NESHA-256, NEw 256-bit Secure Hash Algorithm (Extended Abstract)

Yaser Esmaeili Salehani[1], S. Amir Hossein A.E. Tabatabaei[2],
Mohammad Reza Sohizadeh Abyaneh[3], Mehdi Mohammad Hassanzadeh[3]

[1] Sharif University of Technology, Tehran, Iran
[2] Sadra University, Tehran, Iran
[3] Selmer Center, University of Bergen, PB. 7803, N-5020 Bergen, Norway

yaser.esmaeili@gmail.com, reza.sohizadeh@ii.uib.no, Mehdi.Hassanzadeh@ii.uib.no

Abstract. In this paper, we introduce a new dedicated 256-bit hash function: NESHA-256. The recently contest for hash functions held by NIST, motivates us to design the new hash function which has a parallel structure. Advantages of parallel structures and also using some ideas from the designing procedure of block-cipher-based hash functions strengthen our proposed hash function both in security and in efficiency. NESHA-256 is designed not only to have higher security but also to be faster than SHA-256: the performance of NESHA-256 is at least 38% better than that of SHA-256 in software. We give security proofs supporting our design, against existing known cryptographic attacks on hash functions.

Keywords: Hash function, NIST, Diffusion layer, Differential attack

1 Introduction

A cryptographic hash function is a transformation that takes an arbitrary finite length input and returns a fixed-size string called the hash value. The hash value is a concise representation of the longer message or document from which it was computed. Cryptographic hash functions are a group of cryptographic functions used in message integrity check, digital signatures, e-cash and many other cryptographic schemes and applications.

For a cryptographic hash function, the following general security requirements considerations are needed according to *Complexity Theory*:

Pre-image resistance: It is infeasible to find any input message which hashes to any pre-specified image.

Second pre-image resistance: It is infeasible to find any second input which has the same output as pre-specified input message.

Collision resistance: It is infeasible to find two different messages which hash to one message digest.

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

Since hash functions are desired to be fast in performance, recent designing methods of hash functions are based on sequentially iterating a simple and fast step function. The most popular hash functions, which are called MD-like, have been designed according to this method in an evolutional process. MD4 was the first type of MD-like structure which proposed by Rivest in 1990 [20]. MD4 was a novel design, oriented towards software implementation on 32-bit architectures. Several hashing algorithms were derived from MD4 hash function called MDx-class hash function. MD5 [21], SHA0/1 [17], HAVAL [27] and RIPEMD [1] are some prominent instances (also see: [18,23]). These hash functions are the most popular hash functions because of their performance and trust gained from cryptanalysis techniques [18,23]. All of the mentioned hash functions are based on a serial method except for RIPEMD. The RIPEMD family of hash functions was designed by combining sequential structure and parallel framework. This method of designing is still reliable due to no effective attack so far, except elementary versions of RIPEMD [23,4]. Also, there are several methods to use a block cipher to build a cryptographic hash function [19]. The methods resemble the block cipher modes of operation usually used for encryption. In 2005, security flaws were identified in both algorithms MD5 and SHA0/1 [24,25,26]. In 2007, the NIST announced a contest to design a hash function which will be given the name SHA-3 and the subject of a FIPS standard. This announcement, advantages of the structures of both parallel and block-cipher-based hash functions, motivate us to design NESHA-256.

This paper is organized as follows: In Section 2 and Section 3, we describe the structure of NESHA-256 along with its specifications. Our design rationales are given in Section 4. This explanation is followed by security analysis, statistical tests, and, performance evaluation in Section 5 and Section 6, respectively. Section 7 includes the final results and some concluding remarks. The test vector is given in the Appendix A .

2 Preprocessing

Preprocessing shall take place before hash computation begins. This preprocessing consists of three steps: padding the message, M (Section 2.1), parsing the padded message into message blocks (Section 2.2), and setting the initial hash value, (Section 2.3).

2.1 Padding the message

The message, M, shall be padded before hash computation begins. The purpose of this padding is to ensure that the padded message is a multiple of 512 bits. The padding mechanism is the same as SHA-1 algorithm [17] as follows: Suppose that the length of the message, M, is l bits. Append the bit “1” to the end of the message, followed by k zero bits, where k is the smallest, non-negative solution to the equation $l+1+k = 448 \bmod 512$. Then append the 64-bit block that is equal to the number l

expressed using a binary representation. The length of the padded message should now be a multiple of 512 bits.

2.2 Parsing the padded message

For NESHA-256, the padded message is parsed into N 512-bit blocks, M_0, $M_1,\ldots,M_{N-1}$. Since the 512 bits of the input block may be expressed as sixteen 32-bit words, the first 32 bits of message block i are denoted $M_0^{(i)}$, the next 32 bits $M_1^{(i)}$, and so on up to $M_{15}^{(i)}$.

2.3 Setting the initial hash value

Before hash computation begins for each of the NESHA-256 hash algorithms, the initial hash value, $IV_0= CV_0=(A,B,C,D,E,F,G,H)$ must be set as follows:
A = 0x6a09e667, B = 0xbb67ae85, C = 0x3c6ef372, D = 0xa54ff53a,
E = 0x510e527f, F = 0x9b05688c,G = 0x1f83d9ab, H = 0x5be0cd19.

3 NESHA-256 Algorithm

In this section, we describe the structure of our proposed hash function: NESHA-256. The compression function of NESHA-256 hashes a 512-bit string to a 256-bit string as shown in Fig.1. When each message block is compressed by compression function, it uses previous compression output as its chaining variable. According to Fig. 1 four parallel branch functions are called BRANCH 1 to BRANCH 4. The chaining variable for i^{th} block (i^{th} compression function) is $CV_i=(A,B,C,D,E,F,G,H)$ initialized by IV_0, as mentioned in Section 2.3.

3.1 NESHA-256 preprocessing

1. Pad the message, M, according to Section 2.1.
2. Parse the padded message into N 512-bit message blocks, M_0, M_1, …, $M_{\text{N-1,}}$ according to Section 2.2.
3. Set the initial hash value, IV_0, as specified in Section 2.3.

3.2 NESHA-256 computations

Each message block M is divided to sixteen 32-bit words $M_0,\ldots,M_{15}$ and compressed according to Fig. 1, where $\sum_j (M) = (M_{\sigma_j(0)},\ldots, M_{\sigma_j(15)})$, $j = 1,2,3,4$ is the permutation for message words, selected from Table 1 (Section 3.2.2). The chaining variable CV_i is updated according to the following relation.

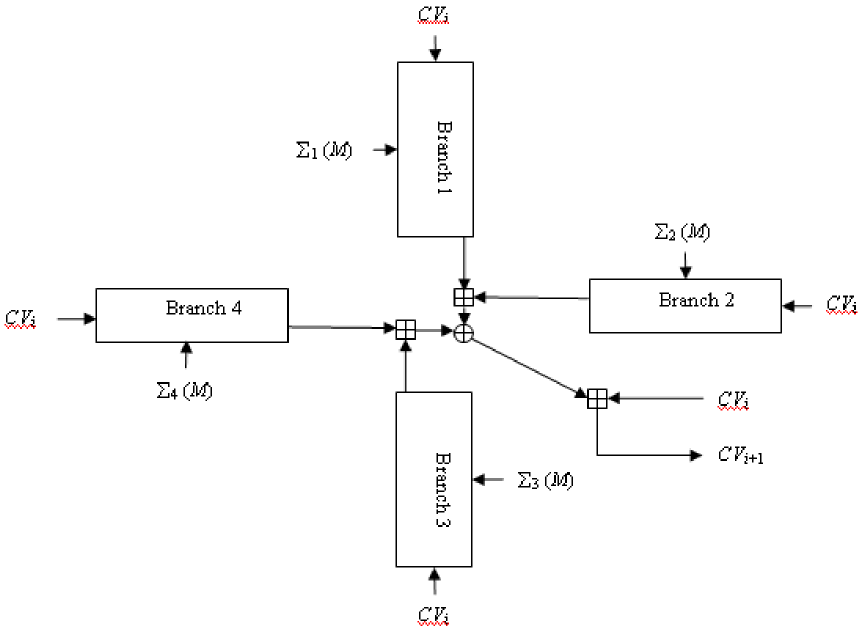

Figure 1. NESHA-256 compression

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

3.2.1 BRANCH FUNCTIONS of NESHA-256

Each branch function (BRANCH j, j=1,…,4) of NESHA-256 contains four step functions which compresses an input message block according to the following instructions.

- Initial variables $V_{j,0}$ are allocated by chaining variables CV_i .
- For k = 0 to 3 step function k+1 (shown in Figure. 2) computes $V_{j,k+1}$ as follows:

$$V_{j,k+1} = STEP_{j,k}(V_{j,k}, M_{\sigma_j(4k)}, M_{\sigma_j(4k+1)}, M_{\sigma_j(4k+2)}, M_{\sigma_j(4k+3)} \\ , \widehat{M}_{\sigma_j(4k)}, \widehat{M}_{\sigma_j(4k+1)}, \widehat{M}_{\sigma_j(4k+2)}, \widehat{M}_{\sigma_j(4k+3)} \\ , \alpha_{j,4k}, \alpha_{j,4k+1}, \alpha_{j,4k+2}, \alpha_{j,4k+3}) \qquad (2)$$

where $\alpha_{j,4k}, \alpha_{j,4k+1}, \alpha_{j,4k+2}, \alpha_{j,4k+3}$ are constant values. In which $\widehat{M}_{\sigma_j(4k)}, \widehat{M}_{\sigma_j(4k+1)}, \widehat{M}_{\sigma_j(4k+2)}, \widehat{M}_{\sigma_j(4k+3)}$ are constructed by other message block words for each branch according to (3) and functions *f* and *g* are word-oriented defined by relation 4.

$$\widehat{M}_{\sigma_j(t)} = M_{\sigma_j\left((t+1) \bmod 4 + 4.\left[\frac{t}{4}\right]\right)} \\ \oplus (M_{\sigma_j\left((3t-8.\left[\frac{t}{4}\right]+4) \bmod 16\right)} \\ + M_{\sigma_j\left((3t-8.\left[\frac{t}{4}\right]+5) \bmod 16\right)} \\ + M_{\sigma_j\left((3t-8.\left[\frac{t}{4}\right]+6) \bmod 4\right)}) \qquad (3)$$

$$t = 0,1,\ldots, 15 \\ j = 1,2,3,4 \text{ (Branch Number)} \\ f(X) = X + (X^2 \vee 7) \qquad (4) \\ g(X) = X^2 + 3X + \beta$$

Where β is a 32-bit constant word assigned to 0xbf597fc7.

3.2.2 Permutations of message words

The permutation of message words in NESHA-256 is designed based on Latin square matrices. Table 1 represents the order of message words $M_0,\ldots,M_{15}$ applied to each four branches.

Table 1. Message words permutation for all branches.

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

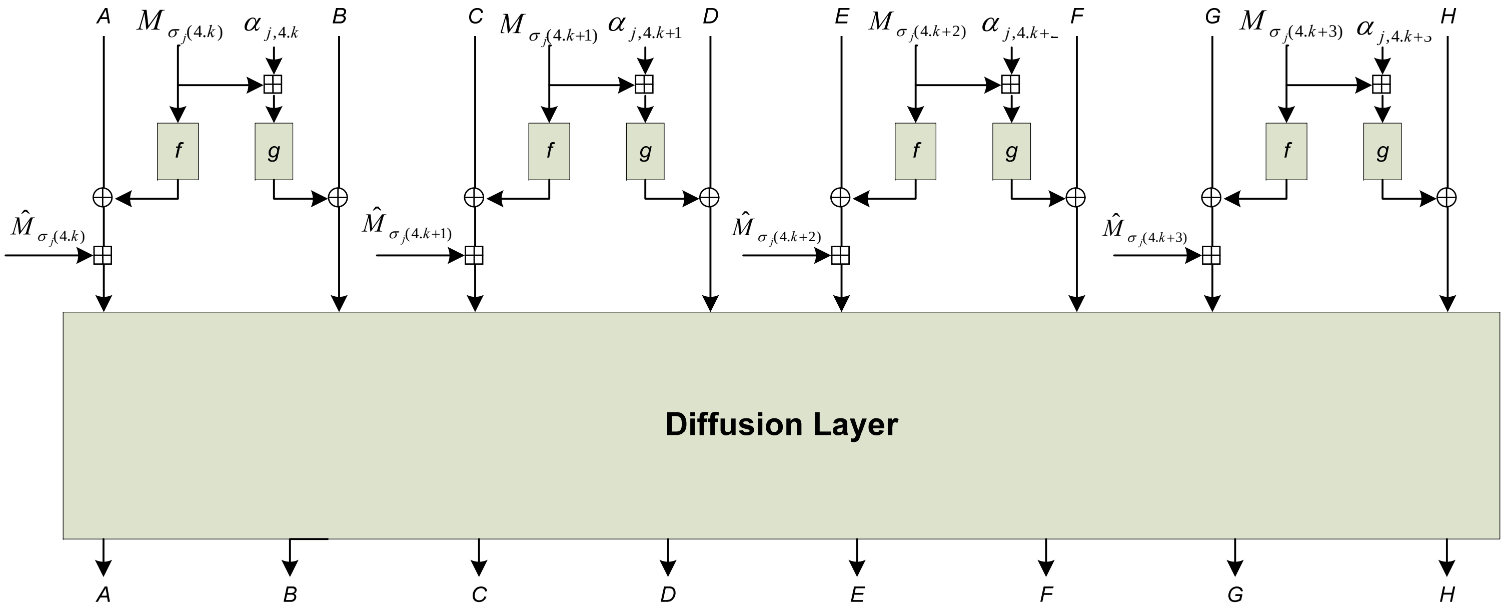

Figure 2. Step function of NESHA-256

3.2.3 Additive constants

The compression function of NESHA 256 uses sixteen additive constants which are given by Table 2.

Table 2. Additive constants in NESHA-256.

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

The constant values of Table 2 are utilized in each branch as additive constants for the compression functions, according to the following permutation table (Table 3).

Table 3. Permutation table for using additive constants.

Step No.	$(\alpha_{1,k})$	$(\alpha_{2,k})$	$(\alpha_{3,k})$	$(\alpha_{4,k})$
0	$\delta_0,\delta_1,\delta_2,\delta_3$	$\delta_4,\delta_5,\delta_6,\delta_7$	$\delta_8,\delta_9,\delta_{10},\delta_{11}$	$\delta_{12},\delta_{13},\delta_{14},\delta_{15}$
1	$\delta_4,\delta_5,\delta_6,\delta_7$	$\delta_8,\delta_9,\delta_{10},\delta_{11}$	$\delta_{12},\delta_{13},\delta_{14},\delta_{15}$	$\delta_0,\delta_1,\delta_2,\delta_3$
2	$\delta_8,\delta_9,\delta_{10},\delta_{11}$	$\delta_{12},\delta_{13},\delta_{14},\delta_{15}$	$\delta_0,\delta_1,\delta_2,\delta_3$	$\delta_4,\delta_5,\delta_6,\delta_7$
3	$\delta_{12},\delta_{13},\delta_{14},\delta_{15}$	$\delta_0,\delta_1,\delta_2,\delta_3$	$\delta_4,\delta_5,\delta_6,\delta_7$	$\delta_8,\delta_9,\delta_{10},\delta_{11}$

3.2.4 Diffusion layer

The diffusion layer of NESHA-256 consists of a three layer Pseudo Hadamard Transformation (PHT) which implements on eight 32-bit words, four times in each step function. Each 2-PHT box indicates a function which maps two 32-bit words to other two 32-bit words according to the formula $2\text{-}PHT(b_1, b_2) = (2a_1+a_2, a_1+a_2)$. In which summation operation '+' denotes summation mod 2^{32}. The diffusion layer of NESHA-256 is shown in Fig. 3.

4 Design principles

In this section, we describe the security criteria for designing NESHA-256 and the design process based upon these criteria. The design rationales include all building blocks of NESHA-256 completely.

4.1 The basic structure design philosophy

NESHA-256 consists of four branches with parallel structure. This kind of structure refers to RIPEMD family hash function [1]. In this family, the functions with the same message ordering in each chaining variable words are resistant against collision attacks so, using message words with different permutation causes algorithm to be more secure [5, 25]. Recently designed hash function which uses parallel structure is: FORK-256 [4,5]. In FORK-256, each branch uses message words with different ordering. Some weaknesses are reported against FORK-256 compression function [11, 12, 16, 14].

In NESHA-256, interaction between non-linear part and diffusion layer of each step function causes algorithm to be more resistant against attacks which are based on partitioning two parts of each step. Moreover, this structure strengthens the algorithm against known attacks based on different message-ordering in branch functions.

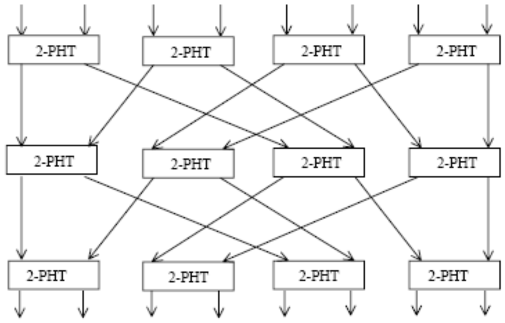

Figure 3. Diffusion Layer

4.2 Selection of Additive Constants

According to the description of NESHA-256 in Section 3, every step function uses four additive constants; so 16 additive constants are entered in each branch with different ordering. Using constant values makes the hashing algorithm more resistant against micro-collision finding attacks. The main criteria of selecting these constants are their independency; therefore, these constants represent the first 32 bits of the fractional parts of the cubic roots of the first 16 prime numbers, which have no interrelationship.

4.3 Inserting of message words

Since NESHA-256 has parallel structure, it should necessarily tolerate simultaneous collision finding attacks in parallel branches. Getting this case, insertion includes of two parts: *re-ordering* of messages for four branches and inserting of *mixed* message in each branch.

In the first case, if an attacker constructs an intended differential characteristic for one branch function, the ordering of message words will cause unintended differential patterns in the other branch functions; thus, finding specific differences for patterns would not be straightforward. There are some important criteria for designing this message permutation such as: balance of upper and lower part, balance of left and right part and balance of sums in input indices [5].

Also four words involving the whole message block are added to each step function prior to entering diffusion layer. In each step function of Branch *j*, four mixed message words ($\widehat{M}_{\sigma_j(4k)}, \widehat{M}_{\sigma_j(4k+1)}, \widehat{M}_{\sigma_j(4k+2)}, \widehat{M}_{\sigma_j(4k+3)}$) are depended on all 16 input message words as mentioned in relation 3. Any changes in any 32-bit message word causes at least one mixed inserted word to be altered so, three out of eight 32-bit positions will be activated due to branch number of the diffusion layer. Subsequently, tracking two different message words which lead to inner collisions are difficult.

Moreover, according to the massage inserting rule passing the same differential pattern through two different branches has become hard. Further details are explained in Section 5.

4.4 Word-oriented functions

Almost all dedicated hash functions use Boolean functions with three or more variables. Therefore, the weaknesses of these bit oriented functions could be exploited by attackers [5]. Some of the most well-known examples of these hash functions are MD4 [20], MD5 [21], HAVAL [27], RIMEMD [1] and, SHA0/1 [17] which are presented in [18, 23].

Also, some new hash functions such as FORK-256 [4,5] uses two nonlinear word-oriented functions, *f* and *g*, which work on a single 32-bit variable. Designers of this algorithm claim that these functions affect all of the chaining variable words during each step so, the attacker cannot divide each step to isolated left and right parts; this point causes resistance to the existing attacks on two branches [16].

NESHA-256 uses two T-function based word-oriented functions which work on a single 32-bit variable word. New applications of T-functions in hash functions [8] have been analyzed in [7,13,15]. According to that, the main weakness of T-function based hash function is reported in case of using truncated T-functions. So, we use T-functions in NESHA-256 in a different way (T-function in NESHA-256 are just 32-bit to 32-bit mapping). As far as we know, there is no any reported attack on this using mode of T-functions.

4.5 The diffusion layer design criteria

The diffusion layer is one of the most important building blocks of our hash function. It plays an important role in elevation of total security in hashing algorithm. Constructing an efficient diffusion layer for a hashing algorithm is important due to software and hardware performance reasons. The diffusion layer in NESHA-256 consists of a three layer Pseudo Hadamard Transformation (PHT) which implements on eight 32-bit words, four times in each step function. PHT has its own advantages according to performance cost. Initially PHT can be characterized by a recursive linear transformation defined by the following relationship [22].

$$H_0 = 1, \quad H_n = \begin{bmatrix} 2.H_{n-1} & H_{n-1} \\ H_{n-1} & H_{n-1} \end{bmatrix} \qquad for \quad n \geq 1$$

Using PHT as a diffusion layer has been observed in SAFER family block ciphers [10]. Applying PHT as the diffusion layer of NESHA-256 is a novel idea. Compared to other hash functions which have parallel structure, using PHT achieves more diffusion. The parameter specifying diffusion value of a transformation is its branch number. Reaching the highest branch number can be gained by using transformation whose matrix is an MDS code generator matrix. Although, this technique increases the security against differential and linear cryptanalysis, it may impose a high cost software implementation. Asymptotically a PHT of dimension $n \times n$ has a branch number of $O(\sqrt{n})$ whereas by comparison an MDS code would have a branch number of exactly $n+1$ [10].

It has been proved that PHT has an efficient implementation for various platforms compared to an equal dimension MDS code [10]. So we prefer to use PHT as NESHA-256 diffusion layer where more efficiency is gained as well as diffusion.

5 Security considerations of NESHA-256

In this section, security analysis of NESHA-256 hash function is given. Besides, statistical evaluations of NESHA-256 are presented. Our analysis on NESHA-256 hashing algorithm concerns its resistance against relevant attacks on hash functions. NESHA-256 algorithm such as FORK-256 hash function has parallel structure. So, its security considerations are like those in FORK hash function and its updated version [4]. Also the security of the algorithm against the most recently attacks [6] on hash functions will be considered. Due to these similarities, we investigate the following considerations:

1- Security analysis for a single branch of NESHA-256 against collision attacks.
2- Security analysis for more than one branch of NESHA-256 against collision attacks.
3- Security considerations for whole structure of NESHA-256 algorithm against attacks, using inner collision patterns.
4- Security considerations for NESHA-256 against pre-image attacks.
5- Security considerations for NESHA-256 against second pre-image attacks.
6- Statistical evaluations of NESHA-256.

5.1 Security analysis for a single branch of NESHA-256 against collision attacks

Collision finding attacks on single branch of NESHA-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 NESHA-256. In this attack, finding compatible IVs together with appropriate massage differences can be led to collision.

Here, we claim that this type of attack is not applicable on one branch of NESHA-256 algorithm. Gaining collision in one branch is possible by finding a non-zero 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. Considering the first branch assume that we select two message words M_0 and M'_0 with nonzero difference ΔM_0, it is easy to find appropriate initial values which cause zero input to the first 2-PHT layer due to forced nonzero differences $\Delta M_{1,}$ ΔM_2 and ΔM_3. Pushing forward the zero outputs to the second step, in the second step all of the message word pairs (M_4 , M'_4), (M_5 , M'_5), (M_6 , M'_6) and (M_7 , M'_7) must have non-zero appropriate differences to maintain zero differences.. Finding proper differences is possible by solving a set of simultaneously semi linear equations of f_1 and f_2 with the complexity of order 2^{128}.

Ordinary collision attack on single branch of NESHA-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 strategies:

1. 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.
2. 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 two steps and the last two steps (with zero value starting) in opposite direction meet each others at the end of second step.

Let us consider the strategies. Suppose that the attacker inserts one or more non-zero-difference message words as input to the first step. Looking at the structure of each step reveals that the entire message words are involved in each step and changing in a message word at the beginning of the step causes at least three inputs in the PHT-layer to be altered. This property compensates for the natural diffusion weaknesses of PHT-layer and makes the attackers decision for altering messages gaining to collision too complex. By the way, in this case all of the message words must be changed to frustrate the effects of assigned non-zero message words differences at the output of the branch. This makes the career of the attacker too hard due to arisen complexity in simultaneously equations. This complexity is not less than what it is in birthday attack due to existence of some good properties of functions f_1 and f_2, according to 3.4.

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 simultaneously equations than the first strategy will grow.

5.2 Security analysis for more than one branch of NESHA-256 against collision attacks

Assume that, the attacker wants to find a collision for NESHA-256 consisting of only two branches: Branch 1 & Branch 2. He must find two output differences Δ_1

and Δ_2 so that Δ_1= -Δ_2. Considering the structure of a branch of NESHA-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^{128} operations of NESHA-256 according to the birthday problem. In [MRD02] a new approach of *k*-dimensional birthday attack on hash functions was proposed. In that approach, *k* (*k*>2) sets of messages (instead of two sets in the usual birthday attack) are generated for a hashing algorithm having a specific property. According to [MRD02], Liskov et al. show that if the hash function can be formulated as *k*-term summation (*e.g.* $H(M)=h(m_1)+ h(m_2)+ \ldots + h(m_k)$) then the complexity of the attack will be reduced to $O(2^{n/3})$. The operation + can be modular summation.

It is notable that neither the compress function *h* nor the hash function of NESHA-256 is in the form of the mentioned structure. So, it is not needed to be concerned for NESHA-256 in this matter. Also, parallel structure of NESHA-256 will prevent the algorithm from being vulnerable against amplified boomerang attack on hash functions proposed by Joux and Peyrin at CRYPTO 2007 [6].

Regarding the above considerations, it is believed that the attacker can not organize any attack applicable to more than one branch.

5.3 Security considerations for whole structure of NESHA-256 algorithm against attacks, using inner collision patterns

Idea of using inner collision patterns for finding collision is the main idea of attacks done on some well known hash functions like SHA0/1. Somehow the above scenarios are using the concept of inner collision pattern. Let us first define the concept of inner collision pattern.

Definition: Any differential characteristic ending to zero difference is called inner collision pattern.

One of the most important criteria for a hash function to be resistant against collision attacks is not to be existent of inner collision patterns with high probability. So, if we find NESHA-256 as a hash function with no high weighted inner collision patterns, our claim about NESHA-256 will be verified. Finding inner collision pattern for one independent step is not difficult for the attacker, but its application for collision finding attack, is not straight forward in this case. Such attacks use inner collision patterns by repeating those patterns which goes back to the second strategy in section 5.1, discussed before. Moreover, assume that someone wants to find inner collision patterns for two or more steps; in this case, the attacker must solve a set of simultaneously equations because all message words in each step are involved. The complexity of solving this set of equations is at least equal to the complexity of birthday attack on NESHA-256 i.e. 2^{128}.

5.4 Security considerations for NESHA-256 against pre-image attacks

Resistance against pre-image attack can be gained by constructing one way structure. There are many methods for building such structures applied in hash functions ranging from basic hash functions to their advanced ones. Using one way functions and one way transformations or mixing them is one of the most common methods in construction hashing algorithm. When it comes to pre-image resistance, we want to compute the difficulties of finding a pre-image of randomly chosen message digest of output set. Since we have a noninvertible function in each step and due to the composition of building blocks in a branch, each branch is not invertible. The complexity of finding pre-image for a randomly chosen output for each step is about 2^{256} NESHA-256 because of involving all message words in each step. Considering the similar case for each branch gives complexity around 2^{1024} which is greater than the complexity of exhaustive search 2^{512}. So, if we see whole algorithm NESHA-256 with its four branches this result is held.

Due to the complexity cost, using meet in the middle technique is also unlikely for pre-image attack on NESHA-256. This is because, if we can bypass the operations after the branches in reverse mode to access to their output, finding their pre-image is not possible due to its complexity. Existing word-oriented balanced functions within the hashing algorithm structure strengthens NESHA-256 against pre-image attack which uses the weakness of some bit-oriented Boolean functions for building the attack scenario [9].

5.5 Security considerations for NESHA-56 against second pre-image attacks

In general, collision resistance property provides second pre-image resistance for a hashing algorithm except for scenarios which are dedicated for finding second pre-image of an output digest. Security considerations against collision attacks have been discussed in subsection 5.1. On the other hand, at the time being, there is no any scenario concerning pre-image attack for such parallel structures like NESHA-256. So there does not exist any second pre-image attack with complexity lower than 2^{256}.

5.6 Statistical evaluations of NESHA-256

In this section, the results of 3 statistical tests based on the distribution of coefficients in ***A****lgebraic* ***N****ormal* ***F****orm of a* ***R****andom* ***B****oolean* ***F****unction* (ANFRBF) are presented. The basic idea is to select a subset of input bits (consisting of key and IV) as variables, denoted $iv_0, \ldots, iv_{n-1}$, while the other values of input are fixed. Any output bit can be considered as a Boolean function of the selected input variables. By running through all possible values of these bits and creating an output keystream of each, the truth table of this Boolean function is determined. By using the truth table, the ANF of this Boolean function can be computed. Now, hope that the distribution of this Boolean function's coefficients is similar to the distribution of coefficients in random Boolean function.

According to this idea, three different tests are proposed by Filiol [3] and Englund et al. [2]. The first one is called the d-Monomial test which proposed by Filiol in 2001. The others are called the monomial distribution test and the maximal degree monomial test which presented in 2007 by Englund et al. The output of NESHA-256 is examined by them. The results are illustrated in Table 4.

Table 4. The results of 3 statistical tests.

Test	**N** (input variables)	**P** (Number of Generated Polynomials)	**α** (Level of significant)	**Result**
d-Monomial Test	14	1	0.05	Pass
Monomial Distribution Test	14	2^6	0.05	Pass
Maximal Degree Monomial Test	14	2^6	0.05	Pass

6 Performance analysis of NESHA-256

In this section, we compare the performance of NESHA-256 in software with those of other of hash functions, SHA-256 and FORK-256. The performance comparison is accomplished using Pentium IV, 2.8 GHz, 512MB RAM/ Microsoft Windows XP Professional v. 2002/ Microsoft Visual C++ Ver. 6.0. Table 5 indicates the results of this software performance testing.

Table 5. Comparison of NESHA-256 performance with the other hash functions, implemented on P4/WinXP/VC.

Algorithm	FORK-256	SHA-256	NESHA-256
Performance (in Mbps)	488.28	393.23	542.53

The software implementation of NESHA-256 in this evaluation is not well-optimized, thus we expect some improvement in performance of any prospective optimized version of this algorithm. However, the simulation results in Table 5 implies that NESHA-256 is about 38% faster than SHA-256 and 11% faster than FORK-256 on a Pentium PC.

7 Conclusions

This paper deals with designing a new dedicated hash function with 256-bit output length called NESHA-256. Our designing scheme has been based on parallel structure and is inspired from some ideas from block-cipher-based hash functions.

The security analysis and performance simulation results indicate that our introduced hash function is not only more secure but also more efficient in software performance in comparison with the standard hashing algorithm, SHA-256 and other functions such as FORK-256 [4].

It is believed that NESHA-256 is secure against currently known attacks on hash functions especially Wang et al.'s attack [24,25,26] and recently proposed attacks

[6,9]. However, the extensive analysis of our new hash function is required. We encourage the readers to give any further analysis on the security of NESHA-256.

References

1. H. Dobbertin, A. Bosselaers and B. Preneel, "RIPEMD-160, a strengthened version of RIPEMD", FSE'96, LNCS 1039, Springer-Heidelberg, pp. 71–82, 1996.
2. H. Englund, T. Johansson, and M. S. Turan, "A Framework for Chosen IV Statistical Analysis of Stream Ciphers", INDOCRYPT'07, LNCS 4859, Springer-Heidelberg, pp. 268–281, 2007.
3. E. Filiol, "A new statistical testing for symmetric ciphers and hash functions", International Conference on Information, Communications and Signal Processing, LNCS 2119, Springer-Heidelberg, pp. 21–35, 2001.
4. D. Hong, D. Chang, J. Sung, S. Lee, S. Hong, J. Lee, D. Moon, and S. Chee, "New FORK-256", 2007. http://eprint.iacr.org/2007/185.
5. D. Hong, J. Sung, S. Lee, and D. Moon, "A new dedicated 256-bit hash function: FORK-256", FSE'06, LNCS 4047, Springer-Heidelberg, pp. 195–209, 2006.
6. A. Joux , T. Peyrin, "Hash Function and the (amplified) Boomerang Attack", CRYPTO'07, LNCS 4622, Springer-Heidelberg, pp. 244-263, 2007.
7. S. Künzli, P. Junod, W. Meier, "Distinguishing Attacks on T-Functions", Mycrypt'05, LNCS 3715, Springer-Heidelberg, pp. 2–15, 2005.
8. A. Klimov, A. Shamir, "New Applications of T-functions in Block Ciphers and Hash Functions", FSE'05, LNCS 3557, Springer-Heidelberg, pp. 18–31, 2005.
9. G. Leurent, " MD5 Is Not One-Way", FSE'08, Springer-Heidelberg, 2008.
10. H. Lipmaa, "On differential Properties of Pseudo-Hadamard Transform and related Mappings", Indocrypt'02, LNCS 2551, Springer-Heidelberg, pp.48-61, 2002.
11. K. Matusiewicz, S. Contini, J. Pieprzyk, "Collisions for Two Branches of FORK-256", Cryptology ePrint Archive 2006/317 (First version), Sep. 2006.
12. K. Matusiewicz, S. Contini, J. Pieprzyk, "Weaknesses of the FORK-256 Compression Function", Cryptology ePrint Archive 2006/317 (Second version), Nov. 2006.
13. H. Molland, T. Helleseth, "A linear weakness in the Klimov-Shamir T-function", ISIT 2005, IEEE International Symposium on Information Theory, pp. 1106 – 1110, 2005.
14. F. Mendel, J. Lano, B. Preneel, "Cryptanalysis of Reduced Variants of the FORK-256 Hash Function", CT-RSA'07, LNCS 4377, Springer-Heidelberg, pp. 85–100, 2007.
15. F. Muller, T. Peyrin, "Cryptanalysis of T-function-Based Hash functions", ICISC'06, LNCS 4296, Springer-Heidelberg, pp. 267-285, 2006.
16. K. Matusiewicz, T. Peyrin, O. Billet, S. Contini, J. Pieprzyk, "Cryptanalysis of FORK-256", FSE'07, LNCS 4593, Springer-Heidelberg, pp. 19-38, 2007.
17. NIST/NSA, "FIPS 180-2: Secure Hash Standard (SHS)", Aug. 2002 (change notice: February 2004).
18. B. Preneel, "Analysis and design of cryptographic hash functions", PhD thesis, Katholieke University Leuven, 1993.
19. J. Pieprzyk and B. Sadeghian, "Design of Hashing Algorithms", LNCS, Springer-Heidelberg, ISBN-10: 0387575006, 1993.
20. R.L. Rivest, "The MD4 Message Digest Algorithm", Crypto'90,LNCS 537, Springer-Heidelberg, pp. 303–311, 1991.
21. R L. Rivest, "The MD5 Message-Digest Algorithm", IETF Request for Comments, April 1992.
22. T. St Denis, "Fast Pseudo-Hadamard Transformations", 2004. www.eprint.iacr.org/2004/010.pdf.

23. B. Van Rompay, “Analysis and design of cryptographic hash functions, MAC algorithms and block ciphers”, PhD thesis, K. U. Leuven, Januvary 2004.

24. X. Wang, H. Yu and Y. L. Yin, “Efficient Collision Search Attacks on SHA-0”, CRYPTO'05, LNCS 3621, Springer-Heidelberg, pp. 1–16, 2005.

25. X. Wang, Y.L. Yin, H. Yu, “Finding Collisions in the Full SHA-1”, CRYPTO'05, LNCS 3621, Springer-Heidelberg, pp. 17–36, 2005.

26. X. Wang , H. Yu, “How to Break MD5 and Other Hash Functions”, EUROCRYPT'05, LNCS 3494, Springer-Heidelberg, pp. 19–35, 2005.

27. Y. Zheng, J. Pieprzyk and J. Seberry, “HAVAL – A One-Way Hashing Algorithm with Variable Length of Output”, AUSCRYPT’92, LNCS 718, Springer-Verlag, pp. 83–104, 1993.

Appendix A: Test vector

```
//INITIALIZATION
CV[0] = 0x6a09e667;CV[1] = 0xbb67ae85;
CV[2] = 0x3c6ef372;CV[3] = 0xa54ff53a;
CV[4] = 0x510e527f;CV[5] = 0x9b05688c;
CV[6] = 0x1f83d9ab;CV[7] = 0x5be0cd19;
//MESSAGE 1
M[0]=0x4105ba8c;M[1]=0xd8423ce8;M[2]=0xac484680;
M[3]=0x07ee1d40;M[4]=0xbc18d07a;M[5]=0x89fc027c;
M[6]=0x5ee37091;M[7]=0xcd1824f0M[8]=0x878de230;
M[9]=0xdbbaf0fc;M[10]=0xda7e4408;M[11]=0xc6c05bc0;
M[12]=0x33065020;M[13]=0x7367cfc5;M[14]=0xf4aa5c78;
M[15]=0xe1cbc780;
//MESSAGE' 1
Mp[0]=0x7cba7f6f;Mp[1]=0x9c28be9c;Mp[2]=0xd3aaf2a8;
Mp[3]=0x08d84931;Mp[4]=0xb43b1548;Mp[5]=0x33cd0b34;
Mp[6]=0xda63fa74;Mp[7]=0x306070d2;Mp[8]=0x40a28ca1;
Mp[9]=0x216dfafc;Mp[10]=0xb68f6ffa;Mp[11]=0x327a75cd;
Mp[12]=0xb6f7f231;Mp[13]=0xbaa8ac4e;Mp[14]=0x5242b031;
Mp[15]=0x4fffc0e4;
//OUTPUT 1
CV[0] = 0x1f44d356;CV[1] = 0x12c1aa81;
CV[2] = 0xd442f582;CV[3] = 0xd50f34b0;
CV[4] = 0xb21ea455;CV[5] = 0x12854997;
CV[6] = 0xcd067ddc;CV[7] = 0x8fdd5ea7;
//MESSAGE 2
memset(M,0,16*4);//M[0~15]=0
//MESSAGE' 1
memset(MP,0,16*4);//MP[0~15]=0
//OUTPUT 2
CV[0]=0xca2dd4f8;CV[1]=0x1a0d6f9d;CV[2]=0x7712c014;
CV[3]=0x0645874a;CV[4]=0x19f8073c;CV[5]=0xf343ab81;
CV[6]=0x30012a91;CV[7]=0x13b069b1;
```