New FORK-256

Deukjo Hong[1], Donghoon Chang[1], Jaechul Sung[2], Sangjin Lee[1], Seokhie Hong[1], Jesang Lee[1], Dukjae Moon[3], and Sungtaek Chee[3]

[1] Center for Information Security Technologies(CIST),
Korea University, Seoul, Korea
{hongdj,pointchang,sangjin,hsh,jslee}@cist.korea.ac.kr
[2] Department of Mathematics, University of Seoul, Seoul, Korea
jcsung@uos.ac.kr
[3] National Security Research Institute
{djmoon,chee}@etri.re.kr

Abstract. The hash function FORK-256 was published at the first NIST hash workshop and FSE 2006. It consists of simple operations so that its performance is better than that of SHA-256. However, recent papers show some weaknesses of FORK-256. In this paper, we propose newly modified FORK-256 which has no microcoliisions and so is resistant against existing attacks. Furthermore, it is faster than the old one.

1 Introduction

The hash function FORK-256 [1] was introduced at the first NIST hash workshop and at FSE 2006. Its performance is at least 30% better than that of SHA-256 in software. However, several recent papers [2–5] indicate 'microcollisions' exist inside of FORK-256 from the fact that inner functions f and g are not bijective, and suggest collision-finding attack using microcollisions.

- Matusiewicz, Contini, and Pieprzyk introduced microcollisions of FORK-256 by using the fact that the functions f and g in the step function are not bijective. They used microcollisions to find collisions of 2-branch FORK-256 in [2], and later, collisions of full FORK-256 with complexity of $2^{126.6}$ in [3].
- Independently, Mendel, Lano, and Preneel [5] published the collision-finding attack on 2-branch FORK-256 using microcollisions and raised possibility of its expansion.
- At FSE 2007 [4], Matusiewicz, Peyrin, Billet, Contini, and Pieprzyk published the result of [2, 3] and another attack which finds a collision with complexity of 2^{108} and memory of 2^{64}.

In this paper, we propose newly modified FORK-256 which has no microcoliisions and so is resistant against existing attacks. Furthermore, it is faster than the old one.

2 Modification of FORK-256

In this section, we describe modified points with the modification strategy. The compression function of FORK-256 consists of 4 parallel branch functions. Each branch function consists of 8 sequential step functions. Each step function has two different simple functions f and g with 32-bit inputs and outputs. In new FORK-256, f and g are modified as follows.

Old		New
$f(x) = x \boxplus (x^{\lll 7} \oplus x^{\lll 22})$	$\Rightarrow$	$f(x) = x \oplus x^{\lll 15} \oplus x^{\lll 27}$
$g(x) = x \oplus (x^{\lll 13} \boxplus x^{\lll 27})$		$g(x) = x \oplus (x^{\lll 7} \boxplus x^{\lll 25})$

Especially, the function f is changed from nonbijective to bijective. This change eliminates microcollisions in the step transformation, which have been crucial points of the attacks on old FORK-256. Moreover, f and g propagate the difference of a message word to the chaining variables.

We also modify the step function slightly. Two additions and two XORs are removed. 4 shift rotations are modified. We searched all the case and found candidate values so that the rank of the linearized step function is maximal.

3 Specification of New FORK-256

In this section, we describe the whole algorithm of new FORK-256. The following notations are used for the description of new FORK-256.

$$\begin{aligned} \boxplus &: \text{addition mod } 2^{32} \\ \oplus &: \text{XOR (eXclusive OR)} \\ A^{\lll s} &: s\text{-bit left shift rotation for a 32-bit string } A \\ |A|_{512} &: \text{the number of 512-bit blocks in a string } A \end{aligned}$$

3.1 Construction of FORK-256

FORK-256 employs Merkle-Damgård construction with the compression function $\mathsf{FORK256COMP}(\cdot,\cdot)$ and the padding method $\mathsf{PAD}(\cdot)$ as follows, where $CV_0 = IV$ is the initial value and M is the message.

```
FORK256HASH(CV_0, M)
    n ← |PAD(M)|_512;
    Partition PAD(M) into n 512-bit blocks M_0, ··· , M_{n−1};
    For i = 0 to n − 1
        CV_{i+1} ← FORK256COMP(CV_i, M_i);
    Return CV_n;
```

3.2 Message Block Length and Padding

The message block length of the compression function $\mathsf{FORK256COMP}$ is 512 bits. PAD pads a message by appending a single bit 1 next to the least significant bit of the message, followed by zero or more bit 0's until the length of the message is 448 modulo 512, and then appends to the message the 64-bit original message length modulo 2^{64}.

3.3 Structure of FORK-256 Compression Function

Fig. 1 depicts the outline of the compression function $\mathsf{FORK256COMP}$. $\mathsf{FORK256COMP}$ hashes a 768-bit string (a 512-bit message block plus a 256-bit chaining variable) to a 256-bit string. It consists of four parallel branch functions, BRANCH_1, BRANCH_2, BRANCH_3, and BRANCH_4. Let $CV_i = (CV_i[0], CV_i[1], \cdots, CV_i[7])$ where $CV_i[j]$ is a 32-bit word. The initial value CV_0 is set as follows:

$CV_0[0] =$ `0x6a09e667`	$CV_0[1] =$ `0xbb67ae85`
$CV_0[2] =$ `0x3c6ef372`	$CV_0[3] =$ `0xa54ff53a`
$CV_0[4] =$ `0x510e527f`	$CV_0[5] =$ `0x9b05688c`
$CV_0[6] =$ `0x1f83d9ab`	$CV_0[7] =$ `0x5be0cd19`

Let us see the computing procedure of the i-th iteration of $\mathsf{FORK256COMP}$. The message block M_i is partitioned to 16 32-bit words $(M_i[0], \cdots, M_i[15])$. Let $R_j^{(s)} = (R_j^{(s)}[0], \cdots, R_j^{(s)}[7])$ for $1 \le j \le 4$ and $0 \le s \le 8$ where each $R_j^{(s)}[t]$ is a 32-bit word for $0 \le t \le 7$. $R_j^{(8)}$ is the output of BRANCH_j on the inputs CV_i and M_i, for $1 \le j \le 4$ and computed as follows:

$$R_j^{(8)} = \mathsf{BRANCH}_j(CV_i, M_i) \quad \text{for } 1 \le j \le 4$$

where $R_j^{(s)}$'s are used in computation of BRANCH_j for $1 \le j \le 4$ and $0 \le s \le 7$. Consequently, $CV_{i+1} = (CV_{i+1}[0], \cdots, CV_{i+1}[7])$ is the output of the i-th iteration of $\mathsf{FORK256COMP}$ and computed as follows:

$$CV_{i+1}[t] = CV_i[t] \boxplus ((R_1^{(8)}[t] \boxplus R_2^{(8)}[t]) \oplus (R_3^{(8)}[t] \boxplus R_4^{(8)}[t])) \quad \text{for } 0 \le t \le 7.$$

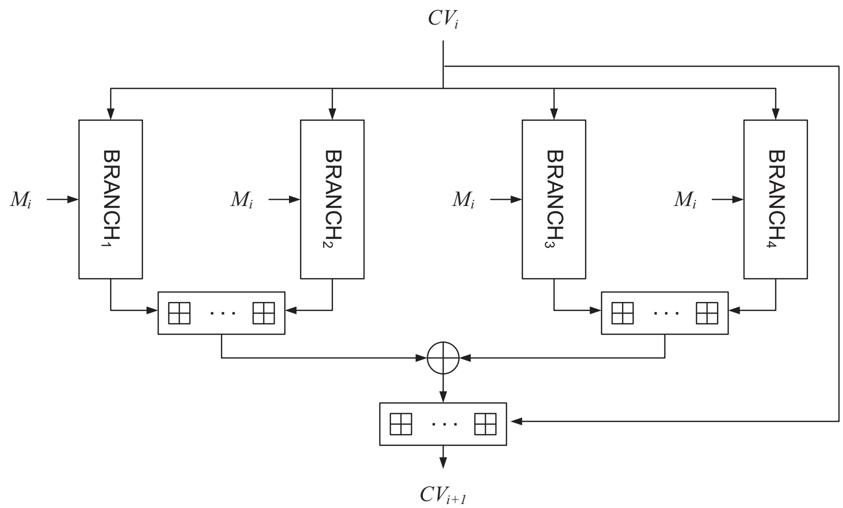

Fig. 1. Compression function of FORK-256, FORK256COMP

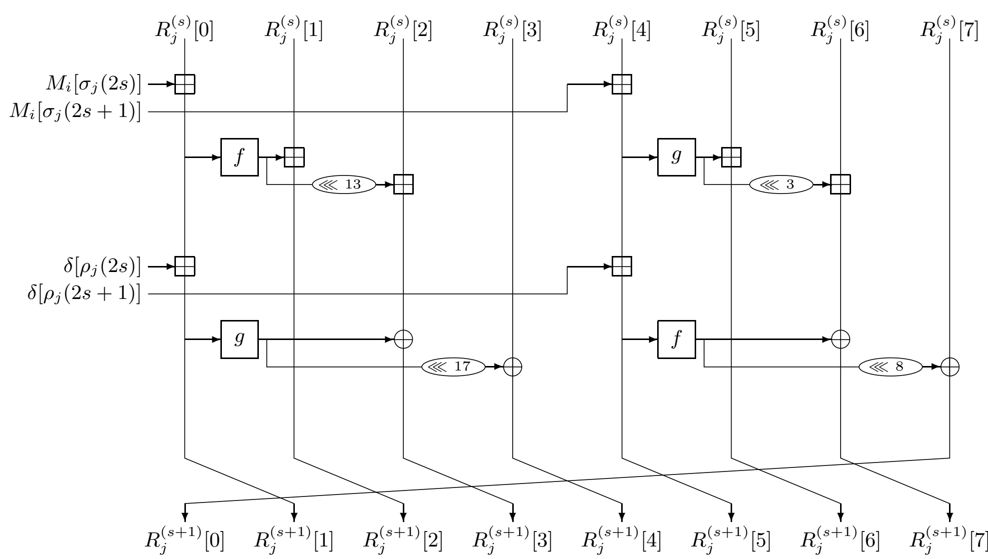

Fig. 2. Step function of FORK-256, STEP $(0 \le s \le 7, 1 \le j \le 4)$

3.4 Branch Function

Each BRANCH_j for $1 \leq j \leq 4$ is computed on the inputs CV_i and M_i as follows:

$$
\begin{array}{|l|}
\hline
\mathsf{BRANCH}_j(CV_i, M_i) \\
\quad R_j^{(0)} \leftarrow CV_i; \\
\quad \text{For } s = 0 \text{ to } 7 \\
\qquad R_j^{(s+1)} \leftarrow \mathsf{STEP}(R_j^{(s)}, M_i[\sigma_j(2s)], M_i[\sigma_j(2s+1)], \delta[\rho_j(2s)], \delta[\rho_j(2s+1)]); \\
\quad \text{Return } R_j^{(8)}; \\
\hline
\end{array}
$$

Message Word Ordering Each BRANCH_j for $1 \leq j \leq 4$ uses the message words $M_i[0], \cdots, M_i[15]$ with different order σ_j.

Table 1. Message word ordering

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\sigma_1(s)$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\sigma_2(s)$	14	15	11	9	8	10	3	4	2	13	0	5	6	7	12	1
$\sigma_3(s)$	7	6	10	14	13	2	9	12	11	4	15	8	5	0	1	3
$\sigma_4(s)$	5	12	1	8	15	0	13	11	3	10	9	2	7	14	4	6

Constants $\mathsf{FORK256COMP}$ totally uses sixteen constants:

$\delta[0] =$ `0x428a2f98`	$\delta[1] =$ `0x71374491`
$\delta[2] =$ `0xb5c0fbcf`	$\delta[3] =$ `0xe9b5dba5`
$\delta[4] =$ `0x3956c25b`	$\delta[5] =$ `0x59f111f1`
$\delta[6] =$ `0x923f82a4`	$\delta[7] =$ `0xab1c5ed5`
$\delta[8] =$ `0xd807aa98`	$\delta[9] =$ `0x12835b01`
$\delta[10] =$ `0x243185be`	$\delta[11] =$ `0x550c7dc3`
$\delta[12] =$ `0x72be5d74`	$\delta[13] =$ `0x80deb1fe`
$\delta[14] =$ `0x9bdc06a7`	$\delta[15] =$ `0xc19bf174`

These constants are used in each BRANCH_j with different order ρ_j for $1 \leq j \leq 4$.

Table 2. Constant ordering

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\rho_1(s)$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\rho_2(s)$	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$\rho_3(s)$	1	0	3	2	5	4	7	6	9	8	11	10	13	12	15	14
$\rho_4(s)$	14	15	12	13	10	11	8	9	6	7	4	5	2	3	0	1

Step Function In the s-th step of BRANCH_j for $1 \le j \le 4$ and $0 \le s \le 7$, STEP outputs $R_j^{(s+1)}$ on the inputs $R_j^{(s)}, M_i[\sigma_j(2s)], M_i[\sigma_j(2s+1)], \delta[\rho_j(2s)]$, and $\delta[\rho_j(2s+1)]$. $R_j^{(s+1)}$ is computed as follows (See Fig. 2):

$$
\begin{aligned}
R_j^{(s+1)}[0] &= R_j^{(s)}[7] \oplus f(R_j^{(s)}[4] \boxplus M_i[\sigma_j(2s+1)] \boxplus \delta[\rho_j(2s+1)])^{\lll 8},\\
R_j^{(s+1)}[1] &= R_j^{(s)}[0] \boxplus M_i[\sigma_j(2s)] \boxplus \delta[\rho_j(2s)],\\
R_j^{(s+1)}[2] &= R_j^{(s)}[1] \boxplus f(R_j^{(s)}[0] \boxplus M_i[\sigma_j(2s)])\\
R_j^{(s+1)}[3] &= R_j^{(s)}[2] \boxplus f(R_j^{(s)}[0] \boxplus M_i[\sigma_j(2s)])^{\lll 13} \oplus g(R_j^{(s)}[0] \boxplus M_i[\sigma_j(2s)] \boxplus \delta[\rho_j(2s)]),\\
R_j^{(s+1)}[4] &= R_j^{(s)}[3] \oplus g(R_j^{(s)}[0] \boxplus M_i[\sigma_j(2s)] \boxplus \delta[\rho_j(2s)])^{\lll 17},\\
R_j^{(s+1)}[5] &= R_j^{(s)}[4] \boxplus M_i[\sigma_j(2s+1)] \boxplus \delta[\rho_j(2s+1)],\\
R_j^{(s+1)}[6] &= R_j^{(s)}[5] \boxplus g(R_j^{(s)}[4] \boxplus M_i[\sigma_j(2s+1)])\\
R_j^{(s+1)}[7] &= R_j^{(s)}[6] \boxplus g(R_j^{(s)}[4] \boxplus M_i[\sigma_j(2s+1)])^{\lll 3} \oplus f(R_j^{(s)}[4] \boxplus M_i[\sigma_j(2s+1)] \boxplus \delta[\rho_j(2s+1)]).
\end{aligned}
$$

4 Performance

Table 3. Comparison of the performance of new FORK-256, old FORK-256, and SHA-256 which are implemented with Visual C++ (Ver 6.0) in Window XP Professional Version 2002, Pentium 4, CPU 3.2 GHz

New FORK-256	Old FORK-256	SHA-256
762.939 Mbps	538.942 Mbps	434.028 Mbps

5 Source Code

Here, we provide a source code for the compression function of FORK-256.

```
unsigned int delta[16] = {
        0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
        0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
        0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
        0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174
        };


#define ROL(x, n)   ( ( (x) << n ) | ( (x) >> (32-n) ) )

#define f(x)    (x ^ ROL(x,15)^ROL(x,27) ) #define g(x)    (x +
(ROL(x,7)^ROL(x,25)))


#define step(A,B,C,D,E,F,G,H, M1,M2,D1,D2)
\
    temp1 = A + M1;                                         \
    temp2 = E + M2;                                         \
    A = temp1 + D1;                                         \
    E = temp2 + D2;                                         \
    temp1 = f(temp1);                                       \
    temp2 = g(temp2);                                       \
    temp3 = g(A);                                           \
    temp4 = f(E);                                           \
    B += temp1;                                             \
    F += temp2;                                             \
    C = (C + ROL(temp1, 13)) ^ temp3;                       \
    G = (G + ROL(temp2, 3)) ^ temp4;                            \
    D ^= ROL(temp3, 17);                                        \
    H ^= ROL(temp4, 8);
```

```
static void FORK256_Compression_Function(unsigned int *CV, unsigned
int *M) {
    unsigned long R1[8],R2[8],R3[8],R4[8];
    unsigned long temp1, temp2, temp3, temp4;

    R1[0] = R2[0] = R3[0] = R4[0] = CV[0];
    R1[1] = R2[1] = R3[1] = R4[1] = CV[1];
    R1[2] = R2[2] = R3[2] = R4[2] = CV[2];
    R1[3] = R2[3] = R3[3] = R4[3] = CV[3];
    R1[4] = R2[4] = R3[4] = R4[4] = CV[4];
    R1[5] = R2[5] = R3[5] = R4[5] = CV[5];
    R1[6] = R2[6] = R3[6] = R4[6] = CV[6];
    R1[7] = R2[7] = R3[7] = R4[7] = CV[7];


    // BRANCH1(CV,M)
    step(R1[0],R1[1],R1[2],R1[3],R1[4],R1[5],R1[6],R1[7],M[0],M[1],delta[0],delta[1]);
    step(R1[7],R1[0],R1[1],R1[2],R1[3],R1[4],R1[5],R1[6],M[2],M[3],delta[2],delta[3]);
    step(R1[6],R1[7],R1[0],R1[1],R1[2],R1[3],R1[4],R1[5],M[4],M[5],delta[4],delta[5]);
    step(R1[5],R1[6],R1[7],R1[0],R1[1],R1[2],R1[3],R1[4],M[6],M[7],delta[6],delta[7]);
    step(R1[4],R1[5],R1[6],R1[7],R1[0],R1[1],R1[2],R1[3],M[8],M[9],delta[8],delta[9]);
    step(R1[3],R1[4],R1[5],R1[6],R1[7],R1[0],R1[1],R1[2],M[10],M[11],delta[10],delta[11]);
    step(R1[2],R1[3],R1[4],R1[5],R1[6],R1[7],R1[0],R1[1],M[12],M[13],delta[12],delta[13]);
    step(R1[1],R1[2],R1[3],R1[4],R1[5],R1[6],R1[7],R1[0],M[14],M[15],delta[14],delta[15]);

    // BRANCH2(CV,M)
    step(R2[0],R2[1],R2[2],R2[3],R2[4],R2[5],R2[6],R2[7],M[14],M[15],delta[15],delta[14]);
    step(R2[7],R2[0],R2[1],R2[2],R2[3],R2[4],R2[5],R2[6],M[11],M[9],delta[13],delta[12]);
    step(R2[6],R2[7],R2[0],R2[1],R2[2],R2[3],R2[4],R2[5],M[8],M[10],delta[11],delta[10]);
    step(R2[5],R2[6],R2[7],R2[0],R2[1],R2[2],R2[3],R2[4],M[3],M[4],delta[9],delta[8]);
    step(R2[4],R2[5],R2[6],R2[7],R2[0],R2[1],R2[2],R2[3],M[2],M[13],delta[7],delta[6]);
    step(R2[3],R2[4],R2[5],R2[6],R2[7],R2[0],R2[1],R2[2],M[0],M[5],delta[5],delta[4]);
    step(R2[2],R2[3],R2[4],R2[5],R2[6],R2[7],R2[0],R2[1],M[6],M[7],delta[3],delta[2]);
    step(R2[1],R2[2],R2[3],R2[4],R2[5],R2[6],R2[7],R2[0],M[12],M[1],delta[1],delta[0]);

    // BRANCH3(CV,M)
    step(R3[0],R3[1],R3[2],R3[3],R3[4],R3[5],R3[6],R3[7],M[7],M[6],delta[1],delta[0]);
    step(R3[7],R3[0],R3[1],R3[2],R3[3],R3[4],R3[5],R3[6],M[10],M[14],delta[3],delta[2]);
    step(R3[6],R3[7],R3[0],R3[1],R3[2],R3[3],R3[4],R3[5],M[13],M[2],delta[5],delta[4]);
    step(R3[5],R3[6],R3[7],R3[0],R3[1],R3[2],R3[3],R3[4],M[9],M[12],delta[7],delta[6]);
    step(R3[4],R3[5],R3[6],R3[7],R3[0],R3[1],R3[2],R3[3],M[11],M[4],delta[9],delta[8]);
    step(R3[3],R3[4],R3[5],R3[6],R3[7],R3[0],R3[1],R3[2],M[15],M[8],delta[11],delta[10]);
    step(R3[2],R3[3],R3[4],R3[5],R3[6],R3[7],R3[0],R3[1],M[5],M[0],delta[13],delta[12]);
    step(R3[1],R3[2],R3[3],R3[4],R3[5],R3[6],R3[7],R3[0],M[1],M[3],delta[15],delta[14]);

    // BRANCH4(CV,M)
    step(R4[0],R4[1],R4[2],R4[3],R4[4],R4[5],R4[6],R4[7],M[5],M[12],delta[14],delta[15]);
    step(R4[7],R4[0],R4[1],R4[2],R4[3],R4[4],R4[5],R4[6],M[1],M[8],delta[12],delta[13]);
    step(R4[6],R4[7],R4[0],R4[1],R4[2],R4[3],R4[4],R4[5],M[15],M[0],delta[10],delta[11]);
    step(R4[5],R4[6],R4[7],R4[0],R4[1],R4[2],R4[3],R4[4],M[13],M[11],delta[8],delta[9]);
    step(R4[4],R4[5],R4[6],R4[7],R4[0],R4[1],R4[2],R4[3],M[3],M[10],delta[6],delta[7]);
    step(R4[3],R4[4],R4[5],R4[6],R4[7],R4[0],R4[1],R4[2],M[9],M[2],delta[4],delta[5]);
    step(R4[2],R4[3],R4[4],R4[5],R4[6],R4[7],R4[0],R4[1],M[7],M[14],delta[2],delta[3]);
    step(R4[1],R4[2],R4[3],R4[4],R4[5],R4[6],R4[7],R4[0],M[4],M[6],delta[0],delta[1]);

    // output
    CV[0] = CV[0] + ((R1[0] + R2[0]) ^ (R3[0] + R4[0]));
    CV[1] = CV[1] + ((R1[1] + R2[1]) ^ (R3[1] + R4[1]));
    CV[2] = CV[2] + ((R1[2] + R2[2]) ^ (R3[2] + R4[2]));
    CV[3] = CV[3] + ((R1[3] + R2[3]) ^ (R3[3] + R4[3]));
    CV[4] = CV[4] + ((R1[4] + R2[4]) ^ (R3[4] + R4[4]));
    CV[5] = CV[5] + ((R1[5] + R2[5]) ^ (R3[5] + R4[5]));
    CV[6] = CV[6] + ((R1[6] + R2[6]) ^ (R3[6] + R4[6]));
    CV[7] = CV[7] + ((R1[7] + R2[7]) ^ (R3[7] + R4[7]));
}
```

6 Test Vector

Message M (1 block)

00112233 44556677 88990011 22334455 66778899 00112233 44556677 88990011
22334455 66778899 00112233 44556677 88990011 22334455 66778899 00112233

Output of Compression Function CV_1

c07dd7ab 444a1014 1f99581e 4e928ebe a6cddbdd 562ca48a 9398df6e 95829af4

Intermediate Values

BRANCH_1

$R_1^{(0)}$ = 6a09e667 bb67ae85 3c6ef372 a54ff53a 510e527f 9b05688c 1f83d9ab 5be0cd19
$R_1^{(1)}$ = 3649eb59 aca53832 f86e9458 fd43d04a b3fe3def 069afd87 8d5fddbd f23a2a1c
$R_1^{(2)}$ = 1e82df40 74a3e739 4b45db72 9b686a81 7818ff37 bfe75de9 6e39c13a 96b943ad
$R_1^{(3)}$ = b0ba8f38 be512a34 efd55dd3 7a23c8e7 5f166af4 d21b335b f9f260d1 abb7dbb7
$R_1^{(4)}$ = 1b692a15 874f7853 2ec19ab9 75e42467 252ed8e3 92cbc9da 96457a85 b9f444d6
$R_1^{(5)}$ = 5fcace7a 15a41902 e2950c2a 0af7641b b191f3f0 9e29bc7d 489a1ddd e85e2428
$R_1^{(6)}$ = 593db6dc 840d766b e31799c7 076d6041 41c5cf9a 4af3d82a d0581432 2ebdb814
$R_1^{(7)}$ = 2934117d 54951461 59bc6a1c 1d35fa9f 5aaad931 e4d7c5ed d13af1af 6b9769f7
$R_1^{(8)}$ = b2cafcdd 2b87a0bd 4b729574 ecd01a66 f8163082 1c57ecd8 d4dc872c 7bb9a63b

BRANCH_2

$R_2^{(0)}$ = 6a09e667 bb67ae85 3c6ef372 a54ff53a 510e527f 9b05688c 1f83d9ab 5be0cd19
$R_2^{(1)}$ = 6bb63387 921d6074 1cecbabd a4449f5f 64b50658 ecfb7b59 d73d44ff 9f72ebba
$R_2^{(2)}$ = a85718d8 30ea4bfc 1b91fda8 0ebd3d60 4953e303 3deaec65 2df92c42 bd1de9ba
$R_2^{(3)}$ = 3d158131 1f96daf0 bb33367d 720b1e5b 6fe9a8b2 6d968af4 656042c9 e223d70b
$R_2^{(4)}$ = 0869dfd5 71cc2087 2f0886fe aa0b04eb 8b0bab24 ae68dbe3 eb2c23c8 8ce74364
$R_2^{(5)}$ = 07f9718b 3c1f3ebb 3c45a21f e93d02d3 ef3668e6 3f7e721d c7d58c64 591ab9e2
$R_2^{(6)}$ = 39b1e94d 61fba5af edb501e1 f1a6befb 646908ca 289e4d74 bee10117 6d8aaf67
$R_2^{(7)}$ = edc74112 67bd2b69 5c10f068 e31bbb6d 053a56d4 a2c304aa 4c7ec036 a864dac6
$R_2^{(8)}$ = a08152a0 e79785b4 b5002383 32b1413c 1542a827 8c19ece3 3da07cd3 4562640a

BRANCH_3

$R_3^{(0)}$ = 6a09e667 bb67ae85 3c6ef372 a54ff53a 510e527f 9b05688c 1f83d9ab 5be0cd19
$R_3^{(1)}$ = 4e0cf14c 63da2b09 0173369f cede67c3 12af3262 d7ede88e 8d5fddbd e7426f06
$R_3^{(2)}$ = 12de60a2 37d3ef24 21ab6ff4 f37c8ab0 870dd51b 2ee7b6ca b6c3d452 20faa7ec
$R_3^{(3)}$ = 5925e255 8f02b6e8 9696a27c a75e53de a3593420 48fd9787 ca0467a3 36d0845b
$R_3^{(4)}$ = 9c34ff93 6ab9c9c3 0e19955f 368ec9b7 108cda08 be31b6d5 103dc8b5 ccaf1562
$R_3^{(5)}$ = 2465c3e1 f30dc10b ef450f42 f2ddda76 8983c19f 4f0c0d39 f61571f4 4cb36b78
$R_3^{(6)}$ = 021b7064 798363d7 e96d042a 3f6f5f5f 50780bfa cfe88bb2 2d98a78b f634ec91
$R_3^{(7)}$ = 7b62b399 830b4495 6cf9daec 89e06245 7ed53c00 c3478ba1 3ea7bed3 ab1f16c9
$R_3^{(8)}$ = d6a3af57 81540b84 ba58c9b1 39c6140e 62562af3 3ce486fc 935247c6 20801cb8

BRANCH_4

$R_4^{(0)}$ = 6a09e667 bb67ae85 3c6ef372 a54ff53a 510e527f 9b05688c 1f83d9ab 5be0cd19
$R_4^{(1)}$ = c0f34804 05f70f41 f86e9458 66aa06cf 1ef50a3c 9b434404 66c6c1e5 6015b7fc
$R_4^{(2)}$ = 30cc3fc0 78070bef 905678ed b0d92039 6d9eac35 c207008f 9310aad2 196121d4
$R_4^{(3)}$ = a52e356e 550ee7b1 91a91e81 e8401137 eaf555e8 c2bc4c2b 36f33aa1 80ce3471
$R_4^{(4)}$ = f0890839 9f69245b baca796e a0f6e8da af85571b 41ce1760 0d07c379 c13eb22f
$R_4^{(5)}$ = b7dd0903 a4fbcf32 e3d7cc0f 326b3c05 db0b938e 5ab2d823 47c81c53 10a108aa
$R_4^{(6)}$ = a1f77e40 57ab53f7 5b64096c 79ad23df 306c1961 bd95a590 aae5f258 23d1bb8b
$R_4^{(7)}$ = 71a78449 e0517a20 f497bce2 a14e51ad 20e5b041 80997d9f d9768192 4bbea457
$R_4^{(8)}$ = 2e9c0ee2 1aa93c7a 290012aa 7cfdae18 f6912704 d6725b49 d315be76 d83daae6

7 Erratum in FSE 2006 version of FORK-256

The figure of the step function in [1] is totally wrong, but that in the preproceeding version of FSE 2006 is correct. Please be careful for referring to them.

References

1. D. Hong, D. Chang, J. Sung, S. Lee, S. Hong, J. Lee, D. Moon, S. Chee, "A New Dedicated 256-Bit Hash Function: FORK-256", *FSE 2006*, LNCS 4047, Springer-Verlag, pp. 195–209, 2006.

2. K. Matusiewicz, S. Contini, J. Pieprzyk, "Collisions for Two Branches of FORK-256", Cryptology ePrint Archive 2006/317 (First version), Sep., 2006.
3. K. Matusiewicz, S. Contini, J. Pieprzyk, "Weaknesses of the FORK-256 Compression Function", Cryptology ePrint Archive 2006/317 (Second version), Nov., 2006.
4. K. Matusiewicz, T. Peyrin, O. Billet, S. Contini, and J. Pieprzyk, "Cryptanalysis of FORK-256", Preproceeding of *FSE 2007*, 2007.
5. F. Mendel, J. Lano, B. Preneel, "Cryptanalysis of Reduced Variants of the FORK-256 Hash Function", *CT-RSA*, LNCS 4377, Springer-Verlag, pp. 85–100, 2007.