Cent. Eur. J. Comp. Sci. • 4(2) • 2014 • 67-85
DOI: 10.2478/s13537-014-0204-7

VERSITA

Central European Journal of **Computer Science**

Building a 256-bit hash function on a stronger MD variant

Research Article

Harshvardhan Tiwari*, Krishna Asawa

Jaypee Institute of Information Technology, India

Received 03 December 2012; accepted 08 April 2014

Abstract: Cryptographic hash functions are important cryptographic techniques and are used widely in many cryptographic applications and protocols. All the MD4 design based hash functions such as MD5, SHA-1, RIPEMD-160 and FORK-256 are built on Merkle-Damgård iterative method. Recent differential and generic attacks against these popular hash functions have shown weaknesses of both specific hash functions and their underlying Merkle-Damgård construction. In this paper we propose a hash function follows design principle of NewFORK-256 and based on HAIFA construction. Its compression function takes three inputs and generates a single output of 256-bit length. An extra input to a compression function is a 64-bit counter (number of bits hashed so far). HAIFA construction shows strong resistance against major generic and other cryptanalytic attacks. The security of proposed hash function against generic attacks, differential attack, birthday attack and statistical attack was analyzed in detail. It is shown that the proposed hash function has high sensitivity to an input message and is secure against different cryptanalytic attacks.

Keywords: cryptographic hash function • MD4 • SHA-1 • FORK-256 • Merkle-Damgård construction

1. Introduction

The rapid growth of new digital technologies increased the demand of information security in communication. Cryptographic hash functions are being widely used in different security applications and protocols such as digital signature, message authentication code, SSL, TLS, etc. for ensuring the integrity and authenticity of information. Cryptographic hash functions are functions that compress an input message of arbitrary length to an output with short fixed length, the hash value. Collision resistance, preimage resistance and second preimage resistance are three important security properties of a hash function [1]. Collision resistance means it is computationally infeasible to find two distinct inputs (M, M') with $H(M) = H(M')$. It is practically impossible to find the preimage M of $H(M)$ when $H(M)$ is given, this is referred to as preimage resistance. Finding $M' \neq M$ with $H(M) = H(M')$, when M and $H(M)$ are given, should also be infeasible. This property is called second preimage resistance. 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. Also it

* *E-mail: tiwari.harshvardhan@gmail.com*

requires 2^n hash computations for finding preimage and second preimage. Cryptographic hash functions are classified into unkeyed hash functions and keyed hash functions. Unkeyed hash functions, also known as modification detection codes (MDCs), use message as a single input whereas keyed hash functions, also known as message authentication codes (MACs), can be viewed as hash functions which take two functionally distinct inputs, a message of arbitrary finite length and a fixed length secret key. In this paper, the unkeyed hash functions are discussed and they are simply called hash functions. There are three main categories of hash functions, namely hash functions based on block cipher, hash functions based on modular algorithm and dedicated hash functions [2]. Other approaches for building hash functions are chaos-based hash functions [3–5] and cellular automata-based hash functions [6]. Most widely used hash functions are MD4 [7] design based dedicated hash functions. These hash functions use traditional Merkle-Damgård iterative structure [8, 9]. The input message M is padded to obtain a message of length multiple m-bits and divided into t blocks of equal length. The hash function H can then be described as follows:

$$h_0 = IV, \quad h_i = f(h_{i-1}, M_i), \quad 1 \le i \le t, \quad H(M) = h_t, \tag{1}$$

where f is a collision resistant compression function, h_i is the i^{th} chaining variable and IV is the initial value of the chaining variable. Recent attacks presented by many researchers have exposed flaws in both Merkle-Damgård construction and specific hash functions. Some attacks against Merkle-Damgård construction are fixed point [10], multi-collision attack [11], second preimage attack [12], herding attack [13]. Due to structural weaknesses, researchers have proposed several variants of Merkle-Damgård construction [14–16]. In this paper we proposed a hash function HNF-256 that takes an input message of arbitrary length and converts it into a 256-bit hash value. The compression function of proposed HNF-256 uses HAIFA iterative mode. Biham and Dunkelman [14] introduced HAIFA, the HAsh Iterative FrAmework. HAIFA is a collection of slight tweaks to the original Merkle-Damgård construction. HAIFA construction is obtained by adding a bit counter and salt to the compression function of Merkle-Damgård construction. The salt is a randomly chosen parameter, which shall be fixed just before hashing. It is an optional input. The counter represents the sum of the number of message bits that have been hashed so far and the number of message bits in the current block to be hashed. Its padding includes digest length. The iterative structure of this design provides good resistance against generic and other cryptanalytic attacks. 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 HNF-256. The performance analysis of hash function is presented in Section 5. Section 6 contains the source code for hash function and the paper is concluded in Section 7.

2. Related work

A number of hash functions have been proposed, most of them have been influenced by the design of the MD4 hash function. The MD4 hash function was proposed by Rivest in 1991 [7]. The algorithm produces hash values of 128-bits in length. It is a very fast hash function optimized for 32-bit architectures. Extended version of MD4 generates 256-bit hash value. MD4 pads a message by appending single bit 1 followed by variable number of 0's until the length of the message is 448 modulo 512 and then the 64-bit length of the message is appended as two 32-bit words. Other MD4 variants also use the same padding rule. The compression function of MD4 takes as input 128-bit chaining variable and a 512-bit message block and maps this to a new chaining variable. Each run of a compression function consists of three rounds and 48 sequential steps (each round consists of 16 steps), where each step is used to update the value of one of the four registers. Every round of MD4 compression function uses a different non-linear boolean function. den Boer and Bosselaers [17] published the first attack on MD4. The attack was on the last two rounds of MD4. They showed that if the first round is omitted, then collision in MD4 can be found easily. Merkle showed an attack on the first two rounds of MD4, but this work was never published. Dobbertin [18] showed that MD4 is not a collision resistant hash function. He also showed that the first two rounds of MD4 are not one-way.
MD5 was also designed by Rivest [19] as a strengthen version of MD4. It generates 128-bit hash value. Padding, parsing and processing of MD5 is almost similar to MD4, but some changes have been made to MD4. Changes include the addition of one extra round along with a new round function and redefined second round function. The compression function uses four rounds, each round has sixteen steps. Four non-linear boolean functions and 64 different additive constants are used in MD5. Each message, like the MD4, after it has been appended by padding bits, processed in blocks of 512 bits. den Boer and Bosselaers [20] found pseudo-collision for MD5. Dobbertin [21] published an attack

that found a collision in MD5. At Crypto'04, Wang *et al.* [22] announced collision in MD5 as well as collisions in other hash functions such as MD4, RIPEMD and HAVAL-128.
Another MD4 design based hash functions are from SHA-family. The original design of the hash function SHA-0 was designed by NSA and published by NIST as FIPS PUB 180[1].Two years later, SHA-0 was withdrawn due to a flaw found in it and replaced by SHA-1, published by NIST as FIPS PUB 180-1[2]. Both SHA-0 and SHA-1 produce a hash value of 160 bits. The only difference between these two versions is that, SHA-1 uses a single bitwise rotation in its message schedule. Padding is done in the same way, then a 512-bit message block is split into sixteen 32-bit words and expands it into eighty 32-bit words using a message expansion relation. Each block is processed in 4 rounds consisting of 20 steps each. These four rounds are structurally similar to one another with the only difference that each round uses a different boolean function and one of four different additive constants. A complete round of SHA-0/1 is made up of 80 steps. NIST published SHA-2 family as FIPS PUB 180-2[3].SHA-2 family consists of four hash functions, SHA-224, SHA-256, SHA-384, and SHA-512. SHA-224 and SHA-384 are the truncated versions of SHA-256 and SHA-512 respectively.
The first result of cryptanalysis of SHA-0 was presented at CRYPTO'98. Chabaud and Joux [23] found collision with complexity 2^{61}. This was a differential attack and faster than generic birthday paradox attack. Biham and Chen [24] found two near-collisions of the full compression function of SHA-0. Biham *et al.* [25] presented collision for the full SHA-0 and reduced SHA-1 algorithms. Wang *et al.* [26] showed collision in SHA-0 in 2^{39} operations. Rijmen and Oswald [27] published an attack on a reduced version of SHA-1. Wang *et al.* [28] presented collisions in full SHA-1 with less than 2^{69} hash operations.
The RIPEMD hash function was designed in the framework of the European Race Integrity Primitives Evaluation (RIPE) project. The design of RIPEMD is based on MD4; its compression function consists essentially of two parallel schemes of the MD4 compression function. It generates 128-bit message digest. Later two strengthen versions of RIPEMD are released, RIPEMD-128 and RIPEMD-160. RIPEMD-128 also produces 128-bit message digest as its predecessor. The RIPEMD-160 hash function [29] processes 512-bit input message blocks and produces a 160-bit hash value. Both RIPEMD-128 and RIPEMD -160 are extended to RIPEMD-256 and RIPEMD-320 respectively.
FORK family hash functions can be viewed as the further extension of RIPEMD family. FORK-256 [30] was the first hash function in FORK family, introduced in the first NIST hash workshop and at FSE 2006. NewFORK-256 [31] hash function was introduced in 2007. In this new version they modified step operations, removed some additions and XORs and changed non-linear operations of FORK-256. It includes bijective function in step operation. 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. The four branches are structurally equivalent, but differ in scheduling of the message words and round constants. Each branch is computed in eight steps. Each step utilizes two message words and two round constants. Matusiewicz *et al.* [32] attacked FORK-256 by using the fact that the functions f and g in the step operation were not bijective. They used micro-collisions to find collisions of 2-branch FORK-256 and collisions of full FORK-256 with complexity of $2^{126.6}$. Mendel *et al.* [33] published the collision-finding attack on 2-branch FORK-256 using micro collisions and raised possibility of its expansion. FORK-256 was optimized by Danda [34]. In [35] a collision attack against NewFORK-256 using meet-in-the-middle technique is presented. 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 competition, similar to the AES contest, for new cryptographic hash algorithms [36]. The intent of the competition is to identify modern secure hash functions and to define the new SHA-3 family. 56 algorithms were submitted, of which NIST accepted 51 for the first round of evaluations. In 2009, out of 51 candidates, 14 candidates were selected for the second round of the competition. After the second round of evaluations, the list of candidates was reduced to 5 for the final round evaluation. These five candidates are Blake, Groestl, Keccak, JH, Skein.

[1] *NIST, Secure hash standard (SHS), Federal Information Processing Standards 180, 1993*
[2] *NIST, Secure hash standard (SHS), Federal Information Processing Standards 180-1, 1995*
[3] *NIST, Secure hash standard (SHS), Federal Information Processing Standards 180-2, 2002*

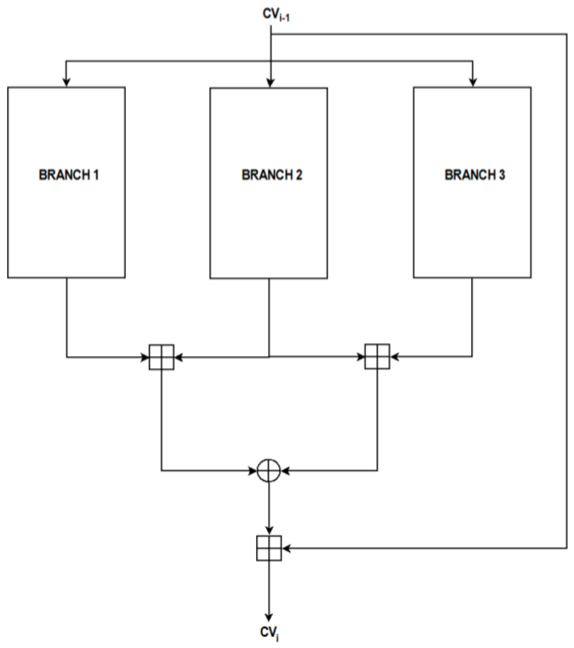


Figure 1. Compression function consists of three parallel branches, each one processing the set of message words in different order.

Table 1. Basic notations.

Notation	Description
$x \boxplus y$	$(x+y) \bmod 2^{32}$
$x <<< y$	Circular left shift of 32-bit word x by y bits
$x \oplus y$	Bitwise exclusive-or (XOR) between x and y
$\left(\alpha_j^{(s)}, \beta_j^{(s)}\right)$	32-bit constant in step s of branch j
$\left(a_j^{(s)}, b_j^{(s)}\right)$	32-bit message word in step s of branch j
X	64-bit counter value

In the last quarter of 2012, NIST announced Keccak as the winner of SHA-3 competition[4].

3. Description of proposed hash function HNF-256

In this section we describe details of the proposed hash function HNF-256. In the proposal of HNF-256, it is aimed to construct a hash function which satisfies all the essential properties of hash function. To achieve this, HAIFA is chosen as the construction method. HNF-256 is entirely built on shift, exclusive-or (XOR), and addition operations on 32-bit words. HNF-256 uses parallel branch structure like FORK-256 and NewFORK-256. FORK-256 and NewFORK-256 use four branches. HNF-256 consists of three branches. By reducing one redundant computation of branches used in compression function we make it more efficient than its parent algorithms. Since branch 2 is used here in left as well

[4] *NIST, Cryptographic Hash Algorithm Competition, http://www.nist.gov/hash-competition*

Table 2. Initialization vector.

CV_0= 0x6A09E667	CV_1= 0xBB67AE85
CV_2= 0x3C6EF372	CV_3= 0xA54FF53A
CV_4= 0x510E527F	CV_5= 0x9B05688C
CV_6= 0x1F83D9AB	CV_7= 0x5BE0CD19

right step structure, so the reduction in branch does not affect over all security of the compression function. Each one of these branches takes in the 256-bit (8-word) chaining value, 64-bit (2-word) counter and a 512-bit (16-word) message block to produce a 256-bit result. These three branch results are combined with the chaining value to produce the final hash result. Figure 1 illustrates the branch structure. The basic notations used in HNF-256 are shown in Table 1.
FORK and NewFORK-256 are based on Merkle-Damgård construction method. Both are vulnerable to generic attacks. We are using HAIFA construction as a mode of operation to build HNF-256 that provides strong resistance to generic attacks. HAIFA construction assures strong platform to iterate the compression function. It is difficult to find fixed points [10] for compression function when it is iterated through HAIFA construction because compression function includes an additional random input counter to compress the message to hash value.
Pre-processing stage contains three steps: message padding, message parsing and initialization of eight chaining variables. Padding procedure of the algorithm is different from other FORK-family based hash functions. Padding of HNF-256 also includes digest length along with message length. Due to inclusion of digest length and message length in the padding rule it is difficult to mount length extension attack on HNF-256.
The purpose of the message padding is to make the total length of a padded message a multiple of 512 because the message block length of the compression function is 512-bit. The message is padded by appending a single bit 1 next to the least significant bit of the message, followed by zero or more bits 0's until the length of the message is 447 modulo 512, and then appends to the message the digest length and message length. Among 65-bits, first bit is set to 1 for representing the 256-bit digest length and other 64-bits represents the length of original message. Padded message is then parsed into 512-bit blocks. Each 512-bit block is a concatenation of sixteen 32-bit words.
The 256-bit chaining variables are used to hold intermediate and final results of the hash function. There are eight chaining variables. The initial values of these chaining variables are exactly the same as values of the eight initial variables used in NewFORK-256 [31]. Hexadecimal values of these initial chaining variables are shown in Table 2.
The counter represents the sum of the number of message bits that have been hashed so far and the number of message bits in the current block to be hashed. HAIFA makes the compression of each block a function of the counter. The counter input adds an extra security layer to the hash function against fixed point attacks. With the inclusion of the counter as an input for each message, the attacker is forced to work harder to find a fixed point. The following example illustrates how the counter value for each message block in the hash function is determined: suppose we have a message with 1021 bits. After padding, this will be broken into 3 blocks of 512, 509 and 0 message bits. The term message bits is important because in the second and third blocks there will be some padding bits, but since we are only considering message bits, we do not consider these. Thus, the counter value for the first block is 512; for the second block it is 512+509 = 1021. For the third the counter value is set to 0 because we consider only message bits, while here third block is containing only padding bits. In the description 64-bit (2-word) counter is represented as $X = [X_0, X_1]$, each word is of 32-bit length. Values of these two 32-bit words depend upon the number of message bits in finally padded message.
The three branches are structurally equivalent, but differ in scheduling of the message words and round constants. Each branch is computed in eight steps, $0 \leq s \leq 7$. Each step utilizes two message words and two round constants. Hash functions either use message permutation or message expansion. It is easy to establish attack on hash function that uses message expansion methods. Also same message ordering in different branches are susceptible to attacks for example, RIPEMD, which consists of two branches and follows the same message ordering in both branches, was fully attacked. On the other hand, in case of RIPEMD-160, there is no attack result because RIPEMD-160 has different message-ordering in branches. So we have used different message ordering in different branches of HNF-256. Message ordering used in three branches is similar to that of message ordering used in first three branches of NewFORK-256. HNF-256 can be implemented efficiently because message ordering is simpler than the message expansion such that

Table 3. Message word ordering.

Step	Branch 1		Branch 2		Branch 3	
S	$a_1^{(s)}$	$b_1^{(s)}$	$a_2^{(s)}$	$b_2^{(s)}$	$a_3^{(s)}$	$b_3^{(s)}$
0	M[0]	M[1]	M[14]	M[15]	M[7]	M[6]
1	M[2]	M[3]	M[11]	M[9]	M[10]	M[14]
2	M[4]	M[5]	M[8]	M[10]	M[13]	M[2]
3	M[6]	M[7]	M[3]	M[4]	M[9]	M[12]
4	M[8]	M[9]	M[2]	M[13]	M[11]	M[4]
5	M[10]	M[11]	M[0]	M[5]	M[15]	M[8]
6	M[12]	M[13]	M[6]	M[7]	M[5]	M[0]
7	M[14]	M[15]	M[12]	M[1]	M[1]	M[3]

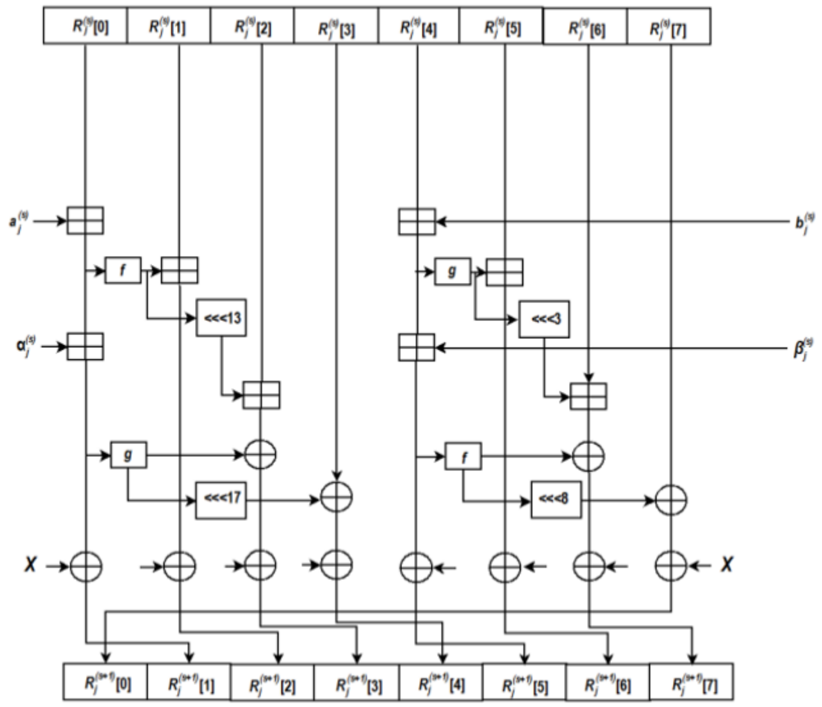


Figure 2. Step function.

of SHA-0/1/2. The scheduling of the message block words $M[O \ldots 15]$ in each branch is given in Table 3. We use left message word $a_j^{(s)}$ and right message word $b_j^{(s)}$ in step s of j^{th} branch (see Figure 2).
Each branch uses sixteen different constants represent the first thirty-two bits of the fractional parts of the cube roots of the first sixteen prime numbers. These constant values are similar to the constants used in NewFORK-256. By using constants we achieve the goal to disturb the attacker who tries to find a good differential characteristic with a relatively high probability. The round constants $\delta[0 \cdots 15]$ are given in Table 4 and their schedule in Table 5 $\alpha_j^{(s)}$ represents left constant and $\beta_j^{(s)}$ represents the right constant.
HNF-256 uses two simple 32-bit functions f and g, which output one word with one input word. Almost all MD4 design based dedicated hash functions use boolean functions which output one word with three words at least. The

Table 4. Round 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] = $ 0x80DEB1E
$\delta[14] = $ 0x9BDC06A7	$\delta[15] = $ 0xC19BF174

Table 5. Constant ordering.

s	$\alpha_1^{(s)}$	$\beta_1^{(s)}$	$\alpha_2^{(s)}$	$\beta_2^{(s)}$	$\alpha_3^{(s)}$	$\beta_3^{(s)}$
0	$\delta[0]$	$\delta[1]$	$\delta[15]$	$\delta[14]$	$\delta[1]$	$\delta[0]$
1	$\delta[2]$	$\delta[3]$	$\delta[13]$	$\delta[12]$	$\delta[3]$	$\delta[2]$
2	$\delta[4]$	$\delta[5]$	$\delta[11]$	$\delta[10]$	$\delta[5]$	$\delta[4]$
3	$\delta[6]$	$\delta[7]$	$\delta[9]$	$\delta[8]$	$\delta[7]$	$\delta[6]$
4	$\delta[8]$	$\delta[9]$	$\delta[7]$	$\delta[6]$	$\delta[9]$	$\delta[8]$
5	$\delta[10]$	$\delta[11]$	$\delta[5]$	$\delta[4]$	$\delta[11]$	$\delta[10]$
6	$\delta[12]$	$\delta[13]$	$\delta[3]$	$\delta[2]$	$\delta[13]$	$\delta[12]$
7	$\delta[14]$	$\delta[15]$	$\delta[1]$	$\delta[0]$	$\delta[15]$	$\delta[14]$

boolean functions can make it easy to control the output one word by adjusting the input several words. The attacks on MD-family, SHA-family and HAVAL are based on this weakness of boolean functions. Additionally, the output words of f and g functions propagate high diffusion to chaining variables. They update other chaining variables whereas in MD or SHA design based output words of boolean functions are used to update only one chaining variable. Functions f and g are used in MNF-256 are same as that of used in NewFORK-256. 32-bit functions f and g, are defined as:

$$\begin{aligned} f(x) &= x \oplus ((x <<< 15) \oplus (x <<< 27)), \\ g(x) &= x \oplus ((x <<<< 7) \boxplus (x <<< 25)). \end{aligned}$$

Let $CV_j[0 \cdots 7]$ be the result of the compression function iteration i and the Initialization Vector given in Table 2. Each branch j processes eight input words to eight output words $R_j^{(8)}, 0 \leq t \leq 7$. $R_j^{(8)}$ is the output of branch. Figure 2 illustrates the step function. For $0 \leq s \leq 7$:

$$
\begin{aligned}
t_1 &= f\left(R_j^{(s)}[0] \boxplus a_j^{(s)}\right), \\
t_2 &= g\left(R_j^{(s)}[0] \boxplus a_j^{(s)} \boxplus \alpha_j^{(s)}\right), \\
t_3 &= g\left(R_j^{(s)}[4] \boxplus b_j^{(s)}\right), \\
t_4 &= f\left(R_j^{(s)}[4] \boxplus b_j^{(s)} \boxplus \beta_j^{(s)}\right), \\
R_j^{(s+1)}[0] &= R_j^{(s)}[7] \oplus (t_4 <<< 8) \oplus X_0, \\
R_j^{(s+1)}[1] &= R_j^{(s)}[0] \boxplus a_j^{(s)} \boxplus \alpha_j^{(s)} \oplus X_1, \\
R_j^{(s+1)}[2] &= R_j^{(s)}[1] \boxplus t_1 \oplus X_0, \\
R_j^{(s+1)}[3] &= \left(R_j^{(s)}[2] \boxplus (t_1 <<< 13)\right) \oplus t_2 \oplus X_1, \\
R_j^{(s+1)}[4] &= R_j^{(s)}[3] \oplus (t_2 <<< 17) \oplus X_0, \\
R_j^{(s+1)}[5] &= R_j^{(s)}[4] \boxplus b_j^{(s)} \boxplus \beta_j^{(s)} \oplus X_1, \\
R_j^{(s+1)}[6] &= R_j^{(s)}[5] \boxplus t_3 \oplus X_0, \\
R_j^{(s+1)}[7] &= \left(R_j^{(s)}[6] \boxplus (t_3 <<< 3)\right) \oplus t_4 \oplus X_1.
\end{aligned} \tag{2}
$$

The final result of the compression function for each word $0 \leq t \leq 7$ is:

$$CV_{i+1}[t] = CV_i[t] \boxplus \left(\left(R_1^{(8)}[t] \boxplus R_2^{(8)}[t]\right) \oplus \left(R_2^{(8)}[t] \boxplus R_3^{(8)}[t]\right)\right).$$

If i is the final iteration, CV_{i+1} is the final hash value.

4. Security analysis

In this section we have discussed the strength of the hash function against known attacks such as first preimage attacks, second preimage attacks, collision attacks, length extension attack and multicollision attack. The preimage attack consists of finding a message which hashes to a given hash value. Resistance against preimage attack can be gained by constructing oneway structure. Using oneway functions and one way transformations or mixing them is one of the most common methods in design of hashing function. Since we have a noninvertible function in each step so as a result each branch is noninvertible. Due to the complexity cost, using meet in the middle technique is also unlikely for preimage attack on HNF-256. This is because, if we can bypass the operations after the branches in reverse mode to access to their output, finding their preimage is not possible due to its complexity. Existing functions within the structure strengthens MNF-256 against preimage attack. There is no method substantially better than brute force search to find the preimage. Thus, brute force search will require about 2^{256} efforts. A second preimage resistance attack is when an attacker is given a message and tries to find another message, where both messages hash to the same value. There is no any scenario concerning preimage attack for such parallel structures like HNF-256. So there does not exist any second preimage attack with complexity lower than 2^{256}. The difficulty of producing two distinct messages having the same hash value is of the order of 2^{128} operations. No collision finding attack is identified against HNF-256 more effective than the birthday attack. There are a few attacks applicable to different spectrum of hash functions. Let us try to analyze the resistance of HNF-256 to these attacks.

4.1. Length extension attack

Most of the generic attacks in some way or other use internal collisions. A lot of hash functions that have an iterative structure suffer from the length extension property. For a function that has this property, once the attacker has one collision, he can easily build many other collisions. For example, if the messages M and M' collide then for any m the messages $M||m$ and $M'||m$ also collide. To overcome this problem, proposed hash design is strengthened by encoding the length of the message and length of the digest into the few last blocks. HNF-256 has strong resistance against the length extension property, because it requires internal collisions to be found in the first place. This way, the first step, *i.e.* finding at least one collision with strong padding rule, of the length extension attack requires at least 2^{256} efforts. Hence the proposal is immune to length extension attack.

4.2. Multicollision attack

The multicollision attack was proposed by Joux. Multicollision set consists of messages that all hash to the same value. The idea is to build collisions one after another, which leads to set of 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 complexity of the attack depends on the size of internal state of compression function since HNF-256 does not use large internal states but they are complex enough and it require at least brute force search to find collision.

4.3. Herding attack

In the Herding attack, the attacker presents the hash value of a message without knowing the beginning of the message. The main idea of the attack is building a diamond structure: a binary tree of collisions. Similarly to the previous multicollision attack, in order to build the diamond structure, internal collisions have to be built. For HAIFA based design it is difficult to generate internal collision using a diamond structure with fixed initial values, random message and a random counter values. Therefore HNF-256 is resistant to the herding attack.

4.4. Fixed point attack

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 preimage is easier than expected. In a hash function, fixed points occur when the intermediate hash value does not change after digesting a given message block.
The complexity of the attack is determined by the complexity of finding expandable messages. Starting from an arbitrary initialization vector, expandable messages are groups of messages of varying lengths whose hash values collide just prior to entering the finalization function *i.e.* just before the digest length and message length are appended. These are messages of varying sizes such that all these messages collide internally for a given initial values. These expandable messages can be quite long, and can be used to generate second preimages for a lot less than 2^{256} work. Fixed point attacks in this form cannot be applied to the HNF-256 because we include the counter values in each iteration of compression function which does not allow to find expandable messages and avoids the existence of trivial fixed point for the design.

4.5. Collision attack

Collision finding attacks on single branch of HNF-256 can be considered in two individual scenarios. The first one is a chosen IV collision attack and the second one is an ordinary collision attack. Chosen IV collision finding attack is an attack which is worth considering on each single branch of HNF-256. In this attack, finding compatible IVs together with appropriate massage differences can be led to collision. Here, this type of attack is not applicable on one branch of HNF-256. Gaining collision in one branch is possible by finding a nonzero XOR difference for some message words and preserving the other message word differences zero. This attack has a complexity not less than what it is in birthday attack.
Ordinary collision attack on single branch of HNF-256 can be successful if someone can insert a differential characteristic through one branch leading to zero differences in the last step. To this aim, the attacker should follow one of the following two strategies: first the attacker inserts one or more non zero difference message words in the first step and expects to meet zero difference words at the end of the last step of one branch. Second, the attacker constructs two individual characteristics for two semi-branches, using meet in the middle technique. In this scenario, the attacker wishes that constructed characteristics for the first four steps and the last four steps in opposite direction meet each others at the end of fourth step.

Let us consider the strategies. Suppose that the attacker inserts one or more nonzero difference message words as input to the first step. Looking at the structure of each step reveals that the two message words along with a counter value are involved in each step. This causes the attackers decision for altering messages gaining to collision too complex. This makes the career of the attacker too hard due to arisen complexity in simultaneous equations. This complexity is not less than what it is in birthday attack (2^{128}) due to existence of some good properties of functions f and g. The second scenario is more complex than the first. In this strategy, he should find two individual and depended characteristics which collide with another in the middle of the branch. So, forced conditions resulted in more simultaneous equations than the first strategy will grow. Moreover, if an attacker inserts the message difference to find a collision in 3-branch then, he expects the following:

$$(\triangle_1 \boxplus \triangle_2) \oplus (\triangle_2 \boxplus \triangle_3) = 0, \tag{3}$$

where, $\triangle_i$ is the output difference of the $BRANCH_1$. 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 $BRANCH_1$ and then expects that, the operation of the output differences in the other branches $\triangle_3$ is equal to $\triangle_1$.
Considering the structure of a branch of HNF-256, it can be easily seen that using functions with good properties, high diffusion structure, and different permutation of input message words for each branch causes the outputs of a branch to be randomized. So it can be expected that finding a collision costs at least 2^{256}.
Strategy-2: To construct two different differential characteristics such that:

$$(\triangle_1 \boxplus \triangle_2) = -(\triangle_2 \boxplus \triangle_3). \tag{4}$$

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-256. 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 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 against different attacks.

5. Performance analysis

5.1. Output hash values

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
Intermediate and final hash values for HNF-256 are given in Table 7.

5.2. Randomness

We have taken an input message M of 512-bit length and computed corresponding hash value. By changing the i^{th} bit of M, new modified messages have been generated, for $1 \leq i \leq 512$. Then we generated hash values of all these new messages using HNF-256 and finally computed Hamming distances between hash values of original message and modified messages. Ideally it should be 128. But we found that these Hamming distances were lying between 108 and

Table 6. Comparison of proposed design with existing designs.

Property	FORK-256	NewFORK-256	HNF-256
Input parameters	2	2	4
Output parameter	1	1	1
Construction	Merkle-Damgård	Merkle-Damgård	HAIFA
Message block size	512-bit	512-bit	512-bit
Word size	32-bit	32-bit	32-bit
Total input bits to the compression function	768-bit	768-bit	832-bit
Output hash value	256-bit	256-bit	256-bit
Number of branches	4	4	3
Message blocks in step	2	2	2
Constants	16	16	16
Number of steps	8	8	8
Shift values	fixed	fixed	fixed
32-bit functions	2	2	2
Bijective function	not present	present	present
Used operations	$\oplus, <<<<, \boxplus$	$\oplus, <<<<, \boxplus$	$\oplus, <<<<, \boxplus$
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
Differential cryptanalysis	complex	complex	complex
Generic attacks	possible	possible	not possible

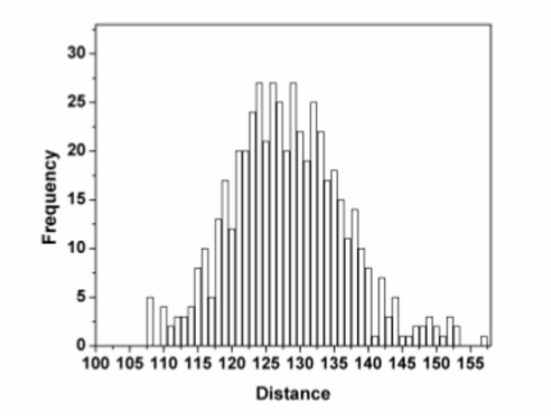
Table 7. Intermediate and final hash values for HNF-256.

	Output							
Branch1	95806BB6	88FE91FC	A3F50D38	E8391DA0	E7C8232C	C687D600	B0D66C14	ABB360FB
Branch2	34513435	9650852E	A68379DC	58AB8E54	C56675D9	1858FAFE	7B53891F	67153F87
Branch3	22742C65	716E8445	EF61CB0A	B022D92B	28D24D9A	4820BF47	61B930CA	41F074C1
Final	91EE5D8	D457CCDE	1C0CB764	ED7AC1C5	9424ACF5	599ED347	10A92685	17ADE1E3

157. Range of distances is given in Table 8. Distribution of distance for HNF-256 and NewFORK-256 is shown in Figure 3. Based on the comparison in Table 8, proposed algorithm has obtained gain over number of hash pairs in the specified range of distances. The analysis shows that HNF-256 generates highly random output and possesses perfect sensitivity property.

Table 8. Range of distances.

	HNF-256		NewFORK-256	
Distances	Hash pairs	Percentage(%)	Hash pairs	Percentage(%)
128±5	262	51.17	231	45.11
128±10	418	81.64	383	74.81
128±15	492	96.09	476	92.96



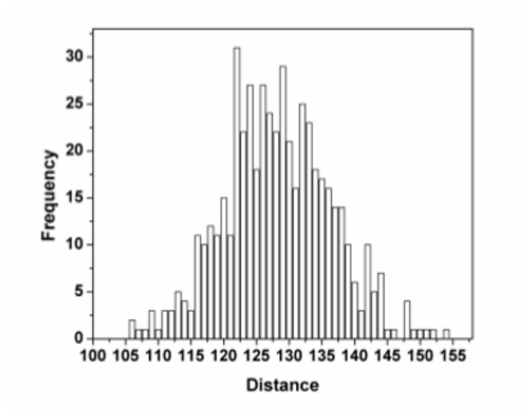


Figure 3. Frequency distribution of distance for HNF-256 (left). Frequency distribution of distance for NewFORK-256 (right).

Table 9. Bit variance analysis.

Hash function	Number of Digests	Mean frequency of 1s (Expected)	Mean frequency of 1s (Calculated)
HNF-256	513	256.50	256.79
NewFORK-256	513	256.50	251.12

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 Pi 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 $i, 1 \leq i \leq n$, 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 results are shown in Table 9.
The above analysis shows that HNF-256 exhibits a reasonably good avalanche effect. Thus it can be used for cryptographic applications.

5.4. Statistical analysis of diffusion

Diffusion is an important criterion in design of hash function. Diffusion means spreading out of the influence of a single plaintext bit so as to hide the statistical structure of the plaintext. It is a general principle to guide the design of 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 input lead to the 50% changing probability of each output bit. We have performed the following diffusion 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 64, 128, 256, 512) times. We used four statistics for this: mean changed bit number $\overline{B}$, mean changed probability P, standard deviation of $\triangle B$ the changed bit number and standard deviation $\triangle P$ [37].
Mean changed bit number:

$$\overline{B} = \frac{1}{N}\sum_{i=1}^{N} B_i. \quad (5)$$

Table 10. Statistics of number of changed bits.

	HNF-256				NewFORK-256			
N	$\overline{B}$	$P\%$	$\triangle B$	$\triangle P$	$\overline{B}$	$P\%$	$\triangle B$	$\triangle P$
64	129.71	50.67	8.69	3.39	127.43	49.78	9.63	3.76
128	128.85	50.33	8.85	3.45	127.91	49.96	8.97	3.51
256	128.78	50.30	9.13	3.56	128.14	50.05	8.53	3.33
512	128.23	50.09	8.46	3.30	127.72	49.89	9.07	3.54
Mean	128.89	50.35	8.78	3.43	127.81	49.92	9.05	3.53

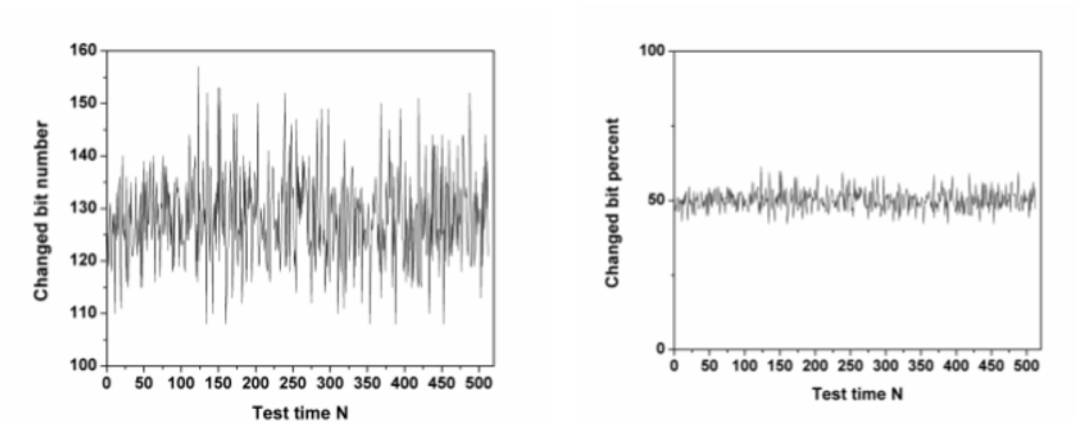


Figure 4. Distribution of changed bit number for HNF-256 (left). Distribution of changed bit percent for HNF-256 (right).

Mean changed probability:

$$P = \left(\overline{B}/256\right) \times 100\%. \tag{6}$$

Standard deviation of the changed bit number:

$$\triangle B = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}\left(B_i - \overline{B}\right)^2}. \tag{7}$$

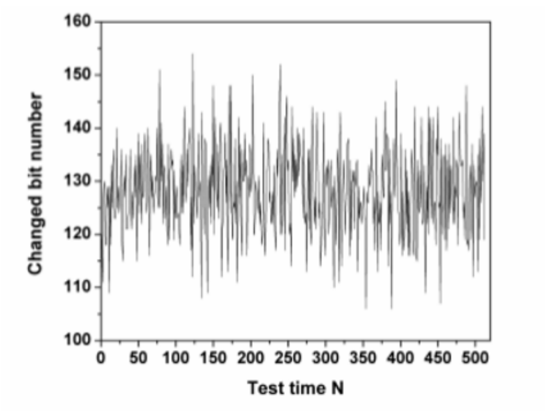
Standard deviation:

$$\triangle P = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(B_i/256 - P)^2} \times 100\%, \tag{8}$$

where, N is total statistic number. $\triangle B$ and $\triangle P$ indicate the stability of diffusion. Through the tests with $N = 64$; 128; 256; 512, respectively, the corresponding data are listed in Table 10. Distribution of changed bit number and percent for HNF-256 and NewFORK-256 is shown in Figure 4 and 5 respectively, where $N = 512$.

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 for finding a pair of messages that results to a same hash value for an n-bit hash function is $2^{n/2}$. 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.



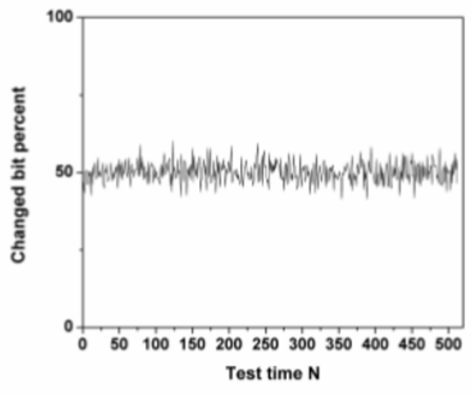


Figure 5. Distribution of changed bit number for NewFORK-256 (left). Distribution of changed bit percent for NewFORK-256 (right).

Table 11. Statistics of absolute difference.

AD	Maximum	Minimum	Mean	Mean/character
HNF-256	3411	1795	2748	85.87
NewFORK-256	3178	1686	2658	83.07

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:

$$AD = \sum_{i=1}^{N} \left| dec(e_i) - dec(e_i') \right| , \tag{9}$$

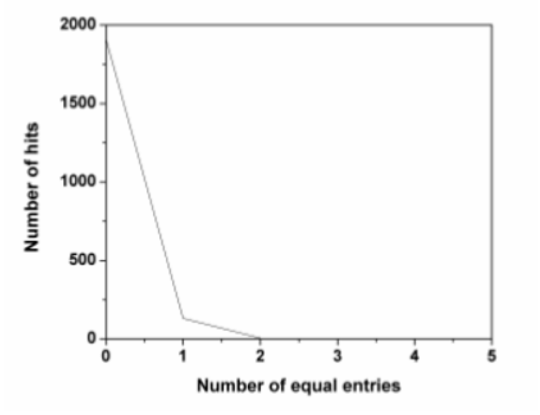
where e_i and e_i' are the i^{th} ASCII character of the original and the new hash value, respectively, $dec()$ converts the entries to their equivalent decimal values. This kind of collision test is performed 2048 times. The maximum, minimum, mean values of AD are listed in Table 11.

Simulation result indicates that the sensitivity property of hash value for HNF-256 is perfect that the absolute difference/character in the final hash value corresponding to any least difference of message will always wave around the theoretical value 85.33.

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 Figure 6. The maximum number of equal entries for HNF-256 is 2 while for NewFORK-256 is 3. So from the Figure 6(a) the hash results could resist collision well. Our proposed algorithm HNF-256 shows a lower value in the maximum number of equal characters at the same location in two hash values than NewFORK-256. It shows that the proposed hash algorithm possesses a strong collision resistance capability.

5.6. 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,



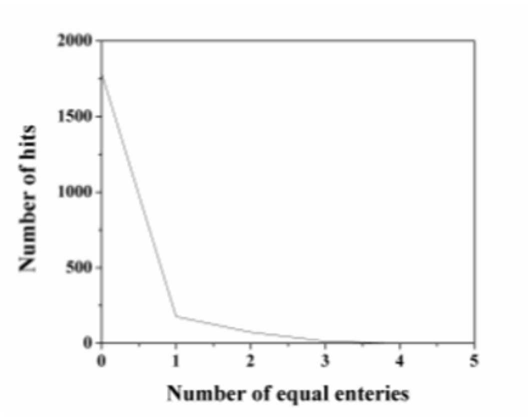


Figure 6. Distribution of the number of ASCII characters with the same value at the same location in the hash value for HNF-256 (left). Distribution of the number of ASCII characters with the same value at the same location in the hash value for NewFORK- 256 (right).

Table 12. Results for differential cryptanalysis.

d	1	2	4	8	16	32
HNF-256(σ)	7.73	7.92	8.47	8.13	8.57	8.51
NewFORK-256(σ)	7.82	7.86	8.92	9.26	8.11	9.89

(*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 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 has better resistance than NewFORK-256 against the differential attack.

5.7. Efficiency

The performance test has been carried out over an Intel Pentium 4 CPU at 1.47 GHz with 1GB 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 HNF-256 are listed in Table 13.These experimental results establish the higher speed of execution of HNF-256 than FORK-256 and NewFORK-256.

Table 13. Computation times.

S(bytes)	64	128	10^4	10^5
FORK-256(sec)	0.0084	0.0657	0.3614	1.1187
NewFORK-256(sec)	0.0063	0.0577	0.3606	0.9656
HNF-256(sec)	0.0043	0.0409	0.2741	0.7145

6. Source code

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

```
using namespace std;
unsigned int delta[16] = {
0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174
};
unsigned int X[]={0x00000000,0x00000200};
#define ROL(x, n) ( ( (x) << n ) | ( (x) >> (32-n) ) )
#define f(x) ( x ^ ROL(x,15) ^ ROL(x,27) )
#define g(x) ( x ^ (ROL(x,7) + ROL(x,25)) )
#define step(A,B,C,D,E,F,G,H, M1,M2,D1,D2,counter) \
temp1 = A + M1; \
temp2 = E + M2; \
A = (temp1 + D1) ^ counter ; \
E = (temp2 + D2) ^ counter ; \
temp1 = f(temp1); \
temp2 = g(temp2); \
temp3 = g(A); \
temp4 = f(E); \
B += temp1 ^ counter ; \
F += temp2  ^ counter; \
C = (C + ROL(temp1, 13)) ^ (temp3) ^ counter ; \
G = (G + ROL(temp2, 3)) ^ (temp4 )^ counter ; \
D ^= ROL(temp3, 17) ^ counter ; \
H ^= ROL(temp4, 8) ^ counter ; \
static void HNF256_Compression_Function(unsigned int *CV, unsigned int *M) {
unsigned long R1[8],R2[8],R3[8];
unsigned long temp1, temp2, temp3, temp4;
R1[0] = R2[0] = R3[0]  = CV[0];
R1[1] = R2[1] = R3[1]  = CV[1];
R1[2] = R2[2] = R3[2]  = CV[2];
R1[3] = R2[3] = R3[3]  = CV[3];
R1[4] = R2[4] = R3[4]  = CV[4];
R1[5] = R2[5] = R3[5]  = CV[5];
R1[6] = R2[6] = R3[6]  = CV[6];
R1[7] = R2[7] = R3[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],X[0]);
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],X[1]);
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],X[0]);
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],X[1]);
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],X[0]);
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],X[1]);
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],X[0]);
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],X[1]);
// 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],X[0]);
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],X[1]);
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],X[0]);
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],X[1]);
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],X[0]);
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],X[1]);
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],X[0]);
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],X[1]);
// 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],X[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],X[1]);
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],X[0]);
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],X[1]);
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],X[0]);
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],X[1]);
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],X[0]);
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],X[1]);
// output
CV[0] = CV[0] + ((R1[0] + R2[0]) ^ (R2[0] + R3[0]));
CV[1] = CV[1] + ((R1[1] + R2[1]) ^ (R2[1] + R3[1]));
CV[2] = CV[2] + ((R1[2] + R2[2]) ^ (R2[2] + R3[2]));
CV[3] = CV[3] + ((R1[3] + R2[3]) ^ (R2[3] + R3[3]));
CV[4] = CV[4] + ((R1[4] + R2[4]) ^ (R2[4] + R3[4]));
CV[5] = CV[5] + ((R1[5] + R2[5]) ^ (R2[5] + R3[5]));
CV[6] = CV[6] + ((R1[6] + R2[6]) ^ (R2[6] + R3[6]));
CV[7] = CV[7] + ((R1[7] + R2[7]) ^ (R2[7] + R3[7]));
}
int main()
{
    int unsigned CV0[8];
    CV0[0] = 0x6a09e667; CV0[1] = 0xbb67ae85;
    CV0[2] = 0x3c6ef372; CV0[3] = 0xa54ff53a;
    CV0[4] = 0x510e527f; CV0[5] = 0x9b05688c;
    CV0[6] = 0x1f83d9ab; CV0[7] = 0x5be0cd19;
unsigned int M[]={0x112233,0x44556677,0x88990011,0x22334455,0x66778899,0x112233,0x44556677,
0x88990011,0x22334455,0x66778899, 0x112233, 0x44556677 ,0x88990011,0x22334455 , 0x66778899,
0x112233};
HNF256_Compression_Function(CV0,M);
return 0;
}
```

7. Conclusions

In this paper NewFORK-256 design based hash functions is proposed. It processes a message of arbitrary length by 512-bit blocks and produce as output a 256-bit hash value or message digest. It is built on HAIFA iterative structure. Compression function of HNF-256 takes three input parameters: 512-bit message block, 256-bit chaining variable and 64-bit counter and produce a single output of 256-bit length. Its iterative structures preserves all the three security properties: collision resistance, pre-image and second pre-image and achieve the high level security against major generic attacks. We have analyzed the proposed hash function for randomness and security. The bit variance test has been performed for one bit changes. Bit variance test results show that HNF-256 has a 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 HNF-256 indicates that it has strong and stable confusion and diffusion capability. The calculated mean changed bit number and mean changed probability for both hash functions are 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. It possesses high message sensitivity and good statistical properties.

References

[1] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone, Handbook of Applied Cryptography (CRC Press, 1997)
[2] B. Schneier, Applied Cryptography (John Wiley & Sons, 1996)
[3] H. S. Kwok, W. K. S. Tang, A Chaos Based Cryptographic Hash Function for Message Authentication, Int. J. Bifur. Chaos 15, 4043–4050, 2005
[4] Y. Li, S. Deng, D. Xiao, A Novel Hash Algorithm Construction Based on Chaotic Neural Network, Neural Comput. Appl. 20, 133–141, 2011
[5] Y. Li, D. Xiao, S. Deng, G. Zhou, Improvement and Performance Analysis of a Novel Hash Function Based on Chaotic Neural Network, Neural Comp. Appl. 22, 391–402, 2013
[6] M. Mihaljevie, Y. Zheng, H. Imai, A Cellular Automaton Based Fast One-Way Hash Function Suitable for Hardware Implementation, PKC '98, LNCS 1431, 217–233, 1998
[7] R. Rivest, The MD4 Message Digest Algorithm, CRYPTO'90, LNCS 537, 303–311, 1991
[8] I. Damgård, A Design Principle for Hash Functions, Crypto'89, LNCS 435, 416–427, 1990
[9] R. Merkle, One Way Hash Functions and DES, CRYPTO'89, LNCS 435, 428–446, 1990
[10] R. D. Dean, Formal Aspects of Mobile Code Security, PhD Thesis (Princeton University, Princeton, 1999)
[11] A. Joux, Multicollisions in Iterated Hash Functions, CRYPTO'04, LNCS 3152, 306–316, 2004
[12] J. Kelsey, B. Schneier, Second Preimages on n-bit Hash Functions for Much Less than 2n Work, EUROCRYPT'05, LNCS 3494, 474–490, 2005
[13] J. Kelsey, T. Kohno, Herding Hash Functions and the Nostradamus Attack, EUROCRYPT'06, LNCS 4004, 183–200, 2006
[14] E. Biham, O. Dunkelman, A Framework for Iterative Hash Functions-HAIFA, Cryptology ePrint Archive, Report2007/278, 2006
[15] R. Rivest, Abelian Square-free Dithering for Iterated Hash Functions, ECRYPT Hash Function Workshop, Cracow, June 21, 2005
[16] S. Hirose, J. H. Park, A. Yun, A Simple Variant of the Merkle- Damgård Scheme with a Permutation, Asiacrypt'08 4833, 113–129, 2008
[17] B. den Boer, A. Bosselaers, An Attack on the Last Two Rounds of MD4, Crypto'91, LNCS 576, 194–203, 1992
[18] H. Dobbertin, Cryptanalysis of MD4, FSE'96, LNCS 1039, 53–69, 1996
[19] R. Rivest, The MD5 Message Digest Algorithm, Request for Comments (RFC) 1321, Internet Engineering Task Force, 1992
[20] B. den Boer, A. Bosselaers, Collisions for the Compression Function of MD5, Eurocrypt'93, LNCS 765, 293–304, 1994
[21] H. Dobbertin, Cryptanalysis of MD5, Rump Session, EUROCRYPT'96, 1996
[22] X. Wang, F. X. Feng, X. Lai, H. Yu, Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD, Rump Session, CRYPTO'04, Santa Barbara, California, USA, August 17, 2004
[23] F. Chabaud, A. Joux, Differential Collisions in SHA-0, Crypto'98, LNCS 1462, 56–71, 1998
[24] E. Biham, R. Chen, Near-collisions of SHA-0, Crypto'04, LNCS 3152, 290–305, 2004
[25] E. Biham, R. Chen, A. Joux, P. Carribault, C. Lemuet, W. Jalby, Collision of SHA-0 and Reduced SHA-1, Eurocrypt'05, LNCS 3494, 36–57, 2005
[26] X. Wang, Yu, Y. L. Yin, Efficient Collision Search Attacks on SHA-0, CRYPTO'05, LNCS 3621, 1–16, 2005
[27] V. Rijmen, E. Oswald, Update on SHA-1, RSA'05, LNCS 3376, 58–71, 2005
[28] X. Wang, Y. L. Yin, H. Yu, Finding Collisions in the Full SHA-1, CRYPTO'05, LNCS 3621, 17–36, 2005
[29] H. Dobbertin, A. Bosselaers, B. Preneel, RIPEMD-160- A Strengthened Version of RIPEMD, FSE'96, LNCS 1039, 71–82, 1996
[30] D. Hong, D. Chang, J. Sung, S. Lee, S. Hong, J. Lee, D. Moon, S. Chee, A New Dedicated 256-bit Hash Function:

FORK-256, FSE'06, LNCS 4047, 195–209, 2006

[31] D. Hong, D. Chang, J. Sung, S. Lee, S. Hong, J. Lee, D. Moon, S. Chee, NewFORK-256, Cryptology ePrint Archive, Report 2007/185, 2007

[32] K. Matusiewicz, S. Contini, J. Pieprzyk, Weaknesses of the FORK-256 Compression Function, Cryptology ePrint Archive, Report 2006/317, 2006

[33] F. Mendel, J. Lano, B. Preneel, Cryptanalysis of Reduced Variants of the FORK-256 Hash Function, RSA'07, LNCS 4377, 85–100, 2006

[34] M. Danda, Design and Analysis of Hash Functions, Master Thesis (Victoria University, 2007)

[35] M. O. Saarinen, A Meet-In-the-Middle Collision Attack Against the New FORK-256, INDOCRYPT'07, LNCS 4859, 10–17, 2007

[36] B. Preneel, The NIST SHA-3 Competition: A Perspective on the Final Year, AFRICACRYPT'11, LNCS 6737, 383–386, 2011

[37] K. W. Wong, A Combined Chaotic Cryptographic and Hashing Scheme, Phys. Lett. A 307, 292–298, 2003