Journal of Information Security, 2014, 5, 91-113
Published Online July 2014 in SciRes. http://www.scirp.org/journal/jis
http://dx.doi.org/10.4236/jis.2014.53010

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, Haifa, Israel
[2]Intel Corporation, Israel Development Center, Haifa, Israel
Email: shay@math.haifa.ac.il

Received 1 May 2014; revised 1 June 2014; accepted 28 June 2014

Abstract

***j*-lanes tree hashing is a tree mode that splits an input message into *j* slices, computes *j* independent digests of each slice, and outputs the hash value of their concatenation. *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 expose parallelization opportunities in a hashing process that is otherwise serial by nature. As a result, they have a performance advantage on modern processor architectures. This paper provides precise specifications for these hashing modes, proposes appropriate IVs, 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 upon the *j*-lanes tree hashing mode which was proposed in [1]. 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

How to cite this paper: Gueron, S. (2014) Parallelized Hashing via *j*-Lanes and *j*-Pointers Tree Modes, with Applications to SHA-256. *Journal of Information Security*, **5**, 91-113. http://dx.doi.org/10.4236/jis.2014.53010

modern processors that have SIMD (Single Instruction Multiple Data) instructions. Currently deployed SIMD architectures use either 128-bit (e.g., SSE, AVX [3], NEON [4]) or 256-bit (AVX2 [3]) 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 [3] 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 [3] [5] 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 SHA-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, from 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 $m_0m_1\ldots m_{k-1}$.

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-bit value that can be used 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:

- $\lceil x \rceil$: floor(x).
- $\lfloor x \rfloor$: 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 *Hash IV* = 0x6*a*09*e*667, 0x*bb*67*ae*85, 0x3*c*6*ef*372, 0x*a*54*ff*53*a*, 0x510*e*527*f*, 0x9b05688*c*, 0x1*f*83*d*9*ab*, 0x5*be*0*cd*19; 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 ($j \geq 2$) 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 ($k \geq 0$) consecutive Q-bit "words" m_i, $i = 0, 1, \ldots, k - 1$ (if M is the *NULL* message, then $k = 0$).

When $k \geq 1$, the words m_j, $j = 0, 1, \ldots, k - 2$ (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 ($N \bmod Q$) bits.

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

$\mathrm{Buff}_0 = \mathrm{m}_0||\mathrm{m}_j||\mathrm{m}_{j\times 2} \ldots$

$\mathrm{Buff}_1 = \mathrm{m}_1||\mathrm{m}_{j+1}||\mathrm{m}_{j\times 2+1} \ldots$

...

$\mathrm{Buff}_{j-1} = \mathrm{m}_{j-1}||\mathrm{m}_{j\times 2-1}||\mathrm{m}_{j\times 3-1} \ldots$

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 independently applied to each buffer as follows:

$\mathrm{H}_0 = \mathrm{HASH}\ (\mathrm{Buff}_0, \mathrm{length}\ (\mathrm{Buff}_0))$

$\mathrm{H}_1 = \mathrm{HASH}\ (\mathrm{Buff}_1, \mathrm{length}\ (\mathrm{Buff}_1))$

$\mathrm{H}_2 = \mathrm{HASH}\ (\mathrm{Buff}_2, \mathrm{length}\ (\mathrm{Buff}_2))$

…

$\mathrm{H}_{j-1} = \mathrm{HASH}\ (\mathrm{Buff}_{j-1}, \mathrm{length}\ (\mathrm{Buff}_{j-1}))$

The j-lanes digest (H) is defined by

$\mathrm{H} = \mathrm{DIGEST}\ (\mathrm{HASH}, \mathrm{M}, \mathrm{length}\ (\mathrm{M}), j) = \mathrm{HASH}\ (\mathrm{H}_0||\mathrm{H}_1||\mathrm{H}_2||\ldots||\mathrm{H}_{j-1}, j \times \mathrm{D})$

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: *SHA*256*_j_lanes* (*uint*8*_t* hash, uint*8*_t* msg*, *size_tlen*).

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

$\mathrm{Buff}_0 = \mathrm{m}_0||\mathrm{m}_8||\mathrm{m}_{16} \ldots||\mathrm{m}_{120}$

$\mathrm{Buff}_1 = \mathrm{m}_1||\mathrm{m}_9||\mathrm{m}_{17} \ldots||\mathrm{m}_{121}$

$\mathrm{Buff}_2 = \mathrm{m}_2||\mathrm{m}_{10}||\mathrm{m}_{18} \ldots||\mathrm{m}_{122}$

$\mathrm{Buff}_3 = \mathrm{m}_3||\mathrm{m}_{11}||\mathrm{m}_{19} \ldots||\mathrm{m}_{123}$

$\mathrm{Buff}_4 = \mathrm{m}_4||\mathrm{m}_{12}||\mathrm{m}_{20} \ldots||\mathrm{m}_{124}$

$\mathrm{Buff}_5 = \mathrm{m}_5||\mathrm{m}_{13}||\mathrm{m}_{21} \ldots||\mathrm{m}_{125}$

$\mathrm{Buff}_6 = \mathrm{m}_6||\mathrm{m}_{14}||\mathrm{m}_{22} \ldots||\mathrm{m}_{126}$

$\mathrm{Buff}_7 = \mathrm{m}_7||\mathrm{m}_{15}||\mathrm{m}_{23} \ldots||\mathrm{m}_{127}$

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 *mod* 32 = 1, the last word, m_{91}, consists of only a single bit. For $j = 8$, we get

$\mathrm{Buff}_0 = \mathrm{m}_0||\mathrm{m}_8||\mathrm{m}_{16} \ldots||\mathrm{m}_{80}||\mathrm{m}_{88}$

$\mathrm{Buff}_1 = \mathrm{m}_1||\mathrm{m}_9||\mathrm{m}_{17} \ldots||\mathrm{m}_{81}||\mathrm{m}_{89}$

$\mathrm{Buff}_2 = \mathrm{m}_2||\mathrm{m}_{10}||\mathrm{m}_{18} \ldots||\mathrm{m}_{82}||\mathrm{m}_{90}$

$\mathrm{Buff}_3 = \mathrm{m}_3||\mathrm{m}_{11}||\mathrm{m}_{19} \ldots||\mathrm{m}_{83}||\mathrm{m}_{91}$

$\mathrm{Buff}_4 = \mathrm{m}_4||\mathrm{m}_{12}||\mathrm{m}_{20} \ldots||\mathrm{m}_{84}$

$\mathrm{Buff}_5 = \mathrm{m}_5||\mathrm{m}_{13}||\mathrm{m}_{21} \ldots||\mathrm{m}_{85}$

$\mathrm{Buff}_6 = \mathrm{m}_6||\mathrm{m}_{14}||\mathrm{m}_{22} \ldots||\mathrm{m}_{86}$

$\mathrm{Buff}_7 = \mathrm{m}_7||\mathrm{m}_{15}||\mathrm{m}_{23} \ldots||\mathrm{m}_{87}$

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 *mod* 32 = 4, the last word, m_3, consists of only 4 bits. For j = 8, we get

$Buff_0 = m_0$
$Buff_1 = m_1$
$Buff_2 = m_2$
$Buff_3 = m_3$
$Buff_4$ = NULL
$Buff_5$ = NULL
$Buff_6$ = NULL
$Buff_7$ = NULL

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 Σ_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:

H_0 = HASH ($Buff_0$, length ($Buff_0$))
H_1 = HASH ($Buff_1$, length ($Buff_1$))
H_2 = HASH ($Buff_2$, length ($Buff_2$))
...
H_{j-1} = HASH ($Buff_{j-1}$, length ($Buff_{j-1}$))

Produce the output digest

$H = \text{HASH}\ (H_0 || H_1 || H_2 || \ldots || H_{j-1}, j \times D)$

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:

*SHA*256_4_*pointers*(*uint*8_*t** *hash*, *uint*8_*t** *buff*0, *size_tlen*0, *uint*8_*t** *buff*1, *size_tlen*1, *uint*8_*t** *buff*2, *size_tlen*2, *uint*8_*t** *buff*3, *size_tlen*3)

or, alternatively:

*SHA*256_*j*_*pointers*(*uint*8_*t** *hash*, *uint*8_*t*** *buffs*, *size_t***lengths*, *unsigned 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) as j slices. The j-lanes tree mode has a 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 **Figure 1**).

The j-pointers tree mode 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 file-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.

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

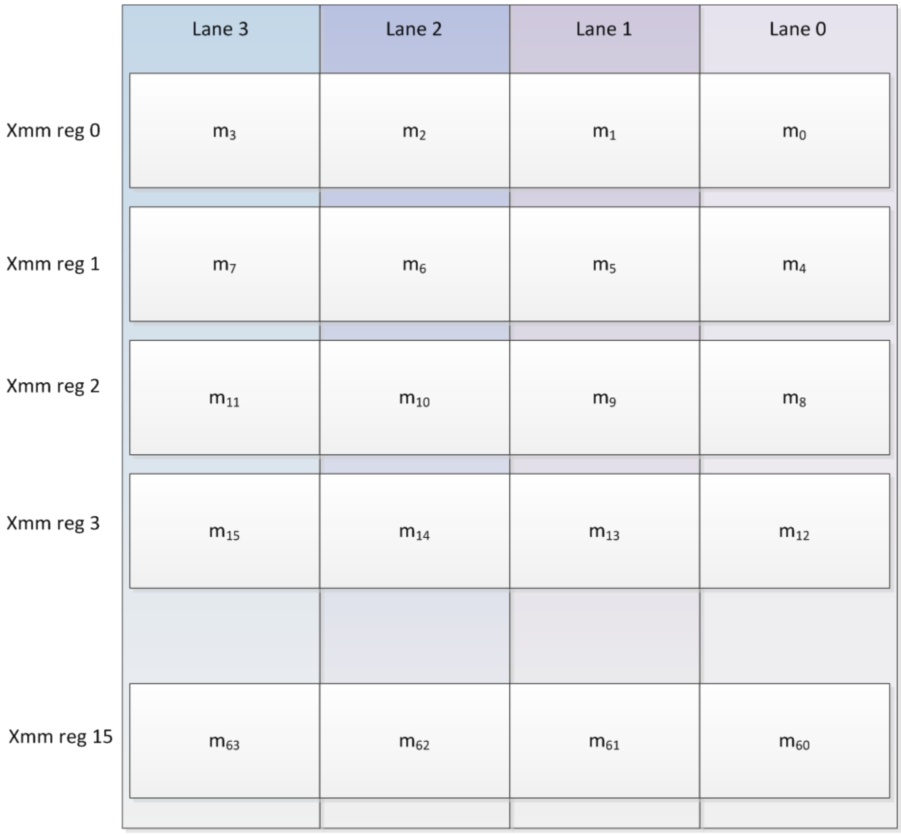

Figure 1. The *j*-lanes tree mode natural data alignment with SIMD architectures (here, with 128-bit registers (xmm'a) as 4 32-bit words).

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

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. A padding block is counted 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 [3]) 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 $\lceil 1024/(4 \times 64) \rceil + 1 + \lceil (8 \times 32 + 9)/6 \rceil = 10$ Updates.

Figures 2-4 show the number of Update calls (some are parallelized). As seen on **Figure 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.

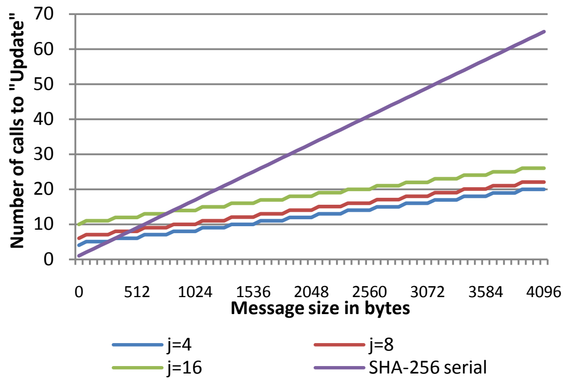

Figure 2. The number of serially computed Updates required on a SIMD architecture supporting 4 lanes (e.g., AVX on a Sandy Bridge architecture), for different message lengths and different choices of *j*.

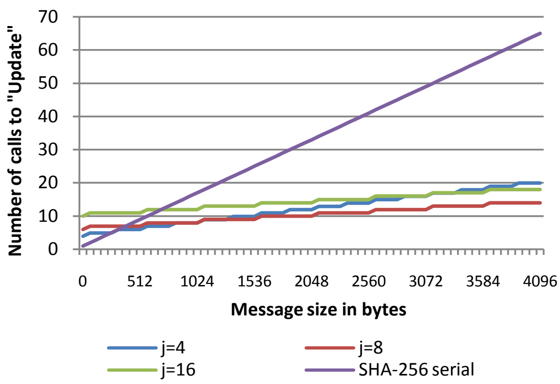

Figure 3. The number of serially computed Updates required on a SIMD architecture supporting 8 lanes (e.g., AVX2 on a Haswell architecture), for different message lengths and different choices of *j*.

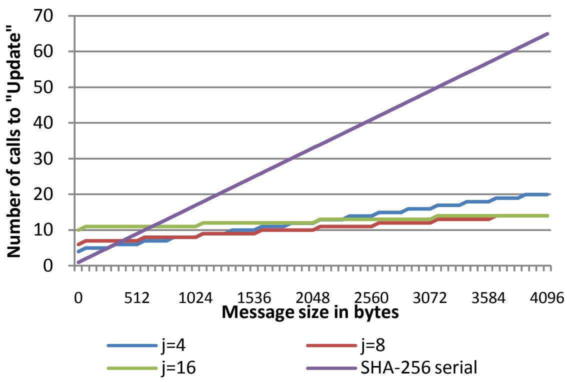

Figure 4. The number of serially computed Updates required on a SIMD architecture supporting 16 lanes (AVX512f—a future architecture), for different message lengths and different choices of *j*.

However, in **Figure 4**, we see the differences when the choice of $j = 16$ becomes the most efficient for message sizes of 4 KB and up, requiring the fewest Updates. For 4 KB messages, both $j = 16$ and $j = 8$ require 14 Updates, $j = 4$ requires 20 updates and the serial SHA-256 requires 65 Updates.

7. The *j*-Lanes Hash and the *j*-Pointers Hash with Different IVs

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

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

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

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 a j-lanes hash, and 0x1 for a j-pointers hash.
- HASH is the name of the underlying hash function, encoded as a string of ASCII characters. For SHA-256 we write HASH = "SHA256" or, as ASCII, 0x53, 0x48, 0x41, 0x32, 0x35, 0x36 (encoding "S" = 0x53, "H" = 0x48, "A" = 0x41 etc.).
- The number of characters (*NCHAR*) in the string that indicates HASH should be such that $NCHAR + 9 \le B$.

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:

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

Remark 5: SHA-256 allows hashing a message of any length less than 2^{64} bits. In the j-lanes/j-pointers modes, the length of the message should be less than $2^{64} - 512$ bits.

Pre-Computing the IVs

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

$$IV_i = \text{compress}\left(\text{Hash}IV, \text{Pre}_i\right); \; i = 0,1\ldots, j \tag{4}$$

Note that the Prefix blocks can also be viewed as a modification of HASH, to use the new IVs instead of a fixed IV. For convenience, denote the hash function that uses IV_i by $HASH'_i$. In that case the SHA-256 padding shall still accommodate the length of the prefix block.

With this notation, the j-lanes hashing can be expressed in terms of HASH' by:

$H_0 = HASH'_0$ ($Buff_0$, length ($Buff_0$))
$H_1 = HASH'_1$ ($Buff_1$, length ($Buff_1$))
$H_2 = HASH'_2$ ($Buff_2$, length ($Buff_2$))
...
$H_{j-1} = HASH'_{j-1}$ ($Buff_{j-1}$, length ($Buff_{j-1}$))
H = $HASH'_j$ ($H_0 \| H_1 \| H_2 \| \ldots \| H_{j-1}$, $j \times D$)

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

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 ($Buff_1$))
...

```
j = 4, type = j-lanes (0), HASH = "SHA256"
Pre0:
0400000000000000005348413235360000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000
Pre1:
0400000001000000005348413235360000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000
Pre2:
0400000002000000005348413235360000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000
Pre3:
0400000003000000005348413235360000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000Pre4:
0400000004000000005348413235360000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000
IV0 =
Presented as 8 integers:
0xf516dd7d 0xcc53773b 0x6a704b3e 0x89f00ca7 0x901d044b 0xa411be1d 0x8a947006 0xa758ccc1
Presented as a string of bytes:
7ddd16f53b7753cc3e4b706aa70cf0894b041d901dbe11a40670948ac1cc58a7IV1 =
Presented as 8 integers:
0x6f8070fd 0x6c2f2e6c 0x297ab335 0x6350bfd7 0x7b824607 0xf72e344b 0xcb5bc352 0x23210247
Presented as a string of bytes:
fd70806f6c2e2f6c35b37a29d7bf50630746827b4b342ef752c35bcb47022123IV2 =
Presented as 8 integers:
0x2940ec18 0x72886f93 0x5b5c5579 0x917315de 0x5696e2f0 0xcacb3551 0xd0b3e70b 0x007675ae
Presented as a string of bytes:
18ec4029936f887279555c5bde157391f0e296565135cbca0be7b3d0ae757600IV3 =
Presented as 8 integers:
0x496c7792 0xb05ad6ed 0x4c00f749 0x98d32ce4 0x363032ec 0x08eacd68 0x410b62b8 0x35a6fe0d
Presented as a string of bytes:
92776c49edd65ab049f7004ce42cd398ec32303668cdea08b8620b410dfea635IV4 =
Presented as 8 integers:
0x57899183 0x99b442ef 0xa5af28ed 0x27de1291 0xb2d00080 0x62ec261d 0xddbac391 0xba39fc75
Presented as a string of bytes:
085b642c34919f260d33b61a13cbd5d114650dee900bfb7915f3c5a004ade274
```

Figure 5. An example for the Prefix blocks and the IVs generation for the 4-lanes SHA-256 hash function.

H_{j-1} = HASH’$_{j+1}$(Buff_{j-1}, length (Buff_{j-1}))
H = HASH’$_{j+1}$($H_0||H_1||H_2||\ldots||H_{j-1}$, $j \times D$)
Figure 6 shows the values of the prefix block and the new IV (for HASH = SHA-256) for the alternative.
Test vectors for *j*-lanes SHA-256 with j = 4, 8, 16 are provided in the Appendix.

8. Performance

This section shows the 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 **Figure 7** and **Figure 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.

```
j = 4, type = j-lanes (0), HASH = "SHA256"
Pre5:
0400000005000000005348413235360000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000
IV5 =
Presented as 8 integers:
0x62d6641f 0x34eb3e8a 0x9e581e8c 0x481cef1c 0x5382d06c 0x0b2d9ace 0x2214143d 0xed1d7412
Presented as a string of bytes:
1f64d6628a3eeb348c1e589e1cef1c486cd08253ce9a2d0b3d14142212741ded
```

Figure 6. An example of 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.

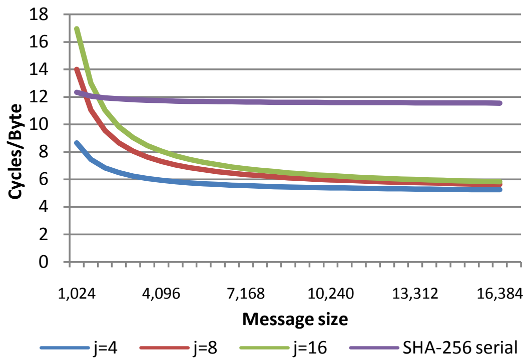

Figure 7. Performance of SHA-256 j-lanes compared to the serial SHA-256 implementation, Intel Architecture Codename Sandy Bridge (S = 4).

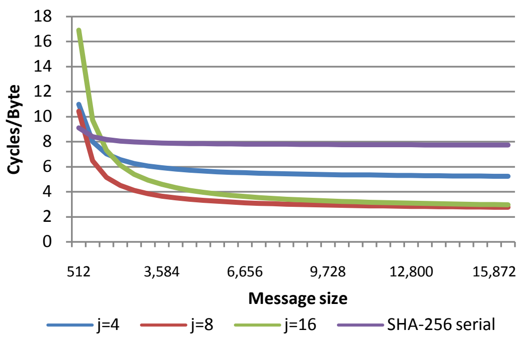

Figure 8. Performance of SHA-256 j-lanes compared to the serial SHA-256 implementation, Intel Architecture Codename Haswell (S = 8).

9. Conclusions

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 Figure 2 and Figure 3 (theoretical approximations) are consistent with Figure 7 and Figure 8 for j = 4 and j = 8 (actual measurements). Therefore, Figure 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] Gueron, S. (2013) A *j*-Lanes Tree Hashing Mode and j-Lanes SHA-256. *Journal of Information Security*, **4**, 7-11.

[2] FIPS (2012) Secure Hash Standard (SHS), Federal Information Processing Standards Publication 180-4. http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

[3] Intel (2013) Intel® Architecture Instruction Set Extensions Programming Reference. http://software.intel.com/en-us/file/319433-017pdf

[4] ARM (2013) Neon, ARM. http://www.arm.com/products/processors/technologies/neon.php

[5] Reinders, J. (2013) AVX-512 Instructions, Intel Developer Zone. 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 (**Figures 9-12**).
uint8_t M[1024];
for (int i = 0; i < 512 ; i++) {M [i * 2] = i >> 8; M [i * 2 + 1] = i & 0 × ff;}

```
The message M (1024 bytes):

0000000100020003000400050006000700080009000a000b000c000d000e000f0010001100120013001400
15001600170018001900 1a001b001c001d001e001f0020002100220023002400250026002700280029002a
002b002c002d002e002f0030003100320033003400350036003700380039003a003b003c003d003e003f00
4000410042004300440045004600470048004900 4a004b004c004d004e004f005000510052005300540055
0056005700580059005a005b005c005d005e005f006000610062006300640065006600670068006900 6a00
6b006c006d006e006f0070007100720073007400750076007700780079007a007b007c007d007e007f0080
0081008200830084008500860087008800 89008a008b008c008d008e008f0090009100920093009400 9500
96009700980099009a009b009c009d009e009f00a000a100a200a300a400a500a600a700a800a900aa00ab
00ac00ad00ae00af00b000b100b200b300b400b500b600b700b800b900ba00bb00bc00bd00be00bf00c000
c100c200c300c400c500c600c700c800c900ca00cb00cc00cd00ce00cf00d000d100d200d300d400d500d6
00d700d800d900da00db00dc00dd00de00df00e000e100e200e300e400e500e600e700e800e900ea00eb00
ec00ed00ee00ef00f000f100f200f300f400f500f600f700f800f900fa00fb00fc00fd00fe00ff01000101
0102010301040105010601070108010901 0a010b010c010d010e010f0110011101120113011401150116 01
1701180119011a011b011c011d011e011f0120012101220123012401250126012701280129012a012b012c
012d012e012f0130013101320133013401350136013701380139013a013b013c013d013e013f0140014101
42014301440145014601470148014901 4a014b014c014d014e014f01500151015201530154015501560157
01580159015a015b015c015d015e015f0160016101620163016401650166016701680169016a016b016c01
6d016e016f0170017101720173017401750176017701780179017a017b017c017d017e017f018001810182
0183018401850186018701880189018a018b018c018d018e018f0190019101920193019401950196019701
980199019a019b019c019d019e019f01a001a101a201a301a401a501a601a701a801a901aa01ab01ac01ad
01ae01af01b001b101b201b301b401b501b601b701b801b901ba01bb01bc01bd01be01bf01c001c101c201
c301c401c501c601c701c801c901ca01cb01cc01cd01ce01cf01d001d101d201d301d401d501d601d701d8
01d901da01db01dc01dd01de01df01e001e101e201e301e401e501e601e701e801e901ea01eb01ec01ed01
ee01ef01f001f101f201f301f401f501f601f701f801f901fa01fb01fc01fd01fe01ff
```

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

```
Lane 0 =
000000010002000300040005000600070008000900 0a000b000c000d000e000f0010001100120013001400150016001700
```

```
000000000000000000000000000000
IV2 =
Presented as 8 integers:
0x2940ec18 0x72886f93 0x5b5c5579 0x917315de 0x5696e2f0 0xcacb3551 0xd0b3e70b 0x007675ae
Presented as a string of bytes:
18ec4029936f887279555c5bde157391f0e296565135cbca0be7b3d0ae757600
H2 =
Presented as 8 integers:
0xa5bff793 0x54e0b9c7 0x38a4abf5 0xf51d6858 0xd4786561 0x51b0b779 0xf92c6680 0x62962ae6
Presented as a string of bytes:
93f7bfa5c7b9e054f5aba43858681df5616578d479b7b05180662cf9e62a9662

Lane 3 =
00600061006200630064006500660067006800690 06a006b006c006d006e006f0070007100720073007400750076007700
780079007a007b007c007d007e007f00e000e100e200e300e400e500e600e700e800e900ea00eb00ec00ed00ee00ef00f0
00f100f200f300f400f500f600f700f800f900fa00fb00fc00fd00fe00ff016001610162016301640165016601670168 01
69016a016b016c016d016e016f017001710172017301740175017601770178017901 7a017b017c017d017e017f01e001e1
01e201e301e401e501e601e701e801e901ea01eb01ec01ed01ee01ef01f001f101f201f301f401f501f601f701f801f901
fa01fb01fc01fd01fe01ff
J = 4, idx = 3, type = 0, Pre3 =
04000000030000000053484132353600000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000
IV3 =
Presented as 8 integers:
0x496c7792 0xb05ad6ed 0x4c00f749 0x98d32ce4 0x363032ec 0x08eacd68 0x410b62b8 0x35a6fe0d
Presented as a string of bytes:
92776c49edd65ab049f7004ce42cd398ec32303668cdea08b8620b410dfea635
H3 =
Presented as 8 integers:
0xda669dfe 0x86fabd5e 0xc9bacdf8 0x1452d42d 0x51daf0a3 0x0e072407 0x4b1e0240 0xc5b4fd16
Presented as a string of bytes:
fe9d66da5ebdfa86f8cdbac92dd45214a3f0da510724070e40021e4b16fdb4c5

The wrapping string (the concatenation of j digests) =
a291b60c1c93e74c55901e2ba918a5b6809ae2b58de7f79629a6f9be3166231cfb4c3a01163982a802a6c26dfd24db111a
e3b4c2f9f50862ef9809e1ff2a25c393f7bfa5c7b9e054f5aba43858681df5616578d479b7b05180662cf9e62a9662fe9d
66da5ebdfa86f8cdbac92dd45214a3f0da510724070e40021e4b16fdb4c5
J = 4, idx = 4, type = 0, Pre4 =
04000000040000000053484132353600000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000
IV4 =
Presented as 8 integers:
0x57899183 0x99b442ef 0xa5af28ed 0x27de1291 0xb2d00080 0x62ec261d 0xddbac391 0xba39fc75
Presented as a string of bytes:
83918957ef42b499ed28afa59112de278000d0b21d26ec6291c3badd75fc39ba
The output digests, H =
Presented as 8 integers:
0x2c645b08 0x269f9134 0x1ab6330d 0xd1d5cb13 0xee0d6514 0x79fb0b90 0xa0c5f315 0x74e2ad04
Presented as a string of bytes:
085b642c34919f260d33b61a13cbd5d114650dee900bfb7915f3c5a004ade274
```

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

```
Lane 0 =
0000000100020003000400050006000700080009000a000b000c000d000e000f0010001100120013001400
15001600170018001900 1a001b001c001d001e001f0100010101020103010401050106010701080109010a
010b010c010d010e010f0110011101120113011401150116011701180119011a011b011c011d011e011f
J = 8, idx = 0, type = 0, Pre0 =
08000000000000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV0 =
Presented as 8 integers:
0x787f6051 0x684c02c0 0xde7ccd48 0x2c6382de 0x903f8cc0 0x74c60570 0xd8e5e679 0xfcad483d
Presented as a string of bytes:
51607f78c0024c6848cd7cdede82632cc08c3f907005c67479e6e5d83d48adfc
H0 =
Presented as 8 integers:
0xd90a9208 0xb1cd8603 0x967e141c 0x9dc938f7 0x28005edc 0x549a7429 0xac6c2d6f 0x576bd8b1
Presented as a string of bytes:
08920ad90386cdb11c147e96f738c99ddc5e002829749a546f2d6cacb1d86b57


Lane 1 =
0020002100220023002400250026002700280029002a002b002c002d002e002f0030003100320033003400
350036003700380039003a003b003c003d003e003f0120012101220123012401250126012701280129012a
012b012c012d012e012f0130013101320133013401350136013701380139013a013b013c013d013e013f
J = 8, idx = 1, type = 0, Pre1 =
08000000010000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV1 =
Presented as 8 integers:
0x39fa5544 0x74d24640 0xf0435922 0xcd1f50b4 0xdfd3eaf6 0x4f295f3a 0xcebedb2a 0xe3126408
Presented as a string of bytes:
4455fa394046d274225943f0b4501fcdf6ead3df3a5f294f2adbbece086412e3
H1 =
Presented as 8 integers:
0x86753c04 0xa3825b56 0xd9dcaa47 0xf84d0f91 0x1b412197 0x1135f42b 0x953a6ba1 0x30d5f9b4
Presented as a string of bytes:
043c7586565b82a347aadcd9910f4df89721411b2bf43511a16b3a95b4f9d530


Lane 2 =
0040004100420043004400450046004700480049004a004b004c004d004e004f0050005100520053005400
550056005700580059005a005b005c005d005e005f0140014101420143014401450146014701480149014a
014b014c014d014e014f0150015101520153015401550156015701580159015a015b015c015d015e015f
J = 8, idx = 2, type = 0, Pre2 =
```

```
08000000020000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV2 =
Presented as 8 integers:
0x6662dd71 0x809cbd72 0x3fe09a5f 0xb75372fa 0x87ef7577 0x7e317792 0x010d9ccf 0xb474ba3b
Presented as a string of bytes:
71dd626672bd9c805f9ae03ffa7253b77775ef879277317ecf9c0d013bba74b4
H2 =
Presented as 8 integers:
0x5181afc2 0x8118f2e6 0x054e5ab0 0xcf2d001e 0x15ad7615 0x6e57d085 0x49d20875 0x315f1180
Presented as a string of bytes:
c2af8151e6f21881b05a4e051e002dcf1576ad1585d0576e7508d24980115f31


Lane 3 =
00600061006200630064006500660067006800690 06a006b006c006d006e006f0070007100720073007400
75007600770078007900 7a007b007c007d007e007f01600161016201630164016501660167016801690 16a
016b016c016d016e016f01700171017201730174017501760177017801790 17a017b017c017d017e017f
J = 8, idx = 3, type = 0, Pre3 =
08000000030000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV3 =
Presented as 8 integers:
0x11ae33b4 0x5dc9d2c8 0xa5f61621 0x48adaed6 0x4baf1946 0xf0092642 0x9202b4da 0x023111fe
Presented as a string of bytes:
b433ae11c8d2c95d2116f6a5d6aead484619af4b422609f0dab40292fe113102
H3 =
Presented as 8 integers:
0xbe9d443c 0x4a550840 0xb28919fd 0x52502fa2 0x226211c8 0x911f847c 0xb97ca0a5 0xa20e2a6a
Presented as a string of bytes:
3c449dbe4008554afd1989b2a22f5052c81162227c841f91a5a07cb96a2a0ea2


Lane 4 =
00800081008200830084008500860087008800890 08a008b008c008d008e008f0090009100920093009400
95009600970098009900 9a009b009c009d009e009f01800181018201830184018501860187018801890 18a
018b018c018d018e018f01900191019201930194019501960197019801990 19a019b019c019d019e019f
J = 8, idx = 4, type = 0, Pre4 =
08000000040000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV4 =
Presented as 8 integers:
0x08ed680a 0xc788f0c0 0x497d599e 0x79c12056 0xf9435c3d 0x28bdd5b0 0xd953912f 0xbe006e9f
```

```
Presented as a string of bytes:

0a68ed08c0f088c79e597d495620c1793d5c43f9b0d5bd282f9153d99f6e00be

H4 =

Presented as 8 integers:

0x971d4f80 0x536dc3ff 0xddbae5f2 0x1aa7f7b9 0x07a9061f 0xbbe4ba2b 0xc7e941b8 0x76fdddf7

Presented as a string of bytes:

804f1d97ffc36d53f2e5baddb9f7a71a1f06a9072bbae4bbb841e9c7f7ddfd76


Lane 5 =

00a000a100a200a300a400a500a600a700a800a900aa00ab00ac00ad00ae00af00b000b100b200b300b400
b500b600b700b800b900ba00bb00bc00bd00be00bf01a001a101a201a301a401a501a601a701a801a901aa
01ab01ac01ad01ae01af01b001b101b201b301b401b501b601b701b801b901ba01bb01bc01bd01be01bf

J = 8, idx = 5, type = 0, Pre5 =

08000000050000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000

IV5 =

Presented as 8 integers:

0xb29eccbd 0xb82ec5ca 0x65166adb 0x85526bfb 0xfca492f1 0x12d2b13d 0xd9b715d1 0xcaea6a44

Presented as a string of bytes:

bdcc9eb2cac52eb8db6a1665fb6b5285f192a4fc3db1d212d115b7d9446aeaca

H5 =

Presented as 8 integers:

0xbf2dbb52 0x59e79dd7 0x8b93e78b 0xf5ddb28c 0x9cd2635e 0xddd27a82 0x80e55ea7 0xa013de40

Presented as a string of bytes:

52bb2dbfd79de7598be7938b8cb2ddf55e63d29c827ad2dda75ee58040de13a0


Lane 6 =

00c000c100c200c300c400c500c600c700c800c900ca00cb00cc00cd00ce00cf00d000d100d200d300d400
d500d600d700d800d900da00db00dc00dd00de00df01c001c101c201c301c401c501c601c701c801c901ca
01cb01cc01cd01ce01cf01d001d101d201d301d401d501d601d701d801d901da01db01dc01dd01de01df

J = 8, idx = 6, type = 0, Pre6 =

08000000060000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000

IV6 =

Presented as 8 integers:

0x0137b128 0xe3aed4ba 0x456f7743 0x591aa3d8 0xc86940d9 0x53ada152 0xdadc486c 0x204e367c

Presented as a string of bytes:

28b13701bad4aee343776f45d8a31a59d94069c852a1ad536c48dcda7c364e20

H6 =

Presented as 8 integers:

0xae024e7e 0x4c2098ba 0x6de0a414 0x35294e44 0x9caf1cbe 0xd9cf1cf0 0x70e9fc43 0x1e3f2d49
```

```
Presented as a string of bytes:
7e4e02aeba98204c14a4e06d444e2935be1caf9cf01ccfd943fce970492d3f1e

Lane 7 =
00e000e100e200e300e400e500e600e700e800e900ea00eb00ec00ed00ee00ef00f000f100f200f300f400
f500f600f700f800f900fa00fb00fc00fd00fe00ff01e001e101e201e301e401e501e601e701e801e901ea
01eb01ec01ed01ee01ef01f001f101f201f301f401f501f601f701f801f901fa01fb01fc01fd01fe01ff
J = 8, idx = 7, type = 0, Pre7 =
08000000070000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV7 =
Presented as 8 integers:
0x4980b595 0x4efa42bb 0x73b812b9 0xd67c0cdd 0x1076165b 0x954dd185 0x62848d5f 0xb14ab123
Presented as a string of bytes:
95b58049bb42fa4eb912b873dd0c7cd65b16761085d14d955f8d846223b14ab1
H7 =
Presented as 8 integers:
0x247779f2 0x88b7a8b4 0x7b18e457 0x6328bc2a 0x5c2903da 0x54028aa2 0x5c284bbe 0x402847b1
Presented as a string of bytes:
f2797724b4a8b78857e4187b2abc2863da03295ca28a0254be4b285cb1472840

The wrapping string (the concatenation of j digests) =
08920ad90386cdb11c147e96f738c99ddc5e002829749a546f2d6cacb1d86b57043c7586565b82a347aadc
d9910f4df89721411b2bf43511a16b3a95b4f9d530c2af8151e6f21881b05a4e051e002dcf1576ad1585d0
576e7508d24980115f313c449dbe4008554afd1989b2a22f5052c81162227c841f91a5a07cb96a2a0ea280
4f1d97ffc36d53f2e5baddb9f7a71a1f06a9072bbae4bbb841e9c7f7ddfd7652bb2dbfd79de7598be7938b
8cb2ddf55e63d29c827ad2dda75ee58040de13a07e4e02aeba98204c14a4e06d444e2935be1caf9cf01ccf
d943fce970492d3f1ef2797724b4a8b78857e4187b2abc2863da03295ca28a0254be4b285cb1472840
J = 8, idx = 8, type = 0, Pre8 =
08000000080000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV8 =
Presented as 8 integers:
0x1caa6939 0x5843a5d3 0xd55f3568 0x9f8b2a9e 0xcd717b92 0xe47c03de 0xa5452624 0xea38329a
Presented as a string of bytes:
3969aa1cd3a5435868355fd59e2a8b9f927b71cdde037ce4242645a59a3238ea
The output digests, H =
Presented as 8 integers:
0xfc872de3 0x5d1ecbd8 0x49305e5e 0xc00977ed 0xc7baa31a 0xe5093d7d 0xf698fd6c 0x22dfe516
Presented as a string of bytes:
e32d87fcd8cb1e5d5e5e3049ed7709c01aa3bac77d3d09e56cfd98f616e5df22
```

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

```
Lane 0 =
0000000100020003000400050006000700080009000a000b000c000d000e000f0010001100120013001400
150016001700180019001a001b001c001d001e001f
J = 16, idx = 0, type = 0, Pre0 =
10000000000000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV0 =
Presented as 8 integers:
0x301b4698 0x421bd4dc 0xa10cc1ce 0x486bb23c 0x51378b93 0x3eecd201 0x9d59a094 0x1bad62fc
Presented as a string of bytes:
98461b30dcd41b42cec10ca13cb26b48938b375101d2ec3e94a0599dfc62ad1b
H0 =
Presented as 8 integers:
0x472d67f1 0x3ed556ca 0x88516bfb 0x0adaae63 0x43af34ce 0x0353eaab 0xb01635dc 0x3af7d38d
Presented as a string of bytes:
f1672d47ca56d53efb6b518863aeda0ace34af43abea5303dc3516b08dd3f73a

Lane 1 =
0020002100220023002400250026002700280029002a002b002c002d002e002f0030003100320033003400
350036003700380039003a003b003c003d003e003f
J = 16, idx = 1, type = 0, Pre1 =
10000000010000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV1 =
Presented as 8 integers:
0xe584be20 0x6be7e96c 0x59d99116 0xcd3cd22e 0xbb9d678c 0x782af33d 0xacec61ac 0x6eedcea4
Presented as a string of bytes:
20be84e56ce9e76b1691d9592ed23ccd8c679dbb3df32a78ac61ecaca4ceed6e
H1 =
Presented as 8 integers:
0x383545a8 0xebccaf84 0xe25793bf 0xaf8d34d3 0x8f07e023 0xf71a2ab5 0x663d4152 0x8798c2ef
Presented as a string of bytes:
a845353884afccebbf9357e2d3348daf23e0078fb52a1af752413d66efc29887

Lane 2 =
0040004100420043004400450046004700480049004a004b004c004d004e004f0050005100520053005400
550056005700580059005a005b005c005d005e005f
J = 16, idx = 2, type = 0, Pre2 =
10000000020000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV2 =
Presented as 8 integers:
0x83d278b8 0x5b8e8240 0x2b4df96c 0xafbeb60a 0xf1a519ee 0xab829302 0xe518a3d9 0x55be0fbd
Presented as a string of bytes:
b878d28340828e5b6cf94d2b0ab6beafee19a5f1029382abd9a318e5bd0fbe55
H2 =
Presented as 8 integers:
0x824307a9 0xda8acc6c 0xffeb1ed6 0xf3315f02 0xb4bb635e 0xafee9d3d 0xbcc2b49d 0x42c3daae
Presented as a string of bytes:
a90743826ccc8adad61eebff025f31f35e63bbb43d9deeaf9db4c2bcaedac342
```

```
Lane 3 =
00600061006200630064006500660067006800690060a006b006c006d006e006f00700071007200730074000
750076007700780079007a007b007c007d007e007f
J = 16, idx = 3, type = 0, Pre3 =
10000000030000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV3 =
Presented as 8 integers:
0x1b5600c7 0x3c12c7a0 0x31df144c 0xd105f19d 0xc5106b02 0xeeb323de 0xb808d185 0xfb6f6550
Presented as a string of bytes:
c700561ba0c7123c4c14df319df105d1026b10c5de23b3ee85d108b850656ffb
H3 =
Presented as 8 integers:
0x419b79f5 0x7e42bc10 0xaf93b5a6 0xa07fc24f 0x2441a0c1 0xe8427787 0xaa3a4d22 0x590e2dbb
Presented as a string of bytes:
f5799b4110bc427ea6b593af4fc27fa0c1a04124877742e8224d3aaabb2d0e59

Lane 4 =
0080008100820083008400850086008700880089008a008b008c008d008e008f00900091009200930094 00
950096009700980099009a009b009c009d009e009f
J = 16, idx = 4, type = 0, Pre4 =
10000000040000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV4 =
Presented as 8 integers:
0x270865f5 0xda90a5e7 0x004ed5ac 0xb399cf28 0x598b21a3 0x4a1b7bd4 0xb298a277 0x0c9d36d9
Presented as a string of bytes:
f5650827e7a590daacd54e0028cf99b3a3218b59d47b1b4a77a298b2d9369d0c
H4 =
Presented as 8 integers:
0x0bdea332 0x489e23c0 0x489f243d 0x08b4404b 0xfdbda480 0x9f019c35 0x0e98ec17 0x1788130e
Presented as a string of bytes:
32a3de0bc0239e483d249f484b40b40880a4bdfd359c019f17ec980e0e138817

Lane 5 =
00a000a100a200a300a400a500a600a700a800a900aa00ab00ac00ad00ae00af00b000b100b200b300b400
b500b600b700b800b900ba00bb00bc00bd00be00bf
J = 16, idx = 5, type = 0, Pre5 =
10000000050000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV5 =
Presented as 8 integers:
0x33aeaf06 0x9d5efd54 0xa98aa21e 0x8df52647 0x730bafe4 0x4e3076af 0xe16e9154 0xbd1d7f07
Presented as a string of bytes:
06afae3354fd5e9d1ea28aa94726f58de4af0b73af76304e54916ee1077f1dbd
H5 =
Presented as 8 integers:
0xb02d93fc 0xe29f0086 0x84fc3565 0x1f300e86 0x1bf43a85 0x71f91ac8 0xd9742ec0 0x179312e5
Presented as a string of bytes:
```

```
fc932db086009fe26535fc84860e301f853af41bc81af971c02e74d9e5129317

Lane 6 =
00c000c100c200c300c400c500c600c700c800c900ca00cb00cc00cd00ce00cf00d000d100d200d300d400
d500d600d700d800d900da00db00dc00dd00de00df
J = 16, idx = 6, type = 0, Pre6 =
10000000060000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV6 =
Presented as 8 integers:
0x03205e22 0x345b8bdb 0xdf24e1ff 0x249e2c65 0xa8c30e39 0x77f91a58 0xe1cb85a4 0x3b6c7448
Presented as a string of bytes:
225e2003db8b5b34ffe124df652c9e24390ec3a8581af977a485cbe148746c3b
H6 =
Presented as 8 integers:
0x83af3937 0x7f30aaf8 0x45fb16ed 0x49ef08ca 0x61a19f7a 0x21b2ecc4 0x2676295c 0x22b1cde4
Presented as a string of bytes:
3739af83f8aa307fed16fb45ca08ef497a9fa161c4ecb2215c297626e4cdb122

Lane 7 =
00e000e100e200e300e400e500e600e700e800e900ea00eb00ec00ed00ee00ef00f000f100f200f300f400
f500f600f700f800f900fa00fb00fc00fd00fe00ff
J = 16, idx = 7, type = 0, Pre7 =
10000000070000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV7 =
Presented as 8 integers:
0x4744b3b3 0xdb42b4ed 0xabe499fb 0xacf298a4 0x929e92ae 0x71c071dc 0xc091cbf5 0xaf33d91f
Presented as a string of bytes:
b3b34447edb442dbfb99e4aba498f2acae929e92dc71c071f5cb91c01fd933af
H7 =
Presented as 8 integers:
0x17fc7d87 0x91385485 0x5892e618 0x2fe0f492 0x4914a63d 0x8e3b87f1 0x24f2a715 0x648e0065
Presented as a string of bytes:
877dfc178554389118e6925892f4e02f3da61449f1873b8e15a7f22465008e64

Lane 8 =
0100010101020103010401050106010701080109010a010b010c010d010e010f0110011101120113011401
150116011701180119011a011b011c011d011e011f
J = 16, idx = 8, type = 0, Pre8 =
10000000080000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV8 =
Presented as 8 integers:
0x88d9773e 0x227b996c 0xb045c986 0x0c6568ca 0x5a31e35f 0xd68a998b 0xa8125b79 0x5e2c81e6
Presented as a string of bytes:
3e77d9886c997b2286c945b0ca68650c5fe3315a8b998ad6795b12a8e6812c5e
H8 =
Presented as 8 integers:
0xd1b89671 0x8ce702c4 0xaf5a504c 0xa8c425fa 0x8d44bc0b 0x2407529d 0xb06eeb14 0x4494bc66
```

```
Presented as a string of bytes:
7196b8d1c402e78c4c505aaffa25c4a80bbc448d9d52072414eb6eb066bc9444

Lane 9 =
0120012101220123012401250126012701280129012a012b012c012d012e012f0130013101320133013401
350136013701380139013a013b013c013d013e013f
J = 16, idx = 9, type = 0, Pre9 =
10000000090000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV9 =
Presented as 8 integers:
0x6dcb53a7 0x91aa3069 0xf594281f 0x5e3c2a06 0xd9d6474b 0x7b5ff762 0x4a96c1ec 0x1cfdbc57
Presented as a string of bytes:
a753cb6d6930aa911f2894f5062a3c5e4b47d6d962f75f7becc1964a57bcfd1c
H9 =
Presented as 8 integers:
0x7aed3aae 0xcc9df426 0x49f39ec1 0xebad8048 0x931b5909 0xd0c15bd9 0x46bb8d14 0x38daf8f2
Presented as a string of bytes:
ae3aed7a26f49dccc19ef3494880adeb09591b93d95bc1d0148dbb46f2f8da38

Lane 10 =
0140014101420143014401450146014701480149014a014b014c014d014e014f0150015101520153015401
550156015701580159015a015b015c015d015e015f
J = 16, idx = 10, type = 0, Pre10 =
100000000a0000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV10 =
Presented as 8 integers:
0x6e7be156 0x9a9e3d68 0x835a8de7 0xbc56536d 0x010759e0 0xeda1e726 0x60cf113a 0xd3b9d737
Presented as a string of bytes:
56e17b6e683d9e9ae78d5a836d5356bce059070126e7a1ed3a11cf6037d7b9d3
H10 =
Presented as 8 integers:
0x66755c29 0xa3a52b52 0xe9d80979 0x1482e24d 0x2c11e40f 0x9302cbc4 0x09b76d4a 0x3816530d
Presented as a string of bytes:
295c7566522ba5a37909d8e94de282140fe4112cc4cb02934a6db7090d531638

Lane 11 =
0160016101620163016401650166016701680169016a016b016c016d016e016f0170017101720173017401
750176017701780179017a017b017c017d017e017f
J = 16, idx = 11, type = 0, Pre11 =
100000000b0000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV11 =
Presented as 8 integers:
0x8cbccae1 0x616bdba3 0xf21abde3 0x551fe2d4 0x0a846720 0x318a5e23 0xcf10df64 0xd3830b54
Presented as a string of bytes:
e1cabc8ca3db6b61e3bd1af2d4e21f552067840a235e8a3164df10cf540b83d3
H11 =
Presented as 8 integers:
```

```
0xf5a975a1 0xc0f9e9e0 0x1a33230f 0x283d99b6 0xdfdf95fb 0x2563b811 0x47e85a61 0x777a92bd
Presented as a string of bytes:
a175a9f5e0e9f9c00f23331ab6993d28fb95dfdf11b86325615ae847bd927a77

Lane 12 =
0180018101820183018401850186018701880189018a018b018c018d018e018f0190019101920193019401
95019601970198019901 9a019b019c019d019e019f
J = 16, idx = 12, type = 0, Pre12 =
100000000c00000000534841323536000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV12 =
Presented as 8 integers:
0xf30aea08 0x6327cb9c 0xe090d724 0xf55be1b1 0x356bac3a 0x259e2a90 0x1474593e 0x317eb9f7
Presented as a string of bytes:
08ea0af39ccb276324d790e0b1e15bf53aac6b35902a9e253e597414f7b97e31
H12 =
Presented as 8 integers:
0x60aa4972 0xb64cf665 0x01e11d8d 0x17a2408b 0xd43ad3b3 0x43c75ac6 0xfcb5a860 0x32fa397d
Presented as a string of bytes:
7249aa6065f64cb68d1de1018b40a217b3d33ad4c65ac74360a8b5fc7d39fa32

Lane 13 =
01a001a101a201a301a401a501a601a701a801a901aa01ab01ac01ad01ae01af01b001b101b201b301b401
b501b601b701b801b901ba01bb01bc01bd01be01bf
J = 16, idx = 13, type = 0, Pre13 =
100000000d00000000534841323536000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV13 =
Presented as 8 integers:
0xeea5ddcb 0xecdc55c4 0xd8554d0a 0xa11207d1 0xe654142b 0xa29989f6 0xdab88e31 0xc66e5305
Presented as a string of bytes:
cbdda5eec455dcec0a4d55d8d10712a12b1454e6f68999a2318eb8da05536ec6
H13 =
Presented as 8 integers:
0xcc4ea4d6 0xe06b3d10 0x9c7cb023 0x36856036 0x6fb62968 0xd7d29ca1 0x62f748f4 0x54b4753f
Presented as a string of bytes:
d6a44ecc103d6be023b07c9c366085366829b66fa19cd2d7f448f7623f75b454

Lane 14 =
01c001c101c201c301c401c501c601c701c801c901ca01cb01cc01cd01ce01cf01d001d101d201d301d401
d501d601d701d801d901da01db01dc01dd01de01df
J = 16, idx = 14, type = 0, Pre14 =
100000000e00000000534841323536000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV14 =
Presented as 8 integers:
0x36c346a5 0x715bbcbc 0xfd5cf994 0x6c4d16b7 0x1a19c99f 0x4c777619 0x412a6300 0x89b25e19
Presented as a string of bytes:
a546c336bcbc5b7194f95cfdb7164d6c9fc9191a1976774c00632a41195eb289
H14 =
```

```
Presented as 8 integers:
0x1d3bf0a0 0x7d399bf7 0xe64b64c0 0xf7ba2434 0xa409ab5d 0x7870f632 0x82e76833 0xc64bcca1
Presented as a string of bytes:
a0f03b1df79b397dc0644be63424baf75dab09a432f670783368e782a1cc4bc6

Lane 15 =
01e001e101e201e301e401e501e601e701e801e901ea01eb01ec01ed01ee01ef01f001f101f201f301f401
f501f601f701f801f901fa01fb01fc01fd01fe01ff
J = 16, idx = 15, type = 0, Pre15 =
100000000f00000000534841323536000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV15 =
Presented as 8 integers:
0xe9f3a324 0x739f1383 0x06e52eaf 0x6608f137 0x61e1a290 0x688ae5a8 0xca111224 0x66b3d938
Presented as a string of bytes:
24a3f3e983139f73af2ee50637f1086690a2e161a8e58a68241211ca38d9b366
H15 =
Presented as 8 integers:
0x8d0f2b54 0x743ac7f1 0x4fd432ef 0x56074b6b 0xa9a16d82 0xb61f363a 0x81518dee 0xd7f215cb
Presented as a string of bytes:
542b0f8df1c73a74ef32d44f6b4b0756826da1a93a361fb6ee8d5181cb15f2d7

The wrapping string (the concatenation of j digests) =
f1672d47ca56d53efb6b518863aeda0ace34af43abea5303dc3516b08dd3f73aa845353884afccebbf9357
e2d3348daf23e0078fb52a1af752413d66efc29887a90743826ccc8adad61eebff025f31f35e63bbb43d9d
eeaf9db4c2bcaedac342f5799b4110bc427ea6b593af4fc27fa0c1a04124877742e8224d3aaabb2d0e5932
a3de0bc0239e483d249f484b40b40880a4bdfd359c019f17ec980e0e138817fc932db086009fe26535fc84
860e301f853af41bc81af971c02e74d9e51293173739af83f8aa307fed16fb45ca08ef497a9fa161c4ecb2
215c297626e4cdb122877dfc178554389118e6925892f4e02f3da61449f1873b8e15a7f22465008e647196
b8d1c402e78c4c505aaffa25c4a80bbc448d9d52072414eb6eb066bc9444ae3aed7a26f49dccc19ef34948
80adeb09591b93d95bc1d0148dbb46f2f8da38295c7566522ba5a37909d8e94de282140fe4112cc4cb0293
4a6db7090d531638a175a9f5e0e9f9c00f23331ab6993d28fb95dfdf11b86325615ae847bd927a777249aa
6065f64cb68d1de1018b40a217b3d33ad4c65ac74360a8b5fc7d39fa32d6a44ecc103d6be023b07c9c3660
85366829b66fa19cd2d7f448f7623f75b454a0f03b1df79b397dc0644be63424baf75dab09a432f6707833
68e782a1cc4bc6542b0f8df1c73a74ef32d44f6b4b0756826da1a93a361fb6ee8d5181cb15f2d7
J = 16, idx = 16, type = 0, Pre16 =
10000000100000000053484132353600000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
IV16 =
Presented as 8 integers:
0xd60422aa 0x2c91c9e5 0xf0e28121 0xeda94d92 0xf24c4a68 0xbf8ab0b9 0x4d118432 0x8354983d
Presented as a string of bytes:
aa2204d6e5c9912c2181e2f0924da9ed684a4cf2b9b08abf3284114d3d985483
The output digests, H =
Presented as 8 integers:
0xf984dec6 0x48df8956 0x50f32833 0x638b076b 0xe4c18b61 0x887a9f35 0xa9ee17d3 0x6668d586
Presented as a string of bytes:
c6de84f95689df483328f3506b078b63618bc1e4359f7a88d317eea986d56866
```

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