Parallelized hashing via j-lanes and j-pointers tree modes, with applications to SHA-256

Shay Gueron[1,2]

[1] Department of Mathematics, University of Haifa, Israel
[2] Intel Corporation, Israel Development Center, Haifa, Israel

March 4, 2014

Abstract. The *j*-lanes tree hashing is a tree mode that splits an input message to *j* slices, computes *j* independent digests of each slice, and outputs the hash value of their concatenation. The *j*-pointers tree hashing is a similar tree mode that receives, as input, *j* pointers to *j* messages (or slices of a single message), computes their digests and outputs the hash value of their concatenation. Such modes have parallelization capabilities on a hashing process that is serial by nature. As a result, they have performance advantage on modern processor architectures. This paper provides precise specifications for these hashing modes, proposes a setup for appropriate IV's definition, and demonstrates their performance on the latest processors. Our hope is that it would be useful for standardization of these modes.

Keywords: Tree mode hashing, SHA-256, SIMD architecture, Advanced Vector Extensions architectures, AVX, AVX2.

1 Introduction

This paper expands the details on the *j*-lanes tree hashing mode which was proposed in [4]. It provides specifications, enhancements, and an updated performance analysis. The purpose is to suggest such modes for standardization. Although the specification is general, we focus on *j*-lanes tree hashing with SHA-256 [2] as the underlying hash function.

The *j*-lanes mode is a particular form of tree hashing, which is optimized for contemporary architectures of modern processors that have SIMD (Single Instruction Multiple Data) instructions. Currently deployed SIMD architectures use either *128*-bit (e.g., SSE, AVX [5], NEON [1]) or *256*-bit (AVX2 [5]) registers. For SHA-256, an algorithm that (by its definition) operates on *32*-bit words, AVX and AVX2 architectures can process *4* or *8* "lanes" in parallel, respectively. The *j*-lanes mode capitalizes on this parallelization capability.

The AVX2 architecture [5] includes all the necessary instructions to implement SHA-256 operations efficiently: *32*-bit shift (*vpsrld*) and add (*vpaddd*), bitwise logical operations (*vpandn, vpand, vpxor*), and the *32*-bit rotation (by combining two shifts (*vpsrld/vpslld*) with a single xor/or (*vpxor*) operation).

The future AVX512f instructions set [5, 6] supports *512*-bit registers, ready for operating on *16* lanes. It also adds a few useful instructions that would increase the parallelized hashing performance: rotation (*vprold*) and ternary-logic operation (*vpternlogd*). The (*vpternlogd*) instruction allows software to use a single instruction for implementing logical functions such as Majority and Choose, which SAH-256 (and other hash algorithms) use. Rotation (*vprold*) can perform the SHA-256 rotations faster than the *vpsrld+vpslld+vpxor* combination.

2 Preliminaries

Hereafter, we focus on hash functions (HASH) that use the Merkle-Damgård construction (SHA-256, SHA-512, SHA-1 are particular examples). Other constructions can be handled similarly. Suppose that HASH produces a digest of d bits, upon an input message M whose length is *length(M)*. The hashing process starts from an initial state, of size i bits, called an Initialization Vector (denoted *HashIV*). The message is first padded with a fixed string plus the encoded length of the message. The resulting (padded) message is then viewed and processed as the concatenation $M||padding = m_0||m_1||\ldots||m_{k-1}$ of k consecutive fixed size blocks $m_0\ m_1 \ldots m_{k-1}$.

The output digest is computed by an iterative invocation of a compression function *compress (H, BLOCK)*. The inputs to the compression function are a chaining variable (H) of i bits, and a block (*BLOCK*) of b bits. Its output is an i-bits value that can be fed as the input to the next iteration. The output digest (of HASH) is $f(H^{k-1})$. We call an invocation of the compression function an "Update" (because it updates the chaining variable).

We use here the following notations:

- $\lfloor x \rfloor$ - floor (x).
- $\lceil x \rceil$ - ceil (x) = floor ($x + 1$).
- *S[y:x]* – bits x through y of S.
- || - string concatenation (e.g., *04||08 = 0408*).
- HASH – the underlying hash function; HASH = HASH *(message, length (message))*.
- *HashIV* the Initialization Vector used for HASH (e.g., for SHA-256 *HashIV = 0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a, 0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19*; when written as *8* integers).
- *compress (H, BLOCK)* - the compression function used by HASH. It consumes a single fixed sized data chunk (*BLOCK*) of the message, a state (H), and updates H (at output) according to a specified algorithm ([2] defines the compression function for SHA-256).
- M – the hashed message.
- N – the length, in bits, of M.
- L – the length, in bytes , of M ($L = \lceil N/8 \rceil$).
- d – the length, in bits, of the digest that *HASH* produces.
- D – the length, in bytes, of the digest that *HASH* produces ($D = \lceil d/8 \rceil$).

- *B* – the length, in bytes, of the message block consumed by the compression function *compress* (e.g., for SHA-256, *B=64*).
- *j* – the number of lanes used by the *j*-lanes hashing process (in this paper, we discuss only *j= 4, 8, 16*).
- *Q* – the size, in bits, of the "word" that *HASH* uses during the computations (*Q=32* for SHA-256, and *Q= 64* for SHA-512).
- *W* – the size, in bytes, of the "word" that *HASH* uses during the computations (*W = Q/8*).
- *S* – the number of lanes that a given architecture supports, with respect to the word size of HASH (e.g., AVX architecture has registers (xmm's) that can hold *128* bits. For HASH = SHA-256, *Q = 32*, therefore, *S = 128/Q = 4*).
- *P* – the length, in bytes, of the minimal padding length of HASH (for SHA-256, a bit "*1*" is concatenated, and then the message bit length (*N*), encoded as an *8*-byte Big Endian integer. Therefore, with SHA-256, we have *P=9*).

3 The j-lanes tree hash

The *j*-lanes tree hash is defined in the context of the underlying hash function HASH, and *j* (*j* ≥ *2*) is a parameter. We are interested here in *j=4, 8, 16*. The input to the *j*-lanes hash function is a message *M* whose length is *N* bits.

This message is (logically) divided into *k (k* ≥ *0)* consecutive "words" m_i, *i=0, 1, …, k-1* (if *M* is the *NULL* message, then *k=0*) as follows:

```
k = ⌈N/Q⌉
i = 0
while i<k-1
  m_i = M [Q×i+Q-1: Q×i]
  i++
endwhile
if k≥1
  m_{k-1} = M [N-1: Q×k-1)]
```

When *k≥1*, the words m_j, *j=0, 1, …, k-2* (if *k-2 < 0*, there are no words in the count) consist of *Q* bits each. If *N* is not divisible by *Q*, then the last word m_{k-1} is incomplete, and consists of only (*N mod Q*) bits.

We then split the original message *M* into the *j* disjoint sub-messages (buffers) $Buff_0$, $Buff_1$, …, $Buff_{j-1}$ as follows:

$$Buff_0 = m_0||m_j||m_{j\times 2} \ldots$$
$$Buff_1 = m_1||m_{j+1}||m_{j\times 2+1} \ldots$$
$$\ldots$$
$$Buff_{j-1} = m_{j-1}||m_{j\times 2-1}||m_{j\times 3-1} \ldots$$

Note if $N \leq Q\times(j-1)$, then one or more buffers $Buff_i$ will be a *NULL* buffer. If $N=0$ all the buffers are defined to be *NULL*, and will be hashed as the empty message (i.e. only the padding pattern is hashed in that case).

After the message is split into j disjoint buffers, as described above, the underlying hash function HASH is applied to each buffer, independently, as follows:

$$
\begin{aligned}
&H_0 = HASH(Buff_0, length(Buff_0))\\
&H_1 = HASH(Buff_1, length(Buff_1))\\
&H_2 = HASH(Buff_2, length(Buff_2))\\
&\ldots\\
&H_{j-1} = HASH(Buff_{j-1}, length(Buff_{j-1}))
\end{aligned}
$$

The j-lanes digest (H) is defined by

$$
\begin{aligned}
&H = DIGEST(HASH, M, length\ (M), j) =\\
&HASH(H_0||H_1||H_2||\ldots||H_{j-1}, j\times D)
\end{aligned}
$$

Remark 1: The final stage of the process is called the wrapping stage. It hashes a message with a fixed size of $j\times D$ bytes. The number of updates required is $\lceil (j\times D+P)/B \rceil$ that are likely to be serial updates.

Remark 2: The API for a j-lanes hash for a fixed j would be the same as for the underlying hash, i.e. for SHA-256, the j-lanes implementation could have the following API: *SHA256_j_lanes (uint8_t* hash, uint8_t* msg, int len)*.

Example 1: Consider a message M with $N=4096$ bits, and the hash function HASH = SHA-256 that operates on 32-bit words ($Q=32$). Here, $k = \lceil 4096/32 \rceil = 128$. For $j = 8$ we get

$$
\begin{aligned}
&Buff_0 = m_0||m_8||m_{16}\ \ldots||m_{120}\\
&Buff_1 = m_1||m_9||m_{17}\ \ldots||m_{121}\\
&Buff_2 = m_2||m_{10}||m_{18}\ \ldots||m_{122}\\
&Buff_3 = m_3||m_{11}||m_{19}\ \ldots||m_{123}\\
&Buff_4 = m_4||m_{12}||m_{20}\ \ldots||m_{124}\\
&Buff_5 = m_5||m_{13}||m_{21}\ \ldots||m_{125}\\
&Buff_6 = m_6||m_{14}||m_{22}\ \ldots||m_{126}\\
&Buff_7 = m_7||m_{15}||m_{23}\ \ldots||m_{127}
\end{aligned}
$$

where each one of the eight buffers is 512 bit long.

Example 2: Consider a message M with $N=2913$ bits, and HASH = SHA-256 ($Q=32$). Here, $k = \lceil 2913/32 \rceil = 92$. Since $2913 \bmod 32 = 1$, the last word, m_{91}, consists of only a single bit. For $j = 8$, we get

$$
\begin{aligned}
Buff_0 &= m_0||m_8||m_{16}\ \ldots||m_{80}||m_{88}\\
Buff_1 &= m_1||m_9||m_{17}\ \ldots||m_{81}||m_{89}\\
Buff_2 &= m_2||m_{10}||m_{18}\ \ldots||m_{82}||m_{90}\\
Buff_3 &= m_3||m_{11}||m_{19}\ \ldots||m_{83}||m_{91}\\
Buff_4 &= m_4||m_{12}||m_{20}\ \ldots||m_{84}\\
Buff_5 &= m_5||m_{13}||m_{21}\ \ldots||m_{85}\\
Buff_6 &= m_6||m_{14}||m_{22}\ \ldots||m_{86}\\
Buff_7 &= m_7||m_{15}||m_{23}\ \ldots||m_{87}
\end{aligned}
$$

Here, $|Buff_0|=|Buff_1|=|Buff_2|=384$ bits, $|Buff_3| = 353$ bits, $|Buff_4| = |Buff_5| = |Buff_6| = |Buff_7| = 352$ bits.

Example 3: Consider a message M with $N=100$ bits, and HASH = SHA-256 ($Q=32$). Here, $k = \lceil 100/32 \rceil = 4$. Since $100 \bmod 32 = 4$, the last word, m_3, consists of only 4 bits. For $j = 8$, we get

$$
\begin{aligned}
Buff_0 &= m_0\\
Buff_1 &= m_1\\
Buff_2 &= m_2\\
Buff_3 &= m_3\\
Buff_4 &= NULL\\
Buff_5 &= NULL\\
Buff_6 &= NULL\\
Buff_7 &= NULL
\end{aligned}
$$

Here, $|Buff_0|=|Buff_1|=|Buff_2|=32$ bits, $|Buff_3| = 4$ bits, $|Buff_4| = |Buff_5| = |Buff_6| = |Buff_7| = 0$ bits.

Remark 3: Similarly to the serial hashing, the j-lanes hashing can process the message incrementally (e.g., when the messages is streamed). Since the parallelized compression operates (in parallel) on consecutive blocks of $j \times B$ bytes, it needs to receive only the "next $j \times B$ bytes" in order to compute an Update.

4 The j-pointers tree hash

An alternative way to define j "slices" of the message M, is to provide j pointers to j disjoint buffers $Buff_0,..,Buff_{j-1}$, of M, together with k values for the length of each buffer. In this case, it is also required that $\Sigma_i\, length(Buff_i) = length\,(M)$.

In this case, the j-pointers tree hash procedure would be the following. Compute the j hash values for each of the disjoint buffers:

$$
\begin{aligned}
H_0 &= HASH(Buff_0,\ length(Buff_0))\\
H_1 &= HASH(Buff_1,\ length(Buff_1))\\
H_2 &= HASH(Buff_2,\ length(Buff_2))
\end{aligned}
$$

```
...
H_{j-1} = HASH(Buff_{j-1}, length(Buff_{j-1}))
```

Produce the output digest $H = HASH(H_0||H_1||H_2||\ldots||H_{j-1}, j \times D)$.

Remark 4: In a software implementation, the API of the *j*-lanes function is the same as the API for any other hash function (see Remark 2). The function computes the buffers and their length internally. On the other hand, the API to a *j*-pointers hash requires a pointer to each buffer and its length, to be provided by the caller. For example:

```
SHA256_4_pointers(uint8_t* hash, uint8_t* buff0, int
len0, uint8_t* buff1, int len1, uint8_t* buff2, int
len2, uint8_t* buff3, int len3)
```

or, alternatively:

```
SHA256_j_pointers(uint8_t* hash, uint8_t** buffs,
int *lengths, int j)
```

5 The difference between j-pointers tree hash and j-lanes tree hash

The *j*-pointers and the *j*-lanes tree modes are essentially the same construction, and the difference is in how the message is viewed (logically) and *j* slices. The *j*-lanes tree has performance advantage when implemented on SIMD architectures, because it supports natural sequential loads into the SIMD registers: each word is naturally placed in the correct lane (see Fig. 1).

The *j*-pointers tree expects the data to be loaded from *j* locations. It is more suitable for implementations on multi-processor platforms, and for hashing multiple independent messages into a single digest (e.g., hashing a complete files-system while keeping a single digest). Of course, a *j*-pointers tree can also be used on a SIMD architecture, but in that case it requires "transposing the data" in order to place the words in the correct position in the registers. This (small) overhead is saved by using the *j*-lanes tree mode.

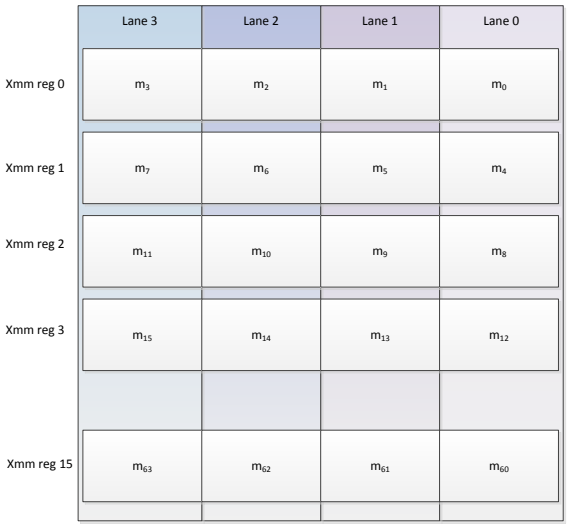

Fig. 1. The *j*-lanes tree mode natural data alignment with SIMD architectures (here, with *128*-bit registers (xmm'a) as *4* 32-bit words).

6 Counting the number of Updates

The performance of a standard (serial) hash function is closely proportional to the number of Updates (*U*) that the computations involve, namely

$$U = \lceil (L+P)/B \rceil \tag{1}$$

In Equation (1), each Update consumes *B* additional bytes of the (padded) message, and the number of bytes in the padded message is at least *L+P* (with no more than a single block added by the padding).

For the *j*-lanes hash (with the underlying function HASH), the number of *serially* computed Updates can be approximated by

$$U \le \lceil L/(\min(j,S)\times B) \rceil + 1 + \lceil (j\times D+P)/B \rceil \tag{2}$$

Note that some of the *j*-lanes Updates are carried out in parallel, compressing *min (S, j)* blocks per one Update call. Equation (2), accounts for parallelizing at most *min(S, j)* block compressions, thus contributing the term $\lceil L/(\min(j,S)\times B) \rceil$, plus one Update for the padding block (we count one for each lane, although (depending on the length of the message), some Updates are redundant). The wrapping step cannot be parallelized (in general) and adds $\lceil (j\times D+P)/B \rceil$ serial Updates to the count.

Example 4: Suppose that HASH = SHA-256, and consider a message of *1024* bytes. The standard SHA-256 function requires $\lceil (1024+9)/64 \rceil = 17$ Updates. We compare this to the count of *j*-lanes Updates for a few values of *j*.

For the AVX2 architecture (Haswell architecture [5]) we have D=32, *B=64*, *P=9*, *S=8*. This implies that the *8*-lanes SHA-256 (*j=8*) is optimal. It requires $\lceil 1024/(8\times 64) \rceil + 1 + \lceil (8\times 32+9)/64 \rceil = 8$ Updates.

For the AVX architecture (Sandy Bridge architecture), we have *S=4*, so, *j=4* is the optimal choice for this setup, and the *4*-lanes SHA-256 (*j=4*) requires $\lceil 1024/(4\times 64) \rceil + 1 + \lceil (4\times 32+9)/64 \rceil = 8$ Updates. Of course, it is possible to use the *8*-lanes SHA-256 on this architecture, but we can only parallelize *4* Updates using the xmm registers. Therefore, the *8*-lanes SHA-256 (*j*=8) on the AVX architecture (where *S=4*) requires $\lceil 1024/(4\times 64) \rceil + 1 + \lceil (8\times 32+9)/64 \rceil = 10$.

Figures 2, 3 and 4 show the number of Update calls (some are parallelized). As seen on Fig. 2, when the number of lanes is limited by the SIMD architecture, the total number of Updates for the different choices of *j*, varies only by the number of Updates that are required by the final wrapping stage.

However in Fig. 4 we see the differences when the choice of *j=16* becomes the most efficient from message size of *4KB* and up, requiring the least amount of Updates. For *4KB* messages, both *j=16* and *j=8* require *14* Updates, *j=4* requires *20* updates and the serial SHA-256 requires *65* Updates.

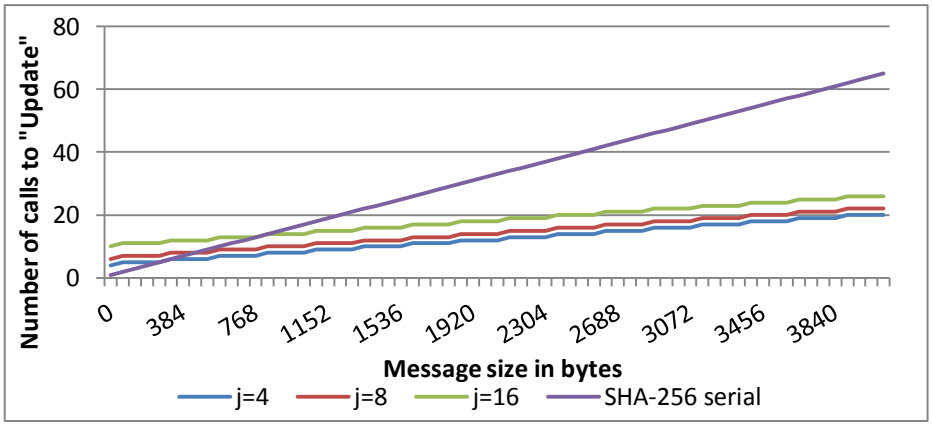

Fig. 2. The number of serially computed Updates required on a SIMD architecture supporting 4 lanes (e.g., AVX on a Sandy Bridge architecture), for different message lengths and different choices of *j*.

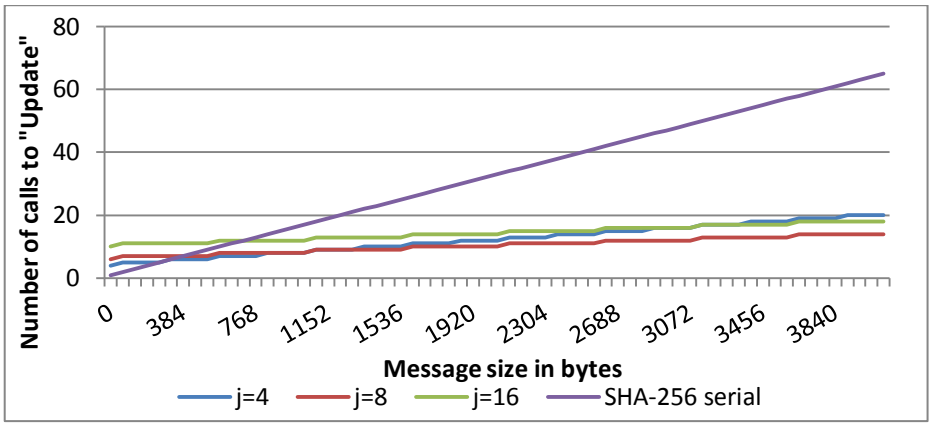

Fig. 3. The number of serially computed Updates required on a SIMD architecture supporting 8 lanes (e.g., AVX2 on a Haswell architecture), for different message lengths and different choices of *j*.

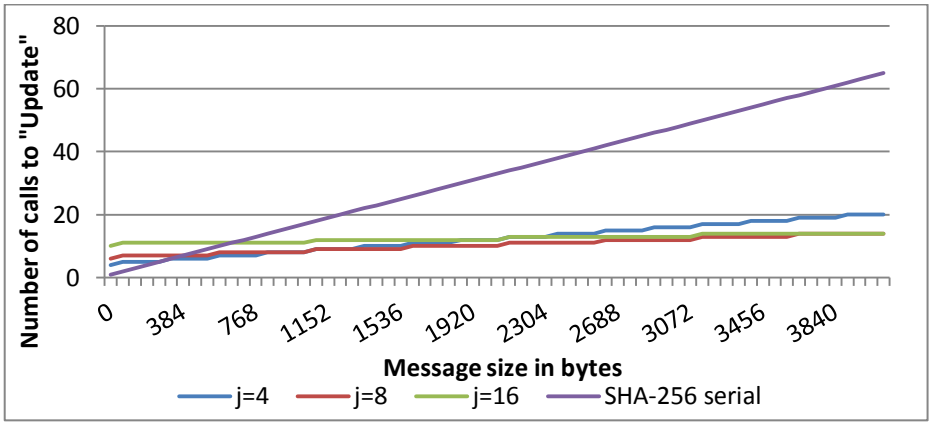

Fig. 4. The number of serially computed Updates required on a SIMD architecture supporting 16 lanes (AVX512f – a future architecture), for different message lengths and different choices of *j*.

7 The *j*-lanes hash and the *j*-pointers hash with different IV's

The Merkle-Damgård construction uses one *d*-bit IV to initialize the computations. For a *j*-lanes hashing, one might prefer to modify the IV's, and this sections proposes a method to achieve that.

Define *j+1* "Prefix" blocks ("*Pre*") as follows.

$$Pre_i = j||i||type||\ HASH||0^{B-NCHAR-9} \qquad i = 0,1\ldots,j \tag{3}$$

where

- j is encoded as a *32*-bit integer in Little Endian notation.
- i in the "index" of the lane, and is encoded as a *32*-bit integer in Little Endian notation. The values $i = 0,\ldots,j-1$ are used for the lanes, and the value $i = j$ is used for the wrapping step.
- *type* is a single byte with the value *0x0* for the *j*-lanes hash, and *0x1* for the *j*-pointers hash.
- HASH, the name of the underlying hash function: it is encoded as a string of characters. For SHA-256 we write HASH = "SHA256" or ASCII *534841323536* (encoding 'S'=*0x53*, 'H'=*0x48*, `A`=*0x41* etc.).
- The number of characters (*NCHAR*) in the string that indicates HASH should be such that $NCHAR+9 \leq B$.

The Prefix blocks are prepended to the $j+1$ hashed messages, and modify the "effective" IV that is being used. In other words, the *j*-lanes algorithm executes the following computations:

$$
\begin{aligned}
&H_0 = \mathrm{HASH}(Pre_0||Buff_0,\ \mathrm{length}(Buff_0)+B)\\
&H_1 = \mathrm{HASH}(Pre_1||Buff_1,\ \mathrm{length}(Buff_1)+B)\\
&H_2 = \mathrm{HASH}(Pre_2||Buff_2,\ \mathrm{length}(Buff_2)+B)\\
&\ldots\\
&H_{j-1} = \mathrm{HASH}(Pre_{j-1}||Buff_{j-1},\ \mathrm{length}(Buff_{j-1})+B)\\
&\\
&H = \mathrm{HASH}(Pre_j||H_0||\ldots||H_{j-1},\ j\times D+B)
\end{aligned}
$$

7.1 Pre-computing the IV's

The Prefix block do not need to be re-computed for each message. Instead, the $j+1$ IV values can be pre-computed by:

$$IV_i = \mathrm{compress}\ (\mathrm{HashIV},\ Pre_i);\ i = 0,1\ldots,j \qquad (4)$$

Note that the can also be viewed as a modification of HASH, to use the new IV's instead of the single IV it is using. For convenience, denote the hash function that uses the IV_i by $\mathrm{HASH`}_i$. With this notation, the *j*-lanes hashing can be expressed in terms of HASH` by:

$$
\begin{aligned}
&H_0 = \mathrm{HASH`}_0(Buff_0,\ \mathrm{length}(Buff_0))\\
&H_1 = \mathrm{HASH`}_1(Buff_1,\ \mathrm{length}(Buff_1))\\
&H_2 = \mathrm{HASH`}_2(Buff_2,\ \mathrm{length}(Buff_2))\\
&\ldots\\
&H_{j-1} = \mathrm{HASH`}_{j-1}(Buff_{j-1},\ \mathrm{length}(Buff_{j-1}))\\
&H = \mathrm{HASH`}_j(H_0||H_1||H_2||\ldots||H_{j-1},\ j\times D)
\end{aligned}
$$

Figure 5 shows the values of the prefix blocks and the new IV's (for HASH = SHA-256).

```
j = 4, type = j-lanes (0), HASH = "SHA256"

Pre0:
0000000400000000005348413235360000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
Pre1:
0000000400000001005348413235360000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
Pre2:
0000000400000002005348413235360000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
Pre3:
0000000400000003005348413235360000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
Pre4:
0000000400000004005348413235360000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000

IV0 =
Presented as 8 integers:
0x58fa599f    0xe4643148    0x4f5ff96d    0x3f090dbd    0x36dcede4
0x392a50b1 0x904a54e2 0xd0f7ed3a
Presented as a string of bytes:
9f59fa58483164e46df95f4fbd0d093fe4eddc36b1502a39e2544a903aedf7d0
IV1 =
Presented as 8 integers:
0x21e0dd66    0x903ebfda    0xeb4b6234    0x7a231591    0xd78a7ed4
0x8897c2dc 0x2c3950b9 0xe134381d
Presented as a string of bytes:
66dde021dabf3e9034624beb9115237ad47e8ad7dcc29788b950392c1d3834e1
IV2 =
Presented as 8 integers:
0xd138e9bf    0x4953c9ec    0xfdf21b4f    0x366c8f44    0x8bfddc06
0x3c01ba6d 0x1ba1fc0f 0x808f1417
Presented as a string of bytes:
bfe938d1ecc953494f1bf2fd448f6c3606dcfd8b6dba013c0ffca11b17148f80
IV3 =
Presented as 8 integers:
0x42f10312    0x2de19fba    0xa07ebae1    0x08e40004    0x377136e7
0x4124af55 0x586ec03e 0x593ce389
Presented as a string of bytes:
1203f142ba9fe12de1ba7ea00400e408e736713755af24413ec06e5889e33c59
IV4 =
Presented as 8 integers:
```

```
0x46f4005b    0xf65bc6dc    0x0018f006    0xb8b3df7e    0xa7fa7585
0x0aefc73c 0xd91912b2 0xc0faaf3a
Presented as a string of bytes:
5b00f446dcc65bf606f018007edfb3b88575faa73cc7ef0ab21219d93aaffac0
```

Fig. 5. An example for the Prefix blocks and the IV's generation for the *4*-lanes SHA-256 hash function.

Remark 5: the following alternative can be considered, for saving the space of storing *j+1* IV values. Instead, use a single (new) IV value for all the *j+1* hash computations. We fixed one value of *idx*, namely *idx = j+1*, and define the *j*-lanes hash by:

H_0 = HASH`$_{j+1}$($Buff_0$, length($Buff_0$))
H_1 = HASH`$_{j+1}$($Buff_1$, length($Buff_1$))
...
H_{j-1} = HASH`$_{j+1}$($Buff_{j-1}$, length($Buff_{j-1}$))
H = HASH`$_{j+1}$(H_0||H_1||H_2||…||H_{j-1}, j×D)

Figure 6 shows the values of the prefix block and the new IV (for HASH=SHA-256) for the alternative.

```
j = 4, type = j-lanes (0), HASH = "SHA256"

Pre5:
0000000400000005005348413235360000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000

IV5 =
Presented as 8 integers:
0x22169f91    0x8947cf0f    0xe023c546    0x2ca32fe0    0xa2ffc63e
0x7af66852 0x64961e97 0xec9e1ae5
Presented as a string of bytes:
919f16220fcf478946c523e0e02fa32c3ec6ffa25268f67a971e9664e51a9eec
```

Fig. 6. An example for the Prefix block and the (single) IV generation, for the *4*-lanes SHA-256 hash function, for the variant that uses only one modified IV.

Test vectors for *j*-lanes SHA-256 with *j=4, 8, 16* are provided in the Appendix.

8 Performance

This section shows the actually measured performance of *j*-lanes SHA-256, for *j = 4, 8, 16*, and compares it to the performance of the serial implementation of SHA-256. The results are shown in Fig. 7 and 8.

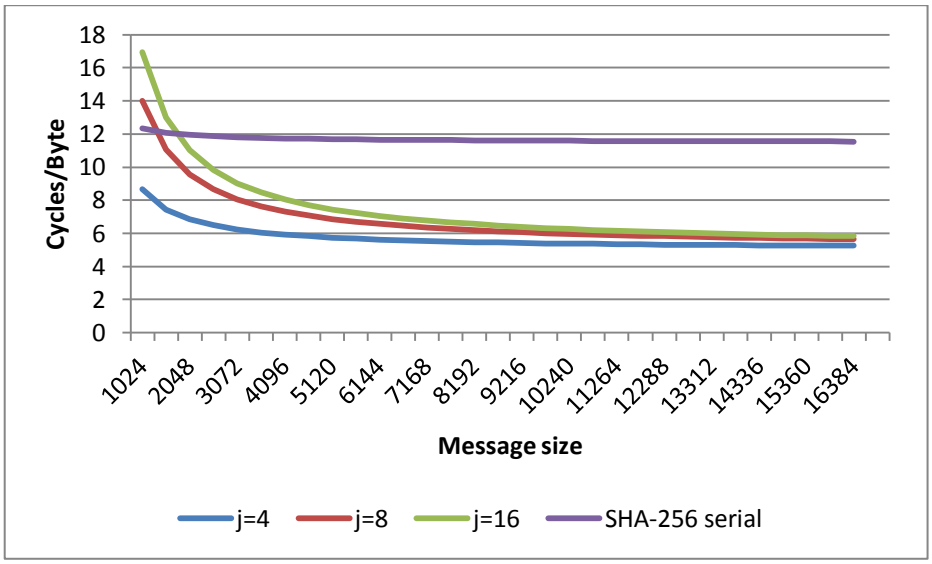

Fig. 7. Performance of SHA-256 j-lanes compared to the serial SHA-256 implementation, Intel Architecture Codename Sandy Bridge (S=4).

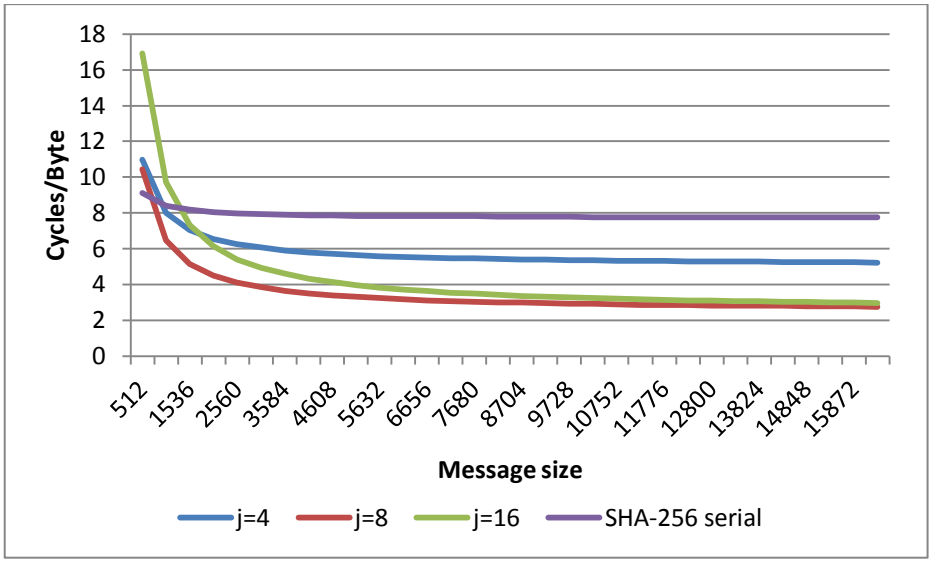

Fig. 8. Performance of SHA-256 j-lanes compared to the serial SHA-256 implementation, Intel Architecture Codename Haswell (S=8).

Clearly, the *j*-lanes SHA-256 has a significant performance advantage over the serial SHA-256, for messages that are at least a few Kilobytes long. The choice of *j* affects the hashing efficiency: for a given architecture, *j*-lanes SHA-256 with $j>S$ is

slower than *j*-lanes SHA-256 with the optimal choice of *j=S*, due to the longer wrapping step. However, the differences become almost negligible for long messages.

9 Conclusion

This paper showed the advantages of a *j*-lanes hashing method on modern processors, and provided information on how it can be easily defined and standardized.

The choice of *j* is a point that needs discussion. If a standard supports different *j* values, then the optimal choice can be selected per platform. This, however, could add an interoperability burden, and we can imagine that a single value of *j* would be preferable. In this context, we point out that Fig. 2 and 3 (theoretical approximations) are consistent with Figures 7 and 8 for *j=4* and *j=8* (actual measurements). Therefore, Fig, 4 can be viewed as a good indication for what can be expected when using *j=16* on the future architectures that would introduce the AVX512f architecture (supporting *S=16*). Furthermore, *j=16* allows better parallelization on multicore platforms. Consequently, our conclusion is that if only one value of *j* is to be specified by a standard, then the choice of *j =16* would be the most advantageous.

References

[1] ARM: Neon, ARM, http://www.arm.com/products/processors/technologies/neon.php
[2] FIPS: Secure Hash Standard (SHS), Federal Information Processing Standards publication 180-4, 2012. http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf
[3] Gueron, S., Krasnov, V.: Simultaneous hashing of multiple messages, Journal of Information Security, Vol. 3 No. 4, 2012, pp. 319-325.
[4] Gueron, S.: A j-Lanes Tree Hashing Mode and j-Lanes SHA-256, Journal of Information Security, Vol. 4 No. 1, 2013, pp. 7-11.
[5] Intel: Intel® Architecture Instruction Set Extensions Programming Reference, Intel, 2013, http://software.intel.com/en-us/file/319433-017pdf
[6] Reinders, J.: AVX-512 instructions, Intel Developer Zone, 2013, http://software.intel.com/en-us/blogs/2013/avx-512-instructions

Appendix: Test vectors

The test vectors provided below use the same *1024* bytes message (*M*) that is defined by

```
uint8_t M[1024];
for(int i=0;i<512;i++) {M[i*2] = 0;M[i*2+1]=i;}
```

```
The message M (1024 bytes):
000000010002000300040005000600070008000900 0a000b000c000d000e000f
001000110012001300140015001600170018001900 1a001b001c001d001e001f
```

```
0020002100220023002400250026002700280029002a002b002c002d002e002f
0030003100320033003400350036003700380039003a003b003c003d003e003f
0040004100420043004400450046004700480049004a004b004c004d004e004f
0050005100520053005400550056005700580059005a005b005c005d005e005f
0060006100620063006400650066006700680069006a006b006c006d006e006f
0070007100720073007400750076007700780079007a007b007c007d007e007f
0080008100820083008400850086008700880089008a008b008c008d008e008f
0090009100920093009400950096009700980099009a009b009c009d009e009f
00a000a100a200a300a400a500a600a700a800a900aa00ab00ac00ad00ae00af
00b000b100b200b300b400b500b600b700b800b900ba00bb00bc00bd00be00bf
00c000c100c200c300c400c500c600c700c800c900ca00cb00cc00cd00ce00cf
00d000d100d200d300d400d500d600d700d800d900da00db00dc00dd00de00df
00e000e100e200e300e400e500e600e700e800e900ea00eb00ec00ed00ee00ef
00f000f100f200f300f400f500f600f700f800f900fa00fb00fc00fd00fe00ff
0100010101020103010401050106010701080109010a010b010c010d010e010f
0110011101120113011401150116011701180119011a011b011c011d011e011f
0120012101220123012401250126012701280129012a012b012c012d012e012f
0130013101320133013401350136013701380139013a013b013c013d013e013f
0140014101420143014401450146014701480149014a014b014c014d014e014f
0150015101520153015401550156015701580159015a015b015c015d015e015f
0160016101620163016401650166016701680169016a016b016c016d016e016f
0170017101720173017401750176017701780179017a017b017c017d017e017f
0180018101820183018401850186018701880189018a018b018c018d018e018f
0190019101920193019401950196019701980199019a019b019c019d019e019f
01a001a101a201a301a401a501a601a701a801a901aa01ab01ac01ad01ae01af
01b001b101b201b301b401b501b601b701b801b901ba01bb01bc01bd01be01bf
01c001c101c201c301c401c501c601c701c801c901ca01cb01cc01cd01ce01cf
01d001d101d201d301d401d501d601d701d801d901da01db01dc01dd01de01df
01e001e101e201e301e401e501e601e701e801e901ea01eb01ec01ed01ee01ef
01f001f101f201f301f401f501f601f701f801f901fa01fb01fc01fd01fe01ff
```

Fig. 9. The message M used for the test vectors

```
Lane 0 =
0000000100020003000400050006000700080009000a000b000c000d000e000f
0010001100120013001400150016001700180019001a001b001c001d001e001f
0080008100820083008400850086008700880089008a008b008c008d008e008f
0090009100920093009400950096009700980099009a009b009c009d009e009f
0100010101020103010401050106010701080109010a010b010c010d010e010f
0110011101120113011401150116011701180119011a011b011c011d011e011f
0180018101820183018401850186018701880189018a018b018c018d018e018f
0190019101920193019401950196019701980199019a019b019c019d019e019f
j = 4, idx = 0, type = 0, Pre0 =
0000000400000000005348413235360000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
IV0 =
```

```
Presented as 8 integers:
0x58fa599f    0xe4643148    0x4f5ff96d    0x3f090dbd    0x36dcede4
0x392a50b1 0x904a54e2 0xd0f7ed3a
Presented as a string of bytes:
9f59fa58483164e46df95f4fbd0d093fe4eddc36b1502a39e2544a903aedf7d0
H0 =
Presented as 8 integers:
0x14cce35e    0xfcf144b0    0xea2bea4b    0x834d936b    0x002c834c
0xd7504a70 0x56ce9ea2 0x14fd91c4
Presented as a string of bytes:
5ee3cc14b044f1fc4bea2bea6b934d834c832c00704a50d7a29ece56c491fd14

Lane 1 =
00200021002200230024002500260027002800290002a002b002c002d002e002f
00300031003200330034003500360037003800390003a003b003c003d003e003f
00a000a100a200a300a400a500a600a700a800a900aa00ab00ac00ad00ae00af
00b000b100b200b300b400b500b600b700b800b900ba00bb00bc00bd00be00bf
01200121012201230124012501260127012801290012a012b012c012d012e012f
01300131013201330134013501360137013801390013a013b013c013d013e013f
01a001a101a201a301a401a501a601a701a801a901aa01ab01ac01ad01ae01af
01b001b101b201b301b401b501b601b701b801b901ba01bb01bc01bd01be01bf
j = 4, idx = 1, type = 0, Pre1 =
00000004000000010053484132353600000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
IV1 =
Presented as 8 integers:
0x21e0dd66    0x903ebfda    0xeb4b6234    0x7a231591    0xd78a7ed4
0x8897c2dc 0x2c3950b9 0xe134381d
Presented as a string of bytes:
66dde021dabf3e9034624beb9115237ad47e8ad7dcc29788b950392c1d3834e1
H1 =
Presented as 8 integers:
0x5bb62958    0x1233bf91    0x4c7c8842    0xbd4d44eb    0xf9adc359
0xcdea8e0d 0xfe31d9b9 0x6860e692
Presented as a string of bytes:
5829b65b91bf331242887c4ceb444dbd59c3adf90d8eeacdb9d931fe92e66068

Lane 2 =
00400041004200430044004500460047004800490004a004b004c004d004e004f
00500051005200530054005500560057005800590005a005b005c005d005e005f
00c000c100c200c300c400c500c600c700c800c900ca00cb00cc00cd00ce00cf
00d000d100d200d300d400d500d600d700d800d900da00db00dc00dd00de00df
01400141014201430144014501460147014801490014a014b014c014d014e014f
01500151015201530154015501560157015801590015a015b015c015d015e015f
01c001c101c201c301c401c501c601c701c801c901ca01cb01cc01cd01ce01cf
```

```
01d001d101d201d301d401d501d601d701d801d901da01db01dc01dd01de01df
j = 4, idx = 2, type = 0, Pre2 =
000000040000000200534841323536000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
IV2 =
Presented as 8 integers:
0xd138e9bf    0x4953c9ec    0xfdf21b4f    0x366c8f44    0x8bfddc06
0x3c01ba6d 0x1ba1fc0f 0x808f1417
Presented as a string of bytes:
bfe938d1ecc953494f1bf2fd448f6c3606dcfd8b6dba013c0ffca11b17148f80
H2 =
Presented as 8 integers:
0x9d28432d    0x99164093    0x0dcf8df0    0xd733e302    0x38a01ea0
0xd52f5c99 0x19a349d5 0x18c12b33
Presented as a string of bytes:
2d43289d93401699f08dcf0d02e333d7a01ea038995c2fd5d549a319332bc118

Lane 3 =
006000610062006300640065006600670068006900 6a006b006c006d006e006f
007000710072007300740075007600770078007900 7a007b007c007d007e007f
00e000e100e200e300e400e500e600e700e800e900ea00eb00ec00ed00ee00ef
00f000f100f200f300f400f500f600f700f800f900fa00fb00fc00fd00fe00ff
016001610162016301640165016601670168016901 6a016b016c016d016e016f
017001710172017301740175017601770178017901 7a017b017c017d017e017f
01e001e101e201e301e401e501e601e701e801e901ea01eb01ec01ed01ee01ef
01f001f101f201f301f401f501f601f701f801f901fa01fb01fc01fd01fe01ff
j = 4, idx = 3, type = 0, Pre3 =
000000040000000300534841323536000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
IV3 =
Presented as 8 integers:
0x42f10312    0x2de19fba    0xa07ebae1    0x08e40004    0x377136e7
0x4124af55 0x586ec03e 0x593ce389
Presented as a string of bytes:
1203f142ba9fe12de1ba7ea00400e408e736713755af24413ec06e5889e33c59
H3 =
Presented as 8 integers:
0x8f5138cb    0xd7c16314    0xfcb27e03    0x4c54a5f9    0xb134f8a1
0xe50a68b1 0x41739296 0xeb7de246
Presented as a string of bytes:
cb38518f1463c1d7037eb2fcf9a5544ca1f834b1b1680ae59692734146e27deb

The wrapping string (the concatenation of j digests)=
5ee3cc14b044f1fc4bea2bea6b934d834c832c00704a50d7a29ece56c491fd14
5829b65b91bf331242887c4ceb444dbd59c3adf90d8eeacdb9d931fe92e66068
```

```
2d43289d93401699f08dcf0d02e333d7a01ea038995c2fd5d549a319332bc118
cb38518f1463c1d7037eb2fcf9a5544ca1f834b1b1680ae59692734146e27deb
j = 4, idx = 4, type = 0, Pre4 =
00000004000000040053484132353600000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
IV4 =
Presented as 8 integers:
0x46f4005b    0xf65bc6dc    0x0018f006    0xb8b3df7e    0xa7fa7585
0x0aefc73c 0xd91912b2 0xc0faaf3a
Presented as a string of bytes:
5b00f446dcc65bf606f018007edfb3b88575faa73cc7ef0ab21219d93aaffac0
The output digests, H =
Presented as 8 integers:
0x546afddd    0x177bd3be    0x47830163    0x44e931fe    0xb9868c76
0x023b42e2 0x723c06f6 0x103a89db
Presented as a string of bytes:
ddfd6a54bed37b1763018347fe31e944768c86b9e2423b02f6063c72db893a10
```

Fig. 10. Test vector for SHA-256 4-lanes

```
Lane 0 =
00000001000200030004000500060007000800090000a000b000c000d000e000f
00100011001200130014001500160017001800190001a001b001c001d001e001f
01000101010201030104010501060107010801090010a010b010c010d010e010f
01100111011201130114011501160117011801190011a011b011c011d011e011f
j = 8, idx = 0, type = 0, Pre0 =
00000008000000000053484132353600000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
IV0 =
Presented as 8 integers:
0x6a3b8129    0x3220ee19    0x0e17b4e0    0xa0b28c13    0xba28aa23
0xfc1e5b6c 0x03d9fa03 0x2bee2831
Presented as a string of bytes:
29813b6a19ee2032e0b4170e138cb2a023aa28ba6c5b1efc03fad9033128ee2b
H0 =
Presented as 8 integers:
0x7d06d05b    0x4978b275    0xb4134996    0x294b2483    0x9c854d1f
0x9a149400 0x8e05432c 0x93b0dabe
Presented as a string of bytes:
5bd0067d75b27849964913b483244b291f4d859c0094149a2c43058ebedab093

Lane 1 =
00200021002200230024002500260027002800290002a002b002c002d002e002f
00300031003200330034003500360037003800390003a003b003c003d003e003f
01200121012201230124012501260127012801290012a012b012c012d012e012f
01300131013201330134013501360137013801390013a013b013c013d013e013f
```

```
j = 8, idx = 1, type = 0, Pre1 =
0000000800000001005348413235360000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
IV1 =
Presented as 8 integers:
0x99820054    0x07369df7    0xa3b0b315    0x1b779945    0x0c76475b
0xcb640190 0x956a869d 0xd9c2758f
Presented as a string of bytes:
54008299f79d360715b3b0a34599771b5b47760c900164cb9d866a958f75c2d9
H1 =
Presented as 8 integers:
0x0dcf339d    0x6c619247    0xc6faf4bb    0x673563ae    0x547af0b5
0x2d935747 0xb4735a2e 0x543f218c
Presented as a string of bytes:
9d33cf0d4792616cbbf4fac6ae633567b5f07a544757932d2e5a73b48c213f54

Lane 2 =
0040004100420043004400450046004700480049004a004b004c004d004e004f
0050005100520053005400550056005700580059005a005b005c005d005e005f
0140014101420143014401450146014701480149014a014b014c014d014e014f
0150015101520153015401550156015701580159015a015b015c015d015e015f
j = 8, idx = 2, type = 0, Pre2 =
0000000800000002005348413235360000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
IV2 =
Presented as 8 integers:
0x631617a3    0x11233e03    0xdce8a816    0xe1430328    0x274c55be
0x8dee62bc 0xd5866ffe 0xc9dd0687
Presented as a string of bytes:
a3171663033e231116a8e8dc280343e1be554c27bc62ee8dfe6f86d58706ddc9
H2 =
Presented as 8 integers:
0x1d8f3659    0xeecb2899    0x0f63271c    0x54594b22    0x6cc74874
0xe56f6be3 0x17326520 0x2d7720e1
Presented as a string of bytes:
59368f1d9928cbee1c27630f224b59547448c76ce36b6fe520653217e120772d

Lane 3 =
0060006100620063006400650066006700680069006a006b006c006d006e006f
0070007100720073007400750076007700780079007a007b007c007d007e007f
0160016101620163016401650166016701680169016a016b016c016d016e016f
0170017101720173017401750176017701780179017a017b017c017d017e017f
j = 8, idx = 3, type = 0, Pre3 =
0000000800000003005348413235360000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
```

```
IV3 =
Presented as 8 integers:
0x19e93312    0x20944725    0x5b566692    0x0ee60d52    0xcb5093d3
0x57d967f5 0xef097f81 0xd4d61120
Presented as a string of bytes:
1233e919254794209266565b520de60ed39350cbf567d957817f09ef2011d6d4
H3 =
Presented as 8 integers:
0x4f13d013    0x805973da    0x38b2c373    0xe49481ef    0xc63cfa28
0xc0dff22a 0x24992236 0x06ff6846
Presented as a string of bytes:
13d0134fda73598073c3b238ef8194e428fa3cc62af2dfc0362299244668ff06

Lane 4 =
008000810082008300840085008600870080089008a008b008c008d008e008f
009000910092009300940095009600970098099009a009b009c009d009e009f
018001810182018301840185018601870188018901a018b018c018d018e018f
019001910192019301940195019601970198019901a019b019c019d019e019f
j = 8, idx = 4, type = 0, Pre4 =
000000080000000400534841323536000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
IV4 =
Presented as 8 integers:
0x22a5a965    0x318b98de    0x92a8a1a9    0xb9563daa    0xb49d9e61
0xf844212b 0x3666bea0 0x602886f6
Presented as a string of bytes:
65a9a522de988b31a9a1a892aa3d56b9619e9db42b2144f8a0be6636f6862860
H4 =
Presented as 8 integers:
0xb8232abe    0x714b48cf    0x95fe4271    0xeb1f7926    0xd9c11739
0x14c4393a 0xf8579a2f 0x9fc01fa5
Presented as a string of bytes:
be2a23b8cf484b717142fe9526791feb3917c1d93a39c4142f9a57f8a51fc09f

Lane 5 =
00a000a100a200a300a400a500a600a700a800a900aa00ab00ac00ad00ae00af
00b000b100b200b300b400b500b600b700b800b900ba00bb00bc00bd00be00bf
01a001a101a201a301a401a501a601a701a801a901aa01ab01ac01ad01ae01af
01b001b101b201b301b401b501b601b701b801b901ba01bb01bc01bd01be01bf
j = 8, idx = 5, type = 0, Pre5 =
000000080000000500534841323536000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
IV5 =
Presented as 8 integers:
0x35d30516    0x37617459    0xc5511c15    0x83fda6c6    0x31032da8
```

```
0x63621c90 0x3d5d2b2f 0x70074543
Presented as a string of bytes:
1605d33559746137151c51c5c6a6fd83a82d0331901c62632f2b5d3d43450770
H5 =
Presented as 8 integers:
0x101eaa1f    0x95ff4557    0xd6014aa7    0x3b0d1b50    0x9a441169
0x8e10a4cf 0x7ea8b4ad 0xcd30792b
Presented as a string of bytes:
1faa1e105745ff95a74a01d6501b0d3b6911449acfa4108eadb4a87e2b7930cd

Lane 6 =
00c000c100c200c300c400c500c600c700c800c900ca00cb00cc00cd00ce00cf
00d000d100d200d300d400d500d600d700d800d900da00db00dc00dd00de00df
01c001c101c201c301c401c501c601c701c801c901ca01cb01cc01cd01ce01cf
01d001d101d201d301d401d501d601d701d801d901da01db01dc01dd01de01df
j = 8, idx = 6, type = 0, Pre6 =
0000000800000006005348413235360000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
IV6 =
Presented as 8 integers:
0x7ee1fc37    0xdf1295a0    0x4a6e41ae    0x025bb582    0x037906c0
0x204b07b5 0x1c920f65 0xe6115920
Presented as a string of bytes:
37fce17ea09512dfae416e4a82b55b02c0067903b5074b20650f921c205911e6
H6 =
Presented as 8 integers:
0xdbf5d023    0xc2520d81    0x1b00f365    0x2103b3d9    0xee9dcfe2
0x0aa717e8 0xc8fd1553 0xf1ca5624
Presented as a string of bytes:
23d0f5db810d52c265f3001bd9b30321e2cf9deee817a70a5315fdc82456caf1

Lane 7 =
00e000e100e200e300e400e500e600e700e800e900ea00eb00ec00ed00ee00ef
00f000f100f200f300f400f500f600f700f800f900fa00fb00fc00fd00fe00ff
01e001e101e201e301e401e501e601e701e801e901ea01eb01ec01ed01ee01ef
01f001f101f201f301f401f501f601f701f801f901fa01fb01fc01fd01fe01ff
j = 8, idx = 7, type = 0, Pre7 =
0000000800000007005348413235360000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
IV7 =
Presented as 8 integers:
0x070ac48e    0x18efe1cf    0x5d569c4e    0x84f637e4    0x938b184c
0xb882483e 0x562f6dc9 0xe9c06ec2
Presented as a string of bytes:
8ec40a07cfe1ef184e9c565de437f6844c188b933e4882b8c96d2f56c26ec0e9
```

```
H7 =
Presented as 8 integers:
0xecf8b1c3    0xf197615c    0x9061f43b    0xe629d544    0xb2e5ee40
0x2a3f7453 0xd81353f2 0x4f3d0969
Presented as a string of bytes:
c3b1f8ec5c6197f13bf4619044d529e640eee5b253743f2af25313d869093d4f

The wrapping string (the concatenation of j digests)=
5bd0067d75b27849964913b483244b291f4d859c0094149a2c43058ebedab093
9d33cf0d4792616cbbf4fac6ae633567b5f07a544757932d2e5a73b48c213f54
59368f1d9928cbee1c27630f224b59547448c76ce36b6fe520653217e120772d
13d0134fda73598073c3b238ef8194e428fa3cc62af2dfc0362299244668ff06
be2a23b8cf484b717142fe9526791feb3917c1d93a39c4142f9a57f8a51fc09f
1faa1e105745ff95a74a01d6501b0d3b6911449acfa4108eadb4a87e2b7930cd
23d0f5db810d52c265f3001bd9b30321e2cf9deee817a70a5315fdc82456caf1
c3b1f8ec5c6197f13bf4619044d529e640eee5b253743f2af25313d869093d4f
j = 8, idx = 8, type = 0, Pre8 =
0000000800000008005348413235360000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
IV8 =
Presented as 8 integers:
0x46cf3b1f    0x5e7170b4    0x6990aa2a    0x5986adff    0x85f37f26
0x42f20d96 0xf11b0673 0xeea04314
Presented as a string of bytes:
1f3bcf46b470715e2aaa9069ffad8659267ff385960df24273061bf11443a0ee
The output digests, H =
Presented as 8 integers:
0xee45c3db    0x0d14ec35    0x98d19bff    0x37913d84    0x3b290b63
0x6cb12aee 0x320cc900 0xbaa6fb77
Presented as a string of bytes:
dbc345ee35ec140dff9bd198843d9137630b293bee2ab16c00c90c3277fba6ba
```

Fig. 11. Test vector for SHA-256 8-lanes

```
Lane 0 =
0000000100020003000400050006000700080009000a000b000c000d000e000f
0010001100120013001400150016001700180019001a001b001c001d001e001f
j = 16, idx = 0, type = 0, Pre0 =
0000001000000000005348413235360000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
IV0 =
Presented as 8 integers:
0x53b18ab3    0x97ab1e53    0x53dc67cf    0xefb881d1    0x20b3a016
0x03b7ebd7 0xfcb2b1f8 0x2c940bed
Presented as a string of bytes:
b38ab153531eab97cf67dc53d181b8ef16a0b320d7ebb703f8b1b2fced0b942c
```

```
H0 =
Presented as 8 integers:
0x719d8a89    0x402b3f17    0x16812e49    0x828ca969    0x43a2f851
0x98c4949d 0xc2203e3e 0xae4ccbdd
Presented as a string of bytes:
898a9d71173f2b40492e811669a98c8251f8a2439d94c4983e3e20c2ddcb4cae

Lane 1 =
00200021002200230024002500260027002800290002a002b002c002d002e002f
00300031003200330034003500360037003800390003a003b003c003d003e003f
j = 16, idx = 1, type = 0, Pre1 =
00000010000000010053484132353600000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
IV1 =
Presented as 8 integers:
0x35c9eb52    0x0e0224d5    0xfe0e32d3    0x9e043748    0xb854c3a5
0x6ac6ccb9 0xd3cf9706 0x293d99b9
Presented as a string of bytes:
52ebc935d524020ed3320efe4837049ea5c354b8b9ccc66a0697cfd3b9993d29
H1 =
Presented as 8 integers:
0x67c8f721    0x1b1940e2    0xb0ae60b3    0xa6490065    0xe98056b5
0xb7c55280 0x7c4ec593 0xbe7546ce
Presented as a string of bytes:
21f7c867e240191bb360aeb0650049a6b55680e98052c5b793c54e7cce4675be

Lane 2 =
00400041004200430044004500460047004800490004a004b004c004d004e004f
00500051005200530054005500560057005800590005a005b005c005d005e005f
j = 16, idx = 2, type = 0, Pre2 =
00000010000000020053484132353600000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
IV2 =
Presented as 8 integers:
0x16c7eac5    0x29771983    0x634f500e    0xbc0521fb    0xc217d151
0xf35bc961 0x888d991d 0x5f9dd744
Presented as a string of bytes:
c5eac716831977290e504f63fb2105bc51d117c261c95bf31d998d8844d79d5f
H2 =
Presented as 8 integers:
0xa0a085b0    0x038d39b3    0x64bc5a94    0x6eeb458e    0xec9e88ff
0xc114e78b 0x83e5d86c 0xb9f851a2
Presented as a string of bytes:
b085a0a0b3398d03945abc648e45eb6eff889eec8be714c16cd8e583a251f8b9
```

```
Lane 3 =
006000610062006300640065006600670068006900 6a006b006c006d006e006f
007000710072007300740075007600770078007900 7a007b007c007d007e007f
j = 16, idx = 3, type = 0, Pre3 =
000000100000000300534841323536000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
IV3 =
Presented as 8 integers:
0x8c517791    0x38b65adf    0xdc9929ef    0x32007b4f    0x285ec615
0xd63c958a 0x512f2371 0x85621ad4
Presented as a string of bytes:
9177518cdf5ab638ef2999dc4f7b003215c65e288a953cd671232f51d41a6285
H3 =
Presented as 8 integers:
0xf8e63cb5    0x5fbe9ff3    0xda626540    0xd0383fd4    0x7199fbbc
0xdefa8cac 0x87e2f5b3 0x512dd863
Presented as a string of bytes:
b53ce6f8f39fbe5f406562dad43f38d0bcfb9971ac8cfadeb3f5e28763d82d51

Lane 4 =
008000810082008300840085008600870088008900 8a008b008c008d008e008f
009000910092009300940095009600970098009900 9a009b009c009d009e009f
j = 16, idx = 4, type = 0, Pre4 =
000000100000000400534841323536000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
IV4 =
Presented as 8 integers:
0xb8be6ae0    0x04ff42c1    0xfe42e0d5    0xffc88363    0xabc81c8b
0x5a1b5d05 0xf4f7ca8b 0x3e3ef932
Presented as a string of bytes:
e06abeb8c142ff04d5e042fe6383c8ff8b1cc8ab055d1b5a8bcaf7f432f93e3e
H4 =
Presented as 8 integers:
0x509c6bf4    0x7703a7bf    0xc6d0d269    0x76d97683    0xda9ca583
0xbf51c4ff 0x67b063ab 0x24eece64
Presented as a string of bytes:
f46b9c50bfa7037769d2d0c68376d97683a59cdaffc451bfab63b06764ceee24

Lane 5 =
00a000a100a200a300a400a500a600a700a800a900aa00ab00ac00ad00ae00af
00b000b100b200b300b400b500b600b700b800b900ba00bb00bc00bd00be00bf
j = 16, idx = 5, type = 0, Pre5 =
000000100000000500534841323536000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
IV5 =
```

```
Presented as 8 integers:
0xe0b0c033   0x11d94944   0xe48e9147   0x1baeb8bb   0xf3c66409
0x5a5ff515 0xe347c8aa 0x101181dc
Presented as a string of bytes:
33c0b0e04449d91147918ee4bbb8ae1b0964c6f315f55f5aaac847e3dc811110
H5 =
Presented as 8 integers:
0xfc2a0cec   0x7ad4a654   0x2fff6fa5   0x2d4e2b45   0x5b55c6d6
0x8b587dd0 0x21da9cfd 0x9f7cb8a3
Presented as a string of bytes:
ec0c2afc54a6d47aa56fff2f452b4e2dd6c6555bd07d588bfd9cda21a3b87c9f

Lane 6 =
00c000c100c200c300c400c500c600c700c800c900ca00cb00cc00cd00ce00cf
00d000d100d200d300d400d500d600d700d800d900da00db00dc00dd00de00df
j = 16, idx = 6, type = 0, Pre6 =
0000001000000006005348413235360000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
IV6 =
Presented as 8 integers:
0x23fb73e0   0x8a0b496c   0x505154fa   0x746fc777   0xe6f998b5
0x67495484 0x2406ffc6 0x4aa43f76
Presented as a string of bytes:
e073fb236c490b8afa54515077c76f74b598f9e684544967c6ff0624763fa44a
H6 =
Presented as 8 integers:
0xc23be33f   0x3bddb744   0x4e47f29d   0xa14164a2   0x74168998
0xf21a7cde 0x3545d839 0x6b5c4bf1
Presented as a string of bytes:
3fe33bc244b7dd3b9df2474ea26441a198891674de7c1af239d84535f14b5c6b

Lane 7 =
00e000e100e200e300e400e500e600e700e800e900ea00eb00ec00ed00ee00ef
00f000f100f200f300f400f500f600f700f800f900fa00fb00fc00fd00fe00ff
j = 16, idx = 7, type = 0, Pre7 =
0000001000000007005348413235360000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
IV7 =
Presented as 8 integers:
0x44fe7e09   0xbe083b6e   0xecaaedbd   0xe467f27e   0x40ac697f
0x9f6d4133 0x2d7b8ff7 0x68cda71a
Presented as a string of bytes:
097efe446e3b08bebdedaaec7ef267e47f69ac4033416d9ff78f7b2d1aa7cd68
H7 =
Presented as 8 integers:
```

```
0x11614256    0x54aeeb83    0xf1226b0f    0x3d1dc0ce    0xaf3d2bdd
0xced3c12c 0xffba137c 0x5a91fb36
Presented as a string of bytes:
5642611183ebae540f6b22f1cec01d3ddd2b3daf2cc1d3ce7c13baff36fb915a

Lane 8 =
01000101010201030104010501060107010801090 10a010b010c010d010e010f
01100111011201130114011501160117011801190 11a011b011c011d011e011f
j = 16, idx = 8, type = 0, Pre8 =
00000010000000080053484132353600000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
IV8 =
Presented as 8 integers:
0x0a821cae    0xbb015410    0x383d76d1    0xfde64151    0xda710bc2
0xcca59f04 0xea760684 0xd2a7352b
Presented as a string of bytes:
ae1c820a105401bbd1763d385141e6fdc20b71da049fa5cc840676ea2b35a7d2
H8 =
Presented as 8 integers:
0x1303de22    0x76973021    0x1001bc13    0x98e17eee    0xb140e47e
0xb5c516d7 0x6c8a93bd 0x59514624
Presented as a string of bytes:
22de03132130977613bc0110ee7ee1987ee440b1d716c5b5bd938a6c24465159

Lane 9 =
01200121012201230124012501260127012 80129012a012b012c012d012e012f
01300131013201330134013501360137013 80139013a013b013c013d013e013f
j = 16, idx = 9, type = 0, Pre9 =
00000010000000090053484132353600000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
IV9 =
Presented as 8 integers:
0x9aac9abf    0x14eca0bd    0xb08afdb2    0xfa34993b    0xadf2b31c
0xfa7e99e5 0x28cb925e 0xe9c24f3f
Presented as a string of bytes:
bf9aac9abda0ec14b2fd8ab03b9934fa1cb3f2ade5997efa5e92cb283f4fc2e9
H9 =
Presented as 8 integers:
0x6effc9a9    0xd0684e09    0xece12ee5    0x43d3d6d5    0x059185fb
0x1d80feb7 0x25e4de30 0x95b73312
Presented as a string of bytes:
a9c9ff6e094e68d0e52ee1ecd5d6d343fb859105b7fe801d30dee4251233b795

Lane 10 =
01400141014201430144014501460147014 80149014a014b014c014d014e014f
```

```
01500151015201530154015501560157015801590l5a015b015c015d015e015f
j = 16, idx = 10, type = 0, Pre10 =
000000100000000a00534841323536000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
IV10 =
Presented as 8 integers:
0x86ad6710    0x87916f83    0x3ab0051f    0x981766d8    0x36338e11
0x30b19cc3 0x15175040 0x598725c8
Presented as a string of bytes:
1067ad86836f91871f05b03ad8661798118e3336c39cb13040501715c8258759
H10 =
Presented as 8 integers:
0x775a098b    0x0aee00ed    0x2b3984ee    0x4e2559b3    0x048c6452
0x38dbd096 0x72f31038 0x0d58e15d
Presented as a string of bytes:
8b095a77ed00ee0aee84392bb359254e52648c0496d0db383810f3725de1580d

Lane 11 =
01600161016201630164016501660167016801690l6a016b016c016d016e016f
01700171017201730174017501760177017801790l7a017b017c017d017e017f
j = 16, idx = 11, type = 0, Pre11 =
000000100000000b00534841323536000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
IV11 =
Presented as 8 integers:
0x8083c5f4    0x139868a9    0x51752174    0xd5e1b54e    0xe5232e68
0xb01f7589 0x184627d5 0x39bc31cc
Presented as a string of bytes:
f4c58380a9689813742175514eb5e1d5682e23e589751fb0d5274618cc31bc39
H11 =
Presented as 8 integers:
0xdcb593e8    0xaf7e4ba4    0xda0da20a    0x620dd0e8    0xb34da956
0xacfeb6e2 0xe882514f 0x581d7b36
Presented as a string of bytes:
e893b5dca44b7eaf0aa20ddae8d00d6256a94db3e2b6feac4f5182e8367b1d58

Lane 12 =
01800181018201830184018501860187018801890l8a018b018c018d018e018f
01900191019201930194019501960197019801990l9a019b019c019d019e019f
j = 16, idx = 12, type = 0, Pre12 =
000000100000000c00534841323536000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
IV12 =
Presented as 8 integers:
0xb354a648    0xb812c666    0xd7d7eaa7    0xadafee79    0x4dc3a6db
```

```
0xf6c09db0 0x5e598c35 0xe0a1a309
Presented as a string of bytes:
48a654b366c612b8a7ead7d779eeafaddba6c34db09dc0f6358c595e09a3a1e0
H12 =
Presented as 8 integers:
0xcd439906    0xef7a7028    0xf93701c5    0xcbf5c4d0    0x16c42056
0xea8bd068 0xeb14edc8 0x036a2777
Presented as a string of bytes:
069943cd28707aefc50137f9d0c4f5cb5620c41668d08beac8ed14eb77276a03

Lane 13 =
01a001a101a201a301a401a501a601a701a801a901aa01ab01ac01ad01ae01af
01b001b101b201b301b401b501b601b701b801b901ba01bb01bc01bd01be01bf
j = 16, idx = 13, type = 0, Pre13 =
0000001000000000d00534841323536000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
IV13 =
Presented as 8 integers:
0xeea77a54    0x2abc2923    0x31b6a769    0x5d7202c9    0x4b58821c
0x9db8a548 0x485cefe7 0x371439b0
Presented as a string of bytes:
547aa7ee2329bc2a69a7b631c902725d1c82584b48a5b89de7ef5c48b0391437
H13 =
Presented as 8 integers:
0xd791ba08    0xa2907c8a    0xd8f5285a    0x969ff8f1    0x8bbd811d
0x050ce9a0 0x55f84313 0xcec38885
Presented as a string of bytes:
08ba91d78a7c90a25a28f5d8f1f89f961d81bd8ba0e90c051343f8558588c3ce

Lane 14 =
01c001c101c201c301c401c501c601c701c801c901ca01cb01cc01cd01ce01cf
01d001d101d201d301d401d501d601d701d801d901da01db01dc01dd01de01df
j = 16, idx = 14, type = 0, Pre14 =
0000001000000000e00534841323536000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
IV14 =
Presented as 8 integers:
0xfd2bc3b8    0xaa9ab770    0x71ff87de    0xd44b81b2    0xd2fee8ac
0x9dff47a5 0x9cc73511 0xfb471d62
Presented as a string of bytes:
b8c32bfd70b79aaade87ff71b2814bd4ace8fed2a547ff9d1135c79c621d47fb
H14 =
Presented as 8 integers:
0x7663cb26    0x9ec92b48    0x89f9ea1e    0x3f203b7b    0x96014a0a
0x9041b76a 0x829d7b31 0x33b4b45a
```

```
Presented as a string of bytes:
26cb6376482bc99e1eeaf9897b3b203f0a4a01966ab74190317b9d825ab4b433

Lane 15 =
01e001e101e201e301e401e501e601e701e801e901ea01eb01ec01ed01ee01ef
01f001f101f201f301f401f501f601f701f801f901fa01fb01fc01fd01fe01ff
j = 16, idx = 15, type = 0, Pre15 =
0000001000000000f00534841323536000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
IV15 =
Presented as 8 integers:
0xd32f91ee    0x77d5c0d5    0x1dc23267    0x3331a841    0xfdda8026
0x3a528977 0x61cf7f28 0xe0916e03
Presented as a string of bytes:
ee912fd3d5c0d5776732c21d41a831332680dafd7789523a287fcf61036e91e0
H15 =
Presented as 8 integers:
0x41710eb5    0x6ffd2d28    0x02635e04    0xafbc0b26    0x1eeb3e7b
0xedf32f61 0xb8a733fa 0xf0e05265
Presented as a string of bytes:
b50e7141282dfd6f045e6302260bbcaf7b3eeb1e612ff3edfa33a7b86552e0f0

The wrapping string (the concatenation of j digests)=
898a9d71173f2b40492e811669a98c8251f8a2439d94c4983e3e20c2ddcb4cae
21f7c867e240191bb360aeb0650049a6b55680e98052c5b793c54e7cce4675be
b085a0a0b3398d03945abc648e45eb6eff889eec8be714c16cd8e583a251f8b9
b53ce6f8f39fbe5f406562dad43f38d0bcfb9971ac8cfadeb3f5e28763d82d51
f46b9c50bfa7037769d2d0c68376d97683a59cdaffc451bfab63b06764ceee24
ec0c2afc54a6d47aa56fff2f452b4e2dd6c6555bd07d588bfd9cda21a3b87c9f
3fe33bc244b7dd3b9df2474ea26441a198891674de7c1af239d84535f14b5c6b
5642611183ebae540f6b22f1cec01d3ddd2b3daf2cc1d3ce7c13baff36fb915a
22de03132130977613bc0110ee7ee1987ee440b1d716c5b5bd938a6c24465159
a9c9ff6e094e68d0e52ee1ecd5d6d343fb859105b7fe801d30dee4251233b795
8b095a77ed00ee0aee84392bb359254e52648c0496d0db383810f3725de1580d
e893b5dca44b7eaf0aa20ddae8d00d6256a94db3e2b6feac4f5182e8367b1d58
069943cd28707aefc50137f9d0c4f5cb5620c41668d08beac8ed14eb77276a03
08ba91d78a7c90a25a28f5d8f1f89f961d81bd8ba0e90c051343f8558588c3ce
26cb6376482bc99e1eeaf9897b3b203f0a4a01966ab74190317b9d825ab4b433
b50e7141282dfd6f045e6302260bbcaf7b3eeb1e612ff3edfa33a7b86552e0f0
j = 16, idx = 16, type = 0, Pre16 =
0000001000000010005348413235360000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
IV16 =
Presented as 8 integers:
0x0b88d460    0x0f6babc4    0xb4ded1a4    0x72ed8427    0x54c4c715
```

```
0x2f9f2775 0x91863c0e 0x4cb44b3e
Presented as a string of bytes:
60d4880bc4ab6b0fa4d1deb42784ed7215c7c45475279f2f0e3c86913e4bb44c
The output digests, H =
Presented as 8 integers:
0x83915ca0    0x348feaf2    0x0f094b8b    0x4c521f88    0xd5a1cc07
0xca7d7437 0xf9788f23 0x550e62a8
Presented as a string of bytes:
a05c9183f2ea8f348b4b090f881f524c07cca1d537747dca238f78f9a8620e55
```

Fig. 12. Test vector for SHA-256 16-lanes