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.

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

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($CV_0, M$)
    $n \leftarrow |\mathsf{PAD}(M)|_{512}$;
    Partition $\mathsf{PAD}(M)$ into $n$ 512-bit blocks $M_0, \cdots, M_{n-1}$;
    For $i = 0$ to $n-1$
        $CV_{i+1} \leftarrow \mathsf{FORK256COMP}(CV_i, M_i)$;
    Return $CV_n$;
```

3.2 Message Block Length and Padding

The message block length of the compression function 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 FORK256COMP. 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 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 FORK256COMP and computed as follows:

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

3.4 Branch Function

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

```
$\mathsf{BRANCH}_j(CV_i, M_i)$
    $R_j^{(0)} \leftarrow CV_i$;
    For $s = 0$ to $7$
        $R_j^{(s+1)} \leftarrow \mathsf{STEP}(R_j^{(s)}, M_i[\sigma_j(2s)], M_i[\sigma_j(2s+1)], \delta[\rho_j(2s)], \delta[\rho_j(2s+1)])$;
    Return $R_j^{(8)}$;
```

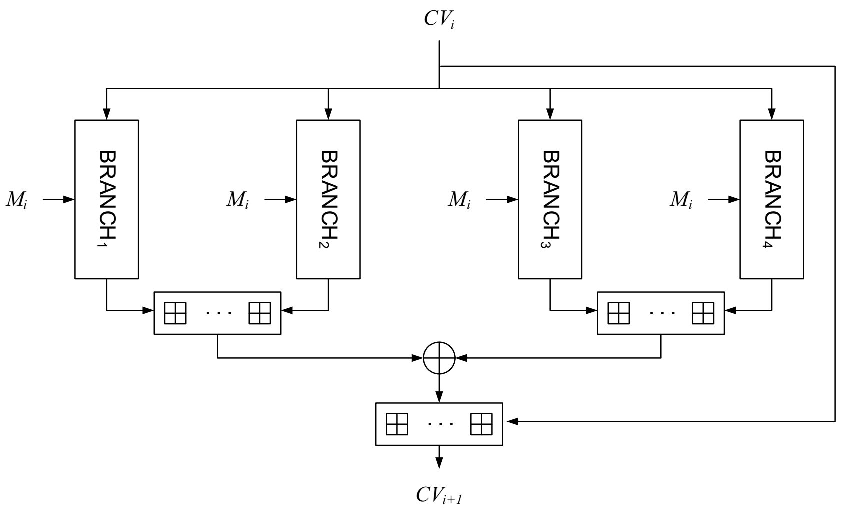

Fig. 1. Compression function of FORK-256, FORK256COMP

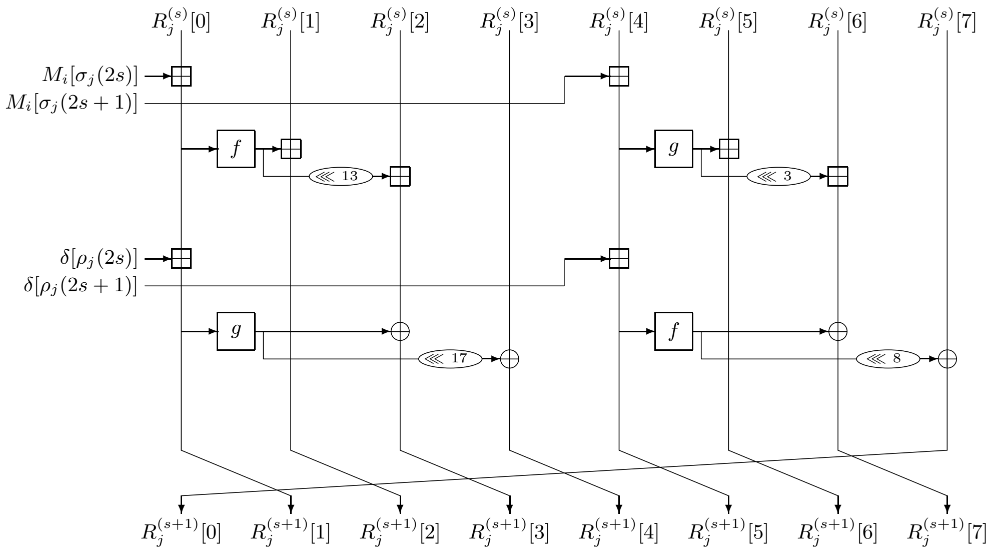

Fig. 2. Step function of FORK-256, STEP $(0 \le s \le 7, 1 \le j \le 4)$

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

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.

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 $1 \leq j \leq 4$ and $0 \leq s \leq 7$, 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
`00df9461 b558ce10 43e8024c 3a10fb1e 2a8fad8d 19ed9c5d a50486cb f15365d9`

Intermediate Values
BRANCH$_1$
$R_1^{(0)} =$ `6a09e667 bb67ae85 3c6ef372 a54ff53a 510e527f 9b05688c 1f83d9ab 5be0cd19`
$R_1^{(1)} =$ `3649eb59 aca53832 f86e9458 027900e6 12a7c39a 069afd87 a56a62d9 32c60535`
$R_1^{(2)} =$ `9cadc5fa 74a3e739 4b45db72 bca9ab9d 051a6018 1e90e394 85d1a7cd 50b1decc`
$R_1^{(3)} =$ `aaeecd3b 3c7c10ee b119ccae 2109fcd1 430be667 5f1c943c 4d71e261 0b23c125`
$R_1^{(4)} =$ `dad7c69a 8183b656 9e63767b 7296fc10 28fc226a 76c1454d 6ecaf485 753510ae`
$R_1^{(5)} =$ `8fdbdfa9 d512b587 89186293 7ffd4bbf cb32643c a1f70604 c649076b ecf3ebb8`
$R_1^{(6)} =$ `672b944b b41e879a c08ee3ab fe6f8df1 641aaada 64944876 c77aa9b3 01edfc9b`
$R_1^{(7)} =$ `82aa4ea4 6282f1d0 f6b34eb6 d9cd7439 c1e68c8f 072ca12d 64de2504 f64ac991`
$R_1^{(8)} =$ `62363776 84fddde4 50791ee4 7f50dc6d b9fef233 8393a036 47d99fac 8dd26400`

BRANCH$_2$
$R_2^{(0)} =$ `6a09e667 bb67ae85 3c6ef372 a54ff53a 510e527f 9b05688c 1f83d9ab 5be0cd19`
$R_2^{(1)} =$ `6bb63387 921d6074 1cecbabd b83ceadf 8fb53ea8 ecfb7b59 4049552f d7936836`
$R_2^{(2)} =$ `215a8d67 30ea4bfc 1b91fda8 edfed2b8 8a9a5004 68eb24b5 595eb921 f2958a97`
$R_2^{(3)} =$ `a252aeef 989a4f7f 7a39eaf3 dbb25dc4 9bd57b3d aedcf7f5 b7502553 b4d67937`
$R_2^{(4)} =$ `05b04ecc d7094e45 b39e0e1b 35511591 2c37d37c da54ae6e 7fa87ed3 c61d3a7e`
$R_2^{(5)} =$ `5323c97c 3965adb2 9c5f11d4 423c0f00 26db49c9 e0aa9a75 70985775 3350f9fc`
$R_2^{(6)} =$ `59d7fdb0 ad25fda0 95e0931a 8070fc92 81defdf9 60432e57 291a578b f87cc723`
$R_2^{(7)} =$ `2da2cc51 87e33fcc bdf5bdb0 80ae063d a66ce60b c038f9d9 baa73962 8eef71ee`
$R_2^{(8)} =$ `a5e62d2c 277310f3 cd9e88e8 9fbd0920 1e217775 2d4c7c1a 4c728465 de58849a`

BRANCH$_3$
$R_3^{(0)} =$ `6a09e667 bb67ae85 3c6ef372 a54ff53a 510e527f 9b05688c 1f83d9ab 5be0cd19`
$R_3^{(1)} =$ `4e0cf14c 63da2b09 0173369f 31752e3d 8152cd34 d7ede88e a56a62d9 27be402f`
$R_3^{(2)} =$ `8efb95fe 37d3ef24 21ab6ff4 2fa47884 9ccf2554 9d8b519c 3f1e16e1 0f4290a1`
$R_3^{(3)} =$ `a47d318e 0b1fec44 7962633a e33db5bc 34965ea3 5ebee7c0 f8c0f955 37606b0a`
$R_3^{(4)} =$ `8ed18ef7 b61118fc 7ad086d0 3b5a4940 93dcda6f 4f6ee158 1ac5cf6f 25418737`
$R_3^{(5)} =$ `3bc564e5 e5aa506f 9519c452 c162be4c 82a0ebf3 d25c0da0 15e12fa3 9c46c52e`
$R_3^{(6)} =$ `d2b9f10a 90e304db 9f2e7f3a a2b40c88 af77cec2 c905b606 32120a9a 37dfe841`
$R_3^{(7)} =$ `563ad18d 53a9c53b 46a6a79c 0ba681fa caa43cc8 22474e69 c9653253 f96c083b`
$R_3^{(8)} =$ `ed33ab54 5c2c2978 f5f0b9c7 7fd9061c 70563438 88b387c4 6d950b4e 4cc746de`

BRANCH$_4$
$R_4^{(0)} =$ `6a09e667 bb67ae85 3c6ef372 a54ff53a 510e527f 9b05688c 1f83d9ab 5be0cd19`
$R_4^{(1)} =$ `c0f34804 05f70f41 f86e9458 614ddaf3 a68d05f3 9b434404 c8012e0d 9af91631`
$R_4^{(2)} =$ `6bd67d34 78070bef 905678ed 7fd52b9f 7d34ee11 499efc46 d4ba32b4 96498ce5`
$R_4^{(3)} =$ `55bad24e 90192525 1712225e 348f8cf9 bfbaa4c8 d2528e07 9ae3ba60 f523d0d8`
$R_4^{(4)} =$ `f2d5eb7d 4ff5c13b f71a0306 0cbb072c a4abf013 16936640 547042ae 0ed6a9a7`
$R_4^{(5)} =$ `24cbd68a a748b276 623e2d63 32707b8e f1396af2 4fd9711b 6602f570 e2c6a14f`
$R_4^{(6)} =$ `c5c85384 c49a217e dfd132f1 fc4e46ae 97c56789 d3c37cf4 d9d4beac 7efa6f0e`
$R_4^{(7)} =$ `50540fb3 04224f64 12731fb7 2295e568 8fc7fcb5 e7f2cbc7 6e0bf871 113634f8`
$R_4^{(8)} =$ `b1961e04 f955c7e4 237def4f 0bf3da4d 914afe6e 4554a7bd a4377de3 ac91297c`

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.