FORK-160: A New 160 - bit Software-Oriented Hash Function

Amir Hossein Tabatabaee[†], **Yaser Esmaeili**[††], **Mohammad Reza Sohizadeh Abyaneh**[††], **Hadi Ahmadi**[††] ,

[†]Zaeim Electronic Industry, Tehran, Iran. [††]Sharif University of Technology, Tehran,Iran.

Summary
There are various cryptographic protocols in which 160-bit message digest is required. SHA-1is the most well-known 160-bit hash function which is still used in protocols despite of its vulnerabilities against collision attacks. Lack of 160-bit hash function structures and disadvantages of truncating outputs of other secure hash functions (security problems and inefficiency) motivated us to introduce a new 160-bit hash function. In this paper, we describe our new software-efficient hash function FORK-160. Hence the name, this function uses basic design principles from the recently proposed hash function FORK-256. However, FORK-160 aims at improving FORK-256 both on security and efficiency. Most notably, FORK-160 uses more secure step function, reasonable message ordering and additive constants which make it resistant against existing cryptanalysis especially local collision attacks.

Key words:
Hash function, collision attack, differential attack, FORK-256.

1. Introduction

Hash functions are a group of cryptographic functions which are used in digital signatures, data integrity, e-cash and many other cryptographic schemes and applications. A one-way hash function maps bit strings of arbitrary finite length into strings of fixed length.

For a cryptographic hash function, the following security requirements according to complexity considerations are needed:
1. Pre-image resistance: It is infeasible to find any input message which hashes to any pre-specified image.
2. Second pre-image resistance: It is infeasible to find any second input which has the same output as pre-specified input message.
3. Collision resistance: It is infeasible to find two different messages which hash to one message digest.

Assume that the output space of a hash function consists of n-bit strings i.e. {0,1}n. For a well-designed hash function, finding pre-image or second pre-image requires about 2n and finding collision requires about 2n/2 hashing operations.

Since hash functions are desired to be fast in performance, recent designing methods of hash functions are based on sequentially iterating a simple and fast step function. The most popular hash functions, which are called MD-like, have been designed according to this method in an evolutional process. The first of this type was MD4, proposed by Rivest in 1990 [6]. MD4 was a novel design, oriented towards software implementation on 32 bit architectures. Several hashing algorithms were derived from MD4 hash function called MDx-class hash functions. MD5, SHA0/1, HAVAL and RIPEMD are some prominent instances [1, 2]. These hash functions are the most popular hash functions because of their performance and trust gained from cryptanalysis techniques [1, 2]. All of the mentioned hash functions are based on a serial method but RIPEMD. The RIPEMD family of hash functions was designed by combining sequential method and parallel structure. This method of designing is still reliable due to no effective attacks so far, except elementary versions of RIPEMD [1, 7]. The most recently proposed hash functions based on this method, FORK-256, motivates us to design a 160-bit output hash function based on its structure. There are several applications of 160-bit hash functions especially in cryptographic protocols. Nonetheless, the most well-known and widely used 160-bit hash function, SHA-1, has been broken by Wang et al [8]. Thus it seems better to use a secure hash function with longer output and truncate the output to a 160-bit string; however, this method is not recommendable due to lack of performance and reasonable security. Consequently, designing a secure dedicated 160-bit hash function could be a major task in hash function area. In this paper, we introduce a 160-bit output hash function based upon a parallel structure like FORK-256. Hence, we named this hash function FORK-160. FORK-160 while being adapted to 160-bit has improved the security of FORK-256 against existing attacks.

The paper is organized as follows: In section 2 we introduce the structure of FORK-160 along with the related functions and parameters. Section 3 explains our

Manuscript received September 5, 2007
Manuscript revised September 20, 2007

design principles for designing each part of the introduced algorithm. This explanation is followed by security analysis and performance evaluation in section 4 and section 5, respectively. Section 6 includes the final results and concluding remarks. It is worth mentioning that the source code of FORK-160 written by C++ programming language with a test vector is given in the appendices.

2. Description of FORK-160

One compression function of FORK-160 consists of four parallel branches; each compresses a twenty-word expanded message to a five-word output. Fig. 1 shows the scheme of the compression function in FORK-160. First, the input message is padded in order to be devisable to 512-bit (16-word) message blocks, corresponding to compression functions. Padding is like SHA-1, i.e. appending a single bit 1 next to the least significant bit of the message followed by zeros until the length of the message is 448 modulo 512, and then appending the original message length modulo 2^{64}. The compression function of FORK-160 hashes a 512-bit string to a 160-bit string. Next, each message block is compressed through the compression function, using the previous compression output as the chaining variable. According to Fig. 1 the four parallel branch functions are called BRANCH 1 to BRANCH 4. The chaining variable for ith block (ith compression function) is CV_i=(A,B,C,D,E) and is initialized to IV_0, represented below:

A= 0x6a09e667, B= 0xbb67ae85, C= 0x3c6ef372,
D= 0xa54ff53a, E= 0x510e527f

Each message block M is divided to sixteen 32-bit words $M_0,\dots,M_{15}$ and is compressed according to Fig. 1, where $\sum_j(M) = (M_{\sigma_j(0)},\dots, M_{\sigma_j(15)})$, j = 1,2,3,4 is the permutation for message words, selected according Table 2 (section 2.2). CV_i is updated through the following relation.

$$CV_{i+1} = CV_i + \{[BRANCH1(CV_i,\textstyle\sum_1(M)) + BRANCH2(CV_i,\textstyle\sum_2(M))] \oplus [BRANCH3(CV_i,\textstyle\sum_3(M)) + BRANCH4(CV_i,\textstyle\sum_4(M))]\} \qquad (1)$$

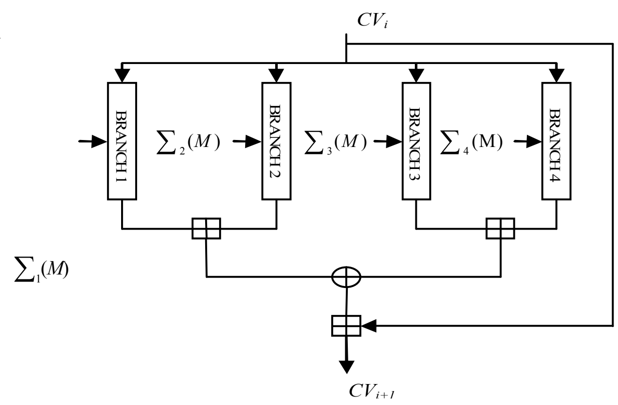

Fig. 1. FORK-160 compression function

2.1 Branch Functions of FORK-160

Each branch contains five step functions and each step function deals with four message blocks; thus each branch uses a set of 20 message words representing a simple expansion and permutation on the input message block.

For the BRANCH j $(1 \le j \le 4)$ the message block is compressed as follows:

1. The chaining variable CV_i is assigned to initial variables $V_{j,0}$.
2. At (k+1)th step function $(0 \le k \le 3)$, the output $V_{j,k+1}$ is computed as follows:

$$V_{j,k+1} = STEP_{j,k}(V_{j,k}, M_{\sigma_j(2k)}, M_{\sigma_j(2k+1)}, M_{\sigma_j(2k+2)}, M_{\sigma_j(2k+3)}, \alpha_{j,2k}, \alpha_{j,2k+1}, \beta_{j,2k}, \beta_{j,2k+1}) \qquad (2)$$

where $\alpha_{j,2k}$, $\alpha_{j,2k+1}$, $\beta_{j,2k}$, $\beta_{j,2k+1}$ are constants. In the fifth step, message words are calculated from the following relations applied to the original sixteen message words:

$$M_{16} = g(M'_{\theta_j(0)} + \delta_{16+4j}) \text{ , } M_{17} = f(M'_{\theta_j(1)} + \delta_{17+4j}),$$
$$M_{18} = f(M'_{\theta_j(2)} + \delta_{18+4j}) \text{ , } M_{19} = g(M'_{\theta_j(3)} + \delta_{19+4j}). \qquad (3)$$

where $\theta_j(t)$ $(0 \le t \le 3)$, $M'_{\theta_j(0)}, M'_{\theta_j(1)}, M'_{\theta_j(2)}, M'_{\theta_j(3)}$ and functions f and g are defined in the Table 1, relation 4 and relation 5, respectively.

Table 1. The contents of $\theta_j(t)$, $(0 \le t \le 3)$, $(1 \le j \le 4)$.

t	0	1	2	3
$\theta_1(t)$	16	17	18	19
$\theta_2(t)$	17	18	19	16
$\theta_3(t)$	18	19	16	17
$\theta_4(t)$	19	16	17	18

$$\begin{aligned}
&M'_{16} = M_0 \oplus M_1 + M_2 \oplus M_3 + M_4 \oplus M_5 + M_6 \oplus M_7 + \\
&M_8 \oplus M_9 + M_{10} \oplus M_{11} + M_{12} \oplus M_{13} + M_{14} \oplus M_{15} \\
&M'_{17} = M_0 + M_1 \oplus M_2 + M_3 \oplus M_4 + M_5 \oplus M_6 + M_7 \oplus \\
&M_8 + M_9 \oplus M_{10} + M_{11} \oplus M_{12} + M_{13} \oplus M_{14} + M_{15} \\
&M'_{18} = M_0 + M_1 + M_2 \oplus M_3 \oplus M_4 + M_5 + M_6 \oplus M_7 \oplus \\
&M_8 + M_9 + M_{10} \oplus M_{11} \oplus M_{12} + M_{13} + M_{14} \oplus M_{15} \\
&M'_{19} = M_0 \oplus M_1 \oplus M_2 + M_3 + M_4 \oplus M_5 \oplus M_6 + M_7 + \\
&M_8 \oplus M_9 \oplus M_{10} + M_{11} + M_{12} \oplus M_{13} \oplus M_{14} + M_{15}
\end{aligned} \tag{4}$$

$$\begin{aligned}
f(X) &= X + (X <<< 7 \oplus X <<< 22) \\
g(X) &= X \oplus (X <<< 13 + X <<< 27)
\end{aligned} \tag{5}$$

Constant values δ_j are defined in section 2.3. Consequently, $V_{j,5}$ is the five-word output of BRANCH j.

$$V_{j,5} = STEP_{j,4}(V_{j,4}, M_{16}, M_{17}, M_{18}, M_{19}, \alpha_{j,8}, \alpha_{j,9}, \beta_{j,8}, \beta_{j,9}) \tag{6}$$

Table 2. Message words permutation for all branches.

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

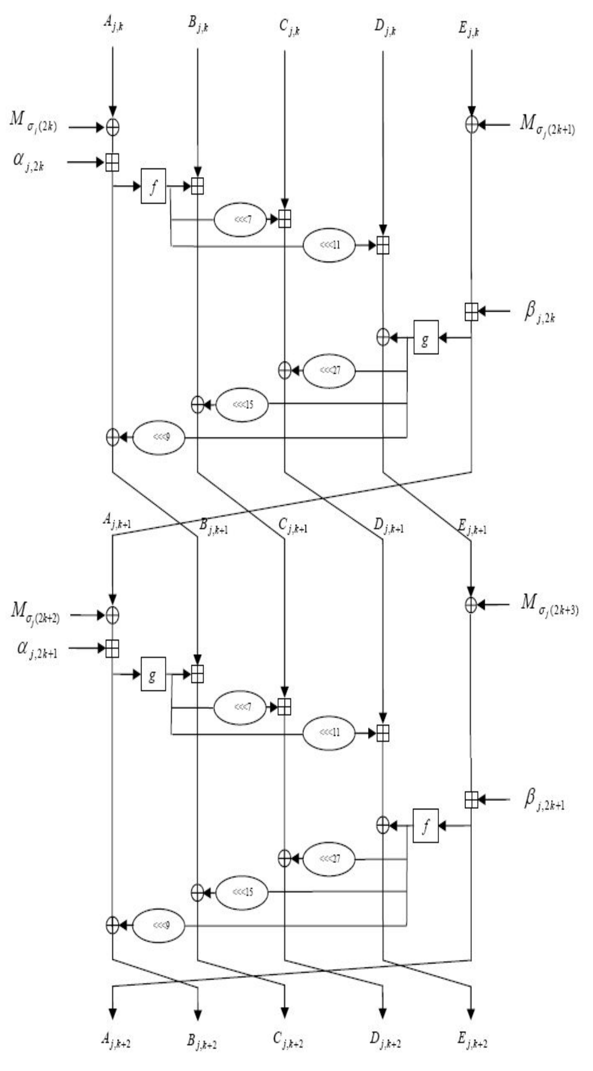

Fig. 2. Step function of FORK-160

2.2 Permutation of Message Words

The permutation of message words in FORK-160 is designed based on Latin square matrices. Table 2 represents the order of message words $M_0,\ldots,M_{15}$ applied to each of four branches.

2.3 Additive Constants

The compression function of FORK-160 uses thirty six additive constants (Table3).

Table 3. Additive constants in FORK-160.

δ_0=0x428a2f98	δ_1=0x71374491
δ_4= 0x3956c25b	δ_5= 0x59f111f1
δ_8= 0xd807aa98	δ_9= 0x12835b01
δ_{12}= 0x72be5d74	δ_{13}= 0x80deb1fe
δ_{16}= 0xe49b69c1	δ_{17}= 0xefbe4786
δ_{20}=0x2de92c6f	δ_{21}= 0x4a7484aa
δ_{24}= 0x983e5152	δ_{25}= 0xa831c66d
δ_{28}= 0xc6e00bf3	δ_{29}= 0xd5a79147
δ_{32}= 0x27b70a85	δ_{33}= 0x2e1b2138
δ_2= 0xb5c0fbcf	δ_3= 0xe9b5dba5
δ_6= 0x923f82a4	δ_7= 0xab1c5ed5
δ_{10}= 0x243185be	δ_{11}= 0x550c7dc3
δ_{14}= 0x9bdc06a7	δ_{15}= 0xc19bf174
δ_{18}= 0x0fc19dc6	δ_{19}= 0x240ca1cc
δ_{22}= 0x5cb0a9dc	δ_{23}= 0x76f988da
δ_{26}= 0xb00327c8	δ_{27}= 0xbf597fc7
δ_{30}= 0x06ca6351	δ_{31}= 0x14292967
δ_{34}= 0x4d2c6dfc	δ_{35}= 0x53380d13

The first 20 constant values of Table 3 are utilized in each branch as additive constants for the compression functions, according to the table 4.

Table 4. Permutation table for using additive constants.

Ste p No.	$(\alpha_{1,0},\ldots,\alpha_{1,9})$	$(\alpha_{2,0},\ldots,\alpha_{2,9})$	$(\alpha_{3,0},\ldots,\alpha_{3,9})$	$(\alpha_{4,0},\ldots,\alpha_{4,9})$
1	δ_0	δ_{19}	δ_1	δ_{18}
	δ_2	δ_{17}	δ_3	δ_{16}
2	δ_4	δ_{15}	δ_5	δ_{14}
	δ_6	δ_{13}	δ_7	δ_{12}
3	δ_8	δ_{11}	δ_9	δ_{10}
	δ_{10}	δ_9	δ_{11}	δ_8
4	δ_{12}	δ_7	δ_{13}	δ_6
	δ_{14}	δ_5	δ_{15}	δ_4
5	δ_{16}	δ_3	δ_{17}	δ_2
	δ_{18}	δ_1	δ_{19}	δ_0
Ste p No.	$(\beta_{1,0},\ldots,\beta_{1,9})$	$(\beta_{2,0},\ldots,\beta_{2,9})$	$(\beta_{3,0},\ldots,\beta_{3,9})$	$(\beta_{4,0},\ldots,\beta_{4,9})$
1	δ_1	δ_{18}	δ_0	δ_{19}
	δ_3	δ_{16}	δ_2	δ_{17}
2	δ_5	δ_{14}	δ_4	δ_{15}
	δ_7	δ_{12}	δ_6	δ_{13}
3	δ_9	δ_{10}	δ_8	δ_{11}
	δ_{11}	δ_8	δ_{10}	δ_9
4	δ_{13}	δ_6	δ_{12}	δ_7
	δ_{15}	δ_4	δ_{14}	δ_5
5	δ_{17}	δ_2	δ_{16}	δ_3
	δ_{19}	δ_0	δ_{18}	δ_1

The remaining 16 constant values of Table 3 are used in the process of message expansion in four branches of FORK-160 (relation (3)).

3. Design Principles

In this section, we describe the security criteria for designing FORK-160 and the design process based upon these criteria. The design criteria include basic structure, additive constants, message expansion, nonlinear functions and rotation values.

3.1 Basic Structure

FORK-160 consists of four branches with parallel structure. This kind of structure refers to RIPEMD family hash function [7]. In this family, the functions with the same message ordering in each chaining variable words are resistant against collision attacks. So using message words with different permutation causes algorithm to be more secure [7, 8]. The second hash function which uses parallel structure is FORK-256. In FORK-256, each branch uses message words with different ordering. Pieprzyk et al. could find some weaknesses in FORK-256 compression function and attacked on two branches of the algorithm [9]. In addition, they showed that the security of algorithm against collision attack is of order $2^{126.6}$ FORK-256 operator [9].

In FORK-160 an improved message permutation with using expanded message words are combined with each other to consolidate the algorithm against two branch attack. Moreover, interaction between two left and right parts of each step causes algorithm to be more resistant against attacks which are based on partitioning two parts of each step.

3.2 Additive Constants

According to the description of FORK-160 in section 2, each step function uses four additive constants; hence, BRANCH 1, which consists of five steps, uses 20 additive constants. Since each new branch uses a new permutation of 20 additive constants of the previous branch, the whole compression function uses 20 additive constants within its structure. It is notable that the last step of each branch uses 4 unique extra constants in its message expansion process; thus there are 16 constants used in message expansion for last steps of four branches. As a consequence, there are totally 36 different additive constants, used in the compression process of FORK-160. Then totally 36 different constant values are applied in the FORK-160 compress function (Table 3). The contents of these 32-bit constants are selected in order to have the best possible diffusion which makes the algorithm more resistant against micro-collision finding attacks. The main criteria of selecting these constants are their independency; therefore, these constants represent the first 32 bits of the fractional parts of the cubic roots of the first 36 prime numbers, which have no interrelationship.

3.3 Message Expansion

FORK-160 uses an expansion process for the fifth step of each branch. In this process, four message words are calculated from the original message block words, by the relation (3). This simple process uses the two word-oriented functions, *f* and *g*, along with the dual XOR and modular sum operations and 16 additive constants. Therefore, the expansion process, while being efficient in performance, causes each branch to tolerate local collision attacks, due to the bit diffusion property. The notable criteria of the introduced expansion process are written as follows:

1. Existence of modular addition causes nonlinear behavior against XOR difference.

2. Using two functions *f* and *g* with one linear and one nonlinear operator causes high differential diffusion in output message words.

3. One way structure of the expansion relation makes it infeasible to inverse the operations.

The introduced message expansion in FORK-160 causes the algorithm to be more resistant than FORK-256 against attacks on single branch which will be considered in the following sections.

3.4 Permutation of Message Words

Since FORK-160 has parallel structure, it should necessarily tolerate simultaneous collision finding attacks in parallel branches. A simple method for this purpose is to use message re-ordering for different branches. In this case, if an attacker constructs an intended differential characteristic for one branch function, the ordering of message words will cause unintended differential patterns in the other branch functions; thus, finding specific differences for patterns would not be straightforward.

There are some important criteria for designing this message permutation such as: balance of upper and lower part, balance of left and right part and balance of sums of in input indices [7]. We have designed the message permutation by inspiration of Latin square matrices, so that all of the former criteria are preserved and even improved in comparison with those of FORK-256. Moreover, by this selection of indices, passing the same differential pattern through two different branches has gotten hard. Further details are explained in the section 4.

3.5 Functions and Shift Rotations

Almost all dedicated hash functions use Boolean functions with three or more variables and therefore the weaknesses of these bit oriented functions could be exploited by attackers [7]. Some of the most well-known examples of these hash functions are MD4, MD5, HAVAL, RIMEMD and SHA0/1 [1,2]. Instead, FORK-160 uses two nonlinear word-oriented functions, *f* and *g*, which work on a single 32-bit variable. These functions are the same as those in FORK-256. On the other hand, these functions affect all of the five chaining variable words during each step and, unlike FORK-256, we cannot divide each step to isolated left and right parts; this point causes resistance to the existing attacks on two branches [9].

In FORK-160 each output of the functions *f* and *g* except the first one is rotated by specific number and then used to update chaining variable words. Using rotation, while having low complexity in software and hardware, causes differential diffusion within steps. This is an essential security principle for any existing dedicated hash function. These rotation constants, are calculated, using a heuristic search method among odd numbers (not devisors of 32), based upon SAC criteria.

4. Security Analysis of FORK-160

In this section, we explain security considerations in designing FORK-160. As stated in section 3, FORK-256 hash function with parallel structure was designed in order to improve weaknesses appeared in the previous well- known hash functions. In spite of having apparent strong structure, FORK-256 contains weak points, especially in message permutation and interaction within chaining variable. Attacks on two branches of FORK-256 and finding near collisions and approximations for full round [9] of the algorithm are evidences of these weaknesses. Due to similarities between FORK-256 and FORK-160, we investigate the following considerations in comparison with the attacks implemented on FORK-256.

1. Security analysis for a single branch of FORK-160

2. Security analysis for two branches of FORK-160 against collision attack

4.1 Security Analysis for a Single Branch of FORK-160

In this section, we consider two possible types of attack on a single branch of FORK-160. The first attack is an ordinary collision finding attack and the second one is a chosen IV collision finding attack. Ignoring the advantage of using expansion for the fifth step of each branch, one can easily find a one branch collision by assigning compatible values for message words within the steps 1 to 4 of the branch. For example, in this case the following algorithm leads to collision for the first branch with four steps:

1. Select two message words M_0 and M'_0 with nonzero XOR difference ΔM_0, which satisfy in the equation $f(M_0+\delta_0) = f(M'_0+\delta_0)$.

2. Preserve zero differences output in 1st to 4th chaining variable words at the output of the second step by assigning zero values to all message words except for M_9 and M'_9.

3. Set: $M_9 = E_{1,4}$ and $M'_9 = E'_{1,4}$ in order to compensate the first differential and obtain the local collision.

It is obvious that attack on a single branch of FORK-160 would have no more complexity than that on FORK-256, without considering message expansion. However, the expansion in the fifth step, adds four new 32-bit check equations in the collision finding scenario which hold with the probability of $2^{4*32}=2^{128}$; consequently, this expansion increases the complexity of such a one branch collision finding attack up to 2^{128} trials which is even more than the complexity of birthday attack (2^{80}). Hence, even one branch of FORK-160 is resistant against such collision attacks.

Another attack which is worth considering on a single branch of FORK-160 is chosen IV collision finding attack. This attack can be applied on any single branch of FORK-160. For example, considering the first branch of FORK-160, one can find a chosen IV collision by implementing the following algorithm:

1. Select two messages M and M' with two message words M_0 and M'_0 and two initial chaining variables IV and IV' with two different first words $IV[0]$ and $IV'[0]$, provided that $M_0 = IV[0]$ and $M'_0 = IV'[0]$,

2. Preserve zero values for outputs of all chaining variable words, by assigning zero to all message words except M_0 and M'_0.

3. If $\Delta A_{1,10}=0$, $\Delta B_{1,10}=0$, $\Delta C_{1,10}=0$, $\Delta D_{1,10}=0$, $\Delta E_{1,10}=0$, then one collision is found; return the collision.

4. Else, return to 1.

The complexity of above algorithm for finding chosen IV collision would be also of O (2^{128}) due to the limitations of four equations in the fifth step of the branch. However, we tried about 2^{37} message and IV differentials and investigated whether there are any collisions in one branch output, through simulation. As a consequence, we could only find one word collision (i.e. an output word with zero differentials) for the branch. This simulation result also reveals that even one branch of FORK-160 is resistant against collision attacks, based upon our knowledge.

4.2 Security Analysis for Two Branches of FORK-160 against Collision Attack

In this section, we investigate the security of two branches of FORK-160 against differential based cryptanalysis to find collisions. In our cryptanalysis by each round, we mean half of a step in which two messages, two additive constants, one function f and one function g is used and finally a word permutation in chaining variable is occurred. In other words, ($A_{j,k+1}$, $B_{j,k+1}$, $C_{j,k+1}$, $D_{j,k+1}$, $E_{j,k+1}$) is the result of one round implementation in BRANCH j on ($A_{j,k}$, $B_{j,k}$, $C_{j,k}$, $D_{j,k}$, $E_{j,k}$), according to Fig. 2.

Since the number of ways for choosing 2 branches from the four branches of FORK-160 is 6, our objective is to investigate how it is hard to find any simultaneous collisions in each of these 6 pairs of branches. To achieve the goal, we consider each branch without the fifth step function. Then, we extend the attack five-step branch pairs. The attack design scenario on four-step branch pairs is written as follows:

For each pair of branches, we activate one differential message in the first branch and trace the effects of the selected differential message over the other branch. In this way, we can omit the influence of activated chaining variables by choosing the other message. Finally, we calculate the alternation of differential chaining variables in each chaining variable (totally 10 chaining variables for 2 branches) while we trace the outputs of forth rounds.

Also we must observe the alternation of differential chaining variables in 10 chaining variables after the last step function. We compare the differences of 2 branches in chaining variables outputs, it is clear that in the case of equality, collisions are gained.

Table 1 shows that differential chaining variables sets that can be calculated for simultaneous equations provided that message pairs (M_i,M_j) are altered.

Table 5. Set of pairs of messages (M_i,M_j) which are used in simultaneous equations

(1,4)	(1,3)	(1,2)	pairs of branch
(M_1,M_{11}) (M_3,M_4) (M_3, M_{14}) (M_4, M_{14})	(M_0,M_{10}) (M_1,M_{11}) (M_4,M_{14}) (M_5, M_{15})	(M_4,M_{14})	pairs of message s
(3,4)	(2,4)	(2,3)	pairs of branch
(M_1, M_6) (M_1, M_{11}) (M_4, M_{14}) (M_6, M_{11})	(M_1, M_{11}) (M_2, M_8) (M_4, M_{14}) (M_7, M_{13}) $(M_{10},$ $M_{15})$	(M_1, M_{11}) (M_4, M_9) (M_4, M_{14}) (M_9, M_{14})	pairs of message s

The equations of selected messages in Table 5, which must be satisfied, can be seen in Table 6. In this table (f^i,g^j) of columns (N,M) means that it is required to build simultaneous collision equation for ith round (considering each two-round steps) of branch number N and simultaneous collision equation for jth round of branch number M. e.g. $(f^6,f^2f^3g^7)$ in the cell (3,4) represents one equation of function f at round 6 in the third branch and three equations of f,f and g at round 2,3 and 7 respectively in the forth round.

Table 6. Formation of simultaneous collision equations for each 2 branches before expansion.

(1,4)	(1,3)	(1,2)	pairs of branch / pairs of messages
-	(f^1g^6,f^1g^6)	-	(M_0, M_{10})
-	-	-	(M_1, M_6)
(g^1g^2,f^3g^7)	(g^1g^2,g^1g^2)	-	(M_1, M_{11})
-	-	-	(M_2, M_8)
$(f^2,g^1g^2f^6)$	-	-	(M_3, M_4)
$(f^2f^3g^7,f^6)$	-	-	(M_3, M_{14})
-	-	-	(M_4, M_9)
(f^3g^7,g^1g^2)	(f^3g^7,f^3g^7)	(f^3g^7,g^2)	(M_4, M_{14})
-	(g^3g^4,g^3g^4)	-	(M_5, M_{15})
-	-	-	(M_6, M_{11})
-	-	-	(M_7, M_{13})
-	-	-	(M_9, M_{14})
-	-	-	(M_{10}, M_{15})
(3,4)	(2,4)	(2,3)	pairs of branch / pairs of messages
-	-	-	(M_0, M_{10})
$(f^6,f^2f^3g^7)$	-	-	(M_1, M_6)
(g^1g^2,f^3g^7)	(f^3g^7,f^3g^7)	(f^3g^7,g^1g^2)	(M_1, M_{11})
-	(g^3g^4,g^3g^4)	-	(M_2, M_8)
-	-	-	(M_3, M_4)
-	-	-	(M_3, M_{14})
-	-	$(f^6,f^2f^3g^7)$	(M_4, M_9)
(f^3g^7,g^1g^2)	(g^2,g^1)	(g^2,f^3g^7)	(M_4, M_{14})
-	-	-	(M_5, M_{15})
(g^1g^2,f^2)	-	-	(M_6, M_{11})
-	(f^1g^5,f^1g^5)	-	(M_7, M_{13})
-	-	(g^2f^6,f^2)	(M_9, M_{14})
-	(f^2f^3,f^4)	-	(M_{10}, M_{15})

According to Table 5 and Table 6, it can be concluded that at least 2 simultaneous equations (in only one position) are required, moreover it is obvious that the last step function added to simultaneous equations causes the propagation of differential messages in 10 output chaining variables of 2 branches to be too high.

5. Performance Analysis of FORK-160

The performance of FORK-160 in software is compared with other hash functions such as MD5, SHA-1, RIPEMD-160 and FORK-256 in Table 7. The performance comparison is accomplished using Pentium IV, 2.8 GHz, 512MB RAM/ Microsoft Windows XP Professional v. 2002/ Microsoft Visual C++ Ver. 6.0.

Table 7. Comparison of FORK-160 performance with the other hash functions, implemented on P4/WinXP/VC (all numbers are in Mbps).

Alg.	MD5	FORK-160	SHA-1	RIPEMD-160	FORK-256
Perf.	1656.49	728.77	616.83	593.10	478.70

The software implementation of FORK-160 in this evaluation is not well-optimized, thus we expect some improvement in performance of any prospective optimized version of this algorithm. However, the simulation results in Table 7 imply that FORK-160 is about 18% faster than SHA-1 and 23% faster than RIPEMD-160 on a Pentium PC.

6. Conclusions

This paper deals with designing a new dedicated hash function with 160-bit output length, which we have called FORK-160. Our designing scheme has been based on parallel structure used in FORK-256. The introduced design criteria yielded more resistance against existing collision attacks in comparison to FORK-256. We analyzed resistance of FORK-160 against single branch and dual branches collision attacks, with regard to attacks which have been implemented on FORK-256 [9]. We have also evaluated performance of FORK-160 and compared it by well-known hashing algorithms, namely MD5, SHA-1, RIPEMD-160 and FORK-256.

The dual security analysis and performance simulation results indicate that our introduced hash function is not only more secure but also more efficient in software performance in comparison to the standard hash algorithm, SHA-1 and other functions such as RIPEMD-160 and FORK-256. As a result, we believe that FORK-160 could replace previous 160-bit hash functions, like SHA-1 in various applications. In fact, from the security point of view, there are various suggestions to improve FORK-160; nonetheless, in these cases the performance might be sacrificed.

References

[1] B. Van Rompay. Analysis and design of cryptographic hash functions, MAC algorithms and block ciphers. PhD thesis, K. U. Leuven, Januvary 2004.

[2] B. Preneel. Analysis and design of cryptographic hash functions, PhD thesis, Katholieke University Leuven, January 1993.

[3] B. Preneel. Cryptographic primitives for information authentication state of the art, in State of the Art in Applied Cryptography (B. Preneel and V. Rijmen, eds.), no. 1528 in Lecture Notes in Computer Science, pp. 50-105, Springer-Verlag, 1998.

[4] B. Preneel. The state of cryptographic hash functions, in Lectures on Data Security. Modern Cryptology in Theory and Practice (I. B. Damgard, ed.), no. 1561 in Lecture Notes in Computer Science, pp. 158-182, Springer-Verlag, 1999.

[5] B. Van Rompay, B. Preneel, and J. Vandewalle. The digital timestamping problem, in Proceedings 20th Symposium on Information Theory in the Benelux (A. Barbe, E. C. van der Meulen, and P. Vanroose, eds.), pp. 71-78, 1999.

[6] R. L. Rivest. The MD4 message digest algorithm, in Advances in Cryptology Crypto'90 (A. Menezes and S. A. Vanstone, eds.), no. 537 in Lecture Notes in Computer Science, pp. 303-311, Springer-Verlag, 1991.

[7] D. Hong, J. Sung, S. Lee, and D. Moon. A new dedicated 256-bit hash function: FORK-256, In Fast Software Encryption-FSE'06, LNCS. Springer-Verlag, 2006.

[8] 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.

[9] K. Matsuesiewicz, S. Contini, and J. Pieprzyk. Weaknesses of the FORK-256 compression function, Available at http://eprint.iacr.org, 2007.

Appendix A: Source Code

```
typedef unsigned int UINT;
//DELTA VALUES
UINT Delta[36]={
0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
0x3956c25b, 0x59f111f1, 0x923f82a4,0xab1c5ed5,
0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13};

//ALPHA VALUES
UINT Alpha[4][10]={
Delta[0],Delta[2],Delta[4],Delta[6],Delta[8],Delta[10],Delt
a[12],Delta[14], Delta[16], Delta[18],
Delta[19],Delta[17],Delta[15],Delta[13],Delta[11],Delta[9
], Delta[7],
Delta[5],Delta[3],Delta[1],Delta[1],Delta[3],Delta[5],Delta
[7], Delta[9], Delta[11],
Delta[13],Delta[15],Delta[17],Delta[19],Delta[18],Delta[1
6],Delta[14], Delta[12],
Delta[10],Delta[8] ,Delta[6],Delta[4],Delta[2],Delta[0] };



//BETA VALUES
UINT Beta[4][10]={
Delta[1],Delta[3],Delta[5],Delta[7],Delta[9],Delta[11],Delt
a[13],Delta[15], Delta[17], Delta[19],
Delta[18],Delta[16],Delta[14],Delta[12],Delta[10],Delta[8
], Delta[6],
Delta[4],Delta[2],Delta[0],Delta[0],Delta[2],Delta[4],Delta
[6],Delta[8], Delta[10],
Delta[12],Delta[14],Delta[16],Delta[18],
Delta[19],Delta[17], Delta[15], Delta[13],
Delta[11],Delta[9], Delta[7],Delta[5],Delta[3],Delta[1] };

//NECESSARY FUNCTION
#define ROL(x,n) (x << n) | (x >> (32-n))     // n-bit left
rotation
#define F(x)    (x + (ROL(x,7)^ROL(x,22)))
#define G(x)    (x ^ (ROL(x,13)+ROL(x,27)))

//STEP FUNCTION
#define
step(A,B,C,D,E,M1,M2,M3,M4,Alpha1,Alpha2,Beta1,
Beta2)
A=(A^M1)+Alpha1;\
temp3=(E^M2)+Beta1;\
temp2 = F(A);\
temp4 = G(temp3);\
temp1 = (temp3^M3)+Alpha2;\
D = (D+ROL(temp2,23))^ temp4;\
C =(C+ROL(temp2,13))^ROL(temp4,5);\
B = (B+temp2)^ROL(temp4,11);\
E=(D^M4)+Beta2;\
temp2 = G(temp1);\
temp3 = F(E);\
D = (C+ROL(temp2,23))^ temp3;\
C = B+ROL(temp2,13))^ROL(temp3,5);\
B = (A+temp2)^ROL(temp3,11);\
A = temp1^ROL(temp3,17);

static void FORK160_Compression_Function(unsigned int
*CV, unsigned int *M) {
unsigned long R1[5],R2[5],R3[5],R4[5];
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];

// BRANCH1(CV,M)
step(R1[0],R1[1],R1[2],R1[3],R1[4],M[0],M[1],M[2],M[
3],Alpha[0][0], Alpha[0][1],Beta[0][0], Beta[0][1]);
step(R1[4],R1[0],R1[1],R1[2],R1[3],M[4],M[5],M[6],M[
7],Alpha[0][2], Alpha[0][3],Beta[0][2], Beta[0][3]);
step(R1[3],R1[4],R1[0],R1[1],R1[2],M[8],M[9],M[10],M
[11],Alpha[0][4], Alpha[0][5],Beta[0][4],Beta[0][5]);
step(R1[2],R1[3],R1[4],R1[0],R1[1],M[12],M[13],M[14]
,M[15],Alpha[0][6], Alpha[0][7], Beta[0][6],Beta[0][7]);
step(R1[1],R1[2],R1[3],R1[4],R1[0],G(M[16]+Delta[20])
,F(M[17]+Delta[21]), F(M[18]+
Delta[22]),G(M[19]+Delta[23]),Alpha[0][8],Alpha[0][9],
Beta[0][8], Beta[0][9]);

// BRANCH2(CV,M)
step(R2[0],R2[1],R2[2],R2[3],R2[4],M[12],M[13],M[14]
,M[15],Alpha[1][0], Alpha[1][1], Beta[1][0],Beta[1][1]);
step(R2[4],R2[0],R2[1],R2[2],R2[3],M[1],M[2],M[3],M[
0],Alpha[1][2], Alpha[1][3],Beta[1][2], Beta[1][3]);
step(R2[3],R2[4],R2[0],R2[1],R2[2],M[5],M[6],M[7],M[
4],Alpha[1][4], Alpha[1][5],Beta[1][4], Beta[1][5]);
step(R2[2],R2[3],R2[4],R2[0],R2[1],M[9],M[10],M[11],
M[8],Alpha[1][6], Alpha[1][7],Beta[1][6],Beta[1][7]);
step(R2[1],R2[2],R2[3],R2[4],R2[0],G(M[17]+Delta[24])
,F(M[18]+Delta[25]),
F(M[19]+Delta[26]),G(M[16]+Delta[27]),Alpha[1][8],Al
pha[1][9],Beta[1][8], Beta[1][9]);
```

```
// BRANCH3(CV,M)
step(R3[0],R3[1],R3[2],R3[3],R3[4],M[10],M[11],M[8],
M[9],Alpha[2][0], Alpha[2][1],Beta[2][0],Beta[2][1]);
step(R3[4],R3[0],R3[1],R3[2],R3[3],M[14],M[15],M[12]
,M[13],Alpha[2][2], Alpha[2][3], Beta[2][2],Beta[2][3]);
step(R3[3],R3[4],R3[0],R3[1],R3[2],M[2],M[3],M[0],M[
1],Alpha[2][4], Alpha[2][5],Beta[2][4], Beta[2][5]);
step(R3[2],R3[3],R3[4],R3[0],R3[1],M[6],M[7],M[4],M[
5],Alpha[2][6], Alpha[2][7],Beta[2][6, Beta[2][7]);
step(R3[1],R3[2],R3[3],R3[4],R3[0],G(M[18]+Delta[28])
,F(M[19]+Delta[29]),
F(M[16]+Delta[30]),G(M[17]+Delta[31]),Alpha[2][8],Al
pha[2][9],Beta[2][8], Beta[2][9]);

// BRANCH4(CV,M)
step(R4[0],R4[1],R4[2],R4[3],R4[4],M[7],M[4],M[5],M[
6],Alpha[3][0], Alpha[3][1],Beta[3][0], Beta[3][1]);
step(R4[4],R4[0],R4[1],R4[2],R4[3],M[11],M[8],M[9],M
[10],Alpha[3][2], Alpha[3][3],Beta[3][2],Beta[3][3]);
step(R4[3],R4[4],R4[0],R4[1],R4[2],M[15],M[12],M[13]
,M[14],Alpha[3][4], Alpha[3][5], Beta[3][4],Beta[3][5]);
step(R4[2],R4[3],R4[4],R4[0],R4[1],M[3],M[0],M[1],M[
2],Alpha[3][6], Alpha[3][7],Beta[3][6], Beta[3][7]);
step(R4[1],R4[2],R4[3],R4[4],R4[0],G(M[19]+Delta[32])
,F(M[16]+Delta[33]),
F(M[17]+Delta[34]),G(M[18]+Delta[35]),Alpha[3][8],Al
pha[3][9],Beta[3][8], Beta[3][9]);
// OUTPUTS
CV[0] + =((R1[0] + R2[0]) ^ (R3[0] + R4[0]));
CV[1] + = ((R1[1] + R2[1]) ^ (R3[1] + R4[1]));
CV[2] + = ((R1[2] + R2[2]) ^ (R3[2] + R4[2]));
CV[3] + =((R1[3] + R2[3]) ^ (R3[3] + R4[3]));
CV[4] + = ((R1[4] + R2[4]) ^ (R3[4] + R4[4]));
}
```

Appendix B: Test vector

```
//IN ITIALIZATION
CV[0]=0x6a09e667;CV[1]=0xbb67ae85;CV[2]=0x3c6ef
372;
CV[3]=0xa54ff53a;CV[4] = 0x510e527f;

//MESSAGE 1
M[0]=0x4105ba8c; M[1]=0xd8423ce8;
M[2]=0xac484680;  M[3]=0x07ee1d40;
M[4]=0xbc18d07a;  M[5]=0x89fc027c;
M[6]=0x5ee37091;  M[7]=0xcd1824f0;
M[8]=0x878de230; M[9]=0xdbbaf0fc;
M[10]=0xda7e4408; M[11]=0xc6c05bc0;
M[12]=0x33065020; M[13]=0x7367cfc5;
M[14]=0xf4aa5c78;M[15]=0xe1cbc780;

//AFTER EXPANSION (MESSAGES FOR THE LAST
STEPS)
//Branch 1:
M[16]=0x64bf34a5;  M[17]= 0xb9252343;
M[18]= 0x84a95a9d;  M[19]= 0x73d6269;
//Branch2.
M[16]= 0xf5f70369;  M[17]= 0xd27c3754;
M[18]= 0x1443c1d9; M[19]= 0xb0eff316;
//Branch3.
M[16]= 0xeb1433a;  M[17]= 0xec2d94f8;
M[18]= 0xe19df64a;  M[19]= 0xbaa53246;
//Branch4.
M[16]= 0x5ea6c2c6; M[17]= 0xa2ed73df;
M[18]= 0xfd5b09a;    M[19]= 0x827d0202;
//OUTPUT 1
CV[0] = 0x6ebd05c2; CV[1] = 0x955a2b42; CV[2] =
0xb86ceabd;
CV[3] = 0xa8af1084; CV[4] = 0xb4ce0111;

//MESSAGE 2
memset(M,0,15*sizeof(UINT)); //M[0~15]=0
//AFTER EXPANSION (MESSAGES FOR THE LAST
STEPS)
//Branch1.
M[16]= 0x5006c190; M[17]= 0x854761cf;
M[18]= 0xdc08980a; M[19]= 0x85464605;
//Branch2.
M[16]= 0x46d5a2dd; M[17]= 0x441d0562;
M[18]= 0xa3c30ca1; M[19]= 0x80a38038;
//Branch3.
M[16]= 0x599f732c;  M[17]= 0xa9a57d35;
M[18]= 0xfc3c1dec;  M[19]= 0x2984c2a8;
//Branch4.
M[16]= 0xcecab673; M[17]= 0x7db6c017;
 M[18]= 0x4c646d3b; M[19]= 0xc883e77c;

//OUTPUT 2
CV[0]=0x5f87ccad;  CV[1]=0xd4b5fdac;
CV[2]=0x6293277f;
CV[3]=0xd25d3bb2; CV[4]=0x7d5ff391;
```