Лабораторная работа № 1

Криптографические хэш-функции

Цель работы

Изучить существующие алгоритмы вычисления дайджестов сообщений и написать программу, реализующую заданный в варианте алгоритм хэширования.

Методические указания к работе

Хэширование (иногда *хеширование*, англ. *hashing*) – преобразование входного массива данных произвольной длины в выходную битовую строку фиксированной длины. Такие преобразования также называются ***хэш-функциями*** или ***функциями свёртки***, а их результаты называют ***хэшем***, ***хэш-кодом*** или ***дайджестом сообщения*** (англ. *message digest*) [1].

Хэширование применяется для сравнения данных: если у двух массивов хэш-коды разные, массивы гарантированно различаются; если одинаковые – массивы, скорее всего, одинаковы. В общем случае, однозначного соответствия между исходными данными и хэш-кодом нет в силу того, что количество значений хэш-функций меньше, чем вариантов входного массива. Также существует множество массивов, дающих одинаковые хэш-коды – так называемые ***коллизии***. Вероятность возникновения коллизий играет немаловажную роль в оценке качества хэш-функций.

Существует множество алгоритмов хэширования с различными характеристиками (разрядность, вычислительная сложность, криптостойкость и т.п.). Выбор той или иной хэш-функции определяется спецификой решаемой задачи. Простейшими примерами хэш-функций могут служить контрольные суммы.

Контрольные суммы – это несложные, крайне быстрые и легко реализуемые аппаратные алгоритмы, используемые для защиты от непреднамеренных искажений, в том числе ошибок аппаратуры. По скорости вычисления они в десятки и сотни раз быстрее, чем криптографические хэш-функции, и значительно проще в аппаратной реализации.

Платой за столь высокую скорость является отсутствие криптостойкости – очень просто подобрать сообщение под заранее известную сумму. Разрядность контрольных сумм (обычно 32 бита) ниже, чем криптографических хэшей (типичные значения – 128, 160 и 256 бит), что означает возможность возникновения непреднамеренных коллизий.

Простейшим случаем такого алгоритма является деление сообщения на 32- или 16-битные слова и их суммирование, что применяется, например, в TCP/IP.

Как правило, к такому алгоритму предъявляются требования отслеживания типичных аппаратных ошибок, таких, как несколько подряд идущих ошибочных бит до заданной длины. К таким алгоритмам относится, например, *CRC32*, применяемый в аппаратуре Ethernet и в формате упакованных файлов ZIP.

Среди множества существующих хэш-функций выделяют криптографически стойкие, применяемые в криптографии. Для того чтобы хэш-функция H считалась ***криптографически стойкой***, она должна удовлетворять трём основным требованиям, на которых основано большинство применений хэш-функций в криптографии:

- ***необратимость***: для заданного значения хэш-функции m должно быть вычислительно неосуществимо найти блок данных X, для которого $H(X) = m$;
- ***стойкость к коллизиям первого рода***: для заданного сообщения M должно быть вычислительно неосуществимо подобрать другое сообщение N, для которого $H(N) = H(M)$;
- ***стойкость к коллизиям второго рода***: должно быть вычислительно неосуществимо подобрать пару сообщений (M, M'), имеющих одинаковый хэш.

Следует отметить, что не доказано существование необратимых хэш-функций, для которых вычисление какого-либо прообраза заданного значения хэш-функции теоретически невозможно. Обычно нахождение обратного значения является лишь вычислительно сложной задачей.

Атака «дней рождения» позволяет находить коллизии для хэш-функции с длиной значений n бит в среднем за $2^{n/2}$ вычислений хэш-функции. Поэтому n-битная хэш-функция считается криптостойкой, если вычислительная сложность нахождения коллизий для неё близка к $2^{n/2}$.

Для криптографических хэш-функций также важно, чтобы при малейшем изменении аргумента значение функции сильно изменялось (это свойство называется ***лавинным эффектом***). В частности, значение хэша не должно давать утечки информации даже об отдельных битах аргумента. Это требование служит залогом криптостойкости алгоритмов, хэширующих пользовательский пароль для получения ключа.

В данной лабораторной работе рассматриваются только криптографические хэш-функции.

Алгоритм MD5

MD5 (англ. *Message Digest 5*) – 128-битный алгоритм хэширования, разработанный Р. Ривестом из Массачусетского технологического института (MIT) в 1991 году [1, 5]. Предназначен для создания «отпечатков» или «дайджестов» сообщений произвольной длины. Является улучшенной в плане безопасности версией алгоритма *MD4*. Используется для проверки подлинности опубликованных сообщений посредством сравнения дайджеста сообщения с опубликованным хэшем. Эту операцию называют ***«проверкой хэша»***.

Алгоритм *MD5* разработан таким образом, чтобы быть достаточно быстрым для выполнения на 32-разрядном процессоре. Алгоритм не требует больших таблиц подстановок и может быть закодирован весьма компактно.

При описании алгоритма под термином ***слово*** понимается 32-битная последовательность, а под термином ***байт*** – 8-битная последовательность.

Последовательность бит может быть интерпретирована естественным образом как последовательность байт, где каждая последовательная группа из 8 бит представляет собой 1 байт. Внутри байта биты располагаются следующим образом: сначала (слева) перечисляются более значимые биты (старшие биты, соответствующие более высокой степени двойки: 2^7, 2^6, …), а в конце (справа) оказываются наименее значимые биты (младшие, соответствующие 2^2, 2^1, 2^0). Такой порядок расположения бит (или байт) называется ***big-endian*** (***порядок от старшего к младшему***).

Последовательность байт может быть интерпретирована как последовательность 32-битных слов, где каждая последовательная группа из 4 байт представляет собой 1 слово. Внутри слова байты располагаются следующим образом: сначала идут наименее значимые байты, затем – наиболее. Такой порядок расположения бит (или байт) называется ***little-endian*** (***порядок от младшего к старшему***).

Например, пусть есть последовательность бит (выделена полужирным шрифтом):

```
|0 0 0 1 0 0 0 1||0 0 1 0 0 0 1 0||0 0 1 1 0 0 1 1||0 1 0 0 0 1 0 0|...
|7 6 5 4 3 2 1 0||7 6 5 4 3 2 1 0||7 6 5 4 3 2 1 0||7 6 5 4 3 2 1 0|
└───────────────┘└───────────────┘└───────────────┘└───────────────┘
       0x11            0x22            0x33            0x44
```

Тогда она может быть интерпретирована как 4 подряд расположенных байта (0x11, 0x22, 0x33, 0x44 – шестнадцатеричные числа) или как одно слово 0x44332211.

Пусть символ `'+'` обозначает сложение по модулю 2^{32}, т.е. $a + b \equiv (a + b) \pmod{2^{32}}$. Пусть выражение `'`$X$ `<<<` s`'` означает циклический сдвиг бит числа X на s позиций влево.

<u>Описание алгоритма</u>.

Пусть имеется сообщение M длиной в b бит, хэш-функцию которого нужно вычислить. Здесь b может быть любым неотрицательным целым числом, не обязательно кратным 8. Тогда сообщение M может быть представлено в виде последовательности бит: $m_0\ m_1\ m_2\ \dots\ m_{b-2}\ m_{b-1}$.

<u>Шаг 1</u>. *Добавление дополнительных бит*.

К концу сообщения добавляется бит $'1'$, вслед за которым добавляются нулевые биты ($'0'$) до тех пор, пока длина L дополненного сообщения не станет удовлетворять условию: $L \equiv 448 \pmod{512}$. Таким образом, может быть добавлено от 1 до 512 бит.

После этого нужно преобразовать сообщение из последовательности бит, сгруппированных в байты, в последовательность слов согласно порядку расположения бит *little-endian*.

Шаг 2. *Добавление исходной длины сообщения*.

К результату предыдущего шага добавляются младшие 64 бита, взятые из побитного представления числа b (исходная длина сообщения). При этом сначала дописываются младшие 4 байта, а затем – старшие. В итоге получается сообщение, длина которого кратна 512 битам, т.е. полученное сообщение можно разбить на блоки, каждый из которых представляется в виде шестнадцати 32-битных слов: $M_0, M_1, M_2, \dots, M_{N-1}$, где N кратно 16.

Шаг 3. *Инициализация буфера*.

Буфер состоит из четырёх 32-битных слов (A, B, C, D) и используется для вычисления хэш-значения. В начале работы алгоритма буфер инициализируется следующими значениями:

A: 01 23 45 67, т.е. A = 0x67452301
B: 89 AB CD EF, т.е. B = 0xEFCDAB89
C: FE DC BA 98, т.е. C = 0x98BADCFE
D: 76 54 32 10, т.е. D = 0x10325476

Также определяются четыре функции, которые будут использоваться в дальнейшем (здесь '∧', '∨', '⊕' и '¬' – побитные операции):

```
F (x, y, z) = (x ∧ y) ∨ (¬x ∧ z)
G (x, y, z) = (x ∧ z) ∨ (¬z ∧ y)
H (x, y, z) = x ⊕ y ⊕ z
I (x, y, z) = y ⊕ (¬z ∨ x)
```

Наконец, определяется таблица констант $T[1 \dots 64]$, i-й элемент которой задаётся следующим образом:

$T[i] = \text{int}\left(4\,294\,967\,296 \cdot |\sin(i)|\right)$, где i – в радианах, а $4\,294\,967\,296 = 2^{32}$.

Шаг 4. *Основной цикл*.

Каждая итерация внешнего цикла (по переменной i) соответствует обработке одного 512-битного блока сообщения. Алгоритм обработки сообщения на псевдокоде имеет вид:

```
For i = 0 to N/16-1 do
  // Копирование блока сообщения номер i в массив X
  For j = 0 to 15 do
    X[j] = M[i*16 + j].
  end

  // Сохранение текущего значения буфера
  AA = A
  BB = B
  CC = C
  DD = D

  // Раунд 1
  // Пусть запись [abcd k s i] обозначает следующее преобразование:
  // a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s)
  // Выполняются следующие 16 преобразований
  [ABCD  0 7  1] [DABC  1 12  2] [CDAB  2 17  3] [BCDA  3 22  4]
  [ABCD  4 7  5] [DABC  5 12  6] [CDAB  6 17  7] [BCDA  7 22  8]
  [ABCD  8 7  9] [DABC  9 12 10] [CDAB 10 17 11] [BCDA 11 22 12]
  [ABCD 12 7 13] [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]

  // Раунд 2
  // Пусть запись [abcd k s i] обозначает следующее преобразование:
  // a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s)
  // Выполняются следующие 16 преобразований
  [ABCD  1 5 17] [DABC  6 9 18] [CDAB 11 14 19] [BCDA  0 20 20]
  [ABCD  5 5 21] [DABC 10 9 22] [CDAB 15 14 23] [BCDA  4 20 24]
  [ABCD  9 5 25] [DABC 14 9 26] [CDAB  3 14 27] [BCDA  8 20 28]
  [ABCD 13 5 29] [DABC  2 9 30] [CDAB  7 14 31] [BCDA 12 20 32]

  // Раунд 3
  // Пусть запись [abcd k s i] обозначает следующее преобразование:
```

```
  // a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s)
  // Выполняются следующие 16 преобразований
  [ABCD  5 4 33] [DABC  8 11 34] [CDAB 11 16 35] [BCDA 14 23 36]
  [ABCD  1 4 37] [DABC  4 11 38] [CDAB  7 16 39] [BCDA 10 23 40]
  [ABCD 13 4 41] [DABC  0 11 42] [CDAB  3 16 43] [BCDA  6 23 44]
  [ABCD  9 4 45] [DABC 12 11 46] [CDAB 15 16 47] [BCDA  2 23 48]

  // Раунд 4
  // Пусть запись [abcd k s i] обозначает следующее преобразование:
  // a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s)
  // Выполняются следующие 16 преобразований
  [ABCD  0 6 49] [DABC  7 10 50] [CDAB 14 15 51] [BCDA  5 21 52]
  [ABCD 12 6 53] [DABC  3 10 54] [CDAB 10 15 55] [BCDA  1 21 56]
  [ABCD  8 6 57] [DABC 15 10 58] [CDAB  6 15 59] [BCDA 13 21 60]
  [ABCD  4 6 61] [DABC 11 10 62] [CDAB  2 15 63] [BCDA  9 21 64]

  // К текущему значению буфера прибавляется значение буфера,
  // сохранённое в начале обработки текущего блока сообщения
  A = A + AA
  B = B + BB
  C = C + CC
  D = D + DD
end
```

Шаг 5. *Результат*.

Результатом вычисления хэш-функции являются биты слов A, B, C, D, т.е. биты результата начинаются с младшего байта слова A и заканчиваются старшим битом слова D.

Примеры значений хэш-функции (в 16-ричном виде):

```
MD5 ("") = d41d8cd9 8f00b204 e9800998 ecf8427e
MD5 ("abc") = 90015098 3cd24fb0 d6963f7d 28e17f72
MD5 ("12345678901234567890123456789012345678901234567890123456789012345678901234567890") =
   57edf4a2 2be3c955 ac49da2e 2107b67a
```

Семейство алгоритмов SHA

Семейство алгоритмов ***SHA*** (*Secure hash standard*) включает в себя 5 алгоритмов вычисления хэш-функции: ***SHA–1***, ***SHA–224***, ***SHA–256***, ***SHA–384***, ***SHA–512***. Четыре последние хэш-функции объединяются в подсемейство ***SHA–2***. Алгоритм *SHA–1* разработан Агентством национальной безопасности США (NSA) в 1995 году. Алгоритмы подсемейства *SHA–2* также разработаны Агентством национальной безопасности США и опубликованы Национальным институтом стандартов и технологий в федеральном стандарте обработки информации FIPS PUB 180–2 в августе 2002 года. Эти алгоритмы используются в *SSL*, *SSH*, *S/MIME*, *DNSSEC*, *X.509*, *PGP*, *IPSec*, при передаче файлов по сети (*BitTorrent*) [1, 2, 3, 6, 7].

Между собой алгоритмы отличаются криптостойкостью, которая обеспечивается для хэшируемых данных, а также размерами блоков и слов данных, используемых при хэшировании. Основные отличия алгоритмов можно представить в виде таблицы 1:

Таблица 1 – Свойства алгоритмов SHA

Алгоритм	Длина дайджеста сообщения (бит)	Длина внутреннего состояния (бит)	Длина блока (бит)	Длина сообщения (бит)	Длина слова (бит)	Количество итераций в цикле
SHA–1	160	160	512	$< 2^{64}$	32	80
SHA–224	224	256	512	$< 2^{64}$	32	64
SHA–256	256	256	512	$< 2^{64}$	32	64
SHA–384	384	512	1024	$< 2^{128}$	64	80
SHA–512	512	512	1024	$< 2^{128}$	64	80

Далее описываются общие для всех алгоритмов особенности.

При преобразовании последовательности бит в байты и слова используется порядок расположения бит *big-endian* (см. описание алгоритма *MD5* выше). В частности, при преобразовании последовательности бит в число наиболее незначимые 4 бита числа представляются четырьмя самыми правыми битами.

В алгоритмах используются побитные операции '$\wedge$', '$\vee$', '$\neg$', '$\oplus$', '$>>$' и '$<<$'. Под операцией '$+$' понимается сложение по модулю 2^{32} (или 2^{64} в зависимости от длины слова), т.е. $(X + Y) \pmod{2^{32}}$.

Также используются следующие обозначения для операций сдвига:

- сдвиг вправо $SHR^n(x) \equiv x >> n$;
- циклический сдвиг вправо $ROTR^n(x) \equiv (x >> n) \vee (x << w - n)$, где w – длина слова;
- циклический сдвиг влево $ROTL^n(x) \equiv (x << n) \vee (x >> w - n)$, где w – длина слова;

В алгоритмах используются следующие функции:

- $Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$;
- $Parity(x, y, z) = x \oplus y \oplus z$;
- $Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$;
- $f(x, y, z) = \begin{cases} Ch(x, y, z), & 0 \le t \le 19, \\ Parity(x, y, z), & 20 \le t \le 39, \\ Maj(x, y, z), & 40 \le t \le 59, \\ Parity(x, y, z), & 60 \le t \le 79. \end{cases}$
- $\Sigma_0^{\{256\}}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$;
- $\Sigma_1^{\{256\}}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$;
- $\sigma_0^{\{256\}}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$;
- $\sigma_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$;
- $\Sigma_0^{\{512\}}(x) = ROTR^{28}(x) \oplus ROTR^{34}(x) \oplus ROTR^{39}(x)$;
- $\Sigma_1^{\{512\}}(x) = ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x)$;
- $\sigma_0^{\{512\}}(x) = ROTR^1(x) \oplus ROTR^8(x) \oplus SHR^7(x)$;
- $\sigma_1^{\{512\}}(x) = ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^6(x)$.

Описание алгоритма.

Пусть имеется сообщение M длиной в l бит, хэш-функцию которого нужно вычислить.

Шаг 1. *Добавление дополнительных бит*.

К концу сообщения добавляется бит $'1'$, вслед за которым добавляются нулевые биты ($'0'$) до тех пор, пока длина L дополненного сообщения не станет удовлетворять условию: $L \equiv 448 \pmod{512}$. Таким образом, может быть добавлено от 1 до 512 бит.

Для *SHA–384* и *SHA–512*: $L \equiv 896 \pmod{1024}$, и соответственно может быть добавлено от 1 до 1024 бит.

Шаг 2. *Добавление исходной длины сообщения*.

К результату предыдущего шага добавляются младшие 64 бита (128 бит для *SHA–384* и *SHA–512*), взятые из побитного представления числа l (исходная длина сообщения). При этом для конвертирования последовательности бит в число используется порядок расположения бит *big-endian*. В итоге получается сообщение, длина которого кратна 512 битам (1024 битам). Например, если длина слова – 32 бита, то сообщение **"abc"**, имеющее длину 24 бита, будет дополнено до следующего сообщения (см. рисунок 1):

$$\underbrace{01100001}_{"a"}\ \underbrace{01100010}_{"b"}\ \underbrace{01100011}_{"c"}\ 1\ \overbrace{00\dots00}^{423}\ \overbrace{00\dots0\underbrace{11000}_{l=24}}^{64}$$

Рисунок 1 – Сообщение "abc", дополненное битами '1' и '0' и исходной длиной сообщения

Полученное сообщение разбивается на блоки длиной 512 (или 1024) бита, которые обозначаются $M^{(1)}, M^{(2)}, \dots, M^{(N)}$. Каждый из блоков подразделяется на 16 32-битных (или 64-битных) слов, обозначаемых $M_0^{(i)}, M_1^{(i)}, \dots, M_{15}^{(i)}$.

Дальнейшие шаги специфичны для каждого алгоритма и приводятся далее в соответствующих разделах.

Алгоритм SHA–1

Шаг 3. *Определение констант*.

SHA–1 использует последовательность 80 32-битных слов $(K_0, K_1, \dots, K_{79})$, где (в 16-ричном виде):

$$K_t = \begin{cases} 5a827999, & 0 \le t \le 19, \\ 6ed9eba1, & 20 \le t \le 39, \\ 8f1bbcdc, & 40 \le t \le 59, \\ ca62c1d6, & 60 \le t \le 79. \end{cases}$$

Начальное значение хэш-функции устанавливается следующими константами:

$H_0^{(0)}$ = 67452301, $H_1^{(0)}$ = efcdab89, $H_2^{(0)}$ = 98badcfe, $H_3^{(0)}$ = 10325476,
$H_4^{(0)}$ = c3d2e1f0.

Шаг 4. *Основной цикл*.

В следующем цикле последовательно обрабатываются все блоки, на которые было разделено дополненное сообщение:

```
For i = 1 to N
  {
   // 1. Подготовка списка преобразованных слов сообщения
```

$$W_t = \begin{cases} M_t, & 0 \le t \le 15, \\ ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}), & 16 \le t \le 79. \end{cases}$$

```
   // 2. Инициализация рабочих переменных
   a = H_0^(i-1)   b = H_1^(i-1)   c = H_2^(i-1)   d = H_3^(i-1)   e = H_4^(i-1)

   // 3. Внутренний цикл
   For t = 0 to 79
     {
      T = ROTL^5(a) + f_t(b,c,d) + e + K_t + W_t
      e = d
      d = c
      c = ROTL^30(b)
      b = a
      a = T
     }

   // 4. Вычисление промежуточного значения хэш-функции
   H_0^(i) = a + H_0^(i-1)   H_1^(i) = b + H_1^(i-1)   H_2^(i) = c + H_2^(i-1)
   H_3^(i) = d + H_3^(i-1)   H_4^(i) = e + H_4^(i-1)
  }
```

Шаг 5. *Результат*.

Результатом вычисления хэш-функции от всего сообщения является:

$$H_0^{(N)} \,||\, H_1^{(N)} \,||\, H_2^{(N)} \,||\, H_3^{(N)} \,||\, H_4^{(N)}.$$

Примеры значений хэш-функции (в 16-ричном виде):

```
SHA1 ("") = da39a3ee 5e6b4b0d 3255bfef 95601890 afd80709
SHA1 ("abc") = a9993e36 4706816a ba3e2571 7850c26c 9cd0d89d
SHA1 ("abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq") =
   84983e44 1c3bd26e baae4aa1 f95129e5 e54670f1
```

Алгоритм SHA–256

Шаг 3. *Определение констант*.

SHA–256 использует последовательность 64 32-битных слов $\left(K_0^{\{256\}}, K_1^{\{256\}}, \dots, K_{63}^{\{256\}}\right)$. Эти слова представляют собой первые 32 бита дробной части кубических корней, взятых от первых 64 простых чисел. В 16-ричном виде эти константы выглядят так:

```
428a2f98 71374491 b5c0fbcf e9b5dba5 3956c25b 59f111f1 923f82a4 ab1c5ed5
d807aa98 12835b01 243185be 550c7dc3 72be5d74 80deb1fe 9bdc06a7 c19bf174
e49b69c1 efbe4786 0fc19dc6 240ca1cc 2de92c6f 4a7484aa 5cb0a9dc 76f988da
```

```
983e5152 a831c66d b00327c8 bf597fc7 c6e00bf3 d5a79147 06ca6351 14292967
27b70a85 2e1b2138 4d2c6dfc 53380d13 650a7354 766a0abb 81c2c92e 92722c85
a2bfe8a1 a81a664b c24b8b70 c76c51a3 d192e819 d6990624 f40e3585 106aa070
19a4c116 1e376c08 2748774c 34b0bcb5 391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3
748f82ee 78a5636f 84c87814 8cc70208 90befffa a4506ceb bef9a3f7 c67178f2
```

Начальное значение хэш-функции устанавливается следующими константами:

$H_0^{(0)}$ = 6a09e667, $H_1^{(0)}$ = bb67ae85, $H_2^{(0)}$ = 3c6ef372, $H_3^{(0)}$ = a54ff53a,
$H_4^{(0)}$ = 510e527f, $H_5^{(0)}$ = 9b05688c, $H_6^{(0)}$ = 1f83d9ab, $H_7^{(0)}$ = 5be0cd19.

Шаг 4. *Основной цикл*.

В следующем цикле последовательно обрабатываются все блоки, на которые было разделено дополненное сообщение.

```
For i = 1 to N
  {
   // 1. Подготовка списка преобразованных слов сообщения
```

$$W_t = \begin{cases} M_t, & 0 \le t \le 15, \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16}, & 16 \le t \le 63. \end{cases}$$

```
   // 2. Инициализация рабочих переменных
```

a = $H_0^{(i-1)}$ b = $H_1^{(i-1)}$ c = $H_2^{(i-1)}$ d = $H_3^{(i-1)}$
e = $H_4^{(i-1)}$ f = $H_5^{(i-1)}$ g = $H_6^{(i-1)}$ h = $H_7^{(i-1)}$

```
   // 3. Внутренний цикл
   For t = 0 to 63
     {
```

T_1 = h + $\Sigma_1^{\{256\}}(e)$ + $Ch(e,f,g)$ + $K_t^{\{256\}}$ + W_t
T_2 = $\Sigma_0^{\{256\}}(a)$ + $Maj(a,b,c)$

```
      h = g
      g = f
      f = e
      e = d + T1
      d = c
      c = b
      b = a
      a = T1 + T2
     }

   // 4. Вычисление промежуточного значения хэш-функции
```

$H_0^{(i)}$ = a + $H_0^{(i-1)}$ $H_1^{(i)}$ = b + $H_1^{(i-1)}$ $H_2^{(i)}$ = c + $H_2^{(i-1)}$ $H_3^{(i)}$ = d + $H_3^{(i-1)}$
$H_4^{(i)}$ = e + $H_4^{(i-1)}$ $H_5^{(i)}$ = f + $H_5^{(i-1)}$ $H_6^{(i)}$ = g + $H_6^{(i-1)}$ $H_7^{(i)}$ = h + $H_7^{(i-1)}$

```
  }
```

Шаг 5. *Результат*.

Результатом вычисления хэш-функции от всего сообщения является:

$$H_0^{(N)} \,||\, H_1^{(N)} \,||\, H_2^{(N)} \,||\, H_3^{(N)} \,||\, H_4^{(N)} \,||\, H_5^{(N)} \,||\, H_6^{(N)} \,||\, H_7^{(N)}.$$

Примеры значений хэш-функции (в 16-ричном виде):

```
SHA256 ("") = e3b0c442 98fc1c14 9afbf4c8 996fb924 27ae41e4 649b934c a495991b 7852b855
SHA256 ("abc") = ba7816bf 8f01cfea 414140de 5dae2223 b00361a3 96177a9c b410ff61 f20015ad
SHA256 ("abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq") =
   248d6a61 d20638b8 e5c02693 0c3e6039 a33ce459 64ff2167 f6ecedd4 19db06c1
```

Алгоритм SHA–224

Шаг 3. *Определение констант*.

SHA–224 использует ту же последовательность 32-битных слов $\left(K_0^{\{256\}}, K_1^{\{256\}}, \ldots, K_{63}^{\{256\}}\right)$, что и *SHA–256*.

Начальное значение хэш-функции устанавливается следующими константами:

$H_0^{(0)}$ = c1059ed8, $H_1^{(0)}$ = 367cd507, $H_2^{(0)}$ = 3070dd17, $H_3^{(0)}$ = f70e5939,
$H_4^{(0)}$ = ffc00b31, $H_5^{(0)}$ = 68581511, $H_6^{(0)}$ = 64f98fa7, $H_7^{(0)}$ = befa4fa4.

Шаг 4. *Основной цикл*.

Основной цикл полностью совпадает с основным циклом алгоритма SHA–256.

Шаг 5. *Результат*.

Результатом вычисления хэш-функции от всего сообщения является:

$$H_0^{(N)} || H_1^{(N)} || H_2^{(N)} || H_3^{(N)} || H_4^{(N)} || H_5^{(N)} || H_6^{(N)}.$$

Примеры значений хэш-функции (в 16-ричном виде):

```
SHA256 ("") = d14a028c 2a3a2bc9 476102bb 288234c4 15a2b01f 828ea62a c5b3e42f
SHA256 ("abc") = 23097d22 3405d822 8642a477 bda255b3 2aadbce4 bda0b3f7 e36c9da7
SHA256 ("abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq") =
   75388b16 512776cc 5dba5da1 fd890150 b0c6455c b4f58b19 52522525
```

Алгоритм SHA–512

Шаг 3. *Определение констант*.

SHA–512 использует последовательность 80 64-битных слов $\left(K_0^{\{512\}}, K_1^{\{512\}}, \dots, K_{79}^{\{512\}}\right)$. Эти слова представляют собой первые 64 бита дробной части кубических корней, взятых от первых 80 простых чисел. В 16-ричном виде эти константы выглядят так:

```
428a2f98d728ae22 7137449123ef65cd b5c0fbcfec4d3b2f e9b5dba58189dbbc
3956c25bf348b538 59f111f1b605d019 923f82a4af194f9b ab1c5ed5da6d8118
d807aa98a3030242 12835b0145706fbe 243185be4ee4b28c 550c7dc3d5ffb4e2
72be5d74f27b896f 80deb1fe3b1696b1 9bdc06a725c71235 c19bf174cf692694
e49b69c19ef14ad2 efbe4786384f25e3 0fc19dc68b8cd5b5 240ca1cc77ac9c65
2de92c6f592b0275 4a7484aa6ea6e483 5cb0a9dcbd41fbd4 76f988da831153b5
983e5152ee66dfab a831c66d2db43210 b00327c898fb213f bf597fc7beef0ee4
c6e00bf33da88fc2 d5a79147930aa725 06ca6351e003826f 142929670a0e6e70
27b70a8546d22ffc 2e1b21385c26c926 4d2c6dfc5ac42aed 53380d139d95b3df
650a73548baf63de 766a0abb3c77b2a8 81c2c92e47edaee6 92722c851482353b
a2bfe8a14cf10364 a81a664bbc423001 c24b8b70d0f89791 c76c51a30654be30
d192e819d6ef5218 d69906245565a910 f40e35855771202a 106aa07032bbd1b8
19a4c116b8d2d0c8 1e376c085141ab53 2748774cdf8eeb99 34b0bcb5e19b48a8
391c0cb3c5c95a63 4ed8aa4ae3418acb 5b9cca4f7763e373 682e6ff3d6b2b8a3
748f82ee5defb2fc 78a5636f43172f60 84c87814a1f0ab72 8cc702081a6439ec
90befffa23631e28 a4506cebde82bde9 bef9a3f7b2c67915 c67178f2e372532b
ca273eceea26619c d186b8c721c0c207 eada7dd6cde0eb1e f57d4f7fee6ed178
06f067aa72176fba 0a637dc5a2c898a6 113f9804bef90dae 1b710b35131c471b
28db77f523047d84 32caab7b40c72493 3c9ebe0a15c9bebc 431d67c49c100d4c
4cc5d4becb3e42b6 597f299cfc657e2a 5fcb6fab3ad6faec 6c44198c4a475817
```

Начальное значение хэш-функции устанавливается следующими константами:

$H_0^{(0)}$ = 6a09e667f3bcc908, $H_1^{(0)}$ = bb67ae8584caa73b,
$H_2^{(0)}$ = 3c6ef372fe94f82b, $H_3^{(0)}$ = a54ff53a5f1d36f1,
$H_4^{(0)}$ = 510e527fade682d1, $H_5^{(0)}$ = 9b05688c2b3e6c1f,
$H_6^{(0)}$ = 1f83d9abfb41bd6b, $H_7^{(0)}$ = 5be0cd19137e2179.

Шаг 4. *Основной цикл*.

В следующем цикле последовательно обрабатываются все блоки, на которые было разделено дополненное сообщение.

```
For i = 1 to N
  {
   // 1. Подготовка списка преобразованных слов сообщения
```

$$W_t = \begin{cases} M_t, & 0 \le t \le 15, \\ \sigma_1^{\{512\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{512\}}(W_{t-15}) + W_{t-16}, & 16 \le t \le 79. \end{cases}$$

// 2. Инициализация рабочих переменных
a = $H_0^{(i-1)}$ b = $H_1^{(i-1)}$ c = $H_2^{(i-1)}$ d = $H_3^{(i-1)}$
e = $H_4^{(i-1)}$ f = $H_5^{(i-1)}$ g = $H_6^{(i-1)}$ h = $H_7^{(i-1)}$

// 3. Внутренний цикл
For t = 0 to 79
{
T_1 = h + $\Sigma_1^{\{512\}}(e)$ + $Ch(e,f,g)$ + $K_t^{\{512\}}$ + W_t
T_2 = $\Sigma_0^{\{512\}}(a)$ + $Maj(a,b,c)$
h = g
g = f
f = e
e = d + T_1
d = c
c = b
b = a
a = T_1 + T_2
}

// 4. Вычисление промежуточного значения хэш-функции
$H_0^{(i)}$ = a + $H_0^{(i-1)}$ $H_1^{(i)}$ = b + $H_1^{(i-1)}$ $H_2^{(i)}$ = c + $H_2^{(i-1)}$ $H_3^{(i)}$ = d + $H_3^{(i-1)}$
$H_4^{(i)}$ = e + $H_4^{(i-1)}$ $H_5^{(i)}$ = f + $H_5^{(i-1)}$ $H_6^{(i)}$ = g + $H_6^{(i-1)}$ $H_7^{(i)}$ = h + $H_7^{(i-1)}$
}

Шаг 5. *Результат*.

Результатом вычисления хэш-функции от всего сообщения является:

$$H_0^{(N)} || H_1^{(N)} || H_2^{(N)} || H_3^{(N)} || H_4^{(N)} || H_5^{(N)} || H_6^{(N)} || H_7^{(N)}.$$

Примеры значений хэш-функции (в 16-ричном виде):

```
SHA512 ("") = cf83e1357eefb8bd f1542850d66d8007 d620e4050b5715dc 83f4a921d36ce9ce
   47d0d13c5d85f2b0 ff8318d2877eec2f 63b931bd47417a81 a538327af927da3e
SHA512 ("abc") = ddaf35a193617aba cc417349ae204131 12e6fa4e89a97ea2 0a9eeee64b55d39a
   2192992a274fc1a8 36ba3c23a3feebbd 454d4423643ce80e 2a9ac94fa54ca49f
SHA512 ("abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmnhijklmnoijklmnopjklmnopqklm
   nopqrlmnopqrsmnopqrstnopqrstu") =
   8e959b75dae313da 8cf4f72814fc143f 8f7779c6eb9f7fa1 7299aeadb6889018
   501d289e4900f7e4 331b99dec4b5433a c7d329eeb6dd2654 5e96e55b874be909
```

Алгоритм SHA–384

Шаг 3. *Определение констант*.

SHA–384 использует ту же последовательность 64-битных слов $\left(K_0^{\{512\}}, K_1^{\{512\}}, \dots, K_{79}^{\{512\}}\right)$, что и *SHA–512*.

Начальное значение хэш-функции устанавливается следующими константами:

$H_0^{(0)}$ = cbbb9d5dc1059ed8, $H_1^{(0)}$ = 629a292a367cd507,
$H_2^{(0)}$ = 9159015a3070dd17, $H_3^{(0)}$ = 152fecd8f70e5939,
$H_4^{(0)}$ = 67332667ffc00b31, $H_5^{(0)}$ = 8eb44a8768581511,
$H_6^{(0)}$ = db0c2e0d64f98fa7, $H_7^{(0)}$ = 47b5481dbefa4fa4.

Шаг 4. *Основной цикл*.

Основной цикл полностью совпадает с основным циклом алгоритма *SHA–512*.

Шаг 5. *Результат*.

Результатом вычисления хэш-функции от всего сообщения является:

$$H_0^{(N)} || H_1^{(N)} || H_2^{(N)} || H_3^{(N)} || H_4^{(N)} || H_5^{(N)}.$$

Примеры значений хэш-функции (в 16-ричном виде):

```
SHA384 ("") = 38b060a751ac9638 4cd9327eb1b1e36a 21fdb71114be0743
   4c0cc7bf63f6e1da 274edebfe76f65fb d51ad2f14898b95b
SHA384 ("abc") = cb00753f45a35e8b b5a03d699ac65007 272c32ab0eded163
   1a8b605a43ff5bed 8086072ba1e7cc23 58baeca134c825a7
```

```
SHA384 ("abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmnhijklmnoijklmnopjklmnopqklm
   nopqrlmnopqrsmnopqrstnopqrstu") = 09330c33f71147e8 3d192fc782cd1b47 53111b173b3b05d2
   2fa08086e3b0f712 fcc7c71a557e2db9 66c3e9fa91746039
```

Алгоритм RIPEMD–160

RIPEMD–160 (от англ. *RACE Integrity Primitives Evaluation Message Digest*) – хэш-функция, разработанная в Лувенском католическом университете Г. Добертином, А. Босселаерсом и Б. Пренелем [1, 4]. *RIPEMD–160* является улучшенной версией *RIPEMD*, которая, в свою очередь, использовала принципы *MD4* и по производительности сравнима с более популярной *SHA–1*.

Также существуют 128-, 256- и 320-битные версии этого алгоритма, которые называются *RIPEMD–128*, *RIPEMD–256* и *RIPEMD–320*. 128-битная версия представляет собой лишь замену оригинальной *RIPEMD*, которая также была 128-битной и в которой были найдены уязвимости. 256- и 320-битные версии отличаются удвоенной длиной дайджеста, что уменьшает вероятность коллизий, но при этом функции не являются более криптостойкими.

RIPEMD–160 разработана в открытом академическом сообществе в отличие от *SHA–1* и *SHA–2*, которые были созданы NSA. Использование *RIPEMD–160* не ограничено какими-либо патентами.

Описание алгоритма.

Пусть имеется сообщение M длиной в b бит, хэш-функцию которого нужно вычислить. Здесь b может быть любым неотрицательным целым числом, необязательно кратным 8. Тогда сообщение M может быть представлено в виде последовательности бит: $m_0\ m_1\ m_2\ \ldots\ m_{b-2}\ m_{b-1}$.

Сообщение состоит из t блоков, каждый из которых состоит из 16 32-битных слов. Каждое 32-битное слово содержит четыре байта, но следуют они не подряд, а наоборот (например, из восьми байт ($a\ b\ c\ d\ e\ f\ g\ h$) получаются два 32-битных слова ($dcba\ hgfe$)). Пусть $M[i][j]$ обозначает *j*-е слово *i*-го блока сообщения.

Пусть символ `'+'` обозначает сложение по модулю 2^{32}, т.е. $a + b \equiv (a + b) \pmod{2^{32}}$. Пусть выражение '$X$ `<<<` s' означает циклический сдвиг бит числа X на s позиций влево.

Шаги 1 и 2. *Добавление дополнительных бит* и *Добавление исходной длины сообщения*.
Эти шаги полностью совпадают с шагами 1 и 2 алгоритма *MD5*.

Шаг 3. *Определение констант и используемых функций*.
Нелинейная побитовая функция f:

$$f(j,x,y,z) = \begin{cases} x \oplus y \oplus z, & 0 \le j \le 15, \\ (x \wedge y) \vee (\neg x \wedge z), & 16 \le j \le 31, \\ (x \vee \neg y) \oplus z, & 32 \le j \le 47, \\ (x \wedge z) \vee (y \wedge \neg z), & 48 \le j \le 63, \\ x \oplus (y \vee \neg z), & 64 \le j \le 79. \end{cases}$$

Добавляемые 16-ричные константы:

$$K1(j) = \begin{cases} 00000000, & 0 \le j \le 15, \\ 5a827999, & 16 \le j \le 31, \\ 6ed9eba1, & 32 \le j \le 47, \\ 8f1bbcdc, & 48 \le j \le 63, \\ a953fd4e, & 64 \le j \le 79. \end{cases}, \qquad K2(j) = \begin{cases} 50a28be6, & 0 \le j \le 15, \\ 5c4dd124, & 16 \le j \le 31, \\ 6d703ef3, & 32 \le j \le 47, \\ 7a6d76e9, & 48 \le j \le 63, \\ 00000000, & 64 \le j \le 79. \end{cases}$$

Номера выбираемых из сообщения 32-битных слов:

```
R1( 0..15) =  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15
R1(16..31) =  7,  4, 13,  1, 10,  6, 15,  3, 12,  0,  9,  5,  2, 14, 11,  8
R1(32..47) =  3, 10, 14,  4,  9, 15,  8,  1,  2,  7,  0,  6, 13, 11,  5, 12
R1(48..63) =  1,  9, 11, 10,  0,  8, 12,  4, 13,  3,  7, 15, 14,  5,  6,  2
R1(64..79) =  4,  0,  5,  9,  7, 12,  2, 10, 14,  1,  3,  8, 11,  6, 15, 13
R2( 0..15) =  5, 14,  7,  0,  9,  2, 11,  4, 13,  6, 15,  8,  1, 10,  3, 12
R2(16..31) =  6, 11,  3,  7,  0, 13,  5, 10, 14, 15,  8, 12,  4,  9,  1,  2
R2(32..47) = 15,  5,  1,  3,  7, 14,  6,  9, 11,  8, 12,  2, 10,  0,  4, 13
R2(48..63) =  8,  6,  4,  1,  3, 11, 15,  0,  5, 12,  2, 13,  9,  7, 10, 14
R2(64..79) = 12, 15, 10,  4,  1,  5,  8,  7,  6,  2, 13, 14,  0,  3,  9, 11
```

Количество бит, на которое будут осуществляться сдвиги:

```
S1( 0..15) = 11, 14, 15, 12,  5,  8,  7,  9, 11, 13, 14, 15,  6,  7,  9,  8
S1(16..31) =  7,  6,  8, 13, 11,  9,  7, 15,  7, 12, 15,  9, 11,  7, 13, 12
S1(32..47) = 11, 13,  6,  7, 14,  9, 13, 15, 14,  8, 13,  6,  5, 12,  7,  5
S1(48..63) = 11, 12, 14, 15, 14, 15,  9,  8,  9, 14,  5,  6,  8,  6,  5, 12
S1(64..79) =  9, 15,  5, 11,  6,  8, 13, 12,  5, 12, 13, 14, 11,  8,  5,  6
S2( 0..15) =  8,  9,  9, 11, 13, 15, 15,  5,  7,  7,  8, 11, 14, 14, 12,  6
S2(16..31) =  9, 13, 15,  7, 12,  8,  9, 11,  7,  7, 12,  7,  6, 15, 13, 11
S2(32..47) =  9,  7, 15, 11,  8,  6,  6, 14, 12, 13,  5, 14, 13, 13,  7,  5
S2(48..63) = 15,  5,  8, 11, 14, 14,  6, 14,  6,  9, 12,  9, 12,  5, 15,  8
S2(64..79) =  8,  5, 12,  9, 12,  5, 14,  6,  8, 13,  6,  5, 15, 13, 11, 11
```

Начальное значение хэш-функции:

```
h0 = 67452301;     h1 = efcdab89;     h2 = 98badcfe;     h3 = 10325476;     h4 = c3d2e1f0;
h5 = 76543210;     h6 = fedcba98;     h7 = 89abcdef;     h8 = 01234567;     h9 = 3c2d1e0f;
```

Шаг 4. *Основной цикл*.

Каждая итерация внешнего цикла (по переменной i) соответствует обработке одного 512-битного блока сообщения. Алгоритм обработки сообщения на псевдокоде имеет вид:

```
For i = 0 to t-1
  {
   A1 = h0;   B1 = h1;   C1 = h2;   D1 = h3;   E1 = h4;
   A2 = h0;   B2 = h1;   C2 = h2;   D2 = h3;   E2 = h4;
   For j = 0 to 79
     {
      T = (A1 + f(j, B1, C1, D1) + M[i][R1(j)] + K1(j)) <<< S1(j) + E1;
      A1 = E1;   E1 = D1;   D1 = C1 <<< 10;   C1 = B1;   B1 = T;
      T = (A2 + f(79-j, B2, C2, D2) + M[i][R2(j)] + K2(j)) <<< S2(j) + E2;
      A2 = E2;   E2 = D2;   D2 = C2 <<< 10;   C2 = B2;   B2 = T;
     }
   T = h1 + C1 + D2;    h1 = h2 + D1 + E2;   h2 = h3 + E1 + A2;
   h3 = h4 + A1 + B2;   h4 = h0 + B1 + C2;   h0 = T;
  }
```

Шаг 5. *Результат*.

Результатом вычисления хэш-функции от всего сообщения является:

```
h0 || h1 || h2 || h3 || h4.
```

Примеры значений хэш-функции (в 16-ричном виде):

```
RIPEMD160 ("") = 9c1185a5 c5e9fc54 61280897 7ee8f548 b2258d31
RIPEMD160 ("abc") = 8eb208f7 e05d987a 9b044a8e 98c6b087 f15a0bfc
RIPEMD160 ("12345678901234567890123456789012345678901234567890123456789012345678901234567890
   ") = 9b752e45 573d4b39 f4dbd332 3cab82bf 63326bfb
```

Алгоритм RIPEMD–128

Описание алгоритма и шаги 1, 2 и 3 аналогичны алгоритму *RIPEMD–160*.

Шаг 4. *Основной цикл*.

Каждая итерация внешнего цикла (по переменной i) соответствует обработке одного 512-битного блока сообщения. Алгоритм обработки сообщения на псевдокоде имеет вид:

```
For i = 0 to t-1
  {
   A1 = h0;   B1 = h1;   C1 = h2;   D1 = h3;
   A2 = h0;   B2 = h1;   C2 = h2;   D2 = h3;
   For j = 0 to 63
     {
      T = (A1 + f(j, B1, C1, D1) + M[i][R1(j)] + K1(j)) <<< S1(j);
      A1 = D1;   D1 = C1;   C1 = B1;   B1 = T;
      T = (A2 + f(63-j, B2, C2, D2) + M[i][R2(j)] + K2(j)) <<< S2(j);
      A2 = D2;   D2 = C2;   C2 = B2;   B2 = T;
     }
```

```
   T = h1 + C1 + D2;    h1 = h2 + D1 + A2;   h2 = h3 + A1 + B2;
   h3 = h0 + B1 + C2;   h0 = T;
  }
```

Шаг 5. *Результат*.

Результатом вычисления хэш-функции от всего сообщения является:

```
h0 || h1 || h2 || h3
```

Примеры значений хэш-функции (в 16-ричном виде):

```
RIPEMD128 ("") = cdf26213 a150dc3e cb610f18 f6b38b46
RIPEMD128 ("abc") = c14a1219 9c66e4ba 84636b0f 69144c77
RIPEMD128 ("12345678901234567890123456789012345678901234567890123456789012345678901234567890
   ") = 3f45ef19 4732c2db b2c4a2c7 69795fa3
```

Алгоритм RIPEMD–256

Описание алгоритма и шаги 1, 2 и 3 аналогичны алгоритму *RIPEMD–160*.

Шаг 4. *Основной цикл*.

Каждая итерация внешнего цикла (по переменной i) соответствует обработке одного 512-битного блока сообщения. Алгоритм обработки сообщения на псевдокоде имеет вид:

```
For i = 0 to t-1
  {
   A1 = h0;   B1 = h1;   C1 = h2;   D1 = h3;
   A2 = h4;   B2 = h5;   C2 = h6;   D2 = h7;
   For j = 0 to 63
     {
      T = (A1 + f(j, B1, C1, D1) + M[i][R1(j)] + K1(j)) <<< S1(j);
      A1 = D1;   D1 = C1;   C1 = B1;   B1 = T;
      T = (A2 + f(63-j, B2, C2, D2) + M[i][R2(j)] + K2(j)) <<< S2(j);
      A2 = D2;   D2 = C2;   C2 = B2;   B2 = T;
      if j == 15
        { T = A1;   A1 = A2;   A2 = T; }
      if j == 31
        { T = B1;   B1 = B2;   B2 = T; }
      if j == 47
        { T = C1;   C1 = C2;   C2 = T; }
      if j == 63
        { T = D1;   D1 = D2;   D2 = T; }
     }
   h0 = h0 + A1;   h1 = h1 + B1;   h2 = h2 + C1;   h3 = h3 + D1;
   h4 = h4 + A1;   h5 = h5 + B2;   h6 = h6 + C2;   h7 = h7 + D2;
  }
```

Шаг 5. *Результат*.

Результатом вычисления хэш-функции от всего сообщения является:

```
h0 || h1 || h2 || h3 || h4 || h5 || h6 || h7
```

Примеры значений хэш-функции (в 16-ричном виде):

```
RIPEMD256 ("") = 02ba4c4e 5f8ecd18 77fc52d6 4d30e37a 2d9774fb 1e5d0263 80ae0168 e3c5522d
RIPEMD256 ("abc") = afbd6e22 8b9d8cbb cef5ca2d 03e6dba1 0ac0bc7d cbe4680e 1e42d2e9 75459b65
RIPEMD256 ("12345678901234567890123456789012345678901234567890123456789012345678901234567890
   ") = 06fdcc7a 409548aa f91368c0 6a6275b5 53e3f099 bf0ea4ed fd6778df 89a890dd
```

Алгоритм RIPEMD–320

Описание алгоритма и шаги 1, 2 и 3 аналогичны алгоритму *RIPEMD–160*.

Шаг 4. *Основной цикл*.

Каждая итерация внешнего цикла (по переменной i) соответствует обработке одного 512-битного блока сообщения. Алгоритм обработки сообщения на псевдокоде имеет вид:

```
For i = 0 to t-1
  {
   A1 = h0;   B1 = h1;   C1 = h2;   D1 = h3;   E1 = h4;
   A2 = h5;   B2 = h6;   C2 = h7;   D2 = h8;   E2 = h9;
   For j = 0 to 79
     {
      T = (A1 + f(j, B1, C1, D1) + M[i][R1(j)] + K1(j)) <<< S1(j) + E1;
      A1 = E1;   E1 = D1;   D1 = C1 <<< 10;   C1 = B1;   B1 = T;
      T = (A2 + f(79-j, B2, C2, D2) + M[i][R2(j)] + K2(j)) <<< S2(j) + E2;
      A2 = E2;   E2 = D2;   D2 = C2 <<< 10;   C2 = B2;   B2 = T;
      if j == 15
        { T = B1;   B1 = B2;   B2 = T; }
      if j == 31
        { T = D1;   D1 = D2;   D2 = T; }
      if j == 47
        { T = A1;   A1 = A2;   A2 = T; }
      if j == 63
        { T = C1;   C1 = C2;   C2 = T; }
      if j == 79
        { T = E1;   E1 = E2;   E2 = T; }
     }
   h0 = h0 + A1;   h1 = h1 + B1;   h2 = h2 + C1;   h3 = h3 + D1;
   h4 = h4 + E1;   h5 = h5 + A2;   h6 = h6 + B2;   h7 = h7 + C2;
   h8 = h8 + D2;   h9 = h9 + E2;
  }
```

Шаг 5. *Результат*.

Результатом вычисления хэш-функции от всего сообщения является:

```
h0 || h1 || h2 || h3 || h4 || h5 || h6 || h7 || h8 || h9
```

Примеры значений хэш-функции (в 16-ричном виде):

```
RIPEMD320 ("") = 22d65d56 61536cdc 75c1fdf5 c6de7b41 b9f27325 ebc61e85 57177d70 5a0ec880
   151c3a32 a00899b8
RIPEMD320 ("abc") = de4c01b3 054f8930 a79d09ae 738e9230 1e5a1708 5beffdc1 b8d11671 3e74f82f
   a942d64c dbc4682d
RIPEMD320 ("12345678901234567890123456789012345678901234567890123456789012345678901234567890
   ") = 557888af 5f6d8ed6 2ab66945 c6d2a0a4 7ecd5341 e915eb8f ea1d0524 955f825d
   c717e4a0 08ab2d42
```

Задание

I. Реализовать приложение с графическим интерфейсом, позволяющее выполнять следующие действия.

1. Вычислять значение хэш-функции, заданной в варианте:
   1) текст сообщения должен считываться из файла;
   2) полученное значение хэш-функции должно представляться в шестнадцатеричном виде и сохраняться в файл;
   3) при работе программы должна быть возможность просмотра и изменения считанного из файла сообщения и вычисленного значения хэш-функции.
2. Исследовать лавинный эффект на сообщении, состоящем из одного блока:
   1) для бита, который будет изменяться, приложение должно позволять задавать его позицию (номер) в сообщении;
   2) приложение должно уметь после каждого раунда (итерации цикла) вычисления хэш-функции подсчитывать число бит, изменившихся в хэше при изменении одного бита в тексте сообщения;
   3) приложение может строить графики зависимости числа бит, изменившихся в хэше, от раунда вычисления хэш-функции, либо графики можно строить в стороннем ПО, но тогда приложение должно сохранять в файл необходимую для построения графиков информацию.

II. С помощью реализованного приложения выполнить следующие задания.
   1. Протестировать правильность работы разработанного приложения.
   2. Исследовать лавинный эффект при изменении одного бита в сообщении: для различных позиций изменяемого бита в сообщении построить графики зависимостей числа бит, изменившихся в хэше, от раунда вычисления хэш-функции (всего в отчёте должно быть 2–3 графика).
   3. Сделать выводы о проделанной работе.

Дополнительные критерии оценивания качества работы

1. Построение графиков:
   1 – программа сама строит графики лавинного эффекта;
   0 – программа только выгружает необходимые для построения графиков данные;
   л.р. не принимается – программа не строит графики и не выгружает данные.

Варианты

1. Алгоритм SHA–512.
2. Алгоритм RIPEMD–320.
3. Алгоритм MD5.
4. Алгоритм SHA–256.
5. Алгоритм RIPEMD–128.
6. Алгоритм RIPEMD–160.
7. Алгоритм SHA–1.
8. Алгоритм MD5.
9. Алгоритм SHA–384.
10. Алгоритм RIPEMD–256.
11. Алгоритм MD5.
12. Алгоритм SHA–224.
13. Алгоритм RIPEMD–160.
14. Алгоритм SHA–512.
15. Алгоритм SHA–256.

Контрольные вопросы

1. Что такое хэш-функция? Когда она является криптографически стойкой? Что такое лавинный эффект?
2. Алгоритм MD5.
3. Семейство алгоритмов SHA.
4. Семейство алгоритмов RIPEMD.

Список литературы

1. Столлингс, В. Криптография и защита сетей: принципы и практика : Пер. с англ. / В. Столлингс. – 2-е изд. – М. : Издательский дом "Вильямс", 2001. – 672 с.
2. Eastlake, D.E. US Secure Hash Algorithm 1 (SHA1) : RFC 3174 / D.E. Eastlake, P.E. Jones. – Milford : Motorola; Research Triangle Park : Cisco Systems, Inc., 2001. – 22 p.
3. Eastlake, D.E. US Secure Hash Algorithms (SHA and HMAC–SHA) : RFC 4634 / D.E. Eastlake, T. Hansen. – Milford : Motorola Laboratories; Middletown : AT&T Laboratories, 2006. – 108 p.
4. RIPEMD–160: A Strengthened Version of RIPEMD [Электронный ресурс] : Article / H. Dobbertin, A. Bosselaers, B. Preneel. – Режим доступа: http://homes.esat.kuleuven.be/~cosicart/pdf/AB-9601/AB-9601.pdf.
5. Rivest, R. The MD5 Message-Digest Algorithm : RFC 1321 / R. Rivest. – Cambridge : MIT Laboratory for Computer Science and RSA Data Security, Inc., 1992. – 21 p.
6. Secure hash standard : FIPS Publication 180-2. – Springfield : National Technical Information Service, 2002. – 79 p.
7. Secure hash standard : FIPS Publication 180-3. – Gaithersburg : National Institute of Standards and Technology, 2008. – 27 p.