Egyptian Informatics Journal (2012) **13**, 199–208

Cairo University

Egyptian Informatics Journal

www.elsevier.com/locate/eij
www.sciencedirect.com

ORIGINAL ARTICLE

A secure and efficient cryptographic hash function based on NewFORK-256

Harshvardhan Tiwari *, **Krishna Asawa**

Department of Computer Science and Engineering, JIIT(Deemed University), Noida, U.P., India

Received 8 June 2012; revised 28 July 2012; accepted 23 August 2012
Available online 29 September 2012

KEYWORDS

Cryptographic hash function;
Digital signature;
MD5;
SHA-1;
NewFORK-256

Abstract Cryptographic hash functions serve as a fundamental building block of information security and are used in numerous security applications and protocols such as digital signature schemes, construction of MAC and random number generation, for ensuring data integrity and data origin authentication. Researchers have noticed serious security flaws and vulnerabilities in most widely used MD and SHA family hash functions. As a result hash functions from FORK family with longer digest value were considered as good alternatives for MD5 and SHA-1, but recent attacks against these hash functions have highlighted their weaknesses. In this paper we propose a dedicated hash function MNF-256 based on the design principle of NewFORK-256. It takes 512 bit message blocks and generates 256 bit hash value. A random sequence is added as an additional input to the compression function of MNF-256. Three branch parallel structure and secure compression function make MNF-256 an efficient, fast and secure hash function. Various simulation results indicate that MNF-256 is immune to common cryptanalytic attacks and faster than NewFORK-256.

1. Introduction

Ensuring privacy and preventing integrity of a message are the basic goals of cryptography. Cryptographic hash functions remain one of the most important cryptographic primitives, and they can be used to guarantee the security of many cryptographic applications and protocols such as digital signature, random number generation, data source authentication, key update and derivation, message authentication code, integrity protection, malicious code recognition, SSL, TLS and S/MIME. Cryptographic hash functions compress an input message of arbitrary length to an output with short fixed length, the hash code. Hash functions are classified into two classes: unkeyed hash function also known as Manipulation Detection Code (MDC) with single parameter – a message and keyed hash function with two distinct inputs – a message and secret key. Keyed hash functions are used to construct the MAC (Message Authentication Code). The MAC is widely used to provide data integrity and data origin authentication. The choice between a MAC and an MDC is application dependent. They are also classified as, namely hash functions based on

* Corresponding author.
E-mail addresses: tiwari.harshvardhan@gmail.com (H. Tiwari), krishna.asawa@jiit.ac.in (K. Asawa).

Peer review under responsibility of Faculty of Computers and Information, Cairo University.

ELSEVIER Production and hosting by Elsevier

1110-8665 © 2012 Faculty of Computers and Information, Cairo University. Production and hosting by Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.eij.2012.08.003

block cipher, hash functions based on modular algorithm and dedicated hash functions. The motivation behind block cipher based schemes is the minimization of the effort to design and implement the hash functions and of the complexity of the equipment. Also the trust that one has in a certain block cipher (such as DES) can be transferred to a hash function. Particularly interesting from a theoretical point of view, modular algorithm based schemes are provably secure, in the sense that their security relies on the hardness of some mathematical problems such as number theory problems. Dedicated hash functions are specially designed from the scratch for the purpose of hashing a plain text with optimized performance and without being constrained to reusing existing system components such as block ciphers and modular arithmetic. These hash functions are not based on hard problems such as factorization and discrete logarithms. The most popular method of designing compression functions of dedicated hash functions is a serial successive iteration of a small step function [1,2]. Our proposed hash function comes under the last category.

Preimage resistance, second preimage resistance and collision resistance are the three classical requirements for security of keyless hash functions. Properties preimage resistance, second preimage resistance and collision resistance are also known as one way, weak collision resistance and strong collision resistance respectively. First of all, it needs to be computationally infeasible to find the preimage X of $H(X)$, when $H(X)$ is given. This is called preimage resistance. Secondly, finding $Y \neq X$ with $H(Y) = H(X)$, when X and $H(X)$ are given, should also be infeasible. This property is called second preimage resistance. Finally, it should be computationally infeasible to find any two distinct messages X and Y with $H(X) = H(Y)$. This is called collision resistance. Additionally, a hash function is often required to behave indistinguishably from a random function. An ideal hash function that generates an n bit hash value requires evaluating about $2^{n/2}$ messages to find any pair of messages having the same hash value. This kind of brute-force search for hash functions is called a birthday attack. Also 2^n hash computations are required for finding preimages and second preimages [3].

In this paper we have proposed a dedicated hash function MNF-256 that takes a message of arbitrary length and converts it into a 256 bit hash value. It is based on the design principle of NewFORK-256 [4]. The compression function of MNF-256 consists of three parallel branches; each branch compresses a sixteen 32 bit words to eight 32 bit words output. Each branch contains eight step operations. FORK-256 [5] and NewFORK-256, both hash functions built on Merkle-Damgård method [6]. In recent years, many attacks have been presented against these hash functions. The compression function of the proposed hash function uses dithering design. Its padding and splitting of message is similar to Merkle-Damgård construction. Dithering construction is obtained by providing an additional input to the Merkle-Damgård construction. This design was given by Rivest [7]. The iterative structure of this design provides good resistance against some typical attacks such as birthday attack, meet-in-the middle attack and preimage attack. The compression function of MNF-256 has three input parameters and one output parameter. The additional input parameter is a dither value. There are many ways to select dither value. Random numbers generated from Park–Miller algorithm [8] are used as dither value in the proposed algorithm. For each message block in the message, there is different dither value sequence. For each message block 16 different 32 bit dither value generated, out of which each branch makes the use of eight 32 bit dither value according to the ordering rule. All the modifications made to overcome the recent attacks on FORK-256 and NewFORK-256.

The rest of this paper is organized as follows; Section 2 presents the related work. The proposed hash function is presented in Section 3. Section 4 contains the security analysis of MNF-256. The performance analysis of MNF-256 is presented in Section 5. The paper is concluded in Section 6.

2. Related work

2.1. Merkle-Damgård construction and its weaknesses

A typical way to build a hash function is to iteratively apply a fixed-input length compression function. Almost all modern hash functions are based on this principle and the most widespread application of an iterative construction is the Merkle-Damgård paradigm. Even though the paradigm lies on solid theoretical foundation, there have been some flaws exposed in the design, turning the MD construction into an insecure model. Several attacks have shown that the iterative paradigm is not equivalent to the random oracle model used in many security proofs of cryptographic protocols. In the MD method a message of arbitrary length is divided into l bit blocks (with a padding process on the last block), that are sequentially injected into an internal collision resistant compression function. The compression function generates an n bit intermediate digest using the input block and the previous intermediate result. The last intermediate digest is defined as the final n bit hash value; after all input blocks are processed.

- *Length-extension attack*: The well-known weakness of the original Merkle-Damgård construction is a length extension property. If digests of messages M and M' collide then adding a common suffix also leads to a collision: $H(M\|\overline{M}) = H(M'\|\overline{M})$. Even if we concatenate an original message with its length $(M' = M\|length(M))$ the attack is valid because M' can be itself considered as an extendable message.
- *Multicollisions*: The multicollision attacks were proposed by Joux [9]. The idea is to build collisions one after another, which leads to 2^k colliding messages after only k trials of the collision search. If a hash function has an iterative structure, the attack can be always maintained. The actual complexity, however, depends on the size of the internal state.
- *Fixed point attacks:* Dean [10] found that fix-points in the compression function can be used for a second-preimage attack against long messages. It is stated by Dean that for an iterative hash function, if the fix points of compression function can be calculated easily, then finding second-preimages is easier than expected. Davies–Meyer construction fits this condition well. Kelsey and Schneier [11] extend this result.

2.2. Dither construction and its security against generic attacks

The main idea behind dithering hash function is to use an additional input to the Merkle-Damgård hash construction in such

a way that this input will change the chaining values of the each stage. This in turn, makes the problem of finding the fixed points much harder and provides more protection against Dean's attack and its extension under any circumstances. For a message M, divided into n-blocks each of length l, that is, $M = m_1, m_2, \ldots, m_n$ the ith chaining value for the dithering hash design can be formally represented as: $IV_i = f(m_i, IV_{i-1}, d_{i-1})$, $i = 1, 2, \ldots, n$, and d_i represents the dither value. The procedure is shown in Fig. 1. The dither value is selected in such a way that it is repetition free which in turn makes the chaining values of the hash function to be repetition free. Thus, finding fixed points become harder for an attacker. Hence, Dean's attack and its extension can be restricted by the dithering hash construction. By inspecting the structure of the dithering hash function it is clear that there is no additional effort required by the attacker to find multi-collisions just as in the case of the normal Merkle-Damgård hash construction. This construction is secure against general message expansion attack.

2.3. Brief history of dedicated hash functions

The MD4 [12] was proposed by Rivest in 1990. Most commonly used hash functions are based on the design principles of MD4. MD5 [13] was also proposed by Rivest in 1992 as a strengthen version of MD4. Both MD4 and MD5 produce 128 bit message digest. MD5 is slightly slower than MD4. The design principles of MD4 are used in SHA family. SHA-0 [14] was developed in 1993 by National Security Agency as the Secure Hash Standard and SHA-1 [15] was introduced in 1995 as a revision of SHA-0. SHA-1 was issued by NIST as FIPS PUB 180-1. Both SHA-0 and SHA-1 produce a message digest of 160 bit. NIST introduced new hash function standard FIPS PUB 180-2 in 2002. Three new hash functions, SHA-256, SHA-384 and SHA-512, collectively known as SHA-2 [16], have been specified in this standard. Later on another hash function SHA-224 was added to this standard. Another popular hash function family is RIPEMD family. 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. The first RIPEMD hash function was introduced in 1992 under the European RIPE (RACE Integrity Primitives Evaluation) project. It produces 128 bit hash value. RIPEMD runs two almost identical copies of MD4 in parallel. Later two strengthen versions of RIPEMD are released, RIPEMD-128 and RIPEMD-160 [17]. RIPEMD-128 also produces 128 bit message digest as its predecessor. Both RIPEMD-128 and RIPEMD-160 are extended to RIPEMD-256 and RIPEMD-320 respectively. In 1998, MD4 was completely broken by Dobbertin [18]. In 2004, a team of researchers led by Wang, announced collisions in MD5 as well as collisions in other hash functions including MD4, RIPEMD, and HAVAL-128 [19]. In 2005, they presented attacks against SHA-0 and SHA-1 [20].

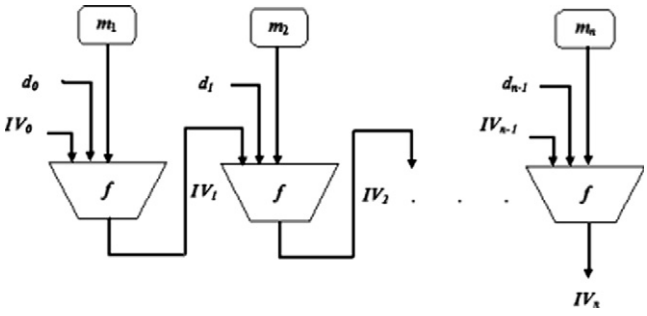

Figure 1 Dither construction.

FORK family hash functions can be viewed as the further extension of RIPEMD family. FORK-256 was the first hash function in FORK family, introduced in the first NIST hash workshop and at FSE 2006. Matusiewicz et al. attacked FORK-256 by using the fact that the functions f and g in the step operation were not bijective They used microcollisions to find collisions of 2-branch FORK-256 and collisions of full FORK-256 with complexity of $2^{126.6}$ [21]. Independently, Mendel et al. [22] published the collision-finding attack on 2-branch FORK-256 using microcollisions and raised possibility of its expansion. At FSE 2007, Matusiewicz et al. another attack which finds a collision with complexity of 2^{108}. FORK-256 was optimized by Danda [23]. NewFORK-256 hash function was introduced in 2007. It includes bijective function in step operation. Saarinen presented collision attack against NewFORK-256 using meet-in-the-middle technique [24]. For this he used a method for finding messages that hash into a significantly smaller subset of possible hash values. The complexity of this collision attack is $2^{112.9}$. This attack is also applicable for FORK-256.

In 2007 NIST introduced a public call for new cryptographic hash algorithms. The intent of the competition is to identify modern secure hash functions and to define the new SHA-3 family [25,26].

2.4. Short description of FORK family

The hash function FORK-256 was introduced at the first NIST hash workshop and at FSE 2006. Later, in 2007 the same team of researchers has published its improved version NewFORK-256. In this new version they modified step operations, removed some additions and XORs and changed non-linear operations of FORK-256. The compression function of FORK-256 and NewFORK-256 consists of four independent branches. Each one of these branches takes in the 256 bit chaining value and a 512 bit message block to produce a 256 bit result. These four branch results are combined with the chaining value to produce the final compression function result. Both algorithms are entirely built on shift, XOR, and addition operations on 32 bit words. Notation for these operations are shown in Table 1.

The four branches are structurally equivalent, but differ in scheduling of the message words and round constants. The Each branch is computed in eight steps, $0 \leqslant k \leqslant 7$. Each step utilizes two message words and two round constants. The scheduling of the message block words $M_0, \ldots, M_{15}$ in each branch is given in Table 2. Each one of the four branches using

Table 1 Basic notations.

Notation	Description
$A \oplus B$	Bitwise exclusive-OR between A and B
$A + B$	$(A + B) \bmod 2^{32}$
$A \lll B$	Circular left shift of 32-bit word A by B bits

Table 2 Message permutation.

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

the same set of chaining variables $CV = (A, B, C, D, E, F, G, H)$. The compression function updates the set of chaining variables according to the following relation:

$$CV_{i+1} = CV_i + \left\{ \begin{array}{l} [BRANCH_1(CV_i, \Sigma_1(M)) + BRANCH_2(CV_i, \Sigma_2(M))] \oplus \\ [BRANCH_3(CV_i, \Sigma_3(M)) + BRANCH_4(CV_i, \Sigma_4(M))] \end{array} \right\} \quad (1)$$

Round constants $\delta_0, \ldots, \delta_{15}$ are given in Table 3 and their schedule in Table 4. FORK-256 uses following two 32 bit Boolean functions f and g:

$$f(x) = x + (x \lll 7 \oplus x \lll 22)$$
$$g(x) = x \oplus (x \lll 13 + x \lll 27)$$

These Boolean functions were redefined for the NewFORK-256 to avoid microcollisions as:

$$f(x) = x \oplus (x \lll 15 \oplus x \lll 27)$$
$$g(x) = x \oplus (x \lll 7 + x \lll 25)$$

For the $BRANCH_j$ $(1 \leqslant j \leqslant 4)$, the message block is compressed as follows:

1. The chaining variable CV_i is assigned to initial variables $V_{j,0}$.
2. At kth step function $(0 \leqslant k \leqslant 7)$, the output $V_{j,k+1}$ is computed as follows:
   $V_{j,k+1} = STEP_{j,k}(V_{j,k}, M_{\sigma_j(2k)}, M_{\sigma_j(2k+1)}, \alpha_{j,k}, \beta_{j,k}), V_{j,8}$ is the output of BRANCH_j.
   $STEP_{j,k}$ uses $V_{j,k}, M_{\sigma_j(2k)}, M_{\sigma_j(2k+1)}, \alpha_{j,k}, \beta_{j,k}$ as inputs and generates the following output:

For FORK-256:

$$A_{j,k+1} = H_{j,k} + g(E_{j,k} + M_{\sigma_j(2k+1)}) \lll 21 \oplus f(E_{j,k} + M_{\sigma_j(2k+1)} + \beta_{j,k}) \lll 17,$$
$$B_{j,k+1} = A_{j,k} + M_{\sigma_j(2k)} + \alpha_{j,k},$$
$$C_{j,k+1} = B_{j,k} + f(A_{j,k} + M_{\sigma_j(2k)}) \oplus g(A_{j,k} + M_{\sigma_j(2k)} + \alpha_{j,k}),$$
$$D_{j,k+1} = C_{j,k} + f(A_{j,k} + M_{\sigma_j(2k)}) \lll 5 \oplus g(A_{j,k} + M_{\sigma_j(2k)} + \alpha_{j,k}) \lll 9,$$
$$E_{j,k+1} = D_{j,k} + f(A_{j,k} + M_{\sigma_j(2k)}) \lll 17 \oplus g(A_{j,k} + M_{\sigma_j(2k)} + \alpha_{j,k}) \lll 21,$$
$$F_{j,k+1} = E_{j,k} + M_{\sigma_j(2k+1)} + \beta_{j,k},$$
$$G_{j,k+1} = F_{j,k} + g(E_{j,k} + M_{\sigma_j(2k+1)}) \oplus f(E_{j,k} + M_{\sigma_j(2k+1)} + \beta_{j,k}),$$
$$H_{j,k+1} = G_{j,k} + g(E_{j,k} + M_{\sigma_j(2k+1)}) \lll 9 \oplus f(E_{j,k} + M_{\sigma_j(2k+1)} + \beta_{j,k}) \lll 5.$$

Table 4 Constant ordering.

STEP-k	$\alpha_{1,k}$	$\beta_{1,k}$	$\alpha_{2,k}$	$\beta_{2,k}$	$\alpha_{3,k}$	$\beta_{3,k}$	$\alpha_{4,k}$	$\beta_{4,k}$
0	δ_0	δ_1	δ_{15}	δ_{14}	δ_1	δ_0	δ_{14}	δ_{15}
1	δ_2	δ_3	δ_{13}	δ_{12}	δ_3	δ_2	δ_{12}	δ_{13}
2	δ_4	δ_5	δ_{11}	δ_{10}	δ_5	δ_4	δ_{10}	δ_{11}
3	δ_6	δ_7	δ_9	δ_8	δ_7	δ_6	δ_8	δ_9
4	δ_8	δ_9	δ_7	δ_6	δ_9	δ_8	δ_6	δ_7
5	δ_{10}	δ_{11}	δ_5	δ_4	δ_{11}	δ_{10}	δ_4	δ_5
6	δ_{12}	δ_{13}	δ_3	δ_2	δ_{13}	δ_{12}	δ_2	δ_3
7	δ_{14}	δ_{15}	δ_1	δ_0	δ_{15}	δ_{14}	δ_0	δ_1

For NewFORK-256:

$$A_{j,k+1} = H_{j,k} \oplus f(E_{j,k} + M_{\sigma_j(2k+1)} + \beta_{j,k}) \lll 8,$$
$$B_{j,k+1} = A_{j,k} + M_{\sigma_j(2k)} + \alpha_{j,k},$$
$$C_{j,k+1} = B_{j,k} + f(A_{j,k} + M_{\sigma_j(2k)}),$$
$$D_{j,k+1} = C_{j,k} + f(A_{j,k} + M_{\sigma_j(2k)}) \lll 13 \oplus g(A_{j,k} + M_{\sigma_j(2k)} + \alpha_{j,k}),$$
$$E_{j,k+1} = D_{j,k} \oplus g(A_{j,k} + M_{\sigma_j(2k)} + \alpha_{j,k}) \lll 17,$$
$$F_{j,k+1} = E_{j,k} + M_{\sigma_j(2k+1)} + \beta_{j,k},$$
$$G_{j,k+1} = F_{j,k} + g(E_{j,k} + M_{\sigma_j(2k+1)}),$$
$$H_{j,k+1} = G_{j,k} + g(E_{j,k} + M_{\sigma_j(2k+1)}) \lll 3 \oplus f(E_{j,k} + M_{\sigma_j(2k+1)} + \beta_{j,k}).$$

3. Proposed hash function

In this section we describe details of the proposed hash function MNF-256. The MNF-256 is a cryptographic dedicated hash function, which compresses three input parameters: 512 bit message block, 256 bit chaining variables and 512 bit dither values into a 256 bit hash value. A structure of three parallel branches has been used in MNF-256. Each branch consists of eight step operations. The basic notations used in MNF-256 are shown in Table 1. The proposed algorithm includes two main stages for the computation of 256 bit hash value: first is preprocessing stage and second one is computation stage. Preprocessing stage contains three steps: message padding, message parsing and initialization of eight chaining variables. Padding procedure of the algorithm is exactly the same as that of SHA-1. In the parsing step the message is divided into N blocks of 512 bit, and the ith block of 512 bit is a concatenation of sixteen 32 bit words. The initial hash value for MNF-256 is the same as that of NewFork-256. The rest of the algorithm details are as follows:

- *Message padding*: The purpose of the message padding is to make the total length of a padded message a multiple of 512. The message M is padded with one bit equal to 1 next to the least significant bit of the message followed by a var-

Table 3 Constants.

δ_0 = 0x428A2F98	δ_1 = 0x71374491	δ_2 = 0xB5C0FBCF	δ_3 = 0xE9B5DBA5
δ_4 = 0x3956C25B	δ_5 = 0x59F111F1	δ_6 = 0x923F82A4	δ_7 = 0xAB1C5ED5
δ_8 = 0xD807AA98	δ_9 = 0x12835B01	δ_{10} = 0x243185BE	δ_{11} = 0x550C7DC3
δ_{12} = 0x72BE5D74	δ_{13} = 0x80DEB1FE	δ_{14} = 0x9BDC06A7	δ_{15} = 0xC19BF174

iable number of zero bits and then appends to the message the 64 bit original message length modulo 2^{64}, so that the total length of the padded message is the exact multiple of 512.

- *Parsing the padded message*: Divide the padded message M into a sequence of N 512 bit blocks. Each message block M_i is expressed as sixteen 32 bit words.
- *Initialization of chaining variables*: There are eight chaining variables A, B, C, D, E, F, G, H. These working variables in each branch are initialized as follows: where

$$A_0 = 0x6A09E667,\ B_0 = 0xBB67AE85,$$

$$C_0 = 0x3C6EF372,\ D_0 = 0xA54FF53A$$

$$E_0 = 0x510E527F,\ F_0 = 0x9B05688C,$$

$$G_0 = 0x1F83D9AB,\ H_0 = 0x5BE0CD19$$

- *Compression function*: The compression function of MNF-256 compresses 512 bit input message block, 256 bit chaining variables and 512 bit dither values to a 256 bit hash value. Each message block M_i is divided into sixteen 32 bit words $M_0, \ldots, M_{15}$ and compressed according to Fig. 2, where $\Sigma_j(M) = (M_{\sigma_j(0)}, \ldots, M_{\sigma_j(15)})$, for $1 \leqslant j \leqslant 3$, is the permutation for the message words, selected from Table 5. The chaining variable CV_i is updated to CV_{i+1} according to following relation:

$$CV_{i+1} = CV_i + \left\{ \begin{array}{l} [BRANCH_1(CV_i, \Sigma_1(M)) + BRANCH_2(CV_i, \Sigma_2(M))] \oplus \\ [BRANCH_2(CV_i, \Sigma_2(M)) + BRANCH_3(CV_i, \Sigma_3(M))] \end{array} \right\} \quad (2)$$

Table 5 Message permutation for MNF-256.

t	0	1	2	3	4	5	6	7
σ_1(t)	0	1	2	3	4	5	6	7
σ_2(t)	14	15	11	9	8	10	3	4
σ_3(t)	7	6	10	14	13	2	9	12
t	8	9	10	11	12	13	14	15
$\sigma_1(t)$	8	9	10	11	12	13	14	15
$\sigma_2(t)$	2	13	0	5	6	7	12	1
$\sigma_3(t)$	11	4	15	8	5	0	1	3

- *Branch function*: Each $BRANCH_j$ for $1 \leqslant j \leqslant 3$, is computed as follows:
  Step 1: the chaining variable CV_i is copied to initial variable $V_{j,0}$ for jth branch.
  Step 2: at kth step of each branch for $0 \leqslant k \leqslant 7$, the step function $STEP_{j,k}$ is computed as follows: $V_{j,k+1} = STEP_{j,k}(V_{j,k}, M_{\sigma_j(2k)}, M_{\sigma_j(2k+1)}, \alpha_{j,k}, \beta_{j,k}, d_{j,k})$,
  where $\alpha_{j,k}$ and $\beta_{j,k}$ are constants and $d_{j,k}$ is the dither input.
- *Constants*: There are total 16 constant values. Each step uses two constant values. Values for constants are given in Table 3. These constants are applied to each branch function. These constants are used in each $BRANCH_j$ with different order. Ordering of constants is shown in Table 6.
- *Step operation*: The input $V_{j,k}$ of $STEP_{j,k}$ is divided as follows:
  $V_{j,k} = (A_{j,k}, B_{j,k}, C_{j,k}, D_{j,k}, E_{j,k}, F_{j,k}, G_{j,k}, H_{j,k})$
  $STEP_{j,k}$ uses $V_{j,k}, M_{\sigma_j(2k)}, M_{\sigma_j(2k+1)}, \alpha_{j,k}, \beta_{j,k}, d_{j,k}$ as inputs and generates the following output:

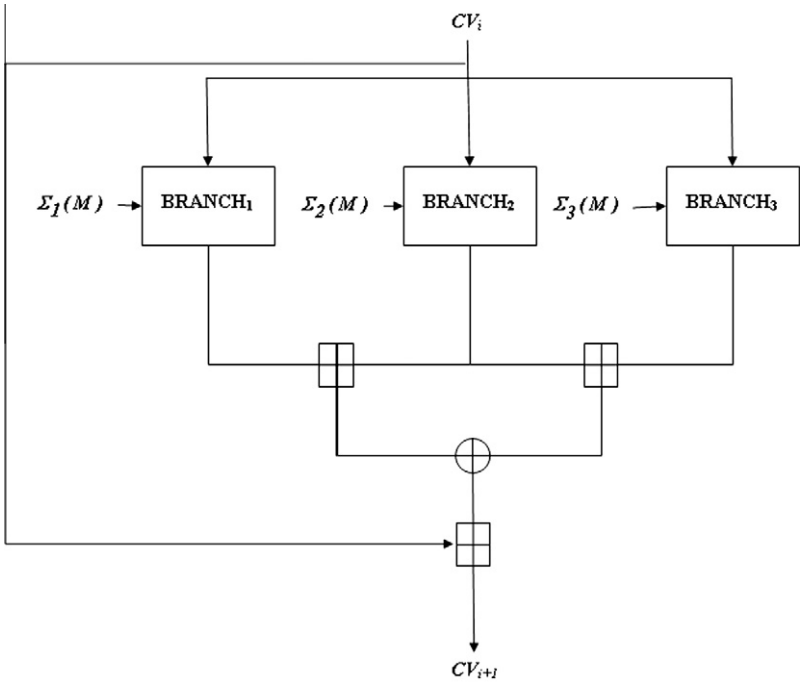

Figure 2 Structure of compression function.

Table 6 Constant ordering for MNF-256.

STEP-k	$\alpha_{1,k}$	$\beta_{1,k}$	$\alpha_{2,k}$	$\beta_{2,k}$	$\alpha_{3,k}$	$\beta_{3,k}$
0	δ_0	δ_1	δ_{15}	δ_{14}	δ_1	δ_0
1	δ_2	δ_3	δ_{13}	δ_{12}	δ_3	δ_2
2	δ_4	δ_5	δ_{11}	δ_{10}	δ_5	δ_4
3	δ_6	δ_7	δ_9	δ_8	δ_7	δ_6
4	δ_8	δ_9	δ_7	δ_6	δ_9	δ_8
5	δ_{10}	δ_{11}	δ_5	δ_4	δ_{11}	δ_{10}
6	δ_{12}	δ_{13}	δ_3	δ_2	δ_{13}	δ_{12}
7	δ_{14}	δ_{15}	δ_1	δ_0	δ_{15}	δ_{14}

$$A_{j,k+1} = H_{j,k} \oplus f(E_{j,k} + M_{\sigma_j(2k+1)} + \beta_{j,k}) \lll 8 \oplus d_{j,k},$$
$$B_{j,k+1} = A_{j,k} + M_{\sigma_j(2k)} + \alpha_{j,k} \oplus d_{j,k},$$
$$C_{j,k+1} = B_{j,k} + f(A_{j,k} + M_{\sigma_j(2k)}) \oplus d_{j,k},$$
$$D_{j,k+1} = C_{j,k} + f(A_{j,k} + M_{\sigma_j(2k)}) \lll 13 \oplus g(A_{j,k} + M_{\sigma_j(2k)} + \alpha_{j,k}) \oplus d_{j,k},$$
$$E_{j,k+1} = D_{j,k} \oplus g(A_{j,k} + M_{\sigma_j(2k)} + \alpha_{j,k}) \lll 17 \oplus d_{j,k},$$
$$F_{j,k+1} = E_{j,k} + M_{\sigma_j(2k+1)} + \beta_{j,k} \oplus d_{j,k},$$
$$G_{j,k+1} = F_{j,k} + g(E_{j,k} + M_{\sigma_j(2k+1)}) \oplus d_{j,k},$$
$$H_{j,k+1} = G_{j,k} + g(E_{j,k} + M_{\sigma_j(2k+1)}) \lll 3 \oplus f(E_{j,k} + M_{\sigma_j(2k+1)} + \beta_{j,k}) \oplus d_{j,k}.$$

The complete step operation is depicted in Fig. 3.

- *Dither values*: For each message block M_i, there is a sequence of 16 different dither values. These dither values are applied to each branch with a different order. Dither value ordering for different branches is shown in Table 7. These values are generated from Park–Miller algorithm. The Park–Miller algorithm is an efficient and fast algorithm for generating good random sequences. It is based on congruential form: $S_{n+1} = aS_n \bmod (2^{31} - 1)$.. The steps of the algorithm are described as follows:

Table 7 Ordering of dither value.

STEP-k	0	1	2	3	4	5	6	7
$d_{1,k}$	d_0	d_2	d_4	d_6	d_8	d_{10}	d_{12}	d_{14}
$d_{2,k}$	d_{15}	d_{13}	d_{11}	d_9	d_7	d_5	d_3	d_1
$d_{3,k}$	d_{14}	d_{12}	d_{10}	d_8	d_6	d_4	d_2	d_0

Step 1: initialize the input seed and parameters, $a = 16{,}807$, $m = 214{,}74{,}83{,}647$, $q = 127{,}773$, $r = 2836$.
Step 2: compute the value of $hi = seed\ div\ q$; $lo = seed \bmod q$.
Step 3: then compute the corresponding test value: $test = a * lo - r * hi$.
Step 4: save the new seed value. If $test > 0$, save test as new seed value, otherwise save $test + m$.
Step 5: output the new seed.
Step 6: iterate, and let the output seed be the new input seed.

- *Shifts*: s bit left shift for a 32 bit string is denoted as $X \lll s$. Four shift values: 13, 17, 3 and 8, have been used in the computation of step operations. They are defined as follows:
  $S1 = X \lll 13,\ S2 = X \lll 17,\ S3 = X \lll 3,\ S4 = X \lll 8.$
- *Nonlinear functions*: MNF-256 uses two nonlinear functions, f and g in the computation of its step operations. Among these two functions f is completely bijective. These functions are defined as follows:

$$f(x) = x \oplus (x \lll 15 \oplus x \lll 27).$$
$$g(x) = x \oplus (x \lll 7 + x \lll 25).$$

Table 8 summarizes the comparative analysis among the FORK-256, NewFORK-256 and MNF-256.

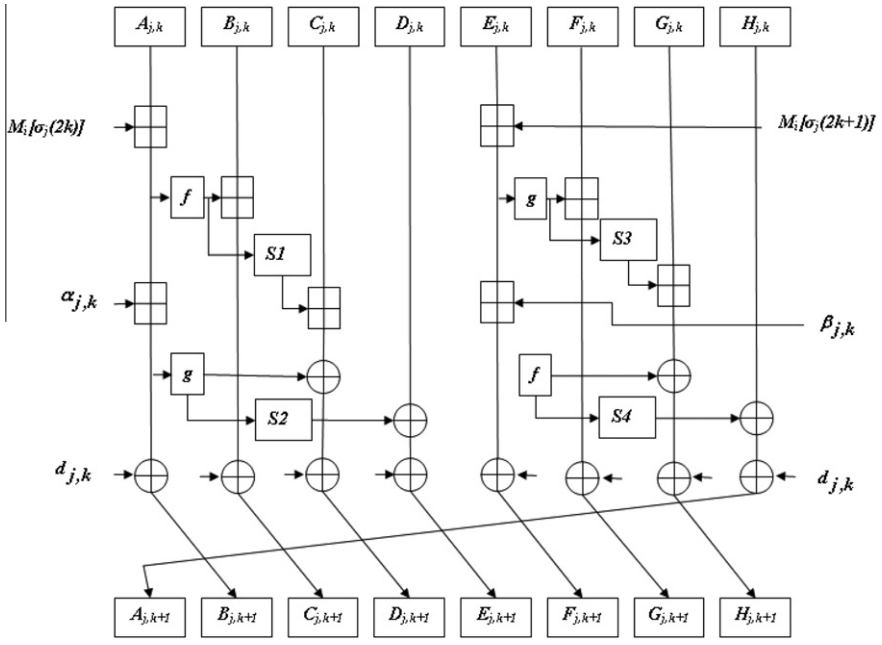

Figure 3 Step operation.

4. Security analysis

First, if an attacker inserts the message difference to find a collision in 3-branch then, he expects the following: $(\Delta_1 + \Delta_2) \oplus (\Delta_2 + \Delta_3) = 0$, where, Δ_i is the output difference of the $BRANCH_i$. To obtain such a differential pattern the attacker should survey the following strategies:

Strategy-1: To construct a differential characteristic with a high probability for a branch function, say $BRANCH_i$ and then expects that, the operation of the output differences in the other branches Δ_3 is equal to Δ_1. Proposed hash function is secure against this strategy because the outputs of each branch function are random; the probability of the event is almost close to 2^{-256}.
Strategy-2: To construct two different differential characteristics such that: $(\Delta_1 + \Delta_2) = -(\Delta_2 + \Delta_3)$. (This can be generated for cancelling the first and second chaining values to obtain the difference between the chaining values as zero, the required condition for generating an attack).

To find an attack using this strategy an attacker has to construct such a differential pattern of the message words. But, for any message words it is computationally hard to find such sequences.

Strategy-3: To insert the message difference which yields same message difference pattern in all the three branches and expect that, same differential characteristics occur simultaneously in three branches.

This strategy is relatively easy for an attacker. However, using the message word reordering this can be avoided just as in the case of FORK and NewFORK. Since the same message word reordering is used in the proposed hash functions same security level can be expected for it against this strategy. Moreover, using different operators (e.g. + and $\oplus$) highly complicates the computation of good differential paths. Addition of message words, parallel mixing structure, rotation of registers and addition of dither value made compression function stronger against different attacks.

5. Performance analysis

5.1. Hash result of a message

For the sake of simplicity, let us consider message, M (1 block, 512 bit), given by:

00112233 44556677 88990011 22334455 66778899
00112233 44556677 88990011 22334455 66778899
00112233 44556677 88990011 22334455 66778899
00112233
Branch1 output:
84fddde4 50791ee4 7f50dc6d b9fef233 8393a036 47d99fac 8dd26400 62363776
Branch2 output:
277310f3 cd9e88e8 9fbd0920 1e217775 2d4c7c1a 4c728465 de58849a a5e62d2c
Branch3 output:
5c2c2978 f5f0b9c7 7fd9061c 70563438 88b387c4 6d950b4e 4cc746de ed33ab54
Final hash result:
50fa289 eb578341 1a07d8d5 a5ebdfeb a7661484 a1e5881a 4dcf854d a315f0fb

5.2. Randomness

We have taken an input message M of 512 bit length and computed corresponding hash value. By changing the ith bit of M, new modified messages M_i have been generated, for $1 \leqslant i \leqslant 512$. Then we generated hash values of all these new messages and finally computed Hamming distances or changed bit numbers between hash values of original message and modified messages. Ideally it should be 128. But we found that these Hamming distances were lying between 106 and 153 for above messages. Range of distances is given in Table 9.

Table 8 Comparison among FORK-256, NewFORK-256 and MNF-256.

Property	FORK-256	NewFORK-256	MNF-256
Input parameters	2	2	3
Output parameter	1	1	1
Construction	Merkle-Damgård	Merkle-Damgård	Dither
Message block size	512 bits	512 bits	512 bits
Word size	32 bits	32 bits	32 bits
Total input bits to the compression function	768 bits	768 bits	1280 bits
Output hash value	256 bits	256 bits	256 bits
Number of branches	4	4	3
Number of message sub-blocks in each step operation	2	2	2
Constants	16	16	16
Number of steps in each branch	8	8	8
Non-linear functions	2	2	2
Bijective function	Absent	Present	Present
Used operations	$+, \lll, \oplus$	$+, \lll, \oplus$	$+, \lll, \oplus$
Efforts required to find preimage	2^{256} operations	2^{256} operations	2^{256} operations
Efforts required to find 2nd-preimage	2^{256} operations	2^{256} operations	2^{256} operations
Efforts required to find collision	2^{128} operations	2^{128} operations	2^{128} operations

Average Hamming distance is 128.0234. Distribution of distance is shown in Fig. 4.

5.3. Bit variance test

The bit variance test consists of measuring the impact on the digest bits by changing input message bits. Bits of an input message are changed and the corresponding message digests (for each changed input) are calculated. Finally from all the digests produced, the probability P_i for each digest bit to take on the value of 1 and 0 is measured. If $P_i(1) = P_i(0) = 1/2$, for all digest bits i $1 \leqslant i \leqslant n$, where n is the digest length, then the hash function under consideration has attained maximum performance in terms of the bit variance test. Therefore, the bit variance test actually measures the uniformity of each bit of the digest. Since it is computationally difficult to consider all input message bit changes, we have evaluated the results for only up to 513 files and found the following results:

Number of digests = 513.
Mean frequency of 1s (expected) = 256.50.
Mean frequency of 1s (calculated) = 256.29.

The above analysis show that MNF-256 exhibits a reasonably good avalanche effect. Thus it can be used for cryptographic applications.

5.4. Statistical analysis of diffusion and confusion

In order to hide message redundancy, Shannon introduced diffusion and confusion. Diffusion means spreading out of the influence of a single plaintext bit so as to hide the statistical structure of the plaintext. Confusion means the use of transformations that complicate dependence of the statistics of ciphertext on the statistics of plaintext. They are two general principles to guide the design of practical cipher, including hash function. For the hash value in binary format, each bit is only 1 or 0. So the ideal diffusion effect should be that any tiny changes in initial conditions lead to the 50% changing probability of each bit.

We have performed the following diffusion and confusion test. A message is randomly chosen and hash value is generated, then a bit in the message is randomly selected and toggled and a new hash value is generated. Two hash values are compared with each other and the number of changed bit is counted as B_i.

This kind of test is performed N (such as 512, 1024, 2048) times. We used four statistics for this: mean changed bit number $\overline{B}$, mean changed probability P, standard deviation of the changed bit number ΔB and standard deviation ΔP.

$$\text{Mean changed bit number} : \overline{B} = \frac{1}{N}\sum_{i=1}^{N} B_i \tag{3}$$

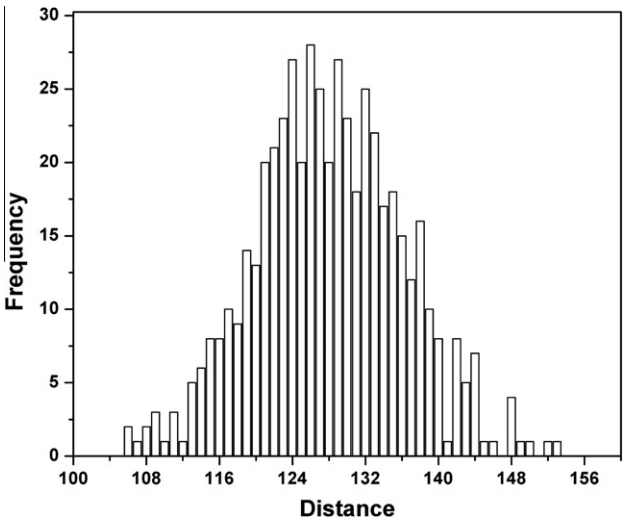

Figure 4 Frequency distribution of distance.

$$\text{Mean changed probability} : P = (\overline{B}/256) \times 100\% \tag{4}$$

Standard deviation of the changed bit number :

$$\Delta B = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(B_i - \overline{B})^2} \tag{5}$$

Standard deviation :

$$\Delta P = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(B_i/256 - P)^2} \times 100\% \tag{6}$$

where N is total statistic number. ΔB and ΔP indicate the stability of diffusion and confusion. Through the tests with N = 512, 1024, 2048, respectively, the corresponding data are listed in Table 10 .

Table 10 indicates that the mean changed bit number and changed percent are 127.94 and 49.97% respectively, which are very close to the ideal value 128 and 50%. While ΔB and ΔP, indicating the stability of diffusion and confusion, is very little, which represent that the capability for diffusion and confusion is stable.

5.5. Analysis of collision resistance

Collision attack is a typical algorithm-independent attack which can apply to any hash function. Collision resistance means that the hash results are identical to different random initial input. Efforts required to find a pair of messages that results to a same hash value for an n-bit hash function is $2^{n/2}$.

Table 9 Range of distances.

Distances	Hash pairs	Percentage
128 ± 5	258	50.39
128 ± 10	413	80.66
128 ± 15	482	94.14
128 ± 20	505	98.63

Table 10 Statistics of number of changed bits.

N	512	1024	2048	Mean
$\overline{B}$	127.68	128.11	128.02	127.94
P	49.87	50.04	50.01	49.97
ΔB	7.40	8.24	8.19	7.94
ΔP%	2.89	3.22	3.19	3.10

Since the length of the hash value is 256 bits, requires 2^{128} operations to find a collision.

Moreover, in order to investigate the collision resistance capability of the hashing approach, we have performed two collision tests. In the first experiment, the hash value for a randomly chosen message is generated and stored in ASCII format. Then a bit in the message is selected randomly and toggled and thus a new hash value is then generated and stored in the same format. Two hash values are compared with each other and the number of character in this format with the same value at the same location in hash value is counted. The absolute difference of the two hash result is calculated by using the following formula:

$$AD = \sum_{i=1}^{N} |dec(e_i) - dec(e_i')| \tag{7}$$

where e_i and e_i' are the ith ASCII character of the original and the new hash value, respectively, $dec()$ converts the entries to their equivalent decimal values. This kind of collision test is performed 2048 times. The maximum, minimum, mean values of AD are listed in Table 11.

In the second experiment, the hash value for a randomly chosen message is generated and stored in ASCII format similarly. This experiment concentrates on the possibility of colliding between every two hash results, thus every two hash results should be compared. The simulation is performed 2048 times. The plot of the distribution of the number of ASCII characters with the same value at same location is given in Fig. 5. Notice that the maximum number of equal entries in the Fig. 5 is 2. Besides, most of the entries are different in ASCII format. It shows that the hash algorithm proposed possesses a strong collision resistance capability.

5.6. Security against preimage and 2nd-preimage attacks

The one way structure of MNF-256 does not allow the reconstruction of the original message from a hash value. In general, collision resistance property provides second preimage resistance for a hash function. MNF-256 has shown good collision resistance. So both, preimage and 2nd-preimage attacks would require at least 2^{256} operations, implying that the proposed hash function has a strong resistance against such attacks.

5.7. Robustness against differential cryptanalysis

We studied the robustness of the proposed hash function against differential cryptanalysis. This attack analyzes the plaintext pairs along with their corresponding hashes pairs. For example, if the difference between 2 messages be 2 bits, (i.e., say, $d = 2$) then the message digest pair difference d' for the corresponding 2 message digests can be calculated. From the distribution of d' corresponding to different message pairs, the standard deviation (σ) is calculated. If $\sigma < 10\%$, then the hash function is secure against differential cryptanalysis. For the experiment input message of 10 bytes was considered. The experiments were run for all possible $d = \{1, 2, 4, 8, 16, 32\}$ bit differences for an input message. The results in Table 12 show that the proposed hash function is secure against the differential attack.

Table 11 Absolute difference.

AD	Max	Min	Mean	Mean/char
Values	3650	1801	2738.74	85.58

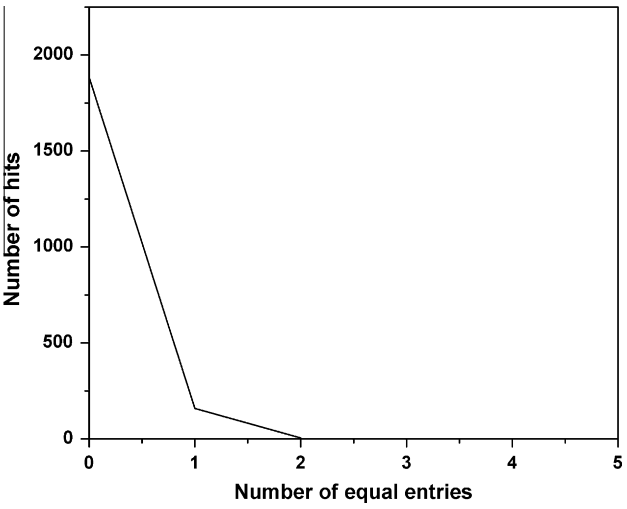

Figure 5 Distribution of the number of equal entries with the same value at the same location in the hash value.

5.8. Efficiency

The performance test has been carried out over an Intel Pentium 4 CPU at 1.47 GHz with 1 GB RAM according to the following procedure: We select a message of size s bytes and generate 1000 random messages of same size. The hash function is applied to each of these 1000 messages, measuring the time required to compute each of them. Finally, we take the average over 1000 samples. In order to compare with FORK-256 and NewFORK-256, the process has been repeated for these algorithms. The average CPU computation times (in sec) obtained for FORK-256, NewFORK-256 and MNF-256 are listed in Table 13.

6. Conclusion

Proposed hash function generates 256 bit hash string. It has a parallel structure consist of three branches. Each branch

Table 12 Results for differential cryptanalysis.

d	1	2	4	8	16	32
σ	8.13	8.07	8.42	7.63	7.79	7.62

Table 13 Computation times.

s (bytes)	FORK-256 (in sec)	NewFORK-256 (in sec)	MNF-256 (in sec)
64	0.0083	0.0062	0.0048
128	0.0638	0.0563	0.0533
10^4	0.3544	0.3519	0.2961
10^5	1.0968	0.9421	0.7193
10^6	8.0742	7.8511	6.2347

computes eight step operations. Algorithm uses 16 constants and two nonlinear functions. Each branch makes the use of 16 message sub blocks and constants with different order. For making the whole structure complicated for the cryptanalyst and secure against known attacks compression function uses three inputs. Dither values are used as third input. These dither values are random numbers generated through Park–Miller algorithm. The one way structure of the algorithm makes it strong against preimage and second preimage attack. Various tests have been performed to check the security level of hash function. The bit variance test has been performed for one bit changes. The result of bit variance test show that MNF-256 exhibits a reasonably good avalanche effect i.e. when a single input bit is complemented, each of the output bits changed with a probability of 0.5. Thus proposed hash function pass the bit variance test. The statistical analysis of MNF-256 indicates that it has strong and stable confusion and diffusion capability. The calculated mean changed bit number and mean changed probability are 127.94 and 49.97% respectively, both very close to the idle value 128 bit and 50% while standard deviation of the changed bit number and standard deviation are very little, which indicates the capability for confusion and diffusion is very stable. The collision test is applied to 2048 different hash pairs. The maximum number of equal characters at the same location in two hash values is only 2. The calculated absolute difference/character of two hash values is 85.58, which is close to the theoretical value 85.33 shows strong collision resistance. Simulation results and rigorous analysis of the hash function guarantee its security against differential and other common known attacks. Further, as future work, we try to improve its efficiency.

References

[1] Schneier B. Applied cryptography. New York: John Wiley & Sons; 1996.

[2] Bakhtiari S, Safavi-Naini R, Pieprzyk J. Cryptographic hash functions: a survey. Technical report 95-09, Department of Computer Science, University of Wollongong; 1995.

[3] Menezes AJ, van Oorschot PC, Vanstone SA. Handbook of applied cryptography. CRC Press; 1997.

[4] Hong D, Chang D, Sung J, Lee S, Hong S, Lee J, et al. NewFORK-256. Cryptology ePrint Archive, report 2007/185; 2007.

[5] Hong D, Chang D, Sung J, Lee S, Hong S, Lee J, et al. A new dedicated 256-bit hash function: FORK-256. In: FSE'06, LNCS, vol. 4047; 2006. p. 195–209.

[6] Damgård I. A design principle for hash functions. In: CRYPTO'89, LNCS, vol. 435; 1989. p. 416–27.

[7] Rivest R. Abelian square-free dithering for iterated hash functions. In: ECRYPT hash function workshop; 2005. <http://csrc.nist.gov/groups/ST/hash/documents/rivest-asf-paper.pdf>.

[8] Park S, Miller K. Random number generators: good ones are hard to find. Commun ACM 1988;31(10):1192–201.

[9] Joux A. Multicollisions in iterated hash functions. In: CRYPTO'04, LNCS, vol. 3152; 2004. p. 306–16.

[10] Dean RD. Formal Aspects of Mobile Code Security. PhD thesis, Princeton University; 1999.

[11] Kelsey J, Schneier B. Second preimages on n-bit hash functions for much less than 2nWork. In: EUROCRYPT'05, LNCS, vol. 3494; 2005. p. 474–90.

[12] Rivest R. The MD4 message digest algorithm. In: CRYPTO'90, LNCS, vol. 537; 1991. p. 303–11.

[13] Rivest R. The MD5 message digest algorithm, request for comments (RFC) 1321. Internet engineering task force; 1992.

[14] NIST, Secure hash standard (SHS), federal information processing standards 180; 1993.

[15] NIST, secure hash standard (SHS), federal information processing standards 180-1; 1995.

[16] NIST, secure hash standard (SHS), federal information processing standards 180-2; 2002.

[17] Preneel B, Bosselaers A, Dobbertin H. RIPEMD-160: a strengthened version of RIPEMD. In: FSE'96, LNCS, vol. 1039; 1997. p. 71–82.

[18] Dobbertin H. Cryptanalysis of MD4. J Cryptol 1998;11(4):253–71.

[19] Wang X, Feng D, Lai X, Yu H. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint archive, report 2004/199; 2004.

[20] Wang X, Yin YL, Yu H. Finding Collisions in the Full SHA-1. In: CRYPTO'05, LNCS, vol. 3621; 2005. p. 17–36.

[21] Matusiewicz K, Contini S, Pieprzyk J. Weaknesses of the FORK-256 compression function. Cryptology ePrint archive, report 2006/317; 2006.

[22] Mendel F, Lano J, Preneel B. Cryptanalysis of reduced variants of the FORK-256 hash function. In: CT-RSA'07, LNCS, vol. 4377; 2006. p. 85–100.

[23] Danda M. Design and analysis of hash functions. Master thesis, Victoria University; 2007.

[24] Saarinen MO. A meet-in-the-middle collision attack against the new FORK-256. In: INDOCRYPT'07, LNCS, vol. 4859; 2007. p. 10–17.

[25] Preneel B. The first 30 years of cryptographic hash functions and the NIST SHA-3 competition. In: RSA'10, LNCS, vol. 5985; 2010. p. 1–14.

[26] Preneel B. The NIST SHA-3 competition: a perspective on the final year. In: AFRICACRYPT'11, LNCS, vol. 6737; 2011. p. 383–6.