Secure and Fast Hashing Algorithm with Multiple Security Levels

Hassan M. Elkamchouchi
Faculty of Engineering, Alexandria, Egypt
Email: helkamchouchi@yahoo.com

Mohammed Nasr and Roayat Ismail
Faculty of Engineering, Tanta, Egypt
Email: {mnasr, roaayat}@yahoo.com

***Abstract*—We propose a new secure and fast hashing algorithm with multiple security levels (SFHA-MSL). It is based on the generic 3C (3 compressions) construction and the 3C-X (3C XOR) hash function which is the simplest and efficient variant of the generic 3C hash function and it is the simplest modification to the Merkle-Damgard (M-D) iterated construction that one can achieve. The design principle of the proposed algorithm is to have variable output length of 128, 192 and 256 bits, variable number of compression functions, variable number of iterations in each compression function and variable compression function structure. The compression function used in this algorithm is more dynamic in the sense that the input controls what happen in the algorithm. This enable us to achieve a novel design principle: when message is changed, different shift rotations are done which causes more complexity for someone trying to create a collision. Instead of mixing a single word of a message block, four words are mixed per iteration which achieve faster data diffusion and hence better avalanching effect. There is no message expansion in the proposed scheme and it doesn't use Boolean functions but uses only addition, XOR and rotations to achieve its security. This in addition to increasing the algorithm efficiency, it distributes non-linearity among all blocks in a round.**

Index Terms—**hashing algorithm, compression function, iterated hash function, the generic 3C construction, the 3C-X hash function, Merkle-Damgard iterated construction**

I. INTRODUCTION

A hash function accepts a variable-size message *m* as input and produces a fixed-size hash code H(*m*), sometimes called a message digest, as output. The hash code is a function of all the bits of the message and provides an error-detection capability: a change to any bit or bits in the message results in a change to the hash code [1]. Historically, the first designs for hash functions have been based on block ciphers; several successful proposals are still widely in use. A second approach has been the use of modular arithmetic. In order to obtain a better performance, cryptographers started in the late eighties to design efficient custom hash function [2]. The hash function must have three desirable properties:

One-way: for any given code *h*, it is computationally infeasible to find *x* such that H(*x*) = *h*.

Weak collision resistance: for any given block *x*, it is computationally infeasible to find $y \neq x$ with H(*y*) = H(*x*).

Strong collision resistance: it is computationally infeasible to find any pair (*x* , *y*) such that H(*x*) = H(*y*).

Most of hash functions are iterated hash function and most of compression functions are iterated by Merkle-Damgard (M-D) construction with constant IV (initial variable) as shown in Fig. 1.

In this construction the hash function takes an input message and partitions it into $L - 1$ fixed-sized block of *b* bits each. The final block is padded to *b* bits. The final block also includes the value of the total length of the input to the hash function. The inclusion of the length makes the job of the opponent more difficult. Either the opponent must find two message of equal length that hash to the same value or two messages of differing lengths that, together with their length values, hash to the same value.

The hash algorithm involves repeated use of a compression function, f, that takes two inputs (an *n*-bit input from the previous step, called the chaining variable; CV and a *b*-bit block) and produces an *n*-bit output. At the start of hashing, the chaining variable has an initial value (CV_O) that is specified as part of the algorithm. The final value of the chaining variable is the hash value. Usually, $b > n$; hence the term compression. The hash function can be summarized as follows:

$$CV_0 = IV = \text{initial } n\text{-bit value.} \quad (1)$$

$$CV_i = f(CV_{i-1}, Y_{i-1}) \qquad 1 \leq i \leq L. \quad (2)$$

$$H(m) = CV_L . \quad (3)$$

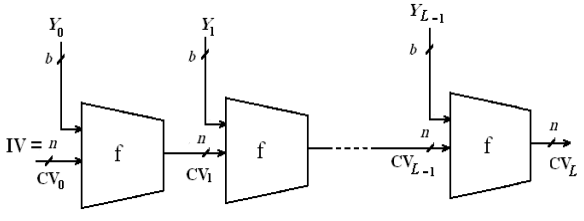

Figure1. Merkle-Damgard iterated construction.

doi:10.4304/jsw.4.9.935-942

where the input to the hash function is a message m consisting of the blocks Y_0, Y_1… Y_{L-1}. Keep in mind that for any hash function there must exist collisions, because we are mapping a message of length at least equal to the block size b into a hash code of length n, where $b > n$ [3]. Cryptanalysis of hash functions proves that the M-D hash construction is not immune to extend attack, fix point attack and multi-blocks attack, moreover, some slight weakness in compression function may results in failure of hash function so some improved constructions such as wide-pipe, double-pipe and the generic 3C (3 compression) constructions are developed. But the generic 3C construction prevents extension attacks in a more efficient way than both the wide-pipe and double-pipe constructions without increasing the size of the internal state [4]. This construction as shown in Fig. 2 consists of two compression functions: f which is iterated in the cascade chain and f ` which is iterated in the accumulation chain. First, the message is processed in an iterated manner over the function f which itself is based on the M-D construction then this output is padded using the standard padding technique (appending Z with a bit 1 and some 0's followed by the binary encoded representation of the length of Z) to make it a block of b bits. This is denoted with ZPAD operation in Fig.2.

Finally the accumulated output of the function f ` is padded then fed as input to the external application f. It should be noted that the padding is performed twice for the 3C construction; once for the basic cascade construction and next on Z to get the accumulation chain block. The construction is called 3C as it requires at least three applications of the compression function to process the message and three being the least when there is only one message block, the first application of the compression function processes that single block, the second application processes the padded block and the third application processes the block due to the accumulation chain function [5]. If the function f ` is replaced with XOR operation we will obtain the 3C-X hash function shown in Fig. 3. The 3C-X hash function is the simplest efficient modification to M-D construction.

In this paper we introduce a new design for a secure one-way hashing algorithm based on both the generic 3C and the 3C-X hash construction with an efficient and dynamic compression function and different levels of security to resist the advanced attacks, such as preimage, second preimage and collision attacks. All simulation programs are performed by using MATHEMATICA.5 and test values are given.

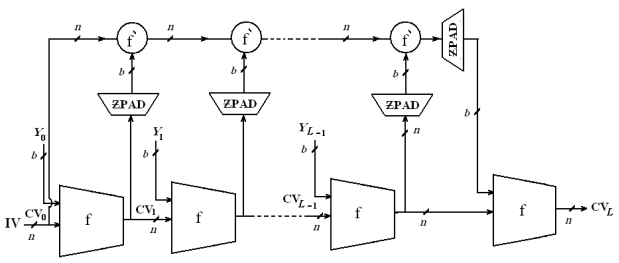

Figure 2. The Generic 3C-hash function construction [5]

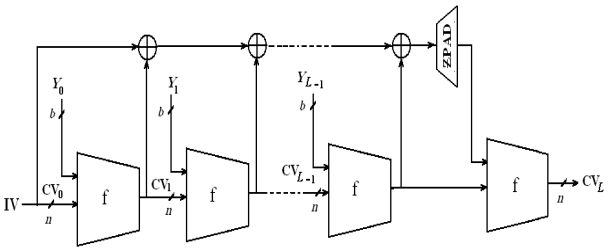

Figure 3. The 3C-X hash function [5]

II. An Overview Of SFHA-MSL

The overall structure of the proposed SFHA-MSL as shown in Fig. 4 consists of the combination of two sections (three compression functions). One of them is the generic 3C hash function section with the compression function f_1 in the cascade chain and the compression function f_1` in the accumulation chain and its output is C_1. The other is 3C-X hash function section with the compression function f_2 in the cascade chain and XOR operations in the accumulation chain and its output is C_2.

The overall processing of SFHA-MSL consists of four steps: set up, pre-processing, iterated processing and output transformation. The set up step is used to select secret parameters (output length, number of compression functions, and number of iterations in each compression function) which are used in the processing step. Both the sender and the verifier use the same secret parameters. Pre-processing step involves padding a message, parsing the padded message and setting initialization values to be used in the iterated processing step. The output transformation is used in a final step to map the n bits to variable lengths. The following operations will be used in the processing and all of these operators act on 32 bit word

+ : addition mod 2^{32}, $\oplus$: XOR, $\wedge$: AND, $\vee$: OR, $A^{<<<s}$: s-bit left shift rotation a 32-bit string.

III. SFHA-MSL Pre-processing

The pre-processing step is used to prepare the message before the SFHA-MSL processing step. It consists of three steps: padding the message, parsing the padded message into message blocks, and setting the initial hash values.

A. Padding the Message

An input message is processed by 512-bit block. SFHA-MSL pads a message 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 448 modulo 512, and then appends to the message the 64-bit original message length modulo 2^{32}.

B. Parsing the Padded Message

Parse the message m into L 512 bit blocks Y_0, Y_1… .....Y_{L-1}. Each of Y_i parsed into 16 32-bit words M_0, M_1… M_{15}. The message blocks are processed one at a

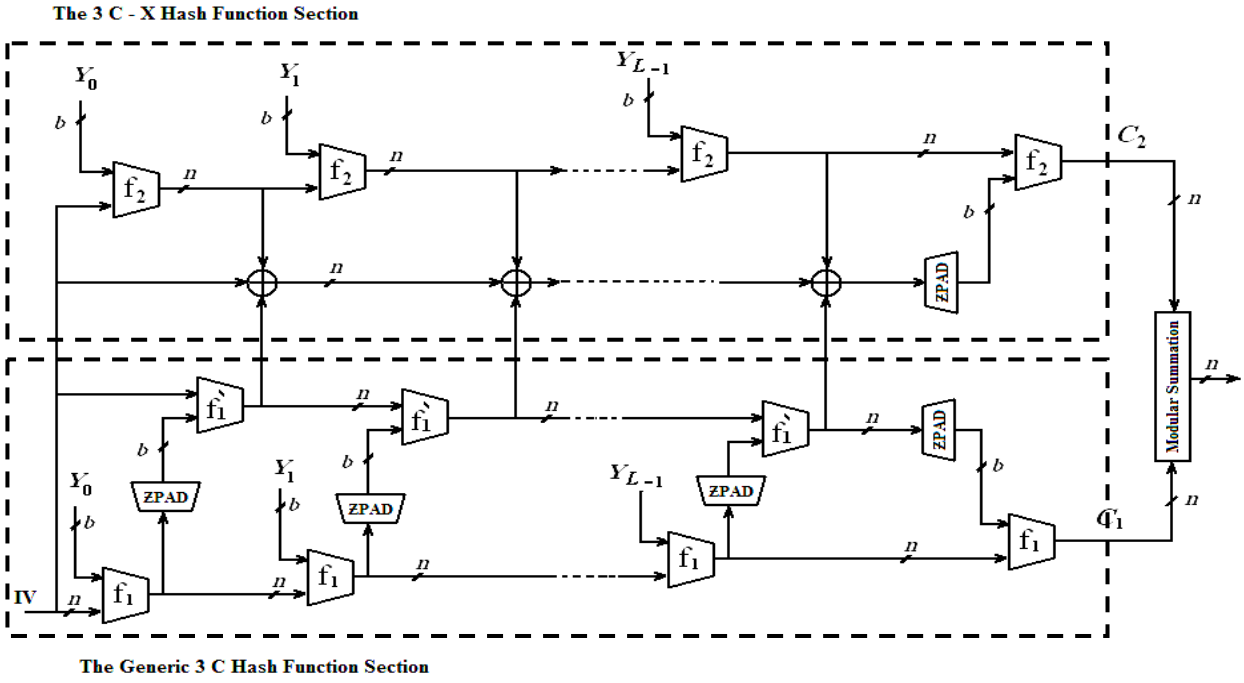

Figure.4 The Overall structure of SFHA – MSL

time, beginning with the initial hash value called the message digest buffer (MDB).

C. Setting the Initial Hash Value

Before hash computation begins for SFHA-MSL, the initial hash values (MDB) must be set. The MDB is 512 bits used to hold intermediate and final results. The MDB can be represented as eight 32-bit words registers A_0, B_0, C_0, D_0, E_0, F_0, G_0, H_0 which are obtained by taking the fractional part of the square roots of the first eight primes in hexagonal values:

A_0 = 6a09e66 B_0 = bb67ae8 C_0 = 3c6ef372,
D_0 = a54ff53a E_0 = 510e527f F_0 = 9b05688c
G_0 = 1f83d9ab H_0 = 5be0cd19

IV. SFHA-MSL Iterated Processing

The iterated processing step depends upon the secret parameters used.

A. Iterated Compression Functions

SFAH-MSL has three compression functions: f_1 and $f_1^{\backslash}$ of the generic 3C hash function section and f_2 of the 3C-X hash function section. Each of f_1 and f_2 consists of two parallel branches; $Branch_1$ and $Branch_2$. So the attacker who tries to break the function should aim simultaneously the two branches. The function $f_1^{\backslash}$ consists of a single branch ($Branch_1$) of the function f_1. Let $CV_i = [A, B, C, D, E, F, G, H]$ be the chaining variable of the compression function. Each successive 512-bit message block M is divided into sixteen 32-bit words M_0, M_1… M_{15} and the following computation is performed to update CV_i to CV_{i+1}.

$$CV_{i+1} = Branch_1(CV_i, M) \oplus Branch_2(CV_i, M) \quad (4)$$

where M is the re-ordering of message words for the two branches as follows:
$Branch_1$: $M = (M_0, M_1 \ldots M_{15})$
$Branch_2$: $M = (M_{15}, M_{14} \ldots M_0)$, where the order is reversed.

B. Constants

The compression functions of SFHA-MSL uses sixteen constants for $Branch_1$ which are ordered as following:

β_0 = 428A2F98 β_1 = 71374491 β_2 = b5c0fbcf
β_3 = e9b5dba5 β_4 = 3956c25b β_5 = 59f111f1
β_6 =923f82a4 β_7 = ab1c5ed5 β_8 = d807aa98
β_9 = 12835b01 β_{10} = 243185be β_{11} = 550c7dc3
β_{12} = 72be5d74 β_{13} = 80deb1fe β_{14} = 9bdc06a7
β_{15} = c19bf174.

For $Branch_2$ the order is reversed.

By using these constants we achieve the goal to disturb the attacker who tries to find good differential characteristics with a relatively high probability. So, we prefer the constants which represent the first thirty-two bits of the fractional parts of the cube roots of the first sixteen prime numbers.

C. Iterated Function

The registers content of the k-th iteration is divided into eight 64-bit words: (A_k , B_k , C_k , D_k , E_k , F_k , G_k , H_k). SFHA-MSL has two different iteration function: the generic 3C and the 3C-X iteration functions.

The generic 3C iterated function:

For the k + 1 iteration the following computations are done (see Figure. 5):

$$A_{k+1} = [H_k + (G_k \oplus M_{(4k+3)\bar{}15})^{<<<s7}] \oplus [(G_k \oplus M_{(4k+3)\bar{}15}) + \beta_{(4k+3)\bar{}15}]^{<<<s8}. \quad (5)$$

$$B_{k+1} = (A_k \oplus M_{(4k)\bar{}15}) + \beta_{(4k)\bar{}15}. \quad (6)$$

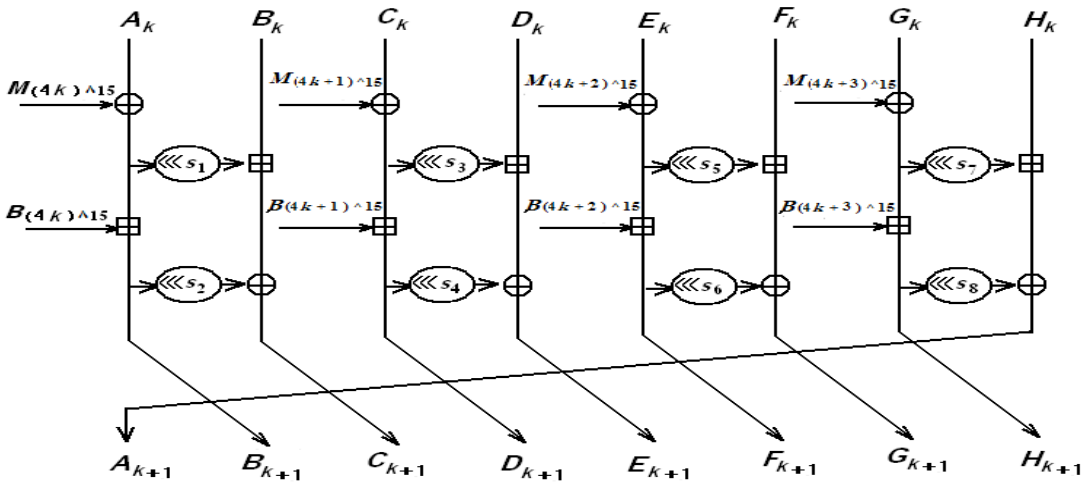

Figure.5 Iteration function of the generic 3C compression function.

$$C_{k+1} = [B_k + (A_k \oplus M_{(4k)^{\wedge}15})^{<<< s1}] \oplus [(A_k \oplus M_{(4k)^{\wedge}15}) + \beta_{(4k)^{\wedge}15}]^{<<< s2}. \quad (7)$$

$$D_{k+1} = (C_k \oplus M_{(4k+1)^{\wedge}15}) + \beta_{(4k+1)^{\wedge}15}. \quad (8)$$

$$E_{k+1} = [D_k + (C_k \oplus M_{(4k+1)^{\wedge}15})^{<<< s3}] \oplus [(C_k \oplus M_{(4k+1)^{\wedge}15}) + \beta_{(4k+1)^{\wedge}15}]^{<<< s4}. \quad (9)$$

$$F_{k+1} = (E_k \oplus M_{(4k+2)^{\wedge}15}) + \beta_{(4k+2)^{\wedge}15}. \quad (10)$$

$$G_{k+1} = [F_k + (E_k \oplus M_{(4k+2)^{\wedge}15})^{<<< s5}] \oplus [(E_k \oplus M_{(4k+2)^{\wedge}15}) + \beta_{(4k+2)^{\wedge}15}]^{<<< s6}. \quad (11)$$

$$H_{k+1} = (G_k \oplus M_{(4k+3)^{\wedge}15}) + \beta_{(4k+3)^{\wedge}15}. \quad (12)$$

where $0 \le k \le (R`/4) - 1$ and $R`$ is a secret parameter used by both the sender and the verifier. $R`$ can be one of three possible values : 16 , 32 , 48.

The 3C-X iteration function:

For the $k + 1$ iteration the following computations are done (see Figure 6):

$$A_{k+1} = (B_k \oplus M_{(4k)^{\wedge}15}) + \beta_{(4k)^{\wedge}15}. \quad (13)$$

$$B_{k+1} = [C_k + (D_k \oplus M_{(4k+1)^{\wedge}15})^{<<< s5}] \oplus [(D_k \oplus M_{(4k+1)^{\wedge}15}) + \beta_{(4k+1)^{\wedge}15}]^{<<< s6}. \quad (14)$$

$$C_{k+1} = (D_k \oplus M_{(4k+1)^{\wedge}15}) + \beta_{(4k+1)^{\wedge}15}. \quad (15)$$

$$D_{k+1} = [E_k + (F_k \oplus M_{(4k+2)^{\wedge}15})^{<<< s3}] \oplus [(F_k \oplus M_{(4k+2)^{\wedge}15}) + \beta_{(4k+2)^{\wedge}15}]^{<<< s4}. \quad (16)$$

$$E_{k+1} = (F_k \oplus M_{(4k+2)^{\wedge}15}) + \beta_{(4k+2)^{\wedge}15}. \quad (17)$$

$$F_{k+1} = [G_k + (H_k \oplus M_{(4k+3)^{\wedge}15})^{<<< s1}] \oplus [(H_k \oplus M_{(4k+3)^{\wedge}15}) + \beta_{(4k+3)^{\wedge}15}]^{<<< s2}. \quad (18)$$

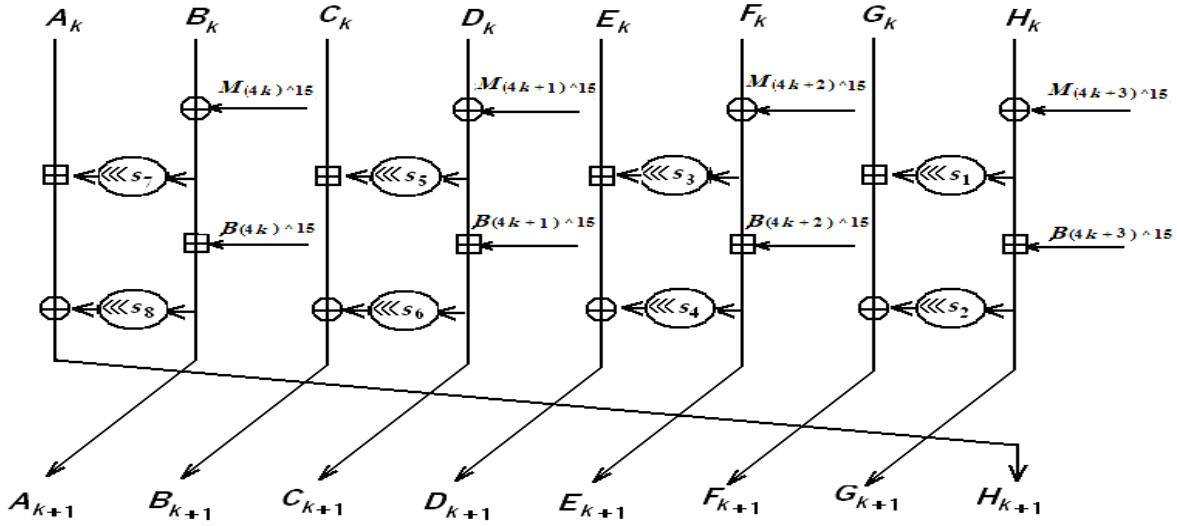

Figure.6 Iteration function of the 3C-X compression function.

$$G_{k+1} = (H_k \oplus M_{(4k+3)\bar{}15}) + \beta_{(4k+3)\bar{}15}. \quad (19)$$

$$H_{k+1} = [A_k + (B_k \oplus M_{(4k)\bar{}15})^{<<< s7}] \oplus [(B_k \oplus M_{(4k)\bar{}15}) + \beta_{(4k)\bar{}15}]^{<<< s8}. \quad (20)$$

where $0 \le k \le (L`/4) - 1$ and $L`$ is a secret parameter used by both the sender and the verifier. $L`$ can be one of three possible values : 16 , 32 , 48.

In these equations k represents the number of iteration of the compression function. Initially for k = 0 the amounts of shift rotations for the generic 3C and the 3C-X iteration function are constant values then for the successive values of k the shift rotations become dynamic as it depends on the input (the content of the registers) as shown in Table I.

For the two iteration functions there is no message expansion part, instead the sixteen words of each message block (M_0, M_1… M_{15}) are mixed more than one time depending on the secret parameters used $R`$ and $L`$ as shown in Table II.

The final output of the generic 3C hash function section (Double Compression Function, DCF) is given by:

$$\text{Hash}^N(C_1) = A^N(C_1), B^N(C_1), C^N(C_1), D^N(C_1), E^N(C_1), F^N(C_1), G^N(C_1), H^N(C_1) \quad (21)$$

The final output of the 3C-X hash function section (Triple Compression Function, TCF) is given by:

$$\text{Hash}^N(C_2) = A^N(C_2), B^N(C_2), C^N(C_2), D^N(C_2), E^N(C_2), F^N(C_2), G^N(C_2), H^N(C_2) \quad (22)$$

where N is the number of blocks in the padded message.

V. SFHA-MSL Output Transformation

The output transformation step is a modular summation used to map the final output of the two output (C_1, C_2) of 256 bit to variable output length (n = 128, 192, 256 bits). This length is also a secret parameter. The final output is given by:

A. SFHA-MSL with DCF Output

Case 128-bit digest:

$$\text{Hash}^N = A^N(C_1), B^N(C_1), C^N(C_1), D^N(C_1). \quad (23)$$

Case 192-bit digest:

$$\text{Hash}^N = A^N(C_1), B^N(C_1), C^N(C_1), D^N(C_1), E^N(C_1), F^N(C_1). \quad (24)$$

Case 256-bit digest:

$$\text{Hash}^N = A^N(C_1), B^N(C_1), C^N(C_1), D^N(C_1), E^N(C_1), F^N(C_1), G^N(C_1), H^N(C_1). \quad (25)$$

B. SFHA-MSL with TCF Output

Case 128-bit digest:

$$\text{Hash}^N = A^N(C_1)+A^N(C_2), B^N(C_1) + B^N(C_2), C^N(C_1) + C^N(C_2), D^N(C_1)+D^N(C_2). \quad (26)$$

TABLE I.
DYNAMIC SHIFT ROTATION FOR THE GENERIC 3C AND THE 3C-X ITERATIONS.

Dynamic shift rotation	Generic 3C iteration		3C-X iteration	
	Initial value	Successive values	Initial value	Successive values
s_1	5	$A_k + B_k + C_k$	15	$H_k + A_k + B_k$
s_2	9	$B_k + C_k + D_k$	19	$G_k + H_k + A_k$
s_3	3	$C_k + D_k + E_k$	13	$F_k + G_k + H_k$
s_4	11	$D_k + E_k + F_k$	20	$E_k + F_k + G_k$
s_5	8	$E_k + F_k + G_k$	18	$D_k + E_k + F_k$
s_6	13	$F_k + G_k + H_k$	23	$C_k + D_k + E_k$
s_7	7	$G_k + H_k + A_k$	27	$B_k + C_k + D_k$
s_8	10	$H_k + A_k + B_k$	19	$A_k + B_k + C_k$

TABLE II.
NUMBER OF ITERATIONS AND WORD MIXING FOR DIFFERENT SECRET PARAMETERS VALUES.

Value of $R`$ and $L`$	No. of iterations	No. of word mixing
16	4	1
32	8	2
48	12	3

Case 192-bit digest:

$$\text{Hash}^N = A^N(C_1)+A^N(C_2), B^N(C_1) + B^N(C_2), C^N(C_1) + C^N(C_2), D^N(C_1)+D^N(C_2) E^N(C_1) + E^N(C_2), F^N(C_1) + F^N(C_2). \quad (27)$$

Case 256-bit digest:

$$\text{Hash}^N = A^N(C_1)+A^N(C_2), B^N(C_1)+B^N(C_2), C^N(C_1) + C^N(C_2), D^N(C_1)+D^N(C_2) E^N(C_1) + E^N(C_2), F^N(C_1) + F^N(C_2), G^N(C_1) + G^N(C_2), H^N(C_1) + H^N(C_2). \quad (28)$$

VI. SFHA-MSL Test Results

Two tests were run on both SFHA-MSL and SHA-256 for comparison between the collision resistance and uniformity of each.

A. Collision and Avalanche Tests

To test for collisions and avalanching effect of the algorithms two different sets of data were run through both SFHA-MSL and SHA-256. Each set of data consists of a single block of data (512 bit) without padding and has a single bit of change from input to input.

The First set of data:

Its basic input is a file of 64 byte (512 = 64 byte × 8 bit) consists of alternate letters through all the letters value: "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijkl", in binary:

<u>01100001</u>01100010011000110110010001100101011001
10011001110110100001101001011010100110101101100
11000110110101101110011011110111000001110001010
11001001110011011101000111101011101100111011100
11110000111100101111010011000010110001001100010
10110010001100101<u>01100110</u>01100111011010000110100
00101101010011010110111011000110110101101110011
01111011100000111000101110010011100110111010000
11101010111011001110111011110000111100101111010
00110000101100010011000110110010001100101011001

11001100111011010000110100101101010011010110110100".

We applied 24 successive inputs (with a single bit changed from input to input) to the two algorithms divided as following:

Eight inputs (from $input_1$ to $input_8$): With a single bit changed from input to input moving forward from the most significant bit to the least significant bit of the underlined most significant byte, 01100001.

Eight inputs (from $input_9$ to $input_{16}$): With a single bit changed from input to input moving forward from the most significant bit to the least significant bit of the underlined middle (32^{nd}) byte,01100110.

Eight inputs (from $input_{17}$ to $input_{24}$) : With a single bit changed from input to input moving forward from the most significant bit to the least significant bit of the underlined least significant byte, 01101100.

The Second set of data:

Its basic input is a file consists of 64 letter all are the same letter, "A", in binary, 01000001, and we repeat the same steps as in the first set of data**.**

The results of comparing the hashed values of the successive inputs for the two sets of data are listed in Table III. These hashed values were first checked for collisions, none were found. Next the bits of the hash were compared from hash to successive hash, in position, checking to see how many bits were different due to a single bit change in the input, and then the average is taken.

In a perfect avalanche: a single bit change from input to input causes half of the bits (128 bit) change from hash to hash [5]. So the optimum number of differing bits = 128. Looking at the results in Table III we conclude that SFHA-MSL with both DCF and TCF has a larger average number of differing bits for the two sets of data and so it has a better avalanching effect than SHA-256.

B. Uniform Distribution Tests

This test was used to test the distribution of hashes into the possible hash values. SFHA-MSL with an output of 256 bit has 2^{256} possible hash value ranges from 0 to 2^{256}-1. For a perfect uniform distribution for this output, the average value of the hash values equals $[(2^{256}-1)/2] \cong 5.7896\times10^{76}$. Practically, it is impossible to obtain a perfect uniform distribution hash algorithm. Instead, the less the deviation from the uniform distribution the more secure the algorithm is. Three sets of messages were used for these tests: the first, middle, and last ten possible input values for both SFHA-MSL and SHA-256. These ranges are: 0x00….00 to 0x00…...09, 0x10…..00 to 0x10…..09 and 0xff….06 to 0xff…..ff. Then the average hash value for each of these sets is computed and compared with that of the uniform distribution then the deviation is computed for the two algorithms**.** In Table IV we give the results of the comparison between SFHA-MSL and SHA-256.

The deviation from optimal uniform distribution
= [(uniform distribution average value – actual average value)/ uniform distribution average value]**.**

From these results, SFHA-MSL does better than SHA-256 for all three sets of data. The most important data

TABLE III
NUMBER OF DIFFERING BITS BETWEEN SUCCESSIVE HASHES

The two compared hashes	Number of differing bits					
	The first set of data			The second set of data		
	SFHA-MSL		SHA 256	SFHA-MSL		SHA 256
	DCF	TCF		DCF	TCF	
$Hash_1$ and $Hash_2$	141	126	119	127	134	121
$Hash_2$ and $Hash_3$	130	131	122	120	128	139
$Hash_3$ and $Hash_4$	143	127	128	129	116	135
$Hash_4$ and $Hash_5$	143	112	121	142	127	135
$Hash_5$ and $Hash_6$	138	140	137	126	125	131
$Hash_6$ and $Hash_7$	129	132	132	126	124	120
$Hash_7$ and $Hash_8$	115	128	134	138	133	131
$Hash_8$ and $Hash_9$	130	134	130	129	132	127
$Hash_9$ and $Hash_{10}$	128	128	129	134	136	129
$Hash_{10}$ and $Hash_{11}$	130	130	122	135	127	119
$Hash_{11}$ and $Hash_{12}$	118	127	124	135	139	120
$Hash_{12}$ and $Hash_{13}$	128	113	118	137	134	130
$Hash_{13}$ and $Hash_{14}$	131	119	124	141	132	121
$Hash_{14}$ and $Hash_{15}$	125	116	128	131	132	135
$Hash_{15}$ and $Hash_{16}$	136	131	121	118	119	132
$Hash_{16}$ and $Hash_{17}$	125	146	127	135	115	117
$Hash_{17}$ and $Hash_{18}$	116	133	129	127	131	100
$Hash_{18}$ and $Hash_{19}$	128	127	133	128	127	132
$Hash_{19}$ and $Hash_{20}$	138	128	131	138	134	121
$Hash_{20}$ and $Hash_{21}$	122	145	128	124	128	121
$Hash_{21}$ and $Hash_{22}$	132	127	124	135	139	123
$Hash_{22}$ and $Hash_{23}$	126	128	129	119	121	125
$Hash_{23}$ and $Hash_{24}$	123	132	125	129	135	113
Average value	129.348	128.696	126.739	130.565	129.043	125.087

TABLE IV
DEVIATION FROM OPTIMAL DISTRIBUTION

The compared values	The first ten hash values			The middle ten hash values			The last ten hash values		
	SFHA-MSL		SHA	SFHA-MSL		SHA	SFHA-MSL		SHA
	DCF	TCF	256	DCF	TCF	256	DCF	TCF	256
Actual average hash value ($\times 10^{76}$)	5.6899	5.4807	7.2954	5.7057	6.5712	7.5985	5.3428	4.6612	7.0252
Deviation from optimal	0.0172	0.0533	0.2600	0.0145	0.1349	0.3124	0.0772	0.1948	0.2134

point being the deviation from the optimal uniform distribution. SFHA-MSL deviation is smaller than SHA-256. This means that SFHA-MSL has a more uniformly distributed output than SHA-256 and so it is more secure than it.

VII. SFHA-MSL TIME COMPLEXITY

In this section we give a complexity analysis on the operations used in SFHA-MSL with DCF and TCF and value of $R`$ and $L`$ =16 then compare it with SHA-256 to indicate the efficiency of SFHA-MSL. This comparison is indicated in Table V.

A. SFHA-MSL with DCF

As we mentioned previously the generic 3C hash function section consists of two compression functions are f_1 and $f_1^{\backslash}$. The function f_1 consists of two XORed branches, while $f_1^{\backslash}$ consists of a single branch. So the overall compression function consists of three branches.

Each branch has a step function that is iterated at least four times (k = 4). Each step function contains 20 addition, 8 XOR and 8 shift rotation operation (see Figure 5).

So the overall compression function contains:
$(20 \times 4 \times 3) = 240$ addition,
$(8 \times 4 \times 3 + 1) = 97$ XOR
and $(8 \times 4 \times 3) = 96$ shift rotation operation.
But the generic 3C construction has an extra stage of f_1 that contains:
$(20 \times 4 \times 2) = 160$ addition,
$(8 \times 4 \times 2 + 1) = 65$ XOR
and $(8 \times 4 \times 2) = 64$ shift rotation operation.
So SFHA-MSL with DCF contains:
$(240 + 160) = 400$ addition,
$(97 + 65) = 162$ XOR
and $(96 + 64) = 160$ shift rotation operation.

B. SFHA-MSL with TCF

With the addition of the 3C-X hash function to the generic 3C hash function we obtain a TCF (f_1, $f_1^{\backslash}$ and f_2). The compression function f_2 consists of two XORed branches and an extra stage.

So the TCF contains:
$[400 + (20 \times 4 \times 2) \times 2] = 720$ addition,
$[162 + (8 \times 4 \times 2 + 1) \times 2] = 292$ XOR
and $[160 + (8 \times 4 \times 2) \times 2] = 288$ shift rotation operation.
These results show that SFHA-MSL is an efficient algorithm compared with SHA-256.

VIII. SFHA-MSL SIMULATED RESULTS

For a simple message M = "abc", the 8-bit ASCII message "abc" has length l = 24 bits and it is given by "01100001 01100010 01100011". It is padded with a one "1", so 24 + 1+ k = 448 mod 512 and then k = 423 zero bits. Append the 64-bit block that is equal to the number l written in binary "0000……011000" and then its length are 512-bits padded message.

The "abc" message has a single (N =1) 512 bit block. Parse the 512 bits into 16 32-bit words M_0, M_1… M_{15} (in hexadecimal).

61626380 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000018

These 16 32-bit words are mixed more than one time depending on the secret parameter used with no message expansion.

The SFHA-MSL with different cases will be represented by SFHA-MSL\$N$\$R`$\$L`$\$K$, where N, $R`$, $L`$ and K are secret dynamic parameters used by the sender and verifier.

N (2 or 3): the number of compression functions (DCF or TCF).
$R`$ (16 or 32 or 48): the number of iteration of the generic 3C compression function.
$L`$ (16 or 32 or 48): the number of iteration of the 3C-X compression function.
K (128 or 192 or 256): the length of the output hash

SFHA-MSL\3\16\32\256; it means that the SFHA-MSL has "3" compression functions (TCF), number of iterations in the generic 3C compression functions are "16" and "32" in the 3C-X compression function and the output length is "256" bits. The simulated test vectors for

TABLE V
COMPARISON BETWEEN NUMBER OF OPERATIONS OF SFHA-MSL AND SHA-256

Operation	SFHA-MSL with DCF	SFHA-MSL with TCF	SHA-256
Addition (+)	400	720	600
Bitwise Operation ($\oplus,\wedge,\vee$)	162	292	1024
Shift (<< , >>)	-	-	96
Shift rotation (<<< , >>>)	160	288	576
Total	722	1300	2296

TABLE VI
SFHA-MSL TEST VECTOR FOR DIFFERENT CASES WITH THE STRING "abc"

SFHA-MSL Version	Hash value (as hex byte string)
SFHA-MSL\2\16\256	D703AF16 0B3F3407 7E82FC19 C0FBF257 A0976776 D3E6D71F 3C5AE9B8 3185A7DB
SFHA-MSL\2\16\192	D703AF16 0B3F3407 7E82FC19 C0FBF257 A0976776 D3E6D71F
SFHA-MSL\2\16\128	D703AF16 0B3F3407 7E82FC19 C0FBF257
SFHA-MSL\3\16\16\256	FE520E80 CB006687 B3DD1F9D 7FE37276 B8FBC27C CB697A52 DABB761D 2098FD46
SFHA-MSL\3\16\16\192	FE520E80 CB006687 B3DD1F9D 7FE37276 B8FBC27C CB697A52
SFHA-MSL\3\16\16\128	FE520E80 CB006687 B3DD1F9D 7FE37276

the SFHA-MSL are given in Table VI.

X. CONCLUSIONS

This paper presents a new secure and efficient hashing algorithm with different security levels called SFHA-MSL.

It is based on the generic 3C construction and the 3C-X hash function which is the simplest and efficient variant of the generic 3C hash function and it is the simplest modification to the M-D iterated construction that one can achieve.

All famous secure hash algorithms (SHA) given by the National Institute of Standards and Technology (NIST) have fixed structure and fixed output length but SFHA-MSL has dynamic structure and variable output length so it can provide many choices for practical applications with different levels of security to resist the advanced attacks such as preimage, second preimage and collision attacks.

SFHA-MSL tests showed that it has a better avalanching effect and a more uniform distribution than SHA-256.

The time complexity of SFHA-MSL is compared with that of the SHA-256 and it gives excellent results.

REFERENCES

[1] A. J. Menzes, P.C. Oroschot , and S.A. Vanstone, "Handbook of Applied Cryptography", Boca Raton, Florida: *CRC Press LLC*, 1997.

[2] A. Bosselaers, H. Dobbertin, B. Preneel, "The Cryptographic Hash Function RIPEMD-160", *CryptoBytes, RSA laboratories*, 1997.

[3] William Stallings, "Cryptography and Network Security: Practice", *Prentice-Hall Inc*, 2003.

[4] Duo Lei, Li Chao, Chen Song, "The Design Principle of Iterated Hash Function Satisfying Failure Tolerant Principle", Department of Science National University of Defense Technology, Changsha, China, 2005.

[5] P. Gauranvaram, W. Millan, J.G. Neito, E. Dawson, "3C-Aprovably Secure Pseudorandom Function and Message Authentication Code", Information Security Institute (ISI), QUT, Australia, 2006.