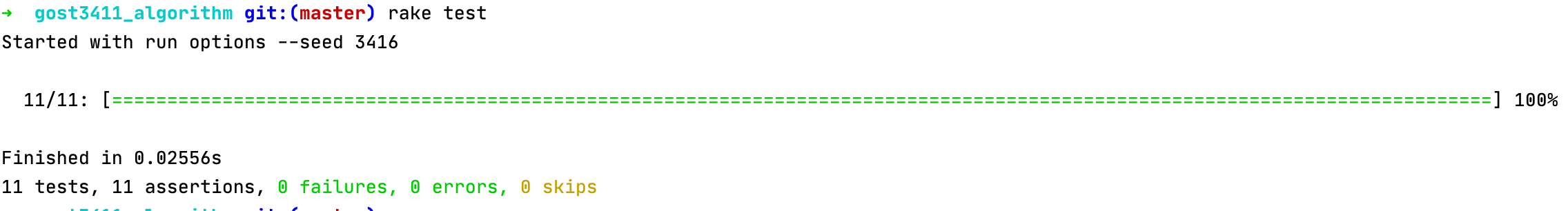
Самый левый, т.е. самый последний блок М' обрабатывается так:

1. Блок добавляется слева нулями так, чтобы его длина стала равна 256 битам.
2. Вычисляется Σ = Σ ' Mi.
3. L рассматривается как неотрицательное целое число, к этому числу прибавляется длина исходного сообщения М и находится остаток от деления результата сложения на 2256 .
4. Вычисляется Н = Χ(М', Н).
5. Вычисляется Н = Χ(L, Н).
6. Вычисляется Н = Χ(Σ, Н). Значением функции хэширования является Н.

# **3. Скриншоты ввода данных и результатов выполнения программы**

Чтобы упростить демонстрацию работы программы и его проверку в папке **test** добавлены тесты.

Запуск тестов осуществляется при помощи команды ***rake test***.



Мы можем вклинится в этот процесс при помощи **“binding.pry”**



# **4. Исходные файлы:**

Реализация алгоритма:

## Файл **gost3411.rb**

module *Gost3411Algorithm*

class *Gost3411*

def *initialize*(*digest\_size* = 64)

*digest\_size* = 64 if *digest\_size* != 32

@digest\_size = *digest\_size*

*#* ***@resH*** *- [8] uint64*

if @digest\_size == 64 then

@resH = [0, 0, 0, 0, 0, 0, 0, 0]

else

*e* = uint8ToUint64("\x01" \* 8)

@resH = [*e*, *e*, *e*, *e*, *e*, *e*, *e*, *e*]

end

*# [8] uint64*

@sigma = [0, 0, 0, 0, 0, 0, 0, 0]

*# [8] uint64*

@nN = [0, 0, 0, 0, 0, 0, 0, 0]

*# byte string*

@block = ''

@block\_len = 0

end

def *update*(*data*)

*bytes* = *data*.dup.force\_encoding('BINARY')

*bytes\_len* = *bytes*.length

*# Nothing to do for empty string*

if *bytes\_len* > 0 then

*len* = @block\_len + *bytes\_len*

if *len* < 64 then

@block += *bytes*

@block\_len = *len*

else

*index* = 0

while *len* >= 64

@block += *bytes*[*index*...(*index* + 64-@block\_len)]

transform(64)

*index* += 64 - @block\_len

*len* -= 64

@block = ''

@block\_len = 0

end

if *len* > 0 then

@block = *bytes*[*index*...*index* + *len*]

@block\_len = *len*

end

end

end

return self

end

def *final*

@block += 1.chr

while @block.length < 64

@block += 0.chr

end

*# Don't increment* ***@block\_len*** *for padding!*

transform(@block\_len)

*zZ* = [0,0,0,0,0,0,0,0]

funcG(@resH, @nN, *zZ*)

funcG(@resH, @sigma, *zZ*)

*dgst* = ''

@resH.each {|*n*| *dgst* += uint64ToUint8LE(*n*)}

if @digest\_size == 32

*dgst* = *dgst*[32..-1]

end

return *dgst*

end

def *digest*(*data*)

update(*data*)

*dgst* = final

end

*private*

*Two64* = 1 << 64

*Two512* = 1 << 512

*# Unload 64-bit number to 8-byte string*

*# (little-endian, adding leading zeroes)*

def *uint64ToUint8LE*(*n*)

[*n* % 0x100000000].pack("V") + [*n* / 0x100000000].pack("V")

end

*# Unpack 8-byte (little-endian) string to 64-bit number (native-endian)*

def *uint8ToUint64*(*bytes*)

*bytes*.unpack('Q\*')[0]

end

*# out, temp - arrays[8]*

def *strido*(*out*, *temp*, *i*)

*t* = *Gost3411Table*[0][(*temp*[0] >> (*i* \* 8)) & 0xff]

(1...8).each{|*j*| *t* ^= *Gost3411Table*[*j*][(*temp*[*j*] >> (*i* \* 8)) & 0xff]}

*out*[*i*] = *t*

end

*# out, a, b, temp - arrays[8]*

def *funcLPSX*(*out*, *a*, *b*)

*temp* = [0,0,0,0,0,0,0,0]

(0...*a*.length).each{|*i*| *temp*[*i*] = *a*[*i*] ^ *b*[*i*]}

(0...*temp*.length).each{|*i*| strido(*out*, *temp*, *i*)}

end

*# h, m, n - arrays[8] of 64-bit numbers*

def *funcG*(*h*, *m*, *n*)

*kk* = [0,0,0,0,0,0,0,0]

*tt* = [0,0,0,0,0,0,0,0]

funcLPSX(*kk*, *h*, *n*)

funcLPSX(*tt*, *kk*, *m*)

funcLPSX(*kk*, *kk*, *C16*[0])

(1...12).each do |*i*|

funcLPSX(*tt*, *kk*, *tt*)

funcLPSX(*kk*, *kk*, *C16*[*i*])

end

(0...8).each do |*i*|

*h*[*i*] ^= *tt*[*i*] ^ *kk*[*i*] ^ *m*[*i*]

end

end

def self.*add512*(*x*, *y*)

*x512* = *x*[7]

(1...8).each do |*i*|

*x512* = *x512*\**Two64* + *x*[7-*i*]

end

*y512* = *y*[7]

(1...8).each do |*i*|

*y512* = *y512*\**Two64* + *y*[7-*i*]

end

*r512* = (*x512* + *y512*)%*Two512*

*r* = []

(0...8).each do |*i*|

*r* << *r512*%*Two64*

*r512* = (*r512*-*r*[-1])/*Two64*

end

*r*

end

def *transform*(*nbytes*)

*count512* = [*nbytes*\*8,0,0,0,0,0,0,0]

*mM* = []

(0...8).each do |*i*|

*mM* << uint8ToUint64(@block[(*i* \* 8)...(*i* \* 8 + 8)])

end

funcG(@resH, *mM*, @nN)

@nN = self.class.add512(@nN, *count512*)

@sigma = self.class.add512(@sigma, *mM*)

end

def self.*printBytes*(*bytes*, *line\_size* = 16)

*bytes*.unpack('H\*')[0].scan(**/**.{1,#{*line\_size*}}**/**).each{|*s*| puts(*s*)}

end

def *printCtx*

puts '='\*16 + ' Digest context ' + '='\*16

puts "digest\_size = #{@digest\_size}"

puts "block\_len = #{@block\_len}"

puts 'block:'

self.class.printBytes(@block)

puts 'resH:'

(0...8).each do |*i*|

printf("0x%X\n", @resH[*i*])

end

puts 'nN:'

(0...8).each do |*i*|

printf("0x%X\n", @nN[*i*])

end

puts 'sigma:'

(0...8).each do |*i*|

printf("0x%X\n", @sigma[*i*])

end

puts "="\*40

end

end

end

Тестирование алгоритма:

## Файл **gost3411\_algorithm\_test.rb**

*require* "test\_helper"

*include Gost3411Algorithm*

*# GOST R 3411 test data:*

*M1* = [

0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,

0x38, 0x39, 0x30, 0x31, 0x32, 0x33, 0x34, 0x35,

0x36, 0x37, 0x38, 0x39, 0x30, 0x31, 0x32, 0x33,

0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x30, 0x31,

0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39,

0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,

0x38, 0x39, 0x30, 0x31, 0x32, 0x33, 0x34, 0x35,

0x36, 0x37, 0x38, 0x39, 0x30, 0x31, 0x32

].pack('C\*')

*Et1\_32* = [

0x9d, 0x15, 0x1e, 0xef, 0xd8, 0x59, 0x0b, 0x89,

0xda, 0xa6, 0xba, 0x6c, 0xb7, 0x4a, 0xf9, 0x27,

0x5d, 0xd0, 0x51, 0x02, 0x6b, 0xb1, 0x49, 0xa4,

0x52, 0xfd, 0x84, 0xe5, 0xe5, 0x7b, 0x55, 0x00

].pack('C\*')

*Et1\_64* = [

0x1b, 0x54, 0xd0, 0x1a, 0x4a, 0xf5, 0xb9, 0xd5,

0xcc, 0x3d, 0x86, 0xd6, 0x8d, 0x28, 0x54, 0x62,

0xb1, 0x9a, 0xbc, 0x24, 0x75, 0x22, 0x2f, 0x35,

0xc0, 0x85, 0x12, 0x2b, 0xe4, 0xba, 0x1f, 0xfa,

0x00, 0xad, 0x30, 0xf8, 0x76, 0x7b, 0x3a, 0x82,

0x38, 0x4c, 0x65, 0x74, 0xf0, 0x24, 0xc3, 0x11,

0xe2, 0xa4, 0x81, 0x33, 0x2b, 0x08, 0xef, 0x7f,

0x41, 0x79, 0x78, 0x91, 0xc1, 0x64, 0x6f, 0x48

].pack('C\*')

*M2* = [

0xd1, 0xe5, 0x20, 0xe2, 0xe5, 0xf2, 0xf0, 0xe8,

0x2c, 0x20, 0xd1, 0xf2, 0xf0, 0xe8, 0xe1, 0xee,

0xe6, 0xe8, 0x20, 0xe2, 0xed, 0xf3, 0xf6, 0xe8,

0x2c, 0x20, 0xe2, 0xe5, 0xfe, 0xf2, 0xfa, 0x20,

0xf1, 0x20, 0xec, 0xee, 0xf0, 0xff, 0x20, 0xf1,

0xf2, 0xf0, 0xe5, 0xeb, 0xe0, 0xec, 0xe8, 0x20,

0xed, 0xe0, 0x20, 0xf5, 0xf0, 0xe0, 0xe1, 0xf0,

0xfb, 0xff, 0x20, 0xef, 0xeb, 0xfa, 0xea, 0xfb,

0x20, 0xc8, 0xe3, 0xee, 0xf0, 0xe5, 0xe2, 0xfb

].pack('C\*')

*Et2\_32* = [

0x9d, 0xd2, 0xfe, 0x4e, 0x90, 0x40, 0x9e, 0x5d,

0xa8, 0x7f, 0x53, 0x97, 0x6d, 0x74, 0x05, 0xb0,

0xc0, 0xca, 0xc6, 0x28, 0xfc, 0x66, 0x9a, 0x74,

0x1d, 0x50, 0x06, 0x3c, 0x55, 0x7e, 0x8f, 0x50

].pack('C\*')

*Et2\_64* = [

0x1e, 0x88, 0xe6, 0x22, 0x26, 0xbf, 0xca, 0x6f,

0x99, 0x94, 0xf1, 0xf2, 0xd5, 0x15, 0x69, 0xe0,

0xda, 0xf8, 0x47, 0x5a, 0x3b, 0x0f, 0xe6, 0x1a,

0x53, 0x00, 0xee, 0xe4, 0x6d, 0x96, 0x13, 0x76,

0x03, 0x5f, 0xe8, 0x35, 0x49, 0xad, 0xa2, 0xb8,

0x62, 0x0f, 0xcd, 0x7c, 0x49, 0x6c, 0xe5, 0xb3,

0x3f, 0x0c, 0xb9, 0xdd, 0xdc, 0x2b, 0x64, 0x60,

0x14, 0x3b, 0x03, 0xda, 0xba, 0xc9, 0xfb, 0x28

].pack('C\*')

*# Digest of NULL message, 256 bits, little-endian.*

*Et0\_32* = [

0x3f, 0x53, 0x9a, 0x21, 0x3e, 0x97, 0xc8, 0x02,

0xcc, 0x22, 0x9d, 0x47, 0x4c, 0x6a, 0xa3, 0x2a,

0x82, 0x5a, 0x36, 0x0b, 0x2a, 0x93, 0x3a, 0x94,

0x9f, 0xd9, 0x25, 0x20, 0x8d, 0x9c, 0xe1, 0xbb

].pack('C\*')

*# Digest of NULL message, 512 bits, little-endian.*

*Et0\_64* = [

0x8e, 0x94, 0x5d, 0xa2, 0x09, 0xaa, 0x86, 0x9f,

0x04, 0x55, 0x92, 0x85, 0x29, 0xbc, 0xae, 0x46,

0x79, 0xe9, 0x87, 0x3a, 0xb7, 0x07, 0xb5, 0x53,

0x15, 0xf5, 0x6c, 0xeb, 0x98, 0xbe, 0xf0, 0xa7,

0x36, 0x2f, 0x71, 0x55, 0x28, 0x35, 0x6e, 0xe8,

0x3c, 0xda, 0x5f, 0x2a, 0xac, 0x4c, 0x6a, 0xd2,

0xba, 0x3a, 0x71, 0x5c, 0x1b, 0xcd, 0x81, 0xcb,

0x8e, 0x9f, 0x90, 0xbf, 0x4c, 0x1c, 0x1a, 0x8a

].pack('C\*')

*M64* = [

0xd1, 0xe5, 0x20, 0xe2, 0xe5, 0xf2, 0xf0, 0xe8,

0x2c, 0x20, 0xd1, 0xf2, 0xf0, 0xe8, 0xe1, 0xee,

0xe6, 0xe8, 0x20, 0xe2, 0xed, 0xf3, 0xf6, 0xe8,

0x2c, 0x20, 0xe2, 0xe5, 0xfe, 0xf2, 0xfa, 0x20,

0xf1, 0x20, 0xec, 0xee, 0xf0, 0xff, 0x20, 0xf1,

0xf2, 0xf0, 0xe5, 0xeb, 0xe0, 0xec, 0xe8, 0x20,

0xed, 0xe0, 0x20, 0xf5, 0xf0, 0xe0, 0xe1, 0xf0,

0xfb, 0xff, 0x20, 0xef, 0xeb, 0xfa, 0xea, 0xfb

].pack('C\*')

*Et64\_32* = [

0xd7, 0xc7, 0x94, 0x8e, 0x58, 0xa0, 0x58, 0xd4,

0xae, 0x53, 0xb1, 0xe0, 0xf5, 0x87, 0x7a, 0x6e,

0xf2, 0x58, 0x38, 0xa3, 0x8c, 0xb0, 0x08, 0x63,

0x8c, 0xcf, 0xe1, 0x1a, 0x47, 0x34, 0x43, 0x38

].pack('C\*')

*Et64\_64* = [

0x7f, 0xae, 0x62, 0x22, 0x1e, 0xac, 0xc1, 0x47,

0x0a, 0x34, 0x4f, 0x5f, 0xf3, 0xbd, 0x1d, 0x30,

0x85, 0x19, 0x6d, 0xb2, 0x8f, 0x24, 0xee, 0x2c,

0x0d, 0x60, 0x43, 0x74, 0x47, 0xe7, 0xad, 0x82,

0xa0, 0x44, 0x0c, 0xf9, 0x81, 0xcf, 0x80, 0x8c,

0x19, 0xcb, 0xff, 0x43, 0xe5, 0x57, 0xbd, 0x8a,

0xd4, 0x09, 0x36, 0x50, 0x5f, 0x53, 0xef, 0xc6,

0x04, 0xb6, 0x73, 0xe2, 0x6e, 0x77, 0x6d, 0x78,

].pack('C\*')

class *Gost3411AlgorithmTest* < *Minitest*::*Test*

def *test\_that\_it\_has\_a\_version\_number*

refute\_nil ::*Gost3411Algorithm*::*VERSION*

end

def *test\_M1\_32*

*digest* = *Gost3411*.new(32).update(*M1*).final

assert *digest* == *Et1\_32*

end

def *test\_M1\_64*

*digest* = *Gost3411*.new(64).update(*M1*).final

assert *digest* == *Et1\_64*

end

def *test\_M2\_32*

*digest* = *Gost3411*.new(32).update(*M2*).final

assert *digest* == *Et2\_32*

end

def *test\_M2\_64*

*digest* = *Gost3411*.new(64).update(*M2*).final

assert *digest* == *Et2\_64*

end

def *test\_M64\_32*

*digest* = *Gost3411*.new(32).update(*M64*).final

assert *digest* == *Et64\_32*

end

def *test\_M64\_64*

*digest* = *Gost3411*.new(64).update(*M64*).final

assert *digest* == *Et64\_64*

end

def *test\_empty\_32*

*digest* = *Gost3411*.new(32).final

assert *digest* == *Et0\_32*

end

def *test\_empty\_64*

*digest* = *Gost3411*.new(64).final

assert *digest* == *Et0\_64*

end

def *test\_size\_not\_set*

*digest* = *Gost3411*.new.final

assert *digest* == *Et0\_64*

end

def *test\_not\_32\_nor\_64*

*digest* = *Gost3411*.new(20).final

assert *digest* == *Et0\_64*

end

end