The Hash Function JH

Hongjun Wu

Institute for Infocomm Research, Singapore
wuhongjun@gmail.com

Contents

1 Introduction

This document specifies four hash algorithms – JH-224, JH-256, JH-384, and JH-512. The hash algorithms are very simple. They are efficient on many platforms ranging from one-bit processor (hardware) to 128-bit processor (SSE2 registers) since they are built on extremely simple components.

The JH hash functions are very efficient in software. With bit-slice implementation using SSE2, the speed of JH is about 16.8 clock cycles/byte on the Intel Core 2 Duo microprocessor running 64-bit operating system (using C implementation and Intel C++ compiler).

The memory required for the hardware implementation of JH hash functions is 1536 bits. With 256 additional memory bits, the round constants of JH can be generated on the fly. JH-224, JH-256, JH-384 and JH-512 share the same compression function, so it is very efficient to implement these four hash algorithms altogether in hardware.

JH is strong in security. Each message block is 64 bytes. A message block passes through the 35.5-round compression function that involves 9216 4-bit-to-4-bit Sboxes. We found that any differential trail in the compression function involves around 600 active Sboxes. The large number of active Sboxes ensures that JH is strong against differential attack [1].

This document is organized as follows. The specifications of JH are given in Sec. 3, 4, 5 and 6. The bit-slice implementation of JH is given in Sec. 7. Section 9 gives the security analysis of JH. The performance of JH is described in Sec. 10. The design rationale and advantage are given in Sec. 11 and Sec. 12, respectively. Sec. 13 concludes this document.

2 Efficient Differential Propagation Design

The compression function in JH is based on a bijective function. The Efficient Differential Propagation (EDP) design is used to design the bijective function in JH. EDP design uses the substitution-permutation network (SPN). The input bits are divided into $\alpha \times 2^d$ elements, and these elements form a d-dimensional array. In the linear layer of the r-th round, Maximum Distance Separable (MDS) code is applied along the $(r \bmod d)$-th dimension. We believe that such design is the simplest approach to achieve efficient differential propagation.

EDP design is not new. AES (Rijndael [6]) is based on EDP design with a two-dimensional array. However, Rijndael with 192-bit and 256-bit block sizes are not based on EDP design since MDS code is not applied to the dimension with 6 (192-bit block size) or 8 (256-bit block size) elements.

We use an eight-dimensional EDP design in the design of the bijective function in JH. The 1024 input bits to the bijective function are divided into 2^8 4-bit elements, and these elements form an eight-dimensional array.

3 Definitions

3.1 Notations

The following notations are used in the JH specifications.

Word	A group of bits.
A^i	The i^{th} bit in the word A. An m-bit word A is represented as $A = A^0 \parallel A^1 \parallel A^2 \parallel \cdots \parallel A^{m-1}$.

3.2 Parameters

The following parameters are used in the JH specifications.

$C_r^{(d)}$	The round constant words used in function E_d with $0 \leq r \leq 5 \times (d-1)$. Each $C_r^{(d)}$ is a 2^d-bit constant word.
d	The dimension of a block of bits. A d-dimensional block consists of 2^d 4-bit elements.
h	Number of bits in a hash value. $h = 1024$.
$H^{(i)}$	The i^{th} hash value, with a size of h bits. $H^{(0)}$ is the initial hash value; $H^{(N)}$ is the final hash value and is truncated to generate the message digest.
$H^{(i),j}$	The j^{th} bit of the i^{th} hash value, where $H^{(i)} = H^{(i),0} \| H^{(i),1} \| \cdots \| M^{(i),h-1}$.
ℓ	Length of the message, M, in bits.
m	Number of bits in a message block $M^{(i)}$. $m = 512$.
M	Message to be hashed.
$M^{(i)}$	Message block i, with a size of m bits.
$M^{(i),j}$	The j^{th} bit of the i^{th} message block, i.e., $M^{(i)} = M^{(i),0} \| M^{(i),1} \| \cdots \| M^{(i),m-1}$.
N	Number of blocks in the padded message.

3.3 Operations

The following operations are used in the JH specifications.

$\&$	Bitwise AND operation.
$\vert$	Bitwise OR ("inclusive–OR") operation.
$\oplus$	Bitwise XOR ("exclusive–OR") operation.
$\neg$	Bitwise complement operation.
$\|$	Concatenation operation.

4 Functions

The following functions are used in the JH specifications.

4.1 S-boxes

S_0 and S_1 are the 4-bit-to-4-bit S-boxes being used in JH. Every round constant bit selects which Sboxes are used (similar to Lucifer [9]).

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$S_0(x)$	9	0	4	11	13	12	3	15	1	10	2	6	7	5	8	14
$S_1(x)$	3	12	6	13	5	7	1	9	15	2	0	4	11	10	14	8

4.2 Linear transformation L

The linear transformation L implements a (4, 2, 3) Maximum Distance Separable (MDS) code over $GF(2^4)$. Here the multiplication in $GF(2^4)$ is defined as the multiplication of binary polynomials modulo the irreducible polynomial $x^4 + x + 1$. Denote this multiplication as '$\bullet$' .

Let A, B, C and D denote 4-bit words. L transforms (A, B) into (C, D) as

$$(C\,, D) = L(A\,, B) = (5\bullet A + 2\bullet B\,,\, 2\bullet A + B)\,.$$

More specifically, the bit-wise computation of L is given as follows. Let A, B, C and D denote 4-bit words, i.e., $A = A^0 \,\|\, A^1 \,\|\, A^2 \,\|\, A^3$, $B = B^0 \,\|\, B^1 \,\|\, B^2 \,\|\, B^3$, $C = C^0 \,\|\, C^1 \,\|\, C^2 \,\|\, C^3$, and $D = D^0 \,\|\, D^1 \,\|\, D^2 \,\|\, D^3$. In polynomial form, A is represented as $A^0x^3 + A^1x^2 + A^2x + A^3$; $2 \bullet A$ is given as $A^1x^3 + A^2x^2 + (A^0 + A^3)x + A^0$. The function $(C, D) = L(A, B)$ is computed as:

$$\begin{aligned}
D^0 &= B^0 \oplus A^1\,; & D^1 &= B^1 \oplus A^2\,;\\
D^2 &= B^2 \oplus A^3 \oplus A^0\,; & D^3 &= B^3 \oplus A^0\,;\\
C^0 &= A^0 \oplus D^1\,; & C^1 &= A^1 \oplus D^2\,;\\
C^2 &= A^2 \oplus D^3 \oplus D^0\,; & C^3 &= A^3 \oplus D^0\,.
\end{aligned}$$

4.3 Permutation P_d

P_d is a simple permutation on 2^d elements. It is constructed from π_d, P'_d and ϕ_d. Denote 2^d input elements as $A = (a_0, a_1, \cdots, a_{2^d-1})$, and 2^d output elements as $B = (b_0, b_1, \cdots, b_{2^d-1})$.

4.3.1 Permutation π_d

π_d operates on 2^d elements. The computation of $B = \pi_d(A)$ is as follows:

$$\begin{aligned}
b_{4i+0} &= a_{4i+0} \quad \text{for } i = 0 \text{ to } 2^{d-2} - 1\,;\\
b_{4i+1} &= a_{4i+1} \quad \text{for } i = 0 \text{ to } 2^{d-2} - 1\,;\\
b_{4i+2} &= a_{4i+3} \quad \text{for } i = 0 \text{ to } 2^{d-2} - 1\,;\\
b_{4i+3} &= a_{4i+2} \quad \text{for } i = 0 \text{ to } 2^{d-2} - 1\,;
\end{aligned}$$

The permutation π_4 is illustrated in Fig. 1.

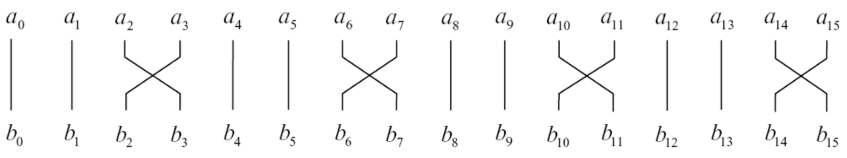

a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14} a_{15}
b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9 b_{10} b_{11} b_{12} b_{13} b_{14} b_{15}

Figure 1: The permutation π_4

4.3.2 Permutation P'_d

P'_d is a permutation on 2^d elements. The computation of $B = P'_d(A)$ is given as follows:

$$\begin{aligned} b_i &= a_{2i} \quad \text{for } i = 0 \text{ to } 2^{d-1} - 1\,; \\ b_{i+2^{N-1}} &= a_{2i+1} \text{ for } i = 0 \text{ to } 2^{d-1} - 1\,; \end{aligned}$$

The permutation P'_4 is illustrated in Fig. 2.

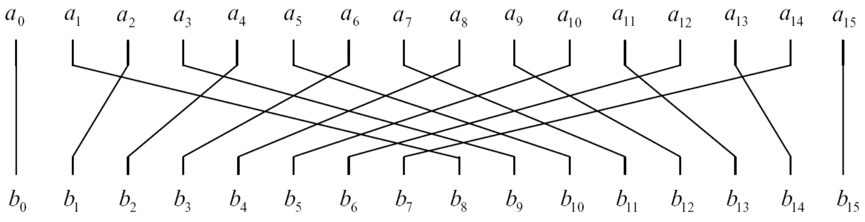

a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14} a_{15}
b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9 b_{10} b_{11} b_{12} b_{13} b_{14} b_{15}

Figure 2: The permutation P'_4

4.3.3 Permutation ϕ_d

ϕ_d is a permutation on 2^d elements. The computation of $B = \phi_d(A)$ is given as follows:

$$\begin{aligned} b_i &= a_i \qquad \text{for } i = 0 \text{ to } 2^{d-1} - 1\,; \\ b_{2i+0} &= a_{2i+1} \quad \text{for } i = 2^{d-1} \text{ to } 2^d - 1\,; \\ b_{2i+1} &= a_{2i+0} \quad \text{for } i = 2^{d-1} \text{ to } 2^d - 1\,; \end{aligned}$$

The permutation ϕ_4 is illustrated in Fig. 3.

4.3.4 Permutation P_d

P_d is the composition of π_d, P'_d and ϕ_d:

$$P_d = \phi_d \circ P'_d \circ \pi_d$$

The permutation P_4 is illustrated in Fig. 4.

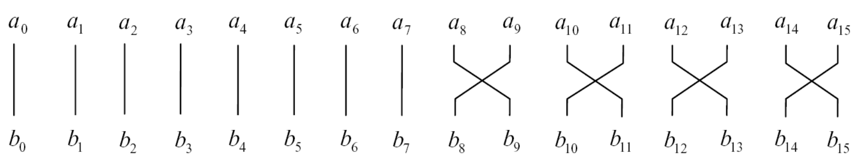

a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14} a_{15}
b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9 b_{10} b_{11} b_{12} b_{13} b_{14} b_{15}

Figure 3: The permutation ϕ_4

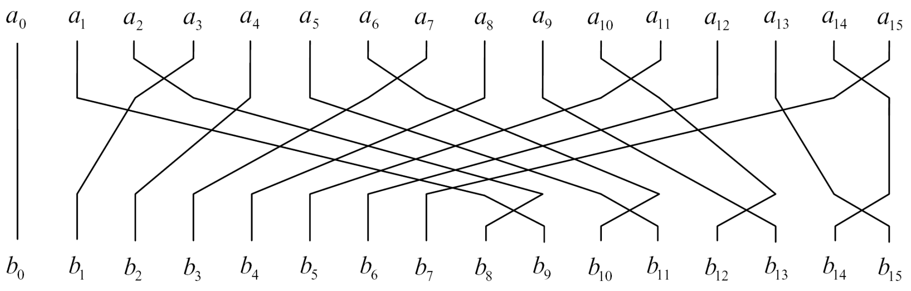

a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14} a_{15}
b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9 b_{10} b_{11} b_{12} b_{13} b_{14} b_{15}

Figure 4: The permutation P_4

4.4 Round function R_d

The round function R_d implements the Efficient Differential Propagation (EDP) design illustrated in Sec. 2. It consists of three layers: the Sbox layer, the linear transform layer and the permutation layer P_d. The input and output sizes of R_d are 2^{d+2} bits. The 2^{d+2}-bit input word is denoted as $A = (a_0 \,\|\, a_1 \,\|\, \cdots \,\|\, a_{2^d-1})$, where each a_i represents a 4-bit word. The 2^{d+2}-bit output word is denoted as $B = (b_0 \,\|\, b_1 \,\|\, \cdots \,\|\, b_{2^d-1})$, where each b_i represents a 4-bit word. The 2^d-bit round constant of the r-th round is denoted as $C_r^{(d)} = C_r^{(d),0} \| C_r^{(d),1} \cdots \| C_r^{(d),2^d-1}$. Let each v_i and w_i $(0 \le i \le 2^d - 1)$ represent a 4-bit word. The computation of $B = R_d(A, C_r^{(d)})$ is given as follows:

```
1.    for $i = 0$ to $2^d - 1$,
      {
          if $C_r^{(d),r} = 0$, then $v_i = S_0(a_i)$ ;
          if $C_r^{(d),i} = 1$, then $v_i = S_1(a_i)$ ;
      }
2.    $(w_{2i}, w_{2i+1}) = L(v_{2i}, v_{2i+1})$    for $0 \le i \le 2^{d-1} - 1$ ;
3.    $(b_0, b_1, \cdots, b_{2^d-1}) = P_d(w_0, w_1, \cdots, w_{2^d-1})$ ;
```

Two rounds of R_4 are illustrated in Fig. 5.

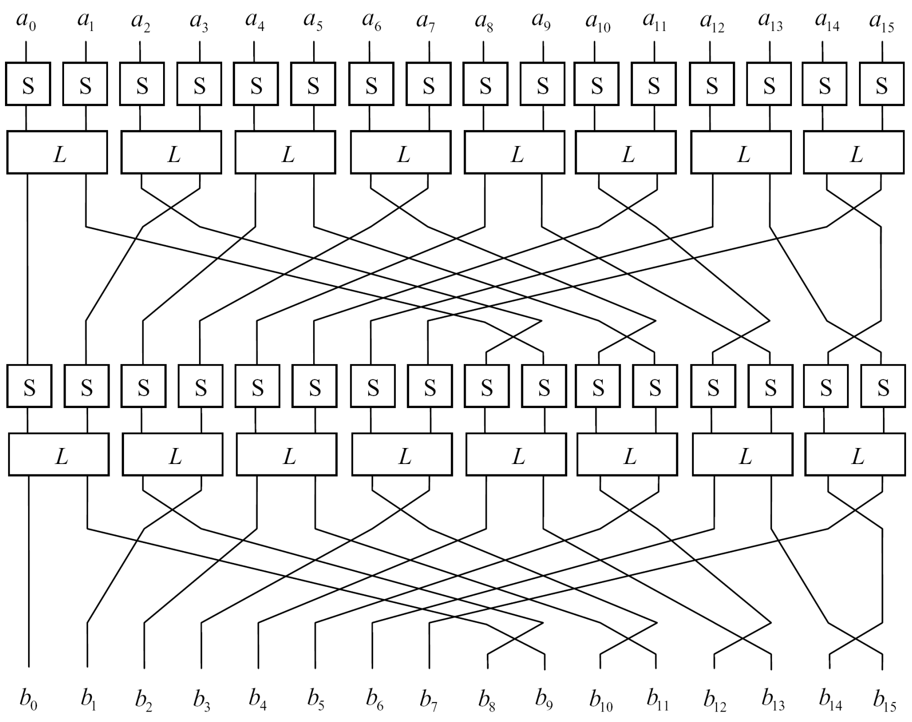

a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14} a_{15}
S
L
b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9 b_{10} b_{11} b_{12} b_{13} b_{14} b_{15}

Figure 5: Two rounds of R_4 (constant bits not shown)

4.5 Bijective function E_d

E_d is based on the d-dimensional EDP design. It is constructed from $5(d-1)$ rounds of R_d, plus an additional Sbox layer. The 2^{d+2}-bit input and output are denoted as A and B, respectively. Let each Q_r denote a 2^{d+2}-bit word for $0 \leq r \leq 4d+1$, and $Q_r = (q_{r,0} \,\|\, q_{r,1} \,\|\, \cdots \,\|\, q_{r,2^d-1})$, where each $q_{r,i}$ denotes a 4-bit word. Let R_d^* denote the round function R_d with the linear transformation and permutation being removed. Let $d' = d-1$. The computation of $B = E_d(A)$ is given as follows:

1. grouping the bits of A into 2^d 4-bit elements to obtain Q_0 ;
2. for $r = 0$ to $5(d-1)-1$, $Q_{r+1} = R_d(Q_r, C_r^{(d)})$;
3. $Q_{5(d-1)+1} = R_d^*(Q_{5(d-1)}, C_{5(d-1)}^{(d)})$;
4. de-grouping the 2^d 4-bit elements in $Q_{5(d-1)+1}$ to obtain B ;

The grouping of bits into 4-bit elements in the first step and the de-grouping in the last step are designed to achieve efficient bit-slice software implementation. The grouping in the first step is given as follows (as shown in Fig. 6):

```
for $i = 0$ to $2^{d-1} - 1$,
{
    $q_{0,2i} = A^i \| A^{i+2^d} \| A^{i+2\cdot 2^d} \| A^{i+3\cdot 2^d}$ ;
    $q_{0,2i+1} = A^{i+2^{d-1}} \| A^{i+2^{d-1}+2^d} \| A^{i+2^{d-1}+2\cdot 2^d} \| A^{i+2^{d-1}+3\cdot 2^d}$ ;
}
```

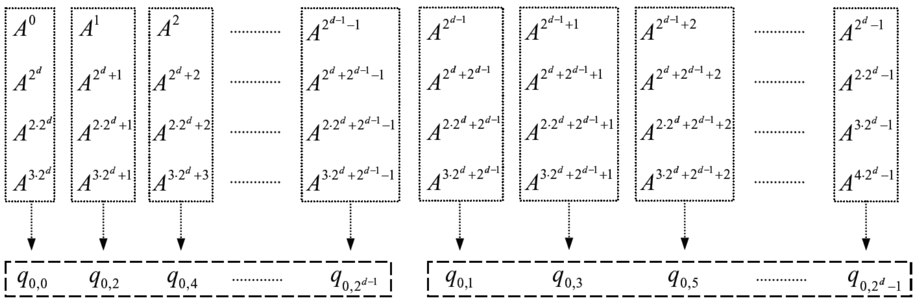

A^0 A^{2^d} $A^{2\cdot 2^d}$ $A^{3\cdot 2^d}$
A^1 A^{2^d+1} $A^{2\cdot 2^d+1}$ $A^{3\cdot 2^d+1}$
A^2 A^{2^d+2} $A^{2\cdot 2^d+2}$ $A^{3\cdot 2^d+3}$
$A^{2^{d-1}-1}$ $A^{2^d+2^{d-1}-1}$ $A^{2\cdot 2^d+2^{d-1}-1}$ $A^{3\cdot 2^d+2^{d-1}-1}$
$A^{2^{d-1}}$ $A^{2^d+2^{d-1}}$ $A^{2\cdot 2^d+2^{d-1}}$ $A^{3\cdot 2^d+2^{d-1}}$
$A^{2^{d-1}+1}$ $A^{2^d+2^{d-1}+1}$ $A^{2\cdot 2^d+2^{d-1}+1}$ $A^{3\cdot 2^d+2^{d-1}+1}$
$A^{2^{d-1}+2}$ $A^{2^d+2^{d-1}+2}$ $A^{2\cdot 2^d+2^{d-1}+2}$ $A^{3\cdot 2^d+2^{d-1}+2}$
A^{2^d-1} $A^{2\cdot 2^d-1}$ $A^{3\cdot 2^d-1}$ $A^{4\cdot 2^d-1}$
$q_{0,0}$ $q_{0,2}$ $q_{0,4}$ $q_{0,2^{d-1}}$
$q_{0,1}$ $q_{0,3}$ $q_{0,5}$ $q_{0,2^d-1}$

Figure 6: The grouping in function E_d

The de-grouping in the last step is given as follows (as shown in Fig. 7):

```
for $i = 0$ to $2^{d-1} - 1$,
{
    $B^i \| B^{i+2^d} \| B^{i+2\cdot 2^d} \| B^{i+3\cdot 2^d} = q_{5(d-1)+1,2i}$ ;
    $B^{i+2^{d-1}} \| B^{i+2^{d-1}+2^d} \| B^{i+2^{d-1}+2\cdot 2^d} \| B^{i+2^{d-1}+3\cdot 2^d} = q_{5(d-1)+1,2i+1}$ ;
}
```

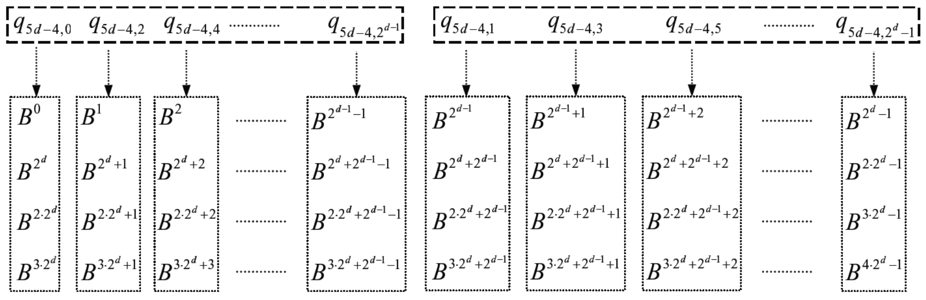

$q_{5d-4,0}$ $q_{5d-4,2}$ $q_{5d-4,4}$ $q_{5d-4,2^{d-1}}$
$q_{5d-4,1}$ $q_{5d-4,3}$ $q_{5d-4,5}$ $q_{5d-4,2^d-1}$
B^0 B^{2^d} $B^{2\cdot 2^d}$ $B^{3\cdot 2^d}$
B^1 B^{2^d+1} $B^{2\cdot 2^d+1}$ $B^{3\cdot 2^d+1}$
B^2 B^{2^d+2} $B^{2\cdot 2^d+2}$ $B^{3\cdot 2^d+3}$
$B^{2^{d-1}-1}$ $B^{2^d+2^{d-1}-1}$ $B^{2\cdot 2^d+2^{d-1}-1}$ $B^{3\cdot 2^d+2^{d-1}-1}$
$B^{2^{d-1}}$ $B^{2^d+2^{d-1}}$ $B^{2\cdot 2^d+2^{d-1}}$ $B^{3\cdot 2^d+2^{d-1}}$
$B^{2^{d-1}+1}$ $B^{2^d+2^{d-1}+1}$ $B^{2\cdot 2^d+2^{d-1}+1}$ $B^{3\cdot 2^d+2^{d-1}+1}$
$B^{2^{d-1}+2}$ $B^{2^d+2^{d-1}+2}$ $B^{2\cdot 2^d+2^{d-1}+2}$ $B^{3\cdot 2^d+2^{d-1}+2}$
B^{2^d-1} $B^{2\cdot 2^d-1}$ $B^{3\cdot 2^d-1}$ $B^{4\cdot 2^d-1}$

Figure 7: The de-grouping in function E_d

The round constants of E_d are given in Sect. 4.6.

4.6 Round constants of E_d

The round constants $C_r^{(d)}$ for E_d are generated from the round function R_{d-2} (with all the round constants of R_{d-2} being set as 0). Each $C_r^{(d)}$ is a 2^d-bit word. They are generated as follows:

$$\begin{aligned} &C_0^{(d)} \text{ is the integer part of } (\sqrt{2}-1)\times 2^{2^d} \text{ (in big endian form) ;} \\ &C_r^{(d)} = R_{d-2}(C_{r-1}^{(d)}, 0) \text{ for } 1 \le r \le 5(d-1)\,. \end{aligned}$$

The values of $C_r^{(8)}$ $(0 \le r \le 35)$ are given in Appendix A.1.

5 Compression Function F_d

Compression function F_d is constructed from the function E_d. F_d compresses the 2^{d+1}-bit message block $M^{(i)}$ and 2^{d+2}-bit $H^{(i-1)}$ into the 2^{d+1}-bit $H^{(i)}$:

$$H^{(i)} = F_d(H^{(i-1)}, M^{(i)})\,.$$

The construction of F_d is shown in Fig. 8. According to the definition of E_d, the input to every first-layer Sbox would be affected by two message bits; and the output from every last-layer Sbox would be XORed with two message bits.

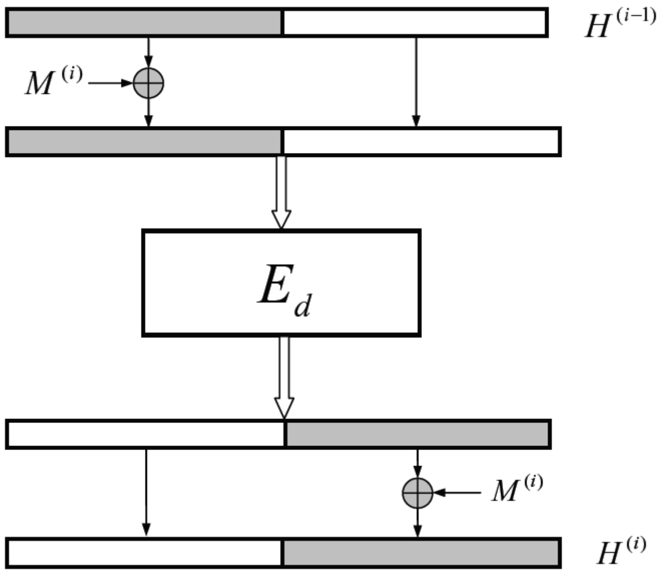


Figure 8: The compression function F_d

5.1 F_8

F_8 is the compression function used in hash function JH. F_8 compresses the 512-bit message block $M^{(i)}$ and 1024-bit $H^{(i-1)}$ into the 1024-bit $H^{(i)}$. F_8 is constructed from E_8. Let A, B denote two 1024-bit words. The computation of $H^{(i)} = F_8(H^{(i-1)}, M^{(i)})$ is given as:

1. $A^j = H^{(i-1),j} \oplus M^{(i),j}$ for $0 \le j \le 511$;
   $A^j = H^{(i-1),j}$ for $512 \le j \le 1023$;

2. $B = E_8(A)$;

3. $H^{(i),j} = B^j$ for $0 \le j \le 511$;
   $H^{(i),j} = B^j \oplus M^{(i),j-512}$ for $512 \le j \le 1023$;

6 JH Hash Algorithms

Hash function JH consists of five steps: padding the message M (Sect. 6.1), parsing the padded message into message blocks (Sect. 6.2), setting the initial hash value $H^{(0)}$ (Sect. 6.3), computing the final hash value $H^{(N)}$ (Sect. 6.4), and generating the message digest by truncating $H^{(N)}$ (Sect. 6.5).

6.1 Padding the message

The message M is padded to be a multiple of 512 bits. Suppose that the length of the message M is ℓ bits. Append the bit "1" to the end of the message, followed by $896-1-(\ell \bmod 512)$ zero bits, then append the 128-bit block that is equal to the number ℓ expressed using a binary representation in big endian form. Thus at least 512 additional bits are padded to the message M.

6.2 Parsing the padded message

After a message has been padded, it is parsed into N 512-bit blocks, $M^{(1)}$, $M^{(2)}$, ..., $M^{(N)}$. The 512-bit message block is expressed as four 128-bit words. The first 128 bits of message block i are denoted as $M_0^{(i)}$, the next 128 bits are $M_1^{(i)}$, and so on up to $M_3^{(i)}$.

6.3 Setting the initial hash value $H^{(0)}$

The initial hash value $H^{(0)}$ is set depending on the message digest size. The first two bytes of $H^{(-1)}$ are set as the message digest size, and the rest bytes of $H^{(-1)}$ are set as 0. Set $M^{(0)}$ as 0. Then $H^{(0)} = F_8(H^{(-1)}, M^{(0)})$.

More specifically, the value of $H_0^{(-1),0} \| H_0^{(-1),1} \| \cdots \| H_0^{(-1),15}$ is $0x00E0$, $0x0100$, $0x0180$, $0x0200$ for JH-224, JH-256, JH-384 and JH-512, respectively. Let $H^{(-1),j} = 0$ for $16 \leq j \leq 1023$. Set the 512-bit $M^{(0)}$ as 0. The 1024-bit initial hash value $H^{(0)}$ is computed as

$$H^{(0)} = F_8(H^{(-1)}, M^{(0)}) \ .$$

6.4 Computing the final hash value $H^{(N)}$

The compression function F_8 is applied to generate $H^{(N)}$ by compressing $M^{(1)}$, $M^{(2)}$, …, $M^{(N)}$ iteratively. The 1024-bit final hash value $H^{(N)}$ is computed as follows:

```
for $i = 1$ to $N$,
    $H^{(i)} = F_8(H^{(i-1)}, M^{(i)})$;
```

6.5 Generating the message digest

The message digest is generated by truncating $H^{(N)}$.

6.5.1 JH-224

The last 224 bits of $H^{(N)}$ are given as the message digest of JH-256:

$$H^{(N),800} \| H^{(N),769} \| \cdots \| H^{(N),1023} \, .$$

6.5.2 JH-256

The last 256 bits of $H^{(N)}$ are given as the message digest of JH-256:

$$H^{(N),768} \| H^{(N),769} \| \cdots \| H^{(N),1023} \, .$$

6.5.3 JH-384

The last 384 bits of $H^{(N)}$ are given as the message digest of JH-384:

$$H^{(N),640} \| H^{(N),641} \| \cdots \| H^{(N),1023} \, .$$

6.5.4 JH-512

The last 512 bits of $H^{(N)}$ are given as the message digest of JH-512:

$$H^{(N),512} \| H^{(N),513} \| \cdots \| H^{(N),1023} \, .$$

7 Bit-Slice Implementation of JH

The description of JH given in Sect. 4 and Sect. 5 are suitable for efficient hardware implementation. In this section, we illustrate the bit-slice implementation of JH. The bit-slice implementation of F_d uses $d-1$ different round function descriptions (the hardware description of F_d uses identical round function description).

7.1 Bit-slice parameters

The following additional parameters are used in the bit-slice implementation of JH.

$C'^{(d)}_r$	The round constant words used in the bit-slice implementation of E_d with $0 \le r \le 5 \times (d-1)$. Each $C'^{(d)}_r$ is a 2^d-bit constant word.
$C'^{(d)}_{r,even}$	Even bits of $C'^{(d)}_r$. $C'^{(d)}_{r,even} = C'^{(d),0}_r \parallel C'^{(d),2}_r \parallel C'^{(d),4}_r \parallel \cdots \parallel C'^{(d),2^d-2}_r$. Each $C'^{(d)}_{r,even}$ is a 2^{d-1}-bit constant word.
$C'^{(d)}_{r,odd}$	Odd bits of $C'^{(d)}_r$. $C'^{(d)}_{r,odd} = C'^{(d),1}_r \parallel C'^{(d),3}_r \parallel C'^{(d),5}_r \parallel \cdots \parallel C'^{(d),2^d-1}_r$. Each $C'^{(d)}_{r,odd}$ is a 2^{d-1}-bit constant word.
$H^{(i)}_j$	The jth 128-bit word of the ith hash value. $H^{(i)}_0$ is the left-most 128-bit word of hash value $H^{(i)}$.
$M^{(i)}_j$	The jth 128-bit word of the ith message block. $M^{(i)}_0$ is the left-most word of message block $M^{(i)}$.

7.2 Bit-slice functions

The following functions are used in the bit-slice implementation of JH.

7.2.1 Sbox

S^{bitsli} implements both S_0 and S_1 in the bit-slice implementation of JH. Let each x_i $(0 \le i \le 3)$ denotes a 2^{d-1}-bit word. Let c denote a 2^{d-1}-bit constant word. let t denote a temporary word. $(x_0, x_1, x_2, x_3) = S^{bitsli}(x_0, x_1, x_2, x_3, c)$ is computed in the following 11 steps:

1. $x_3 = \neg x_3\,;$	2. $x_0 = x_0 \oplus (c \,\&\, (\neg x_2))\,;$
3. $t = c \oplus (x_0 \,\&\, x_1)\,;$	4. $x_0 = x_0 \oplus (x_2 \,\&\, x_3)\,;$
5. $x_3 = x_3 \oplus ((\neg x_1) \,\&\, x_2)\,;$	6. $x_1 = x_1 \oplus (x_0 \,\&\, x_2)\,;$
7. $x_2 = x_2 \oplus (x_0 \,\&\, (\neg x_3))\,;$	8. $x_0 = x_0 \oplus (x_1 \vert x_3)\,;$
9. $x_3 = x_3 \oplus (x_1 \,\&\, x_2)\,;$	10. $x_1 = x_1 \oplus (t \,\&\, x_0)\,;$
11. $x_2 = x_2 \oplus t;$	

7.2.2 Linear Transform

L^{bitsli} implements the linear transform in the bit-slice implementation of JH. Let each a_i and b_i $(0 \le i \le 7)$ denotes a 2^{d-1}-bit word. $(b_0, b_1, \cdots, b_7) = L^{bitsli}(a_0, a_1, \cdots, a_7)$ is computed as follows:

$$\begin{aligned}
b_4 &= a_4 \oplus a_1\,; & b_5 = a_5 \oplus a_2\,;\\
b_6 &= a_6 \oplus a_3 \oplus a_0\,; & b_7 = a_7 \oplus a_0\,;\\
b_0 &= a_0 \oplus b_5\,; & b_1 = a_1 \oplus b_6\,;\\
b_2 &= a_2 \oplus b_7 \oplus b_4\,; & b_3 = a_3 \oplus b_4\,.
\end{aligned}$$

7.2.3 Permutation $\bar{\omega}$

Let $A = (a_0, a_1, \cdots, a_{2\times\alpha\times n-1})$, where α and n are positive integers. Let $B = (b_0, b_1, \cdots, b_{2\times\alpha\times n-1})$. Each a_i and b_i denotes a 4-bit element. The permutation $B = \bar{\omega}(A, n)$ is computed as follows:

```
for $i = 0$ to $\alpha - 1$,
    for $j = 0$ to $n - 1$,
        $b_{2\times i\times n+j} = a_{2\times i\times n+n+j}\,;\; b_{2\times i\times n+n+j} = a_{2\times i\times n+j}\,;$
```

For example, $\bar{\omega}(A, 1)$ swaps element a_{2i} and a_{2i+1}.

7.2.4 Permutation ω

Permutation $\omega(A, n)$ swaps the bits in a word A. It is computed by treating each bit in A as an element, then applying the permutation $\bar{\omega}(A, n)$.

7.2.5 Permutation $\bar{\sigma}_d$

Permutation $\bar{\sigma}_d$ operates on 2^d elements. Let $A = (a_0 \parallel a_1 \parallel \cdots \parallel a_{2^d-1})$, $B = (b_0 \parallel b_1 \parallel \cdots \parallel b_{2^d-1})$. Let $n = 2^\beta$, where β is an integer smaller than $d-1$. $B = \bar{\sigma}_d(A, n)$ permutes the odd elements in A as follows:

$$\begin{aligned}
(b_1, b_3, b_5, \cdots, b_{2^d-1}) &= \bar{\omega}((a_1, a_3, a_5, \cdots, a_{2^d-1})\,, n)\,;\\
(b_0, b_2, b_4, \cdots, b_{2^d-2}) &= (a_0, a_2, a_4, \cdots, a_{2^d-2})\,.
\end{aligned}$$

7.2.6 Permutation σ_d

Permutation $\sigma_d(A, n)$ operates on the bits in a word A. It is computed by treating each bit in A as an element, then applying the permutation $\bar{\sigma_d}(A, n)$.

7.2.7 Round constants

Let IP_d denote the inverse of P_d. Let IP_d^r denote the composition of r permutation IP_d :

$$IP_d^r = \underbrace{IP_d \circ IP_d \circ \cdots \circ IP_d}_{r} \,.$$

Note that IP_d^r has the property that $IP_d^r = IP_d^{r+\alpha\cdot d}$.

Let permutation $\lambda_d^r(A)$ operate on the bits in a word A. It is computed by treating each bit in A as an element, then applying the permutation IP_d^r.

Let η_d^r denote a permutation. Let A, B and V_i denote 2^d-bit words. $B = \eta_d^r(A)$ is computed as follows:

```
$V_0 = A\,;$
for $i = 0$ to $r-1$,    $V_{i+1} = \sigma_d(V_i, 2^{i \bmod (d-1)})\,;$
$B = V_r;$
```

The round constant $C_r'^{(d)}$ is generated from $C_r^{(d)}$ as:

$$C_r'^{(d)} = \eta_d^r \circ \lambda_d^r(C_r^{(d)})\,.$$

The 2^{d-1}-bit constant words $C_{r,even}'^{(d)}$ and $C_{r,odd}'^{(d)}$ are obtained by extracting the even and odd bits of $C_r'^{(d)}$, respectively, as defined in Sec. 7.1. $C_{r,even}'^{(8)}$ and $C_{r,odd}'^{(8)}$ are given in Appendix A.2.

7.2.8 An alternative description of round function R_d

The description of R_d in Sect. 4.4 is suitable for hardware implementation. But that description is not suitable for the bit-slice implementation. We give here an alternative description of R_d, and denote the r-th round function as $R'_{d,r}$. The 2^d-bit round constant of the r-th round is denoted as $C_r'^{(d)}$. Let $V = v_0 \parallel v_1 \parallel \cdots \parallel v_{2^d-1}$, where each v_i denotes a 4-bit word. The computation of $B = R'_{d,r}(A, C_r'^{(d)})$ is given as follows:

```
1.    for $i = 0$ to $2^d - 1$,
      {
          if $C_r'^{(d),i} = 0$, then $v_i = S_0(a_i)\,;$
          if $C_r'^{(d),i} = 1$, then $v_i = S_1(a_i)\,;$
      }
2.    $B = \bar{\sigma}_d(V,\, 2^{r \bmod (d-1)})$
```

Note that $R'_{d,r}$ has the following properties:

1. The description of $R'_{d,r}$ is the same as $R'_{d,r+\alpha\cdot(d-1)}$ except for the different round constants.
2. For the same input passing through multiple rounds, at the end of the $\alpha\cdot(d-1)$-th round, the output from $R'_{d,\alpha\cdot(d-1)}$ is identical to the output from $R_{d,\alpha\cdot(d-1)}$.

Six rounds of $R'_{4,r}$ $(0 \le r \le 5)$ are illustrated in Fig. 9.

7.2.9 Bit-slice implementation of round function R_d

The above description of $R'_{d,r}$ can be implemented efficiently in a bit-slice way. The method used is to separate the odd and even elements of A in $R'_{d,r}$. Denote the bit-slice implementation as $R^{bitsli}_{d,r}$. Let A and B represent two 2^{d+2}-bit words, $A = a_0 \parallel a_1 \parallel a_2 \parallel \cdots \parallel a_7$, and $B = b_0 \parallel b_1 \parallel b_2 \parallel \cdots \parallel b_7$, where each A_i and B_i represents a 2^{d-1}-bit word. Let each v_i and u_i $(0 \le i \le 7)$ denote a 2^{d-1}-bit word. The computation of $B = R^{bitsli}_{d,r}(A, C'^{(d)}_{r,even}, C'^{(d)}_{r,odd})$ is given as follows:

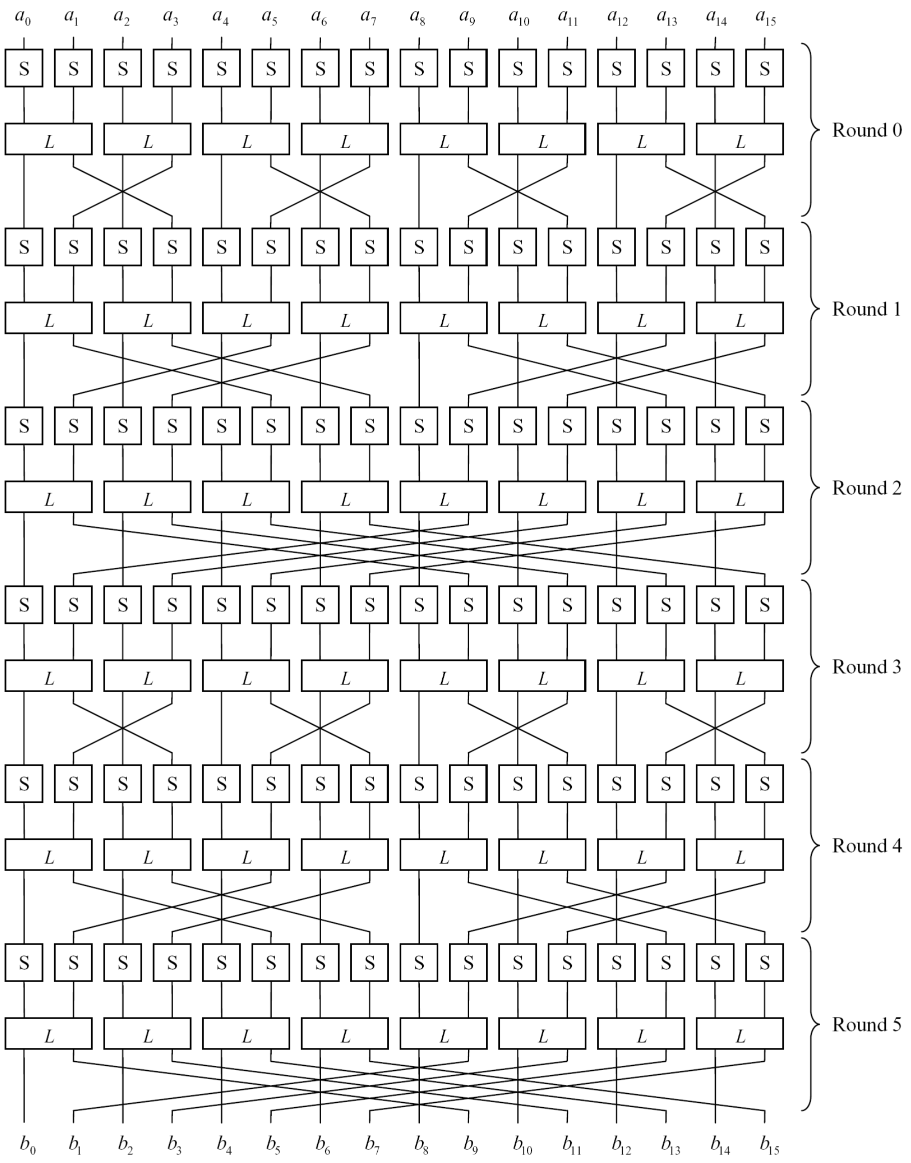
1. $(v_0, v_2, v_4, v_6) = S^{bitsli}(a_0, a_2, a_4, a_6, C'^{(d)}_{r,even})$;
   $(v_1, v_3, v_5, v_7) = S^{bitsli}(a_1, a_3, a_5, a_7, C'^{(d)}_{r,odd})$;
2. $(u_0, u_2, u_4, u_6, u_1, u_3, u_5, u_7) = L^{bitsli}(v_0, v_2, v_4, v_6, v_1, v_3, v_5, v_7)$;
3. $b_0 = u_0$; $b_2 = u_2$; $b_4 = u_4$; $b_6 = u_6$;
   $b_1 = \omega(u_1,\ 2^{r \bmod (d-1)})$;
   $b_3 = \omega(u_3,\ 2^{r \bmod (d-1)})$;
   $b_5 = \omega(u_5,\ 2^{r \bmod (d-1)})$;
   $b_7 = \omega(u_7,\ 2^{r \bmod (d-1)})$;

7.2.10 Bit-slice implementation of E_d

The 2^{d+2}-bit input and output are denoted as A and B, respectively. Let each Q_r denote a 2^{d+2}-bit word for $0 \le r \le 5(d-1)$. Let $R^{*bitsli}_{d,r}$ denote the round function $R^{*bitsli}_{d,r}$ with only the Sbox layer. The computation of $B = E_d(A)$ is given as follows:

1. $Q_0 = A$;
2. for $r = 0$ to $5(d-1)-1$, $Q_{r+1} = R^{bitsli}_{d,r}(Q_r, C'^{(d)}_{r,even}, C'^{(d)}_{r,odd})$;
3. $B = R^{*bitsli}_{d,5(d-1)}(Q_{5(d-1)}, C'^{(d)}_{5(d-1),even}, C'^{(d)}_{5(d-1),odd})$;

The generation of the round constants is given in Sect. 7.2.7.



a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14} a_{15}
S
L
Round 0
Round 1
Round 2
Round 3
Round 4
Round 5
b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9 b_{10} b_{11} b_{12} b_{13} b_{14} b_{15}

Figure 9: An alternative description of 6 rounds of R_4(constant bits not shown)

7.3 Bit-slice implementation of F_8

F_8 compresses the 512-bit message block $M^{(i)}$ and 1024-bit $H^{(i-1)}$ into the 1024-bit $H^{(i)}$. The computation of $H^{(i)} = F_8(H^{(i-1)}, M^{(i)})$ is given as:

1. $A_j = H_j^{(i-1)} \oplus M_j^{(i)}$ for $0 \leq j \leq 3\,;$

   $A_j = H_j^{(i-1)}$ for $4 \leq j \leq 7\,;$

2. $B = E_8(A)$;

3. $H_j^{(i)} = B_j$ for $0 \le j \le 3$;

   $H_j^{(i)} = B_j \oplus M_{j-4}^{(i)}$ for $4 \le j \le 7$;

Note that in round function $R_{8,r}^{bitsli}(A, C'^{(8)}_{r,even}, C'^{(8)}_{r,odd})$, each word is 128-bit and is thus suitable for SSE2 implementation. For a 128-bit word x, $\omega(x, n)$ can be implemented with two AND operations (AND with a constant to extract the bits to be swapped), two shift operations and one OR operations (note that the shift operations would be affected by the endianess of the SSE2 register). In addition, $\omega(x, 32)$ and $\omega(x, 64)$ can be implemented with one SSE2 shuffle operation. Thus the SSE2 implementation of F_8 is very efficient.

8 Variants of JH

The design of JH hash algorithms implies several variants by varying the parameter d or by replacing P_d with P'_d in round function R_d.

8.1 Varying the parameter d

The compression function F_d gives several compression functions by varying the parameter d.

F_6. $d = 6$. The round number is increased from 25 ($= 5(d-1)$) to 30 ($= 6(d-1)$). With 256-bit block size and 128-bit message block size, this compression function is extremely hardware efficient. Hash function using this compression function can achieve 128-bit security level for collision resistance, preimage resistance and second preimage resistance for 256-bit message digest size.

F_7. $d = 7$. With 512-bit block size and 256-bit message block size, this compression is used to generate 256-bit message digest size. The memory required is half of that of F_8, and it achieves 128-bit security level for comllision resistance, 256-bit security for preimage resistance.

F_9. $d = 9$. With 2048-bit block size, this compression function is extremely efficient on the future microprocessors that support shift and binary operations over 256-bit registers.

8.2 Replacing P_d with P'_d

Replacing permutation P_d with P'_d in round function R_d, and change the round number $5(d-1)$ to $5d$ in E_d, we can obtain another family of compression functions. This family of compression functions are slightly simpler in hardware, but its bit-slice implementation requires twice amount of shift operations as required in F_d. A few variants can be obtained by varying the value of d.

9 Security Analysis of JH

The security of JH hash algorithms are stated below ($\bar{l}$ denotes the number of message blocks, the length of a message is less than 2^{128} bits):

	collision	second-preimage	preimage
JH-224	2^{112}	2^{224}	2^{224}
JH-256	2^{128}	2^{256}	2^{256}
JH-384	2^{192}	2^{384}	2^{384}
JH-512	2^{256}	$2^{512-\bar{l}}$	2^{512}

Note that the second-preimage resistance of JH-512 is affected by herding attack [10]. The reason is that the collision resistance of Jh-512 is 2^{256}, although the size of the hash value $H^{(i)}$ is 1024 bits. However, the second-preimage resistance of JH-512 would not be affected by herding attack if birthday attack is applied to find collisions in herding attack.

9.1 Differential cryptanalysis

Differential cryptanalysis is important in analyzing the security of a hash function. It has been applied to break MD4, MD5, SHA-0 and SHA-1 [8, 5, 2, 3, 14, 15, 16, 17].

We study the number of active Sboxes being involved in a differential characteristic in E_d. The symmetry structure of EDP design allows us to determine the number of active Sboxes easily since many differential paths (branches) are equivalent. We can replace P_d with P'_d in E_d to get a simpler variant whose security is equivent to that of the original E_d. We can also study the E_d with small d to learn when the minimum number of active Sboxes would occur. For example, two active Sboxes before the linear transformation L would result in only one active Sboxes after L.

For $d \in 2, 3, 4$, we exhaustively searched for the minimum number of active Sboxes. The minimum number of active Sboxes for $2d+1$ Sbox layers is 10, 20, 38 for $d = 2, 3, 4$, respectively. For $d \geq 4$, we found that the minimum number of active Sboxes for $2d+1$ Sbox layers is 64, 112, 176, 296 for $d = 5, 6, 7, 8$. It shows that the minimum number of active Sboxes does increase significantly as the value of d increases.

For E_8, we found that the minimum number of active Sboxes for 36 Sbox layers is 624 when there are eight active elements in the input of E_8. If we conservatively assume that there are 2^{36} multiple paths for a differential, there are still around 600 effective Sboxes. The large number of active Sboxes shows that JH is strong against the differential cryptanalysis.

9.1.1 Effect of correlated active elements in differential attack

In the differential cryptanalysis of JH, each differential characteristic of an Sbox has a probability of at most $\frac{1}{4}$. Each active Sbox may contribute 2^{-2} to the overall differential probability if the active SBoxes are assumed to be independent. However, when there is correlation between active elements, the overall differential probability may increase.

For the 8-bit-to-8-bit super Sbox (concept from Rijmen and Daemen) consisting of two nonlinear layers (4 Sboxes connected by L), a differential characteristic has a maximum probability of $\frac{12}{256} = 2^{-4.41}$. If we consider that there are 16 combinations of those 4 Sboxes, then the average of those 16 maximum differential probabilities is $\frac{10.875}{256} = 2^{-4.56}$. If only 3 Sboxes are active, then the maximum differential probability is $\frac{10}{256} = 2^{-4.68}$. For the 16-bit-to-16-bit super Sbox consisting of three nonlinear layers, there are 4096 combinations of those 12 Sboxes. If there is only one active Sbox in the first or last Sbox layer, then there are 7 active Sboxes being involved; the maximum differential probability is $\frac{44}{2^{16}} = 2^{-10.54}$, and the average of those 4096 maximum differential probabilities is $2^{-10.98}$. When the minimum number of active Sboxes occurs, we are mainly dealing the 8-bit-to-8-bit super Sbox with 3 active Sboxes, and the 16-bit-to-16-bit super Sbox with 7 active Sboxes. In these situations, we see that the effective differential characteristic of an active Sbox is less than $2^{-1.5}$ (but larger than 2^{-2}).

If we consider that each active Sbox contributes $2^{-1.5}$ to the overall differential probability, then the probability of a differential involves 600 active Sboxes is about 2^{-900}.

9.1.2 Effect of message modification in differential attack

To study the collision resistance of JH, we conservatively assume that an attacker can efficiently eliminate 16 rounds of E_8 with message modification, then there are 20 Sbox layers being left. For 20 Sbox layers of E_8, we found that a differential characteristic involves at least 336 active Sboxes. If we assume that there are 2^{20} multiple paths for a differential, then a differential has probability less than $2^{-1.5\times 336} \times 2^{20} = 2^{-484}$. We thus expect a differential collision attack can not succeed with less than 2^{256} operations.

9.1.3 Second-preimage and preimage differential attacks

For the second-preimage resistance of JH, we note that a differential passing through at least two compression functions F_8 should be considered. The reason is that the 512-bit message block size is only half that of the 1024-bit hash value $H^{(i)}$, and each compression function involves huge number of active Sboxes. Since message modification in second-preimage attack is not as efficient as that in collision attack, and at least two message blocks are involved in a differential second-preimage attack, we expect that a differential exists with probability much less than 2^{-512}, and JH is secure against the differential second-preimage attack.

For the preimage resistance of JH, we note that a differential passing through at least two compression functions F_8 should be considered. The reason is that one more block is padded to the message before generating the message digest. We expect that the complexity of the preimage attack is more than the square of that of the collision attack. We thus expect that JH is secure against the differential preimage attack.

9.2 Truncated differential cryptanalysis

For collision search, truncated differential cryptanalysis [11] may be viewed as the bridge linking differential cryptanalysis and birthday attack. Differential cryptanalysis can be viewed as truncated differential cryptanalysis with input space 2, while birthday cryptanalysis can be viewed as truncated differential cryptanalysis with input space 2^n where n indicates the block size of the compression function. Because of the nature of EDP design, it is necessary to evaluate the security of JH against the truncated differential cryptanalysis.

In the truncated differential cryptanalysis of JH, we focus on whether an element is active or not instead of the value of the difference. Let us consider those four Sboxes connected by a linear transformation L. If only one of the two Sboxes before L is active, then both Sboxes after L are active with probability 1. We call this event as active element expansion. If both two Sboxes before L are active and independent, then the probability that only one Sbox after L is active is 2^{-4}. We call this event as active element shrinking. If there are independent active Sboxes in the last Sbox layer, then the probability that the difference of the output from an active Sbox is cancelled by the message difference (if there is message difference at that location) is 2^{-4}. For a truncated differential characteristic, we count the number of active element shrinking events and the number of active Sboxes in the last Sbox layer of E_8, and denote the sum of these two numbers as TD_8.

Exploiting the symmetry property of E_8, we found in our analysis that the smallest value of TD_8 is 200 when there are eight active elements in the input of E_8. If we assume that the message modification can effectively remove 8 rounds in the truncated differential attack (the message modification in truncated differential attack is a bit difficult), then the smallest value of TD_8 is 144 when there are eight active elements in the input of E_8. Assume that there are 2^{26} multiple paths, it requires around $2^{144\times4-26} = 2^{550}$ difference pairs to generate a collision. Note that 2^{32} messages with eight active elements can generate only 2^{63} difference pairs, the attack would require much more than 2^{256} messages.

Truncated differential cryptanalysis is not that efficient for preimage and second-preimage attack. We thus do not apply truncated differential cryptanalysis to find the preimage and second-preimage of JH.

9.3 Algebraic attacks

Algebraic attacks solve the nonlinear equations in order to recover the key or message. For hash function cryptanalysis, algebraic attacks can be applied to find collision, second preimage and preimage if the algebraic equations of the compression function are very weak.

In the past several years, algebraic attacks have been proposed against block ciphers, but so far there is no evidence that algebraic attacks can break a practical block cipher faster than statistical cryptanalysis techniques, and there is no evidence that the complexity of algebraic attacks against block ciphers would be linear to the round number. The recent cube attacks, developed by Dinur and Shamir [7], can solve nonlinear equations with low degree when a number of equations (involve the same secret key) are available.

To find a collision of JH hash algorithms with algebraic attack, the meet-in-the-middle approach can result in algebraic equations of 18 Sbox layers. To find a second-preimage with algebraic attack, two blocks of message must be considered, and thus an algebraic attack needs to deal with algebraic equations of 36 Sbox layers. Recovering a message from the message digest would involve at least 36 Sbox layers since one more block is padded to the message. Because of the algebraic degree of the Sbox is 3 and the number of rounds being involved is large, we consider that JH is secure against algebraic attacks.

To be conservative, we use constant bits to select Sboxes to further strengthen JH against algebraic attacks.

10 Performance of JH

JH can be implemented efficiently on a wide range of platforms ranging from one-bit processor (hardware) to 128-bit processor (SIMD/SSE2). The reason is that EDP design allows JH being constructed from extremely simple elements. The 5-bit-to-4-bit (including the constant bit) Sbox can be implemented with 20 binary operations (including ANDNOT operation), and the linear transformation L can be implemented with 10 binary operations. The simple Sboxes and linear transformation ensures that JH is extremely hardware and software efficient.

10.1 Hardware

The hardware implementation of JH is extremely simple and efficient due to the simple Sboxes and linear transformation. JH uses 1024-bit memory for storing the state of E_8, 512-bit memory for storing the memory, and 256-bit memory to store a round constant (if the round constants are generated on-the-fly).

Let us compare JH with the ultra-lightweight block cipher PRESENT [4]. The hardware complexity of JH is comparable to that of PRESENT, except for the difference in block sizes. JH uses slightly more complicated Sboxes and linear transformation than PRESENT. The block size of E_8 is about 16 times that of PRESENT, while the size of a round constant in E_8 is only 4 times that of key size of PRESENT. A rough estimation is that E_8 requires 16 times more gates than PRESENT. PRESENT uses about 1570 GE (gate equivalents), so JH may require $1570 \times 16 \approx 25$K GE (estimated).

10.2 8-bit processor

JH can be implemented on 8-bit processor in two approaches. One approach is to implement the hardware description of JH with table lookup for Sboxes. The advantage of this approach is that the constant bits can be generated on-the-fly efficiently. Another approach is to implement the bit-slice description of JH. With 1152-byte precomputed round constants being stored in ROM, this implementation is expected to be quite fast. Given that the SSE2 bit-slice implementation of JH runs at 16.8 cycles/byte on CORE 2 processor, we can roughly estimate the speed of JH on 8-bit processor. The register size of 8-bit processor is 16 times smaller than that of SSE2 register. If we estimate that the number of instructions being processed per clock cycle on 8-bit processor is 5 times less than that on CORE 2 processor, the speed of the bit-slice implementation of JH on 8-bit processor is about $16 \times 5 \times 16.8 = 1344$ cycles/byte (estimated).

10.3 Core 2 processor

The bit-slice implementation of JH is tested on the popular Core 2 processor. The processor being used in the test is Core 2 Duo Mobile Processor P9400 2.53GHz. The Operating systems are 32-bit and 64-bit Windows Vista Business. The compiler being used is the Intel C compiler 10.1.025 (IA-32 version of the compiler is used with the 32-bit Vista, and Intel-64 version of the compiler is used with the 64-bit Vista). The hash speed (for long message) is 16.8 clock cycles/byte with the 64-bit Vista (with optimization option -QxT -O2 of the Intel-64 Intel C compiler); and it is 21.3 clock cycles/byte on the 32-bit Vista (with optimization option -QxT of the IA-32 Intel C compiler).

JH on 64-bit platform is faster than that on 32-bit platform. The reason is that there are sixteen 128-bit registers on the 64-bit platform of Core 2 processor; while there are only eight 128-bit registers on the 32-bit platform of Core 2 processor.

Microsoft Visual C++ 2005 and 2008 are not recommended for compiling the SSE2 codes. It seems that the optimization of SSE2 instructions is not implemented (or very poor) in Microsoft Visual C++ 2005 and 2008. The speed of JH is about 40+ clock cycles/byte with the Microsoft compilers (with optimization option /O2).

11 Design Rationale

We give below the rationale of designing the components of JH.

11.1 Compression function F_d

The construction of compression function F_d from bijective function E_d is new. It gives an extremely simple and efficient approach to construct a compression function from a bijective function.

In F_d, the message block size is half of the block size of E_d. The message is XORed with the first half of the input to E_d, then it is XORed with the second half of the output from E_d to achieve one-wayness. Besides the one-wayness, this construction is very efficient – every bit in the output from E_d is not truncated; and the difference cancellation involving the message is minimized. The message block size is only half of the block size of E_d, it is to prevent copying a collision block to other locations, and it is also helpful to resist attacks launched from the middle of E_d.

In the hash function, at least one more block is appended to the message. The reason is that if the difference of two last message blocks eliminates the difference of the inputs to E_d, then the outputs from F_d are not random. Thus one more F_d operation is needed to randomize the hash value.

11.2 EDP design

EDP design (Sec. 2) being used to construct the bijective function E_d is very simply and efficient. The input to E_d is grouped into a 2^d-dimensional array. The nonlinear layer consists of Sboxes. In the linear layer of the r-th round, MDS code is applied along the $(r \bmod d)$-th dimension of the array. EDP design is the generalization of the AES design [6].

EDP design is easy to analyze due to its symmetrical construction. Round constants are applied to prevent the symmetry property being exploited in attacks.

EDP design is efficient in hardware since E_d can be build upon small components. EDP design is also efficient in software since it can be implemented in a bit-slice approach.

11.3 Round number

The round number of E_8 is $5(8-1) = 35$. The round number is chosen to satisfy two requirements. One requirement is that the round number is the multiple of (d-1) so that the hardware description is simple since at the end of the multiple of (d-1) rounds, the output from the hardware description is identical to that from the bit-slice implementation. Another requirement is that the round number should be larger than $4d$ in order to satisfy the security requirements. We thus set the round number of E_8 as 35.

The round number 35 is used for all the JH algorithms for two reasons – one reason is to achieve the simplicity of description and implementation; another reason is to achieve extremely high security for JH-256 (JH-224) so that it achieves 256-bit (224-bit) security level for preimage and second preimage resistance, and it also eliminates the threat of multicollision attack against JH-224 and JH-256.

11.4 Selecting SBoxes

Two Sboxes are used in JH. Each round constant bit selects which Sboxes are used. Similar design has been used in Feistel's block cipher Lucifer [9] in which a key bit selects which Sboxes are used. The main reason that we use two different Sboxes selected by round constant bits is to increase the complexity of the system algebraic equations so that JH can have better resistance against the future algebraic attack.

11.5 Designing SBoxes

The 4-bit-to-4-bit Sboxes in JH are designed to meet the following requirements:

1. There is no fixed point for each Sbox, i.e., the input is always different from the output. For the same input, the outputs from two Sboxes are different.

2. Each differential characteristic has a probability of at most $\frac{1}{4}$.

3. Each linear characteristic [12] has a probability in the range $\frac{1}{2} \pm \frac{1}{4}$.

4. The nonlinear order of each output bit as a function of the input bits is 3.

5. The algebraic normal forms of the two Sboxes are different.

6. The resulting super Sboxes (formed with more than than one Sbox layer, introduced by Rijmen and Daemen, mainly to address the effect of correlated active elements) are strong against differential cryptanalysis.

Putting two Sboxes together, we have a 5-bit-to-4-bit Sbox with one input bit being the round constant bit that selects which Sboxes are used. This Sbox satisfies the following requirements:

1. Each differential characteristic has a probability of at most $\frac{1}{4}$.
2. Each linear characteristic has a probability in the range $\frac{1}{2} \pm \frac{1}{4}$.

We searched for the 5-bit-to-4-bit Sbox that can be implemented with small number of operations. The 5-bit-to-4-bit Sbox being used in JH can be implemented with 20 binary operations (AND, ANDNOT, XOR, NOT, OR).

11.6 Linear transform

The linear transform L is probably the simplest (4,2,3) MDS code over $GF(2^4)$. It requires only ten XOR operations.

12 Advantages and Limitations

JH hash algorithms have the following advantages:

1. Simple design. Both the hardware and software (bit-slice) descriptions of F_8 are very simple, easy to implement (however, although both the hardware and software descriptions of JH are simple, it requires some efforts to work out the relations between them).
2. The design of the compression F_d gives a simple and efficient way to construct a compression function from a bijective function.
3. EDP design gives a generalized design method of AES.
4. Security analysis can be performed relatively easily. Three approaches are used to achieve this goal. The first approach is to avoid introducing variables into the middle of the compression function so that the differential propagation can be analyzed relatively easily. The second approach is to use the simple EDP design that can greatly simplify the differential cryptanalysis. The third approach is that the EDP involves multidimensional array. The array with low dimension can be easily studied to estimate the strength of the high dimensional array.

5. High efficiency for collision resistance. Three approaches are used. The first approach is to use the EDP design that would likely maximize the difference propagation. The second approach is to minimize the difference cancellation within a compression function. The third approach is to ensure that every operation in a compression function is involved in at least one differential path if there is difference propagation within that compression function.

6. JH can be implemented efficiently over one-bit processor (hardware) to 128-bit processor (SIMD/SSE2). The reason is that EDP design allows JH being built from extremely simple components.

7. Hardware efficient. The hardware description of JH is simple. The internal state size of E_8 is only 1024 bits and the message block size is 512 bits. The round constants can be generated on the fly with 256-bit additional memory. Both the Sboxes and linear transformation in JH are extremely simple.

8. Software efficient. JH is designed to exploit the computational power of modern and widely used microprocessors. The bit-slice description of E_8 can be efficiently implemented with the SIMD/SSE2 instructions.

9. Several variants are available by varying the parameter d. The extremely hardware-efficient F_6 (with 30 rounds) is suitable for achieving 128-bit security for collision resistance, preimage resistance and second-preimage resistance.

10. It is convenient to use JH to substitute SHA2 [13] in almost all the SHA2 applications.

13 Conclusion

In this document, we proposed JH hash algorithms which are both hardware and software efficient. Our analysis shows that JH is very secure. However, the extensive security analysis of any new design requires a lot of efforts from many researchers. We thus invite and encourage researchers to analyze the security of JH. JH is not covered by any patent and JH is freely-available.

Acknowledgement. Part of the design was done when the author was studying at the research group COSIC of the Katholieke Universiteit Leuven.

References

[1] E. Biham, A. Shamir, “Differential Cryptanalysis of DES-like Cryptosystems.” *Advances in Cryptology – Crypto’90*, LNCS 537, pp. 2-21, Springer-Verlag, 1991.

[2] E. Biham, R. Chen, “Near-Collisions of SHA-0.” *Advances in Cryptology – CRYPTO 2004*, pp. 290–305, Springer-Verlag, 2004.

[3] E. Biham, R. Chen, A. Joux, P. Carribault, C. Lemuet, W. Jalby, “Collisions of SHA-0 and Reduced SHA–1.” *Advances in Cryptology – EUROCRYPT 2005*, pp. 36–57, Springer-Verlag, 2005.

[4] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin and C. Vikkelsoe, “PRESENT: An Ultra-Lightweight Block Cipher .” *Cryptographic Hardware and Embedded Systems – CHES 2007*, pp. 450–466, Springer-Verlag, 2007.

[5] F. Chabaud, A. Joux, “Differential Collisions in SHA-0.” *Advances in Cryptology – CRYPTO 1998*, pp. 56-71, Springer-Verlag, 1998.

[6] J. Daeman and V. Rijmen, “AES Proposal: Rijndael.” Available on-line from NIST at http://csrc.nist.gov/encryption/aes/rijndael/

[7] I. Dinur and A. Shamir, “Cube Attacks on Tweakable Black Box Polynomials.” IACR ePrint, 2008. Available at http://eprint.iacr.org/2008/385

[8] H. Dobbertin, “Cryptanalysis of MD4.” *Fast Software Encryption – FSE 1996*, pp. 53–69, Springer-Verlag, 1996.

[9] H. Feistel, “Cryptography and Computer Privacy.” Scientific American, vol.228(5), May 1973, pp 15–23.

[10] J. Kelsey, T. Kohno, “Herding Hash Functions and the Nostradamus Attack.” *Advances in Cryptology – EUROCRYPT 2006*, pp. 183–200, Springer-Verlag, 2006.

[11] L. Knudsen, “Truncated and Higher Order Differentials.” *Fast Software Encryption – FSE94*, pp. 196–211, Springer-Verlag.

[12] M. Matsui, “Linear Cryptanalysis Method for DES Cipher.” *Advances in Cryptology – Eurocrypt’93*, LNCS 765, pp. 386-397, Springer-Verlag, 1994.

[13] National Institute of Standards and Technology, “Secure Hash Standard (SHS).” Available at http://csrc.nist.gov/cryptval/shs.html

[14] X. Wang, X. Lai, D. Feng, H. Chen, X. Yu, “Cryptanalysis of the Hash Functions MD4 and RIPEMD.” *Advances in Cryptology – EUROCRYPT 2005*, pp. 1–18, Springer-Verlag, 2005.

[15] X. Wang, H. Yu, “How to Break MD5 and Other Hash Functions.” *Advances in Cryptology – EUROCRYPT 2005*, pp. 19–35, Springer-Verlag, 2005.

[16] X. Wang, H. Yu, Y. L. Yin, “Efficient Collision Search Attacks on SHA-0.” *Advances in Cryptology – CRYPTO 2005*, pp. 1–16, Springer-Verlag, 2005.

[17] X. Wang, Y. L. Yin, H. Yu, “Finding Collisions in the Full SHA-1.” *Advances in Cryptology – CRYPTO 2005*, pp. 17–36, Springer-Verlag, 2005.

A Round constants of E_8

This section gives the round constants in E_8. E_8 has 36 256-bit round constants.

A.1 Round constants in the hardware implementation of E_8

The round constants are generated from the first round constant using round function R_6 (with the round constants of R_6 being set to 0).

```
C00 = 6a09e667f3bcc908b2fb1366ea957d3e
      3adec17512775099da2f590b0667322a
C01 = bb896bf05955abcd5281828d66e7d99a
      c4203494f89bf12817deb43288712231
C02 = 1836e76b12d79c55118a1139d2417df5
      2a2021225ff6350063d88e5f1f91631c
C03 = 263085a7000fa9c3317c6ca8ab65f7a7
      713cf4201060ce886af855a90d6a4eed
C04 = 1cebafd51a156aeb62a11fb3be2e14f6
      0b7e48de85814270fd62e97614d7b441
C05 = e5564cb574f7e09c75e2e244929e9549
      279ab224a28e445d57185e7d7a09fdc1
C06 = 5820f0f0d764cff3a5552a5e41a82b9e
      ff6ee0aa615773bb07e8603424c3cf8a
C07 = b126fb741733c5bfcef6f43a62e8e570
      6a26656028aa897ec1ea4616ce8fd510
C08 = dbf0de32bca77254bb4f562581a3bc99
      1cf94f225652c27f14eae958ae6aa616
C09 = e6113be617f45f3de53cff03919a94c3
```

```
      2c927b093ac8f23b47f7189aadb9bc67
C10 = 80d0d26052ca45d593ab5fb310250639
      0083afb5ffe107dacfcba7dbe601a12b
C11 = 43af1c76126714dfa950c368787c81ae
      3beecf956c85c962086ae16e40ebb0b4
C12 = 9aee8994d2d74a5cdb7b1ef294eed5c1
      520724dd8ed58c92d3f0e174b0c32045
C13 = 0b2aa58ceb3bdb9e1eef66b376e0c565
      d5d8fe7bacb8da866f859ac521f3d571
C14 = 7a1523ef3d970a3a9b0b4d610e02749d
      37b8d57c1885fe4206a7f338e8356866
C15 = 2c2db8f7876685f2cd9a2e0ddb64c9d5
      bf13905371fc39e0fa86e1477234a297
C16 = 9df085eb2544ebf62b50686a71e6e828
      dfed9dbe0b106c9452ceddff3d138990
C17 = e6e5c42cb2d460c9d6e4791a1681bb2e
      222e54558eb78d5244e217d1bfcf5058
C18 = 8f1f57e44e126210f00763ff57da208a
      5093b8ff7947534a4c260a17642f72b2
C19 = ae4ef4792ea148608cf116cb2bff66e8
      fc74811266cd641112cd17801ed38b59
C20 = 91a744efbf68b192d0549b608bdb3191
      fc12a0e83543cec5f882250b244f78e4
C21 = 4b5d27d3368f9c17d4b2a2b216c7e74e
      7714d2cc03e1e44588cd9936de74357c
C22 = 0ea17cafb8286131bda9e3757b3610aa
      3f77a6d0575053fc926eea7e237df289
C23 = 848af9f57eb1a616e2c342c8cea528b8
      a95a5d16d9d87be9bb3784d0c351c32b
C24 = c0435cc3654fb85dd9335ba91ac3dbde
      1f85d567d7ad16f9de6e009bca3f95b5
C25 = 927547fe5e5e45e2fe99f1651ea1cbf0
      97dc3a3d40ddd21cee260543c288ec6b
C26 = c117a3770d3a34469d50dfa7db020300
      d306a365374fa828c8b780ee1b9d7a34
C27 = 8ff2178ae2dbe5e872fac789a34bc228
      debf54a882743caad14f3a550fdbe68f
C28 = abd06c52ed58ff091205d0f627574c8c
      bc1fe7cf79210f5a2286f6e23a27efa0
C29 = 631f4acb8d3ca4253e301849f157571d
      3211b6c1045347befb7c77df3c6ca7bd
C30 = ae88f2342c23344590be2014fab4f179
      fd4bf7c90db14fa4018fcce689d2127b
C31 = 93b89385546d71379fe41c39bc602e8b
```

```
      7c8b2f78ee914d1f0af0d437a189a8a4
C32 = 1d1e036abeef3f44848cd76ef6baa889
      fcec56cd7967eb909a464bfc23c72435
C33 = a8e4ede4c5fe5e88d4fb192e0a0821e9
      35ba145bbfc59c2508282755a5df53a5
C34 = 8e4e37a3b970f079ae9d22a499a714c8
      75760273f74a9398995d32c05027d810
C35 = 61cfa42792f93b9fde36eb163e978709
      fafa7616ec3c7dad0135806c3d91a21b
```

A.2 Round constants in the bit-slice implementation of E_8

Each round constant used in the bit-slice implementation of E_8 is linked to the corresponding round constant in the hardware implementation through a permutation.

```
C'00_even = 72d5dea2df15f8677b84150ab7231557
C'00_odd  = 81abd6904d5a87f64e9f4fc5c3d12b40
C'01_even = ea983ae05c45fa9c03c5d29966b2999a
C'01_odd  = 660296b4f2bb538ab556141a88dba231
C'02_even = 03a35a5c9a190edb403fb20a87c14410
C'02_odd  = 1c051980849e951d6f33ebad5ee7cddc
C'03_even = 10ba139202bf6b41dc786515f7bb27d0
C'03_odd  = 0a2c813937aa78503f1abfd2410091d3
C'04_even = 422d5a0df6cc7e90dd629f9c92c097ce
C'04_odd  = 185ca70bc72b44acd1df65d663c6fc23
C'05_even = 976e6c039ee0b81a2105457e446ceca8
C'05_odd  = eef103bb5d8e61fafd9697b294838197
C'06_even = 4a8e8537db03302f2a678d2dfb9f6a95
C'06_odd  = 8afe7381f8b8696c8ac77246c07f4214
C'07_even = c5f4158fbdc75ec475446fa78f11bb80
C'07_odd  = 52de75b7aee488bc82b8001e98a6a3f4
C'08_even = 8ef48f33a9a36315aa5f5624d5b7f989
C'08_odd  = b6f1ed207c5ae0fd36cae95a06422c36
C'09_even = ce2935434efe983d533af974739a4ba7
C'09_odd  = d0f51f596f4e81860e9dad81afd85a9f
C'10_even = a7050667ee34626a8b0b28be6eb91727
C'10_odd  = 47740726c680103fe0a07e6fc67e487b
C'11_even = 0d550aa54af8a4c091e3e79f978ef19e
C'11_odd  = 8676728150608dd47e9e5a41f3e5b062
C'12_even = fc9f1fec4054207ae3e41a00cef4c984
C'12_odd  = 4fd794f59dfa95d8552e7e1124c354a5
C'13_even = 5bdf7228bdfe6e2878f57fe20fa5c4b2
C'13_odd  = 05897cefee49d32e447e9385eb28597f
```

```
C’14_even = 705f6937b324314a5e8628f11dd6e465
C’14_odd  = c71b770451b920e774fe43e823d4878a
C’15_even = 7d29e8a3927694f2ddcb7a099b30d9c1
C’15_odd  = 1d1b30fb5bdc1be0da24494ff29c82bf
C’16_even = a4e7ba31b470bfff0d324405def8bc48
C’16_odd  = 3baefc3253bbd339459fc3c1e0298ba0
C’17_even = e5c905fdf7ae090f947034124290f134
C’17_odd  = a271b701e344ed95e93b8e364f2f984a
C’18_even = 88401d63a06cf61547c1444b8752afff
C’18_odd  = 7ebb4af1e20ac6304670b6c5cc6e8ce6
C’19_even = a4d5a456bd4fca00da9d844bc83e18ae
C’19_odd  = 7357ce453064d1ade8a6ce68145c2567
C’20_even = a3da8cf2cb0ee11633e906589a94999a
C’20_odd  = 1f60b220c26f847bd1ceac7fa0d18518
C’21_even = 32595ba18ddd19d3509a1cc0aaa5b446
C’21_odd  = 9f3d6367e4046bbaf6ca19ab0b56ee7e
C’22_even = 1fb179eaa9282174e9bdf7353b3651ee
C’22_odd  = 1d57ac5a7550d3763a46c2fea37d7001
C’23_even = f735c1af98a4d84278edec209e6b6779
C’23_odd  = 41836315ea3adba8fac33b4d32832c83
C’24_even = a7403b1f1c2747f35940f034b72d769a
C’24_odd  = e73e4e6cd2214ffdb8fd8d39dc5759ef
C’25_even = 8d9b0c492b49ebda5ba2d74968f3700d
C’25_odd  = 7d3baed07a8d5584f5a5e9f0e4f88e65
C’26_even = a0b8a2f436103b530ca8079e753eec5a
C’26_odd  = 9168949256e8884f5bb05c55f8babc4c
C’27_even = e3bb3b99f387947b75daf4d6726b1c5d
C’27_odd  = 64aeac28dc34b36d6c34a550b828db71
C’28_even = f861e2f2108d512ae3db643359dd75fc
C’28_odd  = 1cacbcf143ce3fa267bbd13c02e843b0
C’29_even = 330a5bca8829a1757f34194db416535c
C’29_odd  = 923b94c30e794d1e797475d7b6eeaf3f
C’30_even = eaa8d4f7be1a39215cf47e094c232751
C’30_odd  = 26a32453ba323cd244a3174a6da6d5ad
C’31_even = b51d3ea6aff2c90883593d98916b3c56
C’31_odd  = 4cf87ca17286604d46e23ecc086ec7f6
C’32_even = 2f9833b3b1bc765e2bd666a5efc4e62a
C’32_odd  = 06f4b6e8bec1d43674ee8215bcef2163
C’33_even = fdc14e0df453c969a77d5ac406585826
C’33_odd  = 7ec1141606e0fa167e90af3d28639d3f
C’34_even = d2c9f2e3009bd20c5faace30b7d40c30
C’34_odd  = 742a5116f2e032980deb30d8e3cef89a
C’35_even = 4bc59e7bb5f17992ff51e66e048668d3
C’35_odd  = 9b234d57e6966731cce6a6f3170a7505
```