Independent Submission
Request for Comments: 7693
Category: Informational
ISSN: 2070-1721

M-J. Saarinen, Ed.
Queen's University Belfast
J-P. Aumasson
Kudelski Security

November 2015

The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)

Функция BLAKE2 для криптографического хэширования и кодов аутентификации сообщений

Аннотация

В этом документе описана криптографическая хэш-функция BLAKE2 и приведена спецификация алгоритма, а также исходный код на языке C для сообщества Internet. Имеется два основных варианта BLAKE2 - BLAKE2b, оптимизированная для 64-битовых платформ и BLAKE2s для более мелких систем. BLAKE2 можно напрямую связать с ключом, что делает функцию эквивалентом кода аутентификации сообщения (Message Authentication Code или MAC).

Статус документа

Этот документ не является спецификацией какого-либо стандарта Internet и публикуется с информационными целями.

Этот документ документ является вкладом в серию RFC, независимым от какого-либо потока RFC. RFC Editor принял решение о публикации документа по своему усмотрению и не делает каких-либо заявлений о ценности документа для реализации или внедрения. Документы, одобренные для публикации RFC Editor, не претендуют на статус стандартов Internet, см. раздел 2 в RFC 5741.

Информацию о текущем статусе документа, ошибках и способах обратной связи можно найти по ссылке http://www.rfc-editor.org/info/rfc7693.

Оглавление

1. Введение и терминология

Криптографическую функцию BLAKE2 [BLAKE2] разработали Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, Christian Winnerlein. Имеется два основных варианта функции:

- BLAKE2b (или просто BLAKE2) оптимизирована для 64-битовых систем и выдаёт дайджесты от 1 до 64 байтов.
- BLAKE2s оптимизирована для систем от 8 до 32 битов и выдаёт дайджесты от 1 до 32 байтов.

Обе функции BLAKE2b и BLAKE2s считаются очень безопасными и хорошо работают на любой программной или аппаратной платформе. BLAKE2 не требует специальной конструкции HMAC[1] для аутентификации сообщений с ключам, поскольку она включает встроенный механизм ключей.

Хэш-функцию BLAKE2 могут применять алгоритмы цифровых подписей, а также механизмы проверки подлинности и целостности сообщений в таких приложениях, как инфраструктура открытых ключей (Public Key Infrastructure или PKI), защищённые протоколы связи, облачные хранилища, обнаружение вторжений, криминалистические системы, а также системы контроля версий. Набор BLAKE2 представляет собой более эффективную альтернативу американским алгоритмам защищённого хэширования (US Secure Hash Algorithm) SHA и HMAC-SHA [RFC6234]. BLAKE2s-128 особенно подходит для быстрой и более защищённой замены MD5 и HMAC-MD5 в унаследованных приложениях [RFC6151].

Для облегчения реализации в Приложении A представлена трассировка хэширования BLAKE2b-512, а в Приложении B - BLAKE2s-256. Из-за ограниченного размера документ не содержит полного набора тестовых векторов для BLAKE2.

Справочная реализация BLAKE2b на языке C представлена в Приложении C, а BLAKE2s - в Приложении D. Эти реализации можно проверить с помощью более полного тестового модуля из Приложения E.

Ключевые слова **необходимо** (MUST), **недопустимо** (MUST NOT), **требуется** (REQUIRED), **нужно** (SHALL), **не следует** (SHALL NOT), **следует** (SHOULD), **не нужно** (SHOULD NOT), **рекомендуется** (RECOMMENDED), **не рекомендуется** (NOT RECOMMENDED), **возможно** (MAY), **необязательно** (OPTIONAL) в данном документе интерпретируются в соответствии с [RFC2119].

2. Соглашения, переменные и константы

2.1. Параметры

В таблице ниже приведены параметры и диапазоны их значений.

	BLAKE2b	***BLAKE2s***
Число битов слова	w = 64	w = 32
Число раундов F	r = 12	r = 10
Число байтов в блоке	bb = 128	bb = 64
Число байтов хэш-значения	1 <= nn <= 64	1 <= nn <= 32
Число байтов ключа	0 <= kk <= 64	0 <= kk <= 32
Число входных байтов	0 <= ll < 2**128	0 <= ll < 2**64
Циклический сдвиг G	(R1, R2, R3, R4)	(R1, R2, R3, R4)
Константы сдвига	(32, 24, 16, 63)	(16, 12, 8, 7)

2.2. Другие константы и переменные

Ниже указаны переменные и константы, используемые в описании алгоритма.

IV[0..7]
: Вектор инициализации (константа).

SIGMA[0..9]
: Перестановки слов с сообщении (константа).

p[0..7]
: Блок параметров (определяет размеры хэша и ключей).

m[0..15]
: 16 слов из одного блока сообщений.

h[0..7]
: Внутреннее состояние хэша.

d[0..dd-1]
: Дополненные входные блоки по bb байтов.

t
: Смещение байта сообщения в конце текущего блока.

f
: Флаг, указываюий последний блок.

2.3. Арифметические обозначения

Для действительных (вещественных) значений x определены указанные ниже функции.

floor(x)
: Округление вниз до наибольшего целого числа, которое не больше x.

ceil(x)
: Округление вверх до меньшего целого числа, которое не меньше x.

frac(x)
: Положительное значение дробной части x, frac(x) = x - floor(x).

Ниже приведены обозначения операций в псевдокоде.

2**n
: 2 в степени n (2**0=1, 2**1=2, 2**2=4, 2**3=8 и т. д.).

a ^ b
: Побитовая операция исключающее-ИЛИ (XOR) между a и b.

a mod b
: Деление по модулю - остаток при делении a на b (всегда находится в диапазоне [0, b-1]).

x >> n = floor(x / 2**n)
: Логический сдвиг x вправо на n битов.

[1] Hashed Message Authentication Code - хэшированный код аутентификации сообщения.

x << n = (x * 2**n) mod (2**w)

Логический сдвиг x влево на n битов.

x >>> n = (x >> n) ^ (x << (w — n))

Циклический сдвиг x вправо на n битов.

2.4. Интерпретация слов как байтов с порядком Little-Endian

Все математические операции выполняются на 64-битовыми словами в BLAKE2b и 32-битовыми в BLAKE2s. Разрешены также операции над векторами слов. Векторы индексируются от 0, т. е. первым элементом вектора v из n элементов является v[0], последним - v[n - 1]. Все элементы вектора обозначаются v[0..n-1]. Потоки байтов (октетов) интерпретируются как слова с порядком little-endian, где первым является младший байт. Рассмотрим последовательность из восьми шестнадцатеричных байтов

```
x[0..7] = 0x01 0x23 0x45 0x67 0x89 0xAB 0xCD 0xEF
```

При интерпретации как 32-битового слова с начального адреса в памяти байты x[0..3] имеют численное значение 0x67452301 или 1732584193. При интерпретации как 64-битового слова байты x[0..7] имеют численное значение 0xEFCDAB8967452301 или 17279655951921914625.

2.5. Блок параметров

Слова блока параметров p[0..7] задаются в виде

```
смещение байта:    3 2 1 0     (другие байты)
        p[0] = 0x0101kknn     p[1..7] = 0
```

Байт nn задаёт размер хэша в байтах, второй (little-endian) байт блока параметров (kk) указывает размер ключа в байтах. Установка kk = 00 задаёт хэширование без ключа. Байты 2 и 3 имеют значение 01, остальные байты блока параметров имеют значение 0.

> Примечание. В [BLAKE2] определены дополнительные варианты BLAKE2 с такими функциями, как применение затравки (salting), персонализованные хэши и древовидное хэширование. Эти **необязательные** свойства используют поля блока параметров, которые в этом документе не заданы.

2.6. Вектор инициализации

Константа вектора инициализации (Initialization Vector или IV) математически задаётся в виде

```
IV[i] = floor(2**w * frac(sqrt(prime(i+1))))
```

где prime(i) - i-е простое число (2, 3, 5, 7, 11, 13, 17, 19), a sqrt(x) - квадратный корень из x.

Численные значения IV приведены в реализациях Приложений C и D для BLAKE2b и BLAKE2s, соответственно.

> Примечание. BLAKE2b IV совпадает с SHA-512 IV, а BLAKE2s IV - с SHA-256 IV (см. [RFC6234]).

2.7. Распорядок сообщения SIGMA

Перестановки порядка слов в сообщении для каждого раунда в BLAKE2b и BLAKE2s определяются SIGMA. В BLAKE2b две дополнительные перестановки для раундов 10 и 11 - это SIGMA[10..11] = SIGMA[0..1].

Раунд	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***
SIGMA[0]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SIGMA[1]	14	10	4	8	9	15	13	6	1	12	0	2	11	7	5	3
SIGMA[2]	11	8	12	0	5	2	15	13	10	14	3	6	7	1	9	4
SIGMA[3]	7	9	3	1	13	12	11	14	2	6	5	10	4	0	15	8
SIGMA[4]	9	0	5	7	2	4	10	15	14	1	11	12	6	8	3	13
SIGMA[5]	2	12	6	10	0	11	8	3	4	13	7	5	15	14	1	9
SIGMA[6]	12	5	1	15	14	13	4	10	0	7	6	3	9	2	8	11
SIGMA[7]	13	11	7	14	12	1	3	9	5	0	15	4	8	6	2	10
SIGMA[8]	6	15	14	9	11	3	0	8	12	2	13	7	1	4	10	5
SIGMA[9]	10	2	8	4	7	6	1	5	15	11	9	14	3	12	13	0

3. Обработка BLAKE2

3.1. Функция смешивания G

Функция G смешивает 2 входных слова x и y в 4 слова с индексами a, b, c, d в рабочем векторе v[0..15]. Возвращается полностью изменённый вектор. Константы циклического сдвига (R1, R2, R3, R4) указаны в параграф 2.1.

```
FUNCTION G( v[0..15], a, b, c, d, x, y )

   v[a] := (v[a] + v[b] + x) mod 2**w
   v[d] := (v[d] ^ v[a]) >>> R1
   v[c] := (v[c] + v[d])     mod 2**w
   v[b] := (v[b] ^ v[c]) >>> R2
   v[a] := (v[a] + v[b] + y) mod 2**w
   v[d] := (v[d] ^ v[a]) >>> R3
   v[c] := (v[c] + v[d])     mod 2**w
   v[b] := (v[b] ^ v[c]) >>> R4

   RETURN v[0..15]

END FUNCTION.
```

3.2. Функция сжатия F

Функция сжатия F принимает в качестве аргументов вектор состояния h, вектор блока сообщений m (при необходимости последний блок дополняется нулями до полного размера блока), счётчик смещения t размером 2w

битов и флаг индикации финального блока f. При обработке применяется локальный вектор v[0..15]. Функция F возвращает новый вектор состояния. Число раундов r составляет 12 для BLAKE2b и 10 для BLAKE2s. Раунды нумеруются от 0 до r - 1.

```
FUNCTION F( h[0..7], m[0..15], t, f )

      // Инициализация локального рабочего вектора v[0..15]
      v[0..7] := h[0..7]              // Первая половина из состояния.
      v[8..15] := IV[0..7]            // Вторая половина из IV.

      v[12] := v[12] ^ (t mod 2**w)   // Младшее слово смещения.
      v[13] := v[13] ^ (t >> w)       // Старшее слово смещения.

      IF f = TRUE THEN                // Флаг последнего блока?
         v[14] := v[14] ^ 0xFF..FF    // Инверсия всех битов.
      END IF.

      // Cryptographic mixing
      FOR i = 0 TO r - 1 DO           // 10 или 12 раундов.

         // Перестановка выбора слов сообщения для этого раунда.
         s[0..15] := SIGMA[i mod 10][0..15]

         v := G( v, 0, 4,  8, 12, m[s[ 0]], m[s[ 1]] )
         v := G( v, 1, 5,  9, 13, m[s[ 2]], m[s[ 3]] )
         v := G( v, 2, 6, 10, 14, m[s[ 4]], m[s[ 5]] )
         v := G( v, 3, 7, 11, 15, m[s[ 6]], m[s[ 7]] )

         v := G( v, 0, 5, 10, 15, m[s[ 8]], m[s[ 9]] )
         v := G( v, 1, 6, 11, 12, m[s[10]], m[s[11]] )
         v := G( v, 2, 7,  8, 13, m[s[12]], m[s[13]] )
         v := G( v, 3, 4,  9, 14, m[s[14]], m[s[15]] )

      END FOR

      FOR i = 0 TO 7 DO               // XOR двух половин.
         h[i] := h[i] ^ v[i] ^ v[i + 8]
      END FOR.

      RETURN h[0..7]                  // Новое состояние.

END FUNCTION.
```

3.3. Данные заполнения и расчёт дайджеста BLAKE2

Реализации BLAKE2b и BLAKE2s на языке C приведены в Приложениях C и D.

Ввод ключа и данных разбивается (с дополнением) на dd блоков сообщения d[0..dd-1] по 16 слов (bb байтов). При использовании секретного ключа (kk > 0) он заполняется нулевыми байтами и устанавливается как d[0]. В ином случае d[0] - это первый блок данных. Финальный блок d[dd-1] дополняется нулями до bb байтов (16 слов).

Число блоков составляет dd = ceil(kk / bb) + ceil(ll / bb). Однако в особом случае пустого сообщения без ключа (kk = 0, ll = 0) устанавливается dd = 1, d[0] состоит из нулей.

В приведённой ниже процедуре обрабатываются дополненные блоки данных для создания финального хэш-значения из nn байтов. Описания применяемых переменных и констант приведены в разделе 2.

```
 FUNCTION BLAKE2( d[0..dd-1], ll, kk, nn )

      h[0..7] := IV[0..7]          // Вектор инициализации IV.

      // Блок параметров p[0]
      h[0] := h[0] ^ 0x01010000 ^ (kk << 8) ^ nn

      // Процесс дополнения ключа и болков данных.
      IF dd > 1 THEN
            FOR i = 0 TO dd - 2 DO
                  h := F( h, d[i], (i + 1) * bb, FALSE )
            END FOR.
      END IF.

      // Финальный блок.
      IF kk = 0 THEN
            h := F( h, d[dd - 1], ll, TRUE )
      ELSE
            h := F( h, d[dd - 1], ll + bb, TRUE )
      END IF.

      RETURN первые nn байтов из сассива слов h[] (little-endian).

 END FUNCTION.
```

4. Стандартные наборы параметров и идентификаторы алгоритмов

Реализации BLAKE2b и BLAKE2s могут поддерживать приведённые ниже параметры размера дайджеста для функциональной совместимости (например, цифровых подписей), если при таком выборе параметров обеспечивается

достаточный уровень безопасности. Параметры и идентификаторы предназначены для использования в качестве замены MD5 и соответствующих алгоритмов SHA.

Разработчикам, адаптирующим BLAKE2 к форматам сообщений на основе ASN.1, следует использовать терево OID при x = 1.3.6.1.4.1.1722.12.2. Один и тот же идентификатор OID может применяться при хэшировании с ключом и без ключа, поскольку в последнем случае ключ просто имеет нулевой размер.

Идентификатор алгоритма	*Целевая архитектура*	*Защита от конфликтов*	*Размер хэша nn*	*Суффикс хэша ASN.1 OID*
id-blake2b160	64 бита	2**80	20	x.1.5
id-blake2b256	64 бита	2**128	32	x.1.8
id-blake2b384	64 бита	2**192	48	x.1.12
id-blake2b512	64 бита	2**256	64	x.1.16
id-blake2s128	32 бита	2**64	16	x.2.4
id-blake2s160	32 бита	2**80	20	x.2.5
id-blake2s224	32 бита	2**112	28	x.2.7
id-blake2s256	32 бита	2**128	32	x.2.8

```
hashAlgs OBJECT IDENTIFIER ::= {
    iso(1) identified-organization(3) dod(6) internet(1)
    private(4) enterprise(1) kudelski(1722) cryptography(12) 2
}
macAlgs OBJECT IDENTIFIER ::= {
    iso(1) identified-organization(3) dod(6) internet(1)
    private(4) enterprise(1) kudelski(1722) cryptography(12) 3
}

-- Два варианта BLAKE2 --
blake2b OBJECT IDENTIFIER ::= { hashAlgs 1 }
blake2s OBJECT IDENTIFIER ::= { hashAlgs 2 }

-- Идентификаторы BLAKE2b --
id-blake2b160 OBJECT IDENTIFIER ::= { blake2b 5 }
id-blake2b256 OBJECT IDENTIFIER ::= { blake2b 8 }
id-blake2b384 OBJECT IDENTIFIER ::= { blake2b 12 }
id-blake2b512 OBJECT IDENTIFIER ::= { blake2b 16 }

-- Идентификаторы BLAKE2s --
id-blake2s128 OBJECT IDENTIFIER ::= { blake2s 4 }
id-blake2s160 OBJECT IDENTIFIER ::= { blake2s 5 }
id-blake2s224 OBJECT IDENTIFIER ::= { blake2s 7 }
id-blake2s256 OBJECT IDENTIFIER ::= { blake2s 8 }
```

5. Вопросы безопасности

Этот документ предназначен для предоставления удобного доступа сообщества Internet к открытому исходному коду алгоритма криптографического хэширования BLAKE2. Документ не содержит каикх-либо заявлений о безопасности алгоритма. Подробное криптоаналитическое обоснование приведено в [BLAKE] и [BLAKE2].

Для предотвращения разрастания справочные реализации в Приложениях C и D могут не уничтожать в памяти все конфиденциальные данные (например, секретные ключи) сразу же после использования. При необходимости такую очистку можно добавить.

6. Литература

6.1. Нормативные документы

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

6.2. Дополнительная литература

[BLAKE] Aumasson, J-P., Meier, W., Phan, R., and L. Henzen, "The Hash Function BLAKE", January 2015, <https://131002.net/blake/book>.

[BLAKE2] Aumasson, J-P., Neves, S., Wilcox-O'Hearn, Z., and C. Winnerlein, "BLAKE2: simpler, smaller, fast as MD5", January 2013, <https://blake2.net/blake2.pdf>.

[FIPS140-2IG] NIST, "Implementation Guidance for FIPS PUB 140-2 and the Cryptographic Module Validation Program", September 2015, <http://csrc.nist.gov/groups/STM/cmvp/documents/fips140-2/FIPS1402IG.pdf/>.

[RFC6151] Turner, S. and L. Chen, "Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms", RFC 6151, DOI 10.17487/RFC6151, March 2011, <http://www.rfc-editor.org/info/rfc6151>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <http://www.rfc-editor.org/info/rfc6234>.

Приложение A. Пример расчета BLAKE2b

Вычисляется хэш без ключа для 3 байтов ASCII abc с использованием BLAKE2b-512 и показаны детали расчета.

```
m[16] = 0000000000636261 0000000000000000 0000000000000000
        0000000000000000 0000000000000000 0000000000000000
        0000000000000000 0000000000000000 0000000000000000
        0000000000000000 0000000000000000 0000000000000000
        0000000000000000 0000000000000000 0000000000000000
```

```
                0000000000000000

(i= 0) v[16] = 6A09E667F2BDC948 BB67AE8584CAA73B 3C6EF372FE94F82B
                A54FF53A5F1D36F1 510E527FADE682D1 9B05688C2B3E6C1F
                1F83D9ABFB41BD6B 5BE0CD19137E2179 6A09E667F3BCC908
                BB67AE8584CAA73B 3C6EF372FE94F82B A54FF53A5F1D36F1
                510E527FADE682D2 9B05688C2B3E6C1F E07C265404BE4294
                5BE0CD19137E2179

(i= 1) v[16] = 86B7C1568029BB79 C12CBCC809FF59F3 C6A5214CC0EACA8E
                0C87CD524C14CC5D 44EE6039BD86A9F7 A447C850AA694A7E
                DE080F1BB1C0F84B 595CB8A9A1ACA66C BEC3AE837EAC4887
                6267FC79DF9D6AD1 FA87B01273FA6DBE 521A715C63E08D8A
                E02D0975B8D37A83 1C7B754F08B7D193 8F885A76B6E578FE
                2318A24E2140FC64

(i= 2) v[16] = 53281E83806010F2 3594B403F81B4393 8CD63C7462DE0DFF
                85F693F3DA53F974 BAABDBB2F386D9AE CA5425AEC65A10A8
                C6A22E2FF0F7AA48 C6A56A51CB89C595 224E6A3369224F96
                500E125E58A92923 E9E4AD0D0E1A0D48 85DF9DC143C59A74
                92A3AAAA6D952B7F C5FDF71090FAE853 2A8A40F15A462DD0
                572D17EFFDD37358

(i= 3) v[16] = 60ED96AA7AD41725 E46A743C71800B9D 1A04B543A01F156B
                A2F8716E775C4877 DA0A61BCDE4267EA B1DD230754D7BDEE
                25A1422779E06D14 E6823AE4C3FF58A5 A1677E19F37FD5DA
                22BDCE6976B08C51 F1DE8696BEC11BF1 A0EBD586A4A1D2C8
                C804EBAB11C99FA9 8E0CEC959C715793 7C45557FAE0D4D89
                716343F52FDD265E

(i= 4) v[16] = BB2A77D3A8382351 45EB47971F23B103 98BE297F6E45C684
                A36077DEE3370B89 8A03C4CB7E97590A 24192E49EBF54EA0
                4F82C9401CB32D7A 8CCD013726420DC4 A9C9A8F17B1FC614
                559081879775 14A0 5B44273E66B19D27 B6D5C9FCA2579327
                086092CFB858437E 5C4BE2156DBEECF9 2EFEDE99ED4EFF16
                3E7B5F234CD1F804

(i= 5) v[16] = C79C15B3D423B099 2DA2224E8DA97556 77D2B26DF1C45C55
                8934EB09A3456052 0F6D9EEED157DA2A 6FE66467AF88C0A9
                4EB0B76284C7AAFB 299C8E725D954697 B2240B59E6D567D3
                2643C2370E49EBFD 79E02EEF20CDB1AE 64B3EED7BB602F39
                B97D2D439E4DF63D C718E755294C9111 1F0893F2772BB373
                1205EA4A7859807D

(i= 6) v[16] = E58F97D6385BAEE4 7640AA9764DA137A DEB4C7C23EFE287E
                70F6F41C8783C9F6 7127CD48C76A7708 9E472AF0BE3DB3F6
                0F244C62DDF71788 219828AA83880842 41CCA9073C8C4D0D
                5C7912BC10DF3B4B A2C3ABBD37510EE2 CB5668CC2A9F7859
                8733794F07AC1500 C67A6BE42335AA6F ACB22B28681E4C82
                DB2161604CBC9828

(i= 7) v[16] = 6E2D286EEADEDC81 BCF02C0787E86358 57D56A56DD015EDF
                55D899D40A5D0D0A 819415B56220C459 B63C479A6A769F02
                258E55E0EC1F362A 3A3B4EC60E19DFDC 04D769B3FCB048DB
                B78A9A33E9BFF4DD 5777272AE1E930C0 5A387849E578DBF6
                92AAC307CF2C0AFC 30AACCC4F06DAFAA 483893CC094F8863
                E03C6CC89C26BF92

(i= 8) v[16] = FFC83ECE76024D01 1BE7BFFB8C5CC5F9 A35A18CBAC4C65B7
                B7C2C7E6D88C285F 81937DA314A50838 E1179523A2541963
                3A1FAD7106232B8F 1C7EDE92AB8B9C46 A3C2D35E4F685C10
                A53D3F73AA619624 30BBCC0285A22F65 BCEFBB6A81539E5D
                3841DEF6F4C9848A 98662C85FBA726D4 7762439BD5A851BD
                B0B9F0D443D1A889

(i= 9) v[16] = 753A70A1E8FAEADD 6B0D43CA2C25D629 F8343BA8B94F8C0B
                BC7D062B0DB5CF35 58540EE1B1AEBC47 63C5B9B80D294CB9
                490870ECAD27DEBD B2A90DDF667287FE 316CC9EBEEFAD8FC
                4A466BCD021526A4 5DA7F7638CEC5669 D9C8826727D306FC
                88ED6C4F3BD7A537 19AE688DDF67F026 4D8707AAB40F7E6D
                FD3F572687FEA4F1

(i=10) v[16] = E630C747CCD59C4F BC713D41127571CA 46DB183025025078
                6727E81260610140 2D04185EAC2A8CBA 5F311B88904056EC
                40BD313009201AAB 0099D4F82A2A1EAB 6DD4FBC1DE60165D
                B3B0B51DE3C86270 900AEE2F233B08E5 A07199D87AD058D8
                2C6B25593D717852 37E8CA471BEAA5F8 2CFC1BAC10EF4457
                01369EC18746E775

(i=11) v[16] = E801F73B9768C760 35C6D22320BE511D 306F27584F65495E
                B51776ADF569A77B F4F1BE86690B3C34 3CC88735D1475E4B
                5DAC67921FF76949 1CDB9D31AD70CC4E 35BA354A9C7DF448
                4929CBE45679D73E 733D1A17248F39DB 92D57B736F5F170A
                61B5C0A41D491399 B5C333457E12844A BD696BE010D0D889
                02231E1A917FE0BD
```

```
   (i=12) v[16] = 12EF8A641EC4F6D6 BCED5DE977C9FAF5 733CA476C5148639
                  97DF596B0610F6FC F42C16519AD5AFA7 AA5AC1888E10467E
                  217D930AA51787F3 906A6FF19E573942 75AB709BD3DCBF24
                  EE7CE1F345947AA4 F8960D6C2FAF5F5E E332538A36B6D246
                  885BEF040EF6AA0B A4939A417BFB78A3 646CBB7AF6DCE980
                  E813A23C60AF3B82

           h[8] = 0D4D1C983FA580BA E9F6129FB697276A B7C45A68142F214C
                  D1A2FFDB6FBB124B 2D79AB2A39C5877D 95CC3345DED552C2
                  5A92F1DBA88AD318 239900D4ED8623B9

   BLAKE2b-512("abc") = BA 80 A5 3F 98 1C 4D 0D 6A 27 97 B6 9F 12 F6 E9
                        4C 21 2F 14 68 5A C4 B7 4B 12 BB 6F DB FF A2 D1
                        7D 87 C5 39 2A AB 79 2D C2 52 D5 DE 45 33 CC 95
                        18 D3 8A A8 DB F1 92 5A B9 23 86 ED D4 00 99 23
```

Приложение B. Пример расчёта BLAKE2s

Вычисляется хэш без ключа для 3 байтов ASCII abc с использованием BLAKE2b-256 и показаны детали расчета.

```
           m[16] = 00636261 00000000 00000000 00000000 00000000 00000000
                   00000000 00000000 00000000 00000000 00000000 00000000
                   00000000 00000000 00000000 00000000

   (i=0)   v[16] = 6B08E647 BB67AE85 3C6EF372 A54FF53A 510E527F 9B05688C
                   1F83D9AB 5BE0CD19 6A09E667 BB67AE85 3C6EF372 A54FF53A
                   510E527C 9B05688C E07C2654 5BE0CD19

   (i=1)   v[16] = 16A3242E D7B5E238 CE8CE24B 927AEDE1 A7B430D9 93A4A14E
                   A44E7C31 41D4759B 95BF33D3 9A99C181 608A3A6B B666383E
                   7A8DD50F BE378ED7 353D1EE6 3BB44C6B

   (i=2)   v[16] = 3AE30FE3 0982A96B E88185B4 3E339B16 F24338CD 0E66D326
                   E005ED0C D591A277 180B1F3A FCF43914 30DB62D6 4847831C
                   7F00C58E FB847886 C544E836 524AB0E2

   (i=3)   v[16] = 7A3BE783 997546C1 D45246DF EDB5F821 7F98A742 10E864E2
                   D4AB70D0 C63CB1AB 6038DA9E 414594B0 F2C218B5 8DA0DCB7
                   D7CD7AF5 AB4909DF 85031A52 C4EDFC98

   (i=4)   v[16] = 2A8B8CB7 1ACA82B2 14045D7F CC7258ED 383CF67C E090E7F9
                   3025D276 57D04DE4 994BACF0 F0982759 F17EE300 D48FC2D5
                   DC854C10 523898A9 C03A0F89 47D6CD88

   (i=5)   v[16] = C4AA2DDB 111343A3 D54A700A 574A00A9 857D5A48 B1E11989
                   6F5C52DF DD2C53A3 678E5F8E 9718D4E9 622CB684 92976076
                   0E41A517 359DC2BE 87A87DDD 643F9CEC

   (i=6)   v[16] = 3453921C D7595EE1 592E776D 3ED6A974 4D997CB3 DE9212C3
                   35ADF5C9 9916FD65 96562E89 4EAD0792 EBFC2712 2385F5B2
                   F34600FB D7BC20FB EB452A7B ECE1AA40

   (i=7)   v[16] = BE851B2D A85F6358 81E6FC3B 0BB28000 FA55A33A 87BE1FAD
                   4119370F 1E2261AA A1318FD3 F4329816 071783C2 6E536A8D
                   9A81A601 E7EC80F1 ACC09948 F849A584

   (i=8)   v[16] = 07E5B85A 069CC164 F9DE3141 A56F4680 9E440AD2 9AB659EA
                   3C84B971 21DBD9CF 46699F8C 765257EC AF1D998C 75E4C3B6
                   523878DC 30715015 397FEE81 4F1FA799

   (i=9)   v[16] = 435148C4 A5AA2D11 4B354173 D543BC9E BDA2591C BF1D2569
                   4FCB3120 707ADA48 565B3FDE 32C9C916 EAF4A1AB B1018F28
                   8078D978 68ADE4B5 9778FDA3 2863B92E

   (i=10) v[16] = D9C994AA CFEC3AA6 700D0AB2 2C38670E AF6A1F66 1D023EF3
                   1D9EC27D 945357A5 3E9FFEBD 969FE811 EF485E21 A632797A
                   DEEF082E AF3D80E1 4E86829B 4DEAFD3A

           h[8] = 8C5E8C50 E2147C32 A32BA7E1 2F45EB4E 208B4537 293AD69E
                   4C9B994D 82596786

   BLAKE2s-256("abc") = 50 8C 5E 8C 32 7C 14 E2 E1 A7 2B A3 4E EB 45 2F
                        37 45 8B 20 9E D6 3A 29 4D 99 9B 4C 86 67 59 82
```

Приложение C. Код реализации BLAKE2b на языке C

C.1. blake2b.h

```
   <CODE BEGINS>
   // blake2b.h
   // Контекст хэширования и прототипы API для BLAKE2b

   #ifndef BLAKE2B_H
   #define BLAKE2B_H

   #include <stdint.h>
   #include <stddef.h>
```

```
// контекст состояния
typedef struct {
    uint8_t b[128];                  // входной буфер
    uint64_t h[8];                   // измененное состояние
    uint64_t t[2];                   // общее число байтов
    size_t c;                        // указатель для b[]
    size_t outlen;                   // размер дайджеста
} blake2b_ctx;

// Инициализация контекста хэширования ctx с необязательным ключом key.
//      1 <= outlen <= 64 даёт размер дайджеста а байтах.
//      секретный ключ (<= 64 байтов) является необязательным (keylen = 0).
int blake2b_init(blake2b_ctx *ctx, size_t outlen,
    const void *key, size_t keylen);    // секретный ключ

// Добавление в хэш inlen байтов из in.
void blake2b_update(blake2b_ctx *ctx,   // контекст
    const void *in, size_t inlen);      // данные для хэширования

// Генерация дайджеста сообщения (размер задан в init).
//      Результат помещается в out.
void blake2b_final(blake2b_ctx *ctx, void *out);

// Удобная функция "все в одном".
int blake2b(void *out, size_t outlen,   // буфер для возврата дайджеста
    const void *key, size_t keylen,     // необязательный секретный ключ
    const void *in, size_t inlen);      // данные для хэширования

#endif
<CODE ENDS>
```

C.2. blake2b.c

```
<CODE BEGINS>
// blake2b.c
// Простая справочная реализация BLAKE2b.

#include "blake2b.h"

// Циклический сдвиг вправо.

#ifndef ROTR64
#define ROTR64(x, y)  (((x) >> (y)) ^ ((x) << (64 - (y))))
#endif

// Доступ к байтам Little-endian.

#define B2B_GET64(p)                            \
    (((uint64_t) ((uint8_t *) (p))[0]) ^        \
    (((uint64_t) ((uint8_t *) (p))[1]) << 8) ^  \
    (((uint64_t) ((uint8_t *) (p))[2]) << 16) ^ \
    (((uint64_t) ((uint8_t *) (p))[3]) << 24) ^ \
    (((uint64_t) ((uint8_t *) (p))[4]) << 32) ^ \
    (((uint64_t) ((uint8_t *) (p))[5]) << 40) ^ \
    (((uint64_t) ((uint8_t *) (p))[6]) << 48) ^ \
    (((uint64_t) ((uint8_t *) (p))[7]) << 56))

// Функция смешивания G.

#define B2B_G(a, b, c, d, x, y) {   \
    v[a] = v[a] + v[b] + x;         \
    v[d] = ROTR64(v[d] ^ v[a], 32); \
    v[c] = v[c] + v[d];             \
    v[b] = ROTR64(v[b] ^ v[c], 24); \
    v[a] = v[a] + v[b] + y;         \
    v[d] = ROTR64(v[d] ^ v[a], 16); \
    v[c] = v[c] + v[d];             \
    v[b] = ROTR64(v[b] ^ v[c], 63); }

// Вектор инициализации.

static const uint64_t blake2b_iv[8] = {
    0x6A09E667F3BCC908, 0xBB67AE8584CAA73B,
    0x3C6EF372FE94F82B, 0xA54FF53A5F1D36F1,
    0x510E527FADE682D1, 0x9B05688C2B3E6C1F,
    0x1F83D9ABFB41BD6B, 0x5BE0CD19137E2179
};

// Функция сжатия. Флаг last указывает последний блок.
static void blake2b_compress(blake2b_ctx *ctx, int last)
{
    const uint8_t sigma[12][16] = {
        { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 },
        { 14, 10, 4, 8, 9, 15, 13, 6, 1, 12, 0, 2, 11, 7, 5, 3 },
        { 11, 8, 12, 0, 5, 2, 15, 13, 10, 14, 3, 6, 7, 1, 9, 4 },
```

```
        { 7, 9, 3, 1, 13, 12, 11, 14, 2, 6, 5, 10, 4, 0, 15, 8 },
        { 9, 0, 5, 7, 2, 4, 10, 15, 14, 1, 11, 12, 6, 8, 3, 13 },
        { 2, 12, 6, 10, 0, 11, 8, 3, 4, 13, 7, 5, 15, 14, 1, 9 },
        { 12, 5, 1, 15, 14, 13, 4, 10, 0, 7, 6, 3, 9, 2, 8, 11 },
        { 13, 11, 7, 14, 12, 1, 3, 9, 5, 0, 15, 4, 8, 6, 2, 10 },
        { 6, 15, 14, 9, 11, 3, 0, 8, 12, 2, 13, 7, 1, 4, 10, 5 },
        { 10, 2, 8, 4, 7, 6, 1, 5, 15, 11, 9, 14, 3, 12, 13, 0 },
        { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 },
        { 14, 10, 4, 8, 9, 15, 13, 6, 1, 12, 0, 2, 11, 7, 5, 3 }
    };
    int i;
    uint64_t v[16], m[16];

    for (i = 0; i < 8; i++) {           // инициализация рабочих переменных
        v[i] = ctx->h[i];
        v[i + 8] = blake2b_iv[i];
    }

    v[12] ^= ctx->t[0];                 // младшие 64 бита смещения
    v[13] ^= ctx->t[1];                 // старшие 64 бита смещения
    if (last)                           // флаг последнего блока установлен?
        v[14] = ~v[14];

    for (i = 0; i < 16; i++)            // получение слов little-endian
        m[i] = B2B_GET64(&ctx->b[8 * i]);

    for (i = 0; i < 12; i++) {          // 12 раундов
        B2B_G( 0, 4,  8, 12, m[sigma[i][ 0]], m[sigma[i][ 1]]);
        B2B_G( 1, 5,  9, 13, m[sigma[i][ 2]], m[sigma[i][ 3]]);
        B2B_G( 2, 6, 10, 14, m[sigma[i][ 4]], m[sigma[i][ 5]]);
        B2B_G( 3, 7, 11, 15, m[sigma[i][ 6]], m[sigma[i][ 7]]);
        B2B_G( 0, 5, 10, 15, m[sigma[i][ 8]], m[sigma[i][ 9]]);
        B2B_G( 1, 6, 11, 12, m[sigma[i][10]], m[sigma[i][11]]);
        B2B_G( 2, 7,  8, 13, m[sigma[i][12]], m[sigma[i][13]]);
        B2B_G( 3, 4,  9, 14, m[sigma[i][14]], m[sigma[i][15]]);
    }

    for( i = 0; i < 8; ++i )
        ctx->h[i] ^= v[i] ^ v[i + 8];
}

// Инициализация контекста хэширования ctx с необязательным ключом key.
//      1 <= outlen <= 64 даёт размер дайджеста а байтах.
//      секретный ключ (<= 64 байтов) является необязательным (keylen = 0).

int blake2b_init(blake2b_ctx *ctx, size_t outlen,
    const void *key, size_t keylen)         // (keylen=0 - нет ключа)
{
    size_t i;

    if (outlen == 0 || outlen > 64 || keylen > 64)
        return -1;                      // непригодные параметры

    for (i = 0; i < 8; i++)             // состояние, блок параметров
        ctx->h[i] = blake2b_iv[i];
    ctx->h[0] ^= 0x01010000 ^ (keylen << 8) ^ outlen;

    ctx->t[0] = 0;                      // младшее слово счетчика ввода
    ctx->t[1] = 0;                      // старшее слово счетчика ввода
    ctx->c = 0;                         // указатель внутри буфера
    ctx->outlen = outlen;

    for (i = keylen; i < 128; i++)      // нулевой входной блок
        ctx->b[i] = 0;
    if (keylen > 0) {
        blake2b_update(ctx, key, keylen);
        ctx->c = 128;                   // в конце
    }

    return 0;
}

// Добавление в хэш inlen байтов из in.

void blake2b_update(blake2b_ctx *ctx,
    const void *in, size_t inlen)       // байты данных
{
    size_t i;

    for (i = 0; i < inlen; i++) {
        if (ctx->c == 128) {            // буфер полон?
            ctx->t[0] += ctx->c;        // увеличение счетчика
            if (ctx->t[0] < ctx->c)     // перенос при переполнении?
                ctx->t[1]++;            // старшее слово
            blake2b_compress(ctx, 0);   // сжатие (не последнее)
            ctx->c = 0;                 // сброс счётчика в 0
```

```
        }
        ctx->b[ctx->c++] = ((const uint8_t *) in)[i];
    }
}

// Генерация дайджеста сообщения (размер задан в init).
//      Результат помещается в out.

void blake2b_final(blake2b_ctx *ctx, void *out)
{
    size_t i;

    ctx->t[0] += ctx->c;                // маркировка смещения последнего блока
    if (ctx->t[0] < ctx->c)             // перенос при переполнении
        ctx->t[1]++;                    // старшее слово

    while (ctx->c < 128)                // заполнение нулями
        ctx->b[ctx->c++] = 0;
    blake2b_compress(ctx, 1);           // флаг финального блока = 1

    // преобразование в little-endian и сохранение
    for (i = 0; i < ctx->outlen; i++) {
        ((uint8_t *) out)[i] =
            (ctx->h[i >> 3] >> (8 * (i & 7))) & 0xFF;
    }
}

// Удобная функция "все в одном".

int blake2b(void *out, size_t outlen,
    const void *key, size_t keylen,
    const void *in, size_t inlen)
{
    blake2b_ctx ctx;

    if (blake2b_init(&ctx, outlen, key, keylen))
        return -1;
    blake2b_update(&ctx, in, inlen);
    blake2b_final(&ctx, out);

    return 0;
}
<CODE ENDS>
```

Приложение D. Код реализации BLAKE2s на языке C

D.1. blake2s.h

```
<CODE BEGINS>
// blake2s.h
// Контекст хэширования и прототипы API для  BLAKE2s

#ifndef BLAKE2S_H
#define BLAKE2S_H

#include <stdint.h>
#include <stddef.h>

// контекст состояния
typedef struct {
    uint8_t b[64];                      // входной буфер
    uint32_t h[8];                      // измененное состояние
    uint32_t t[2];                      // общее число байтов
    size_t c;                           // указатель для b[]
    size_t outlen;                      // размер дайджеста
} blake2s_ctx;

// Initialize the hashing контекст "ctx" with optional key "key".
//      1 <= outlen <= 32 даёт размер дайджеста a байтах.
//      секретный ключ (<= 32 байтов) является необязательным (keylen = 0).
int blake2s_init(blake2s_ctx *ctx, size_t outlen,
    const void *key, size_t keylen);    // секретный ключ

// Добавление в хэш inlen байтов из in.
void blake2s_update(blake2s_ctx *ctx,   // контекст
    const void *in, size_t inlen);      // данные для хэширования

// Генерация дайджеста сообщения (размер задан в init).
//      Результат помещается в out.
void blake2s_final(blake2s_ctx *ctx, void *out);

// Удобная функция "все в одном".
int blake2s(void *out, size_t outlen,   // буфер для возврата дайджеста
    const void *key, size_t keylen,     // необязательный секретный ключ
    const void *in, size_t inlen);      // данные для хэширования
```

```
#endif
<CODE ENDS>
```

D.2. blake2s.c

```
<CODE BEGINS>
// blake2s.c
// Простая справочная реализация blake2s.

#include "blake2s.h"

// Циклический сдвиг вправо.

#ifndef ROTR32
#define ROTR32(x, y)  (((x) >> (y)) ^ ((x) << (32 - (y))))
#endif

// Доступ к байтам Little-endian.

#define B2S_GET32(p)                            \
    (((uint32_t) ((uint8_t *) (p))[0]) ^        \
    (((uint32_t) ((uint8_t *) (p))[1]) << 8) ^  \
    (((uint32_t) ((uint8_t *) (p))[2]) << 16) ^ \
    (((uint32_t) ((uint8_t *) (p))[3]) << 24))

// Функция смешивания G.

#define B2S_G(a, b, c, d, x, y) {   \
    v[a] = v[a] + v[b] + x;         \
    v[d] = ROTR32(v[d] ^ v[a], 16); \
    v[c] = v[c] + v[d];             \
    v[b] = ROTR32(v[b] ^ v[c], 12); \
    v[a] = v[a] + v[b] + y;         \
    v[d] = ROTR32(v[d] ^ v[a], 8);  \
    v[c] = v[c] + v[d];             \
    v[b] = ROTR32(v[b] ^ v[c], 7); }

// Вектор инициализации.

static const uint32_t blake2s_iv[8] =
{
    0x6A09E667, 0xBB67AE85, 0x3C6EF372, 0xA54FF53A,
    0x510E527F, 0x9B05688C, 0x1F83D9AB, 0x5BE0CD19
};

// Функция сжатия. Флаг last указывает последний блок.
static void blake2s_compress(blake2s_ctx *ctx, int last)
{
    const uint8_t sigma[10][16] = {
        { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 },
        { 14, 10, 4, 8, 9, 15, 13, 6, 1, 12, 0, 2, 11, 7, 5, 3 },
        { 11, 8, 12, 0, 5, 2, 15, 13, 10, 14, 3, 6, 7, 1, 9, 4 },
        { 7, 9, 3, 1, 13, 12, 11, 14, 2, 6, 5, 10, 4, 0, 15, 8 },
        { 9, 0, 5, 7, 2, 4, 10, 15, 14, 1, 11, 12, 6, 8, 3, 13 },
        { 2, 12, 6, 10, 0, 11, 8, 3, 4, 13, 7, 5, 15, 14, 1, 9 },
        { 12, 5, 1, 15, 14, 13, 4, 10, 0, 7, 6, 3, 9, 2, 8, 11 },
        { 13, 11, 7, 14, 12, 1, 3, 9, 5, 0, 15, 4, 8, 6, 2, 10 },
        { 6, 15, 14, 9, 11, 3, 0, 8, 12, 2, 13, 7, 1, 4, 10, 5 },
        { 10, 2, 8, 4, 7, 6, 1, 5, 15, 11, 9, 14, 3, 12, 13, 0 }
    };
    int i;
    uint32_t v[16], m[16];

    for (i = 0; i < 8; i++) {           // инициализация рабочих переменных
        v[i] = ctx->h[i];
        v[i + 8] = blake2s_iv[i];
    }

    v[12] ^= ctx->t[0];                 // младшие 32 бита смещения
    v[13] ^= ctx->t[1];                 // старшие 32 бита смещения
    if (last)                           // флаг последнего блока установлен?
        v[14] = ~v[14];

    for (i = 0; i < 16; i++)            // получение слов little-endian
        m[i] = B2S_GET32(&ctx->b[4 * i]);

    for (i = 0; i < 10; i++) {          // 10 раундов
        B2S_G( 0, 4,  8, 12, m[sigma[i][ 0]], m[sigma[i][ 1]]);
        B2S_G( 1, 5,  9, 13, m[sigma[i][ 2]], m[sigma[i][ 3]]);
        B2S_G( 2, 6, 10, 14, m[sigma[i][ 4]], m[sigma[i][ 5]]);
        B2S_G( 3, 7, 11, 15, m[sigma[i][ 6]], m[sigma[i][ 7]]);
        B2S_G( 0, 5, 10, 15, m[sigma[i][ 8]], m[sigma[i][ 9]]);
        B2S_G( 1, 6, 11, 12, m[sigma[i][10]], m[sigma[i][11]]);
        B2S_G( 2, 7,  8, 13, m[sigma[i][12]], m[sigma[i][13]]);
        B2S_G( 3, 4,  9, 14, m[sigma[i][14]], m[sigma[i][15]]);
    }
```

```
    for( i = 0; i < 8; ++i )
        ctx->h[i] ^= v[i] ^ v[i + 8];
}

// Инициализация контекста хэширования ctx с необязательным ключом key.
//      1 <= outlen <= 32 даёт размер дайджеста в байтах.
//      секретный ключ (<= 32 байтов) необязателен (keylen = 0).

int blake2s_init(blake2s_ctx *ctx, size_t outlen,
    const void *key, size_t keylen)     // (keylen=0 - нет ключа)
{
    size_t i;

    if (outlen == 0 || outlen > 32 || keylen > 32)
        return -1;                      // непригодные параметры

    for (i = 0; i < 8; i++)             // состояние, блок параметров
        ctx->h[i] = blake2s_iv[i];
    ctx->h[0] ^= 0x01010000 ^ (keylen << 8) ^ outlen;

    ctx->t[0] = 0;                      // младшее слово счетчика ввода
    ctx->t[1] = 0;                      // старшее слово счетчика ввода
    ctx->c = 0;                         // указатель внутри буфера
    ctx->outlen = outlen;

    for (i = keylen; i < 64; i++)       // нулевой входной блок
        ctx->b[i] = 0;
    if (keylen > 0) {
        blake2s_update(ctx, key, keylen);
        ctx->c = 64;                    // в конце
    }

    return 0;
}

// Добавление в хэш inlen байтов из in.

void blake2s_update(blake2s_ctx *ctx,
    const void *in, size_t inlen)       // байты данных
{
    size_t i;

    for (i = 0; i < inlen; i++) {
        if (ctx->c == 64) {             // буфер полон?
            ctx->t[0] += ctx->c;        // увеличение счетчика
            if (ctx->t[0] < ctx->c)     // перенос при переполнении?
                ctx->t[1]++;            // старшее слово
            blake2s_compress(ctx, 0);   // сжатие (не последнее)
            ctx->c = 0;                 // сброс счётчика в 0
        }
        ctx->b[ctx->c++] = ((const uint8_t *) in)[i];
    }
}

// Генерация дайджеста сообщения (размер задан в init).
//      Результат помещается в out.

void blake2s_final(blake2s_ctx *ctx, void *out)
{
    size_t i;

    ctx->t[0] += ctx->c;                // маркировка смещения последнего блока
    if (ctx->t[0] < ctx->c)             // перенос при переполнении
        ctx->t[1]++;                    // старшее слово

    while (ctx->c < 64)                 // заполнение нулями
        ctx->b[ctx->c++] = 0;
    blake2s_compress(ctx, 1);           // флаг финального блока = 1

    // преобразование в little-endian и сохранение
    for (i = 0; i < ctx->outlen; i++) {
        ((uint8_t *) out)[i] =
            (ctx->h[i >> 2] >> (8 * (i & 3))) & 0xFF;
    }
}

// Удобная функция "все в одном".

int blake2s(void *out, size_t outlen,
    const void *key, size_t keylen,
    const void *in, size_t inlen)
{
    blake2s_ctx ctx;
    if (blake2s_init(&ctx, outlen, key, keylen))
        return -1;
```

```
    blake2s_update(&ctx, in, inlen);
    blake2s_final(&ctx, out);

    return 0;
}
<CODE ENDS>
```

Приложение E. Код модуля самотестирования BLAKE2b и BLAKE2s

Этот модуль рассчитывает серию хэшей с ключом и без ключа по детерминированно сгенерированным псевдослучайным данным, а также хэш для этих результатов. Это обеспечивает исчерпывающий, компактный и быстрый метод проверки корректности функционирования модуля хэширования.

Такое тестирование **рекомендуется**, особенно при реализации для новой конфигурации целевой платформы. Кроме того, некоторые стандарты безопасности (например, FIPS-140) могут требовать самотестирования при включении (Power-On Self Test или POST) каждый раз, когда загружается криптографический модуль [FIPS140-2IG].

```
<CODE BEGINS>
// test_main.c
// Модули самотестирования для BLAKE2b и BLAKE2s, а также заглушка main().

#include <stdio.h>

#include "blake2b.h"
#include "blake2s.h"

// Детерминированные последовательности (генератор Fibonacci).

static void selftest_seq(uint8_t *out, size_t len, uint32_t seed)
{
    size_t i;
    uint32_t t, a , b;

    a = 0xDEAD4BAD * seed;              // prime
    b = 1;

    for (i = 0; i < len; i++) {         // заполнение буфера
        t = a + b;
        a = b;
        b = t;
        out[i] = (t >> 24) & 0xFF;
    }
}

// Проверка BLAKE2b. При положительном результате возвращает 0.

int blake2b_selftest()
{
    // Хэш результатов хэширования
    const uint8_t blake2b_res[32] = {
        0xC2, 0x3A, 0x78, 0x00, 0xD9, 0x81, 0x23, 0xBD,
        0x10, 0xF5, 0x06, 0xC6, 0x1E, 0x29, 0xDA, 0x56,
        0x03, 0xD7, 0x63, 0xB8, 0xBB, 0xAD, 0x2E, 0x73,
        0x7F, 0x5E, 0x76, 0x5A, 0x7B, 0xCC, 0xD4, 0x75
    };
    // наборы параметров
    const size_t b2b_md_len[4] = { 20, 32, 48, 64 };
    const size_t b2b_in_len[6] = { 0, 3, 128, 129, 255, 1024 };

    size_t i, j, outlen, inlen;
    uint8_t in[1024], md[64], key[64];
    blake2b_ctx ctx;

    // 256-битовый хэш для тестирования
    if (blake2b_init(&ctx, 32, NULL, 0))
        return -1;

    for (i = 0; i < 4; i++) {
        outlen = b2b_md_len[i];
        for (j = 0; j < 6; j++) {
            inlen = b2b_in_len[j];

            selftest_seq(in, inlen, inlen);     // хэш без ключа
            blake2b(md, outlen, NULL, 0, in, inlen);
            blake2b_update(&ctx, md, outlen);   // хэш для хэша

            selftest_seq(key, outlen, outlen);  // хэш с ключом
            blake2b(md, outlen, key, outlen, in, inlen);
            blake2b_update(&ctx, md, outlen);   // хэш для хэша
        }
    }

    // расчет и сравнение хэша хэшей
    blake2b_final(&ctx, md);
    for (i = 0; i < 32; i++) {
        if (md[i] != blake2b_res[i])
```

```
            return -1;
        }

        return 0;
    }

    // Проверка BLAKE2s. При положительном результате возвращает 0.

    int blake2s_selftest()
    {
        // Хэш результатов хэширования
        const uint8_t blake2s_res[32] = {
            0x6A, 0x41, 0x1F, 0x08, 0xCE, 0x25, 0xAD, 0xCD,
            0xFB, 0x02, 0xAB, 0xA6, 0x41, 0x45, 0x1C, 0xEC,
            0x53, 0xC5, 0x98, 0xB2, 0x4F, 0x4F, 0xC7, 0x87,
            0xFB, 0xDC, 0x88, 0x79, 0x7F, 0x4C, 0x1D, 0xFE
        };
        // наборы параметров.
        const size_t b2s_md_len[4] = { 16, 20, 28, 32 };
        const size_t b2s_in_len[6] = { 0,  3,  64, 65, 255, 1024 };

        size_t i, j, outlen, inlen;
        uint8_t in[1024], md[32], key[32];
        blake2s_ctx ctx;

        // 256-битовый хэш для тестирования.
        if (blake2s_init(&ctx, 32, NULL, 0))
            return -1;

        for (i = 0; i < 4; i++) {
            outlen = b2s_md_len[i];
            for (j = 0; j < 6; j++) {
                inlen = b2s_in_len[j];

                selftest_seq(in, inlen, inlen);     // хэш без ключа
                blake2s(md, outlen, NULL, 0, in, inlen);
                blake2s_update(&ctx, md, outlen);   // хэш для хэша

                selftest_seq(key, outlen, outlen);  // хэш с ключом
                blake2s(md, outlen, key, outlen, in, inlen);
                blake2s_update(&ctx, md, outlen);   // хэш для хэша
            }
        }

        // расчет и сравнение хэша хэшей.
        blake2s_final(&ctx, md);
        for (i = 0; i < 32; i++) {
            if (md[i] != blake2s_res[i])
                return -1;
        }

        return 0;
    }

    // Драйвер теста.

    int main(int argc, char **argv)
    {
        printf("blake2b_selftest() = %s\n",
             blake2b_selftest() ? "FAIL" : "OK");
        printf("blake2s_selftest() = %s\n",
             blake2s_selftest() ? "FAIL" : "OK");

        return 0;
    }
    <CODE ENDS>
```

Благодарности

Редактор благодарен за поддержку команде [BLAKE2] в составе Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, Christian Winnerlein. Фрагменты [BLAKE] и [BLAKE2] заимствованы с разрешения.

[BLAKE2] базируется на предложении SHA-3 [BLAKE] от Jean-Philippe Aumasson, Luca Henzen, Willi Meier, Raphael C.-W. Phan. BLAKE2, как и BLAKE, опирается на базовый алгоритм, заимствованный из потокового шифра ChaCha, разработанного Daniel J. Bernstein.

Адреса авторов

Markku-Juhani O. Saarinen (editor)
Queen's University Belfast
Centre for Secure Information Technologies, ECIT
Northern Ireland Science Park
Queen's Road, Queen's Island
Belfast BT3 9DT
United Kingdom

Email: m.saarinen@qub.ac.uk
URI: http://www.csit.qub.ac.uk

Jean-Philippe Aumasson
Kudelski Security
22-24, Route de Geneve
Case Postale 134
Cheseaux 1033
Switzerland
Email: jean-philippe.aumasson@nagra.com
URI: https://www.kudelskisecurity.com

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru