A New Dedicated 256-bit Hash Function: FORK-256

Deukjo Hong[1], Donghoon Chang[1], Jaechul Sung[2], Sangjin Lee[1], Seokhie Hong[1], Jaesang 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. This paper describes a new software-efficient 256-bit hash function, FORK-256. Recently proposed attacks on MD5 and SHA-1 motivate a new hash function design. It is designed not only to have higher security but also to be faster than SHA-256. The performance of the new hash function is at least 30% better than that of SHA-256 in software. And it is secure against any known cryptographic attacks on hash functions.

Keywords: 256-bit Hash Function, FORK-256

1 Introduction

For cryptographic hash function, the following properties are required:

- **preimage resistance**: it is computationally infeasible to find any input which hashes to any pre-specified output.
- **second preimage resistance**: it is computationally infeasible to find any second input which has the same output as any specified input.
- **collision resistance**: it is computationally infeasible to find a collision, i.e. two distinct inputs that hash to the same result.

For an ideal hash function with an m-bit output, finding a preimage or a second preimage requires about 2^m operations and the fastest way to find a collision is a birthday attack which needs approximately $2^{m/2}$ operations.

Most dedicated hash functions which have iterative process use the Merkle-Damgård construction [6, 10] in order to hash inputs of arbitrary length. They work as follows. Let HASH be a hash function. The message M is padded to a multiple of the block length and subsequently divided into n blocks $M_0, \cdots, M_{n-1}$. Then HASH can be described as follows:

$$CV_0 = IV; \qquad CV_{i+1} = \mathsf{COMP}(CV_i, M_i), 0 \le i \le n-1; \qquad \mathsf{HASH}(M) = CV_n,$$

where COMP is the compression function of HASH, CV_i's are chaining variables, and IV is a fixed initial value.

The most popular method of designing compression functions of dedicated hash functions is a serial successive iteration of a small step function, as like round functions of block ciphers. Many hash functions such as MD4 [12], MD5 [13], HAVAL [19], SHA-family [11], etc., follow that idea. Attacks on hash functions have been focused on vanishing the difference of intermediate values caused by the difference of messages. On the other hand, a hash function has been considered secure if it is computationally hard to vanish such difference in its compression function. Usually, the lower the probability of the differential characteristic is, the harder the attack is. Therefore a step function is regarded as a good candidate if it causes a good avalanche effect in the serial structure. A function which has a good diffusion property can not be so light in general. However, most step functions have been developed to be light for efficiency. This may be why MD4-type hash functions including SHA-1 are vulnerable to Wang et al.'s collision-finding attack [15–18].

RIPEMD-family [9] has somewhat different approach for designing a secure hash function. The attacker who tries to break members of RIPEMD-family should aim simultaneously at two ways where the message difference passes. This design strategy is still successful because so far there is not any effective attack on RIPEMD-family except the first proposal of RIPEMD. However, RIPEMD-family have heavier compression functions than hash functions with serial structure. For example, the first proposal of RIPEMD consists of two lines of MD4. Total number of steps is twice as many as that of MD4. Also, the number of steps of RIPEMD-160 is almost twice as many as that of SHA-0.

In this paper, we propose a new dedicated hash function FORK-256. According to the above observation, we determined the design goals (of compression function) as follows.

- It should have a 256-bit output because the security of 2^{128} operations is recommended for symmetric key cryptography as the computing power increases.
- Its structure should be resistant against known attacks including Wang et al.'s attack [1–5, 7, 8, 14–18].
- The performance should be as competitive as that of SHA-256.

2 Description of FORK-256

In this section, we will describe FORK-256. These are basic notations used in FORK-256.

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

2.1 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: For the initial value $CV_0 = IV$ and the message M,

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

2.2 Message Block Length and Padding

The message block length of the compression function $\mathsf{FORK256COMP}$ is 512-bit. 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}.

2.3 Structure of FORK-256 Compression Function

Fig. 1 depicts the outline of the compression function $\mathsf{FORK256COMP}$. The name 'FORK' was originated from the figure. $\mathsf{FORK256COMP}$ hashes a 512-bit string 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:

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

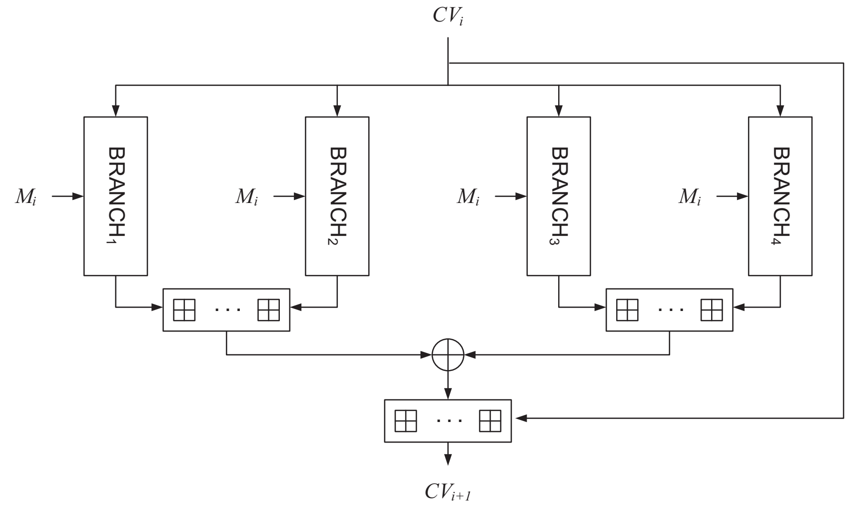

Fig. 1. Outline of the FORK-256 compression function, FORK256COMP

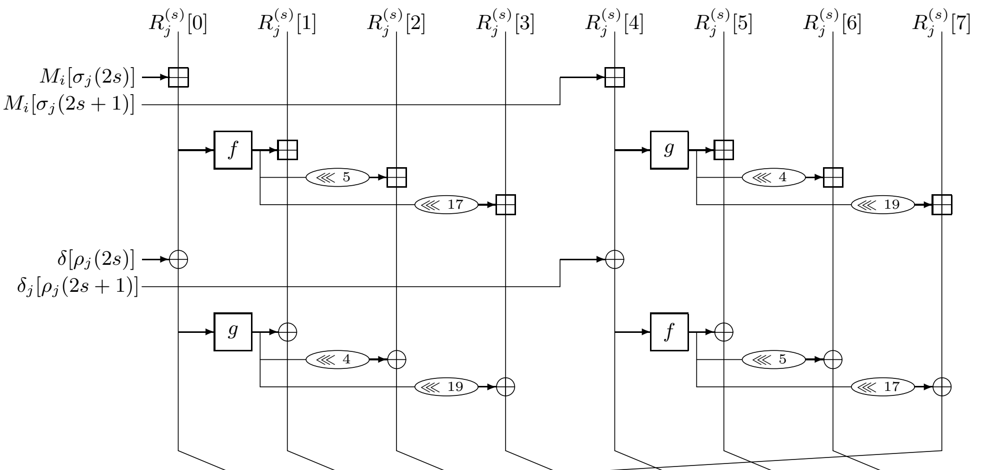

Fig. 2. Step function of FORK-256, STEP $(0 \leq s \leq 7, 1 \leq j \leq 4)$

2.4 Branch Function

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

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

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 $1 \leq j \leq 4$.

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 $R_j^{(s)}, M_i[\sigma_j(2s)], M_i[\sigma_j(2s+1)], \delta[\rho_j(2s)]$,

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

and $\delta[\rho_j(2s+1)]$, and $R_j^{(s+1)}$ is computed as follows (See Fig. 2):

$$\begin{aligned}
R_j^{(s+1)}[0] &= R_j^{(s)}[7] \boxplus g(R_j^{(s)}[4] \boxplus M_i[\sigma_j(2s+1)])^{\lll 21} \\
&\quad \oplus f(R_j^{(s)}[4] \boxplus M_i[\sigma_j(2s+1)] \boxplus \delta[\rho_j(2s+1)])^{\lll 17}, \\
R_j^{(s+1)}[1] &= R_j^{(s)}[0] \boxplus M_i[\sigma_j(2s)] \boxplus \delta[\rho_j(2s)], \\
R_j^{(s+1)}[2] &= R_j^{(s)}[1] \boxplus f(R_j^{(s)}[0] \boxplus M_i[\sigma_j(2s)]) \\
&\quad \oplus g(R_j^{(s)}[0] \boxplus M_i[\sigma_j(2s)] \boxplus \delta[\rho_j(2s)]), \\
R_j^{(s+1)}[3] &= R_j^{(s)}[2] \boxplus f(R_j^{(s)}[0] \boxplus M_i[\sigma_j(2s)])^{\lll 5} \\
&\quad \oplus g(R_j^{(s)}[0] \boxplus M_i[\sigma_j(2s)] \boxplus \delta[\rho_j(2s)])^{\lll 9}, \\
R_j^{(s+1)}[4] &= R_j^{(s)}[3] \boxplus f(R_j^{(s)}[0] \boxplus M_i[\sigma_j(2s)])^{\lll 17} \\
&\quad \oplus g(R_j^{(s)}[0] \boxplus M_i[\sigma_j(2s)] \boxplus \delta[\rho_j(2s)])^{\lll 21}, \\
R_j^{(s+1)}[5] &= R_j^{(s)}[4] \boxplus M_i[\sigma_j(2s+1)] \boxplus \delta[\rho_j(2s+1)], \\
R_j^{(s+1)}[6] &= R_j^{(s)}[5] \boxplus g(R_j^{(s)}[4] \boxplus M_i[\sigma_j(2s+1)]) \\
&\quad \oplus f(R_j^{(s)}[4] \boxplus M_i[\sigma_j(2s+1)] \boxplus \delta[\rho_j(2s+1)]), \\
R_j^{(s+1)}[7] &= R_j^{(s)}[6] \boxplus g(R_j^{(s)}[4] \boxplus M_i[\sigma_j(2s+1)])^{\lll 9} \\
&\quad \oplus f(R_j^{(s)}[4] \boxplus M_i[\sigma_j(2s+1)] \boxplus \delta[\rho_j(2s+1)])^{\lll 5},
\end{aligned}$$

where f and g are nonlinear functions as follows:

$$\begin{aligned}
f(x) &= x \boxplus (x^{\lll 7} \oplus x^{\lll 22}), \\
g(x) &= x \oplus (x^{\lll 13} \boxplus x^{\lll 27}).
\end{aligned}$$

3 Design Strategy

3.1 Motivation for Our Proposal

In 2004, Wang et al.'s attacks on MD4, MD5, HAVAL, and RIPEMD [15, 16] and SHA-0/1 [17, 18] brought a big impact on the field of symmetric key cryptography including hash function. However, RIPEMD-128/160 are the algorithms which are still secure against their attacks. No attacks on them are found so far.

They were designed to have two parallel lines, which is different from MD4, MD5 and SHA-family. This makes an attacker take into account two lines simultaneously. However, since each line needs almost same operation of MD5 and SHA algorithms, its efficiency was degenerated almost half of them. This motivates our design. We use four lines instead of two.

In order to overcome disadvantage of RIPEMD algorithms, we manage to reduce operations for step functions of each line. The message reordering of each branch is deliberately designed to be resistant against Wang et al.'s attack and differential attacks. The function f and g in each step are chosen to have good avalanche effects.

3.2 Design Principle

Structure The compression function $\mathsf{FORK256COMP}$ consists of 4 Branches. In the security aspect, we can give the security against known attacks with the different message ordering in branches. For example, RIPEMD, which consists of 2 branches, was fully attacked by Wang et al. [15] because RIPEMD has the same message ordering in 2 branches. On the other hand, in case of RIPEMD-128/160, there is no attack result because RIPEMD-128/160 have different message ordering in branches. In the implementation aspect, FORK-256 can be implemented efficiently because the message ordering is simpler than the message expansion such as that of SHA-256.

Constants Each BRANCH_j uses 16 different constants $\delta[t]$ for $0 \leq t \leq 15$. By using constants we pursue the goal to disturb the attacker who tries to find a good differential characteristic with a relatively high probability. So, we prefer the constants which represent the first 32 bits of the fractional parts of the cube roots of the first sixteen four prime numbers.

Nonlinear Functions The nonlinear functions f and g of $\mathsf{FORK256COMP}$ output one word on the input of one word, and their outputs are XORed or added modulo 2^{32} to the multiple words in the chaining variables. Due to this property, f and g propagate the difference of a message word to the chaining variables.

Shift Rotations in Nonlinear Functions If the addition is changed into the bitwise xor operation in f and g, nonlinear functions are generalized as $x \oplus (x^{\lll s_1} \oplus x^{\lll s_2})$. We consider all $465 (= \binom{31}{2})$ cases for s_1 and s_2 and want to define shift rotations satisfying the following 7 conditions. HW(w) denotes the Hamming Weight of a 32-bit word w. Let x be an input of f or g and let y be $f(x)$ or $g(x)$.

- The branch number of f and g is 4.
- If HW(x) = 2, then HW(y) $\geq$ 4.
- If HW(x) = 3, then HW(y) $\geq$ 3.

– If HW(x) = 4, then HW(y) $\geq$ 4.
– If HW(y) = 1, then HW(x) $\geq$ 17.
– If HW(y) = 2, then HW(x) $\geq$ 14.
– The interval of shift rotations are greater than or equal to 4.

We determined the shift rotations such that f and g satisfy the above conditions.

Message Word Ordering We adopt the message word ordering instead of the message word extension. If an attacker constructs an intended differential characteristics for one branch function, the ordering of message words will cause unintended differential patterns in the other branch functions. This is the core part of the security in the compression function. We considered the following four conditions in defining the message word ordering.

– Balance of upper (step 0∼3) and lower (step 4∼7) parts : each word should be applied twice to upper and lower parts, respectively.
– Balance of left and right parts : each word should be applied twice to left and right parts, respectively.
– Balance of sums of indices:
  • Each word should be applied four times and indexed by 0∼15.
  • Total sum of indices is 480. Therefore, the average of sum of indices applied to each word is 30.
  • We searched the ordering so that the sum of indices corresponding to each word is around 25∼35.
– Conditions which do not have same differential patterns in all branches:
  • Specific differential pattern used at a branch may be applied to other branches.
  • Therefore, except the case of giving a same difference to all words, we try to find an ordering such that there is no same differential patterns in all branches.

Shift Rotations and Rank In the step function, 5 and 17, the values of shift rotation, are fixed. Then we search all the case and find candidate values (corresponding to 9 and 21) so that the rank of the linearly-changed step function is maximized. The maximum of the rank is 252. Finally we select 9 and 21 among candidate values so that differences generated from the outputs of f and g functions do not overlap when a message word inserted at a step function has an one-bit difference.

4 Security Analysis of FORK-256

4.1 Collision-Finding Attack

We can analyze the collision-finding attacker's behavior in the aspect of message difference. Let $\Delta R_j^{(8)}$ be the output difference of BRANCH_j for $1 \leq j \leq 4$. Usually, the attacker expects the following event for finding collisions:

$$(\Delta R_1^{(8)}[t] \boxplus \Delta R_2^{(8)}[t]) \oplus (\Delta R_3^{(8)}[t] \boxplus \Delta R_4^{(8)}[t]) = 0, \quad \text{for } 0 \leq t \leq 7.$$

For this, he can take several strategies:

1. The attacker constructs a differential characteristic with high probability for a branch function, say BRANCH_1, and then expects that $\Delta R_3^{(8)}[t] \boxplus \Delta R_4^{(8)}[t] \boxminus \Delta R_2^{(8)}[t]$ is equal to $\Delta R_1^{(8)}[t]$ for $0 \leq t \leq 7$.
2. The attacker constructs two distinct differential characteristics, and expects that $\Delta R_1^{(8)} = -\Delta R_2^{(8)}$ and $\Delta R_3^{(8)} = -\Delta R_4^{(8)}$ for $0 \leq t \leq 7$.
3. The attacker inserts the message difference which yields same message difference pattern in four branches, and expects that same differential characteristic occurs simultaneously in four branches. Then the output difference of the compression function vanishes if the hamming weight of the output difference of each branch is small. This is because the final output is generated with using $\oplus$ and $\boxplus$ by turns.

Let us see the first strategy. If we assume that the outputs of each branch function is random, the probability of the event is almost close to 2^{-256}. It is also difficult for the attacker to mount any attack following the second strategy because he should find such differential pattern of the message words.

Third strategy is relatively easy for the attacker to perform. For example, if he inserts the same difference to all the message words, then the same message difference pattern occurs in every branches. However, the message word reordering was designed so that the third strategy is satisfied only if the attacker inserts the same difference to all the message words. Under the assumption that every step is independent, we can compute the upper bound of the probability that such kind of differential characteristic occurs, which frustrates the attacker.

4.2 Attacks Using Inner Collision Patterns

When the attacker inserts the differences to the message words, the event that the difference of the intermediate value becomes zero often occurs. It is called *inner collision*. We call a differential characteristic which causes an inner collision with a probability, *inner collision pattern*.

Note that an inner collision is not a real collision, but the notion of inner collision pattern is important in cryptanalysis of hash function because it can be repeatedly used to yield a real collision with a high probability. The main idea of attacks on SHA-0 and SHA-1 is also the repetition of an inner collision pattern. So, in hash functions with a serial structure it is related to the resistance against collision-finding attack how many time an inner collision can be repeated.

Let us focus on only one branch function. We omit the subscript index in the variables. We can construct 5-step inner collision pattern easily. Let $\Delta R^{(s)}[t]$ be the difference of $R^{(s)}[t]$, and let $\Delta M[\sigma(i)]$ be the difference of $M[\sigma(i)]$. Table 3 and 4 show two among 5-step inner collision patterns in one branch function, which we found. They holds with the probability 2^{-40}, respectively.

If we apply these patterns to BRANCH_1, the output difference $\Delta R_1^{(8)}$ will be zero with the probability 2^{-40}. As mentioned in the previous subsection, however, it is hard to use the pattern for the attack on FORK-256 because the following events seldom occurs: either that the computation of the output differences of the other branches is zero or that the other branches have the same differential pattern in the message words as BRANCH_1.

Table 3. Case 1. 5-step inner collision pattern in a branch: The numbers in the entries of the table denotes the bits in which the difference is 1.

s	0	1	2	3	4
$\Delta M[\sigma(2s)]$	31				
$\Delta M[\sigma(2s+1)]$		1,6,15,16,20,23	3,4,8,11,21,26	6,12,21,26	31
$\Delta R^{(s)}[0]$					
$\Delta R^{(s)}[1]$		31			
$\Delta R^{(s)}[2]$		6,12,21,26	31		
$\Delta R^{(s)}[3]$		3,4,8,11,21,26	6,12,21,26	31	
$\Delta R^{(s)}[4]$		1,6,15,16	3,4,8,11,20,23	6,12,21,26,21,26	31
$\Delta R^{(s)}[5]$					
$\Delta R^{(s)}[6]$					
$\Delta R^{(s)}[7]$					
Prob.	2^{-10}	2^{-16}	2^{-10}	2^{-4}	1

Table 4. Case 2. 5-step inner collision pattern in a branch: The numbers in the entries of the table denotes the bits in which the difference is 1.

s	0	1	2	3	4
$\Delta M[\sigma(2s)]$		1,6,15,16,20,23	3,4,8,11,21,26	6,12,21,26	31
$\Delta M[\sigma(2s+1)]$	31				
$\Delta R^{(s)}[0]$		1,6,15,16,20,23	3,4,8,11,21,26	6,12,21,26	31
$\Delta R^{(s)}[1]$					
$\Delta R^{(s)}[2]$					
$\Delta R^{(s)}[3]$					
$\Delta R^{(s)}[4]$					
$\Delta R^{(s)}[5]$		31			
$\Delta R^{(s)}[6]$		6,12,21,26	31		
$\Delta R^{(s)}[7]$		3,4,8,11,21,26	6,12,21,26	31	
Prob.	2^{-10}	2^{-16}	2^{-10}	2^{-4}	1

5 Efficiency and Performance

In this section we compare the total number of operations and the performance of FORK-256 and SHA-256. The total number of operations is compared in the Table 5.

Table 5. Number of operations used in FORK-256 and SHA-256

operation	FORK-256	SHA-256
addition (+)	472	600
bitwise operation ($\oplus, \wedge, \vee$)	328	1024
shift ($\ll, \gg$)	-	96
shift rotation ($\lll, \ggg$)	512	576

Table 6 shows the performance of FORK-256 and SHA-256 in the two environments. The environments are denoted by *CPU/OS/Compiler* and the following notations are used in description of environments for simplicity.

Pen3 : Pentium III, 801 MHz
Pen4 : Pentium IV, 2.0 GHz
WinXP : Microsoft Windows XP Professional ver 2002
VC : Microsoft Visual C++ Ver 6.0

Table 6. Performance of FORK-256 and SHA-256

environment	FORK-256		SHA-256	
	Mbps	Cycle/Byte	Mbps	Cycle/Byte
Pen3/WinXP/VC	192.010	31.413	132.469	44.581
Pen4/WinXP/VC	521.111	28.755	318.721	46.372

6 Summary

In this paper we have proposed a new dedicated 256-bit hash function FORK-256. The main features are the followings;

- The structure of the compression function consists of 4 parallel branch functions.
- Nonlinear functions f and g are quite different from the boolean functions which have been used in existing hash functions, and updates multiple words.
- The ordering of the message words is simple but well organized such that it is very difficult for any attacker to find good inner collision patterns.

According to our security analysis, FORK-256 looks resistant against existing attacks including Wang et al.'s attack, but we encourage the readers to give any further security analysis. Finally, our performance test shows that the performance of FORK-256 is faster than that of SHA-256, and we expect that the difference between the performance of FORK-256 and SHA-256 would increase after optimization.

Acknowledgement

This research was supported by the MIC(Ministry of Information and Communication), Korea, under the ITRC(Information Technology Research Center) support program supervised by the IITA(Institute of Information Technology Assessment)

References

1. E. Biham and R. Chen, "Near-Collisions of SHA-0," *Advances in Cryptology – CRYPTO 2004*, LNCS 3152, Springer-Verlag, pp. 290–305, 2004.
2. E. Biham, R. Chen, A. Joux, P. Carribault, C. Lemuet and W. Jalby, "Collisions of SHA-0 and Reduced SHA-1," *Advances in Cryptology – EUROCRYPT 2005*, LNCS 3494, Springer-Verlag, pp. 36–57, 2005.
3. B. den Boer and A. Bosselaers, "An Attack on the Last Two Rounds of MD4," *Advances in Cryptology – CRYPTO'91*, LNCS 576, Springer-Verlag, pp. 194–203, 1992.
4. B. den Boer and A. Bosselaers, "Collisions for the Compression Function of MD5," *Advances in Cryptology – CRYPTO'93*, LNCS 765, Springer-Verlag, pp. 293–304, 1994.
5. F. Chabaud and A. Joux, "Differential Collisions in SHA-0," *Advances in Cryptology – CRYPTO'98*, LNCS 1462, Springer-Verlag, pp. 56–71, 1998.
6. I. Damgård, "A Design Priciple for Hash Functions," *Advances in Cryptology – CRYPTO'89*, LNCS 435, Springer-Verlag, pp. 416–427, 1989.
7. H. Dobbertin, "RIPEMD with Two-Round Compress Function is Not Collision-Free," *Journal of Cryptology* 10:1, pp. 51–70, 1997.
8. H. Dobbertin, "Cryptanalysis of MD4," *Journal of Cryptology* 11:4, pp. 253–271, 1998.
9. H. Dobbertin, A. Bosselaers and B. Preneel, "RIPEMD-160, a strengthened version of RIPEMD," *FSE'96*, LNCS 1039, Springer-Verlag, pp. 71–82, 1996.
10. R. C. Merkle, "One way hash functions and DES," *Advances in Cryptology – CRYPTO'89*, LNCS 435, Springer-Verlag, pages 428–446, 1989.
11. NIST/NSA, "FIPS 180-2: Secure Hash Standard (SHS)", August 2002 (change notice: February 2004).
12. R. L. Rivest, "The MD4 Message Digest Algorithm," *Advances in Cryptology – CRYPTO'90*, LNCS 537, Springer-Verlag, pp. 303–311, 1991.
13. R. L. Rivest, "The MD5 Message-Digest Algorithm," IETF Request for Comments, RFC 1321, April 1992.
14. B. Van Rompay, A. Biryukov, B. Preneel and J. Vandewalle, "Cryptanalysis of 3-pass HAVAL," *Advances in Cryptology – ASIACRYPT 2003*, LNCS 2894, Springer-Verlag, pp. 228–245, 2003.

15. X. Wang, X. Lai, D. Feng, H. Chen and X. Yu, "Cryptanalysis of the Hash Functions MD4 and RIPEMD," *Advances in Cryptology – EUROCRYPT 2005*, LNCS 3494, Springer-Verlag, pp. 1–18, 2005.
16. X. Wang and H. Yu, "How to Break MD5 and Other Hash Functions," *Advances in Cryptology – EUROCRYPT 2005*, LNCS 3494, Springer-Verlag, pp. 19–35, 2005.
17. X. Wang, H. Yu and Y. L. Yin, "Efficient Collision Search Attacks on SHA-0," *Advances in Cryptology – CRYPTO 2005*, LNCS 3621, Springer-Verlag, pp. 1–16, 2005.
18. X. Wang, Y. L. Yin and H. Yu, "Finding Collisions in the Full SHA-1," *Advances in Cryptology – CRYPTO 2005*, LNCS 3621, Springer-Verlag, pp. 17-36, 2005.
19. Y. Zheng, J. Pieprzyk and J. Seberry, "HAVAL – A One-Way Hashing Algorithm with Variable Length of Output," *Advances in Cryptology – AUSCRYPT'92*, LNCS 718, Springer-Verlag, pp. 83–104, 1993.

A Source Code

```
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,7)^ROL(x,22)))

#define g(x)    (x ^ (ROL(x,13)+ROL(x,27)))


#define step(A,B,C,D,E,F,G,H,M1,M2,D1,D2)   \
    temp1 = E+M2;   \
    temp2 = g(temp1);   temp3 = f(temp1+D2);    \
    H = (H + ROL(temp2,21)) ^ ROL(temp3,17);    \
    G = (G + ROL(temp2,9)) ^ ROL(temp3,5);  \
    F = (F + temp2) ^ temp3;                \
    E = temp1 +D2;  \
    temp1 = A+M1; \
    temp2 = f(temp1);   temp3 = g(temp1+D1);    \
    D = (D + ROL(temp2,17)) ^ ROL(temp3,21);    \
    C = (C + ROL(temp2,5)) ^ ROL(temp3,9);  \
    B = (B + temp2) ^ temp3;                \
    A = temp1 + D1;


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;

    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]));
}
```

B Test Vector

Message M (1 block)

4105ba8c d8423ce8 ac484680 07ee1d40 bc18d07a 89fc027c 5ee37091 cd1824f0
878de230 dbbaf0fc da7e4408 c6c05bc0 33065020 7367cfc5 f4aa5c78 e1cbc780

Output of Compression Function CV_1

ebcc5b3d d3715534 a6a7a68a e6022b02 49c676ed 639a34b0 b8d978c2 cfdf1a2b

Intermediate Values

BRANCH$_1$

$R_1^{(0)} =$ 6a09e667 bb67ae85 3c6ef372 a54ff53a 510e527f 9b05688c 1f83d9ab 5be0cd19
$R_1^{(1)} =$ 574faabb ed99d08b 55559509 ca832197 cc3e5d3d 9a87d3f8 a53a7eff e5b76844
$R_1^{(2)} =$ 15b6cd3d b958ed0a bc5ec9da 0685ff8e eecd75a9 bde25622 730387f0 8cd537f4
$R_1^{(3)} =$ b37a2f3c 0b266012 421e26a6 c78f6e0b 1cd85800 d2ba8a16 7449f6c0 0f8c7a01
$R_1^{(4)} =$ 31be4596 a49d2271 6ee14e1a e33ff108 11f5f01a 950cdbc5 5dcd1a2a 32aa199f
$R_1^{(5)} =$ 62fd9d8b 9153d25e 4a23586e 9b599483 cf29e3af 00343c17 f33f23cb 9c903e62
$R_1^{(6)} =$ d36228e4 61ad6751 fe55bb69 94720b3c 8a810aa7 eaf6bd32 737155e2 b96a93e9
$R_1^{(7)} =$ 7a779e32 7926d678 3aec6bdd 0e208057 c349f555 7ec78c6a 91ebeb68 1fc96600
$R_1^{(8)} =$ 85c3c25b 0afe0151 60d37e53 93df1ad6 390f9cea 66b1ae49 71de5de6 17ae42cd

BRANCH$_2$

$R_2^{(0)} =$ 6a09e667 bb67ae85 3c6ef372 a54ff53a 510e527f 9b05688c 1f83d9ab 5be0cd19
$R_2^{(1)} =$ 09a80c1a 20503453 b7ce65dc 686c5844 8f7b750a ceb620a6 e84808f4 13a2716f
$R_2^{(2)} =$ e21fd29c 514719d8 47c2c8b0 116c12a7 42ddee6f ddf4c37a 3b2884ee 1b6552ca
$R_2^{(3)} =$ 608f85bc beba328f da492019 ce8cc5ac e939ee3d 418db835 0d4088c0 a4515753
$R_2^{(4)} =$ 9d819935 7b00fdfd d9947c55 0dfccfd7 817088d7 7d5a694f 8da6b62e 3b63944f
$R_2^{(5)} =$ f22fa55e f4e63e8a 2516289f 77d9b888 dc500533 8717db40 6158e3e7 0e922286
$R_2^{(6)} =$ 13ca89c4 8d2671db afbc022b 9580fdfe 356e2f63 9fa2ca0a d2199dee 455937e5
$R_2^{(7)} =$ b8d0fc67 5c63d5fa d2b45236 fad40792 759b52ab b8475022 1cf6c001 6a0cf5f2
$R_2^{(8)} =$ 08283ecb 5d0e9118 da92c996 9316c47c 26167358 9067bf2b 33a76294 a2c36255

BRANCH$_3$

$R_3^{(0)} =$ 6a09e667 bb67ae85 3c6ef372 a54ff53a 510e527f 9b05688c 1f83d9ab 5be0cd19
$R_3^{(1)} =$ 46f81ba6 a8594fe8 f0348c97 749c040f 8e6801dc f27bf2a8 275472bf 0866407e
$R_3^{(2)} =$ 56a9eac1 0b2c3b53 0e98c271 ec010b6c 448475b5 38d35a23 455b10c5 4c819e3b
$R_3^{(3)} =$ 38cd29dc 2402cc77 48018a70 26a5dcf2 3da527e9 2a237e90 2f4dc6a8 33bd5b6f
$R_3^{(4)} =$ a28f637c bfa479ad 68059737 374a7e75 b5e5b8c6 02eafaad 15799680 ae2d5da0
$R_3^{(5)} =$ 64607852 7bd31a3d a54f54b2 4013d658 1fbcbc0a 4a0633d8 972027f7 40a519ed
$R_3^{(6)} =$ b27cf46d 9b38bd95 fb3978fd d52a18c8 1cdbd155 cb7c23f8 d3ce2cdd 5e6705b2
$R_3^{(7)} =$ 317ce148 bd57a8e7 d3b60337 f0dd8789 1a925421 d09fe955 c626a195 8d38ed5d
$R_3^{(8)} =$ 72ec7187 cb5b0fa4 59b04096 55b45924 d54c20ad be5c7808 ec104b46 08d57f3d

BRANCH$_4$

$R_4^{(0)} =$ 6a09e667 bb67ae85 3c6ef372 a54ff53a 510e527f 9b05688c 1f83d9ab 5be0cd19
$R_4^{(1)} =$ ce371d88 8fe1ef8a f4e6891a dd47fbec 8655e369 45b09413 8d2e660f 968ed897
$R_4^{(2)} =$ 015a57e3 1937b7e4 d82e18fe 374895df 3e1357d6 8ec27797 81e87c75 627d168a
$R_4^{(3)} =$ f2619dce 0757a521 b3dc348f a91771d4 00a58535 d4259025 37fc2a18 c5a9d37a
$R_4^{(4)} =$ dc4ebcd3 3dd1182b acb226cd 3ed1c4a9 f6191a1b d9e93bf6 62752a33 d29d946e
$R_4^{(5)} =$ ad2c36d3 767c5cb7 8d977401 ebd447de a0e6e49b 7bb3bcf8 d7b3eadc 71c2d2a4
$R_4^{(6)} =$ b871dbb2 c23dea2a aebfcf21 6de34a20 41d677c5 a7203d0c 14c00db6 d5b6d5ce
$R_4^{(7)} =$ a6072510 3b4afc71 e74b9db3 5120200b b1167426 2036afe2 ddcd1ac5 096735bb
$R_4^{(8)} =$ 99420469 a4aa2522 f7aeb45b 10939176 d252137f 81312948 50c01427 c0ba68f3