A Novel Secured Cryptographic Hash (NSCH) Algorithm

M. Thangavel*, P. Varalakshmi**,
Teaching fellow*, Assistant Professor (Sr.Grade)**,
Department of Information Technology,
MIT Campus, Anna University, Chennai.

M. Kuthalingam,
PG Scholar,
Department of Information Technology,
MIT Campus, Anna University, Chennai.

***Abstract*— Cryptographic hash algorithms are utilized to create a message digest of unique fixed length bit string for the arbitrarily length of the data. Hash functions are considered to be tools, which are used in digital time stamping, digital signatures, and assuring the integrity of the messages. The data that has been outsourced need to be periodically verified that whether the data has been modified by anonymous user or not. In this paper, a novel idea has been proposed for creating a unique message digest of any message by addressing the data integrity problem. A comparison has been made with the existing cryptographic hashing tool MD5, modified approach on MD5 [1] and our proposed efficient algorithm. In the NSCH algorithm, the randomness of the message digest for the input message has been increased which has been proved and justified in the experimental results.**

Keywords—Cryptography; Hash; MD5; message digest; digital signature

I. INTRODUCTION

Computer security refers to the techniques that have been used to secure data from the theft or misuse and also to enable privacy of the user's data. In cloud computing, users store their data in multiple locations in order to increase the data availability but it is not easy to identify whether the data remain unchanged as it is originally stored. It is difficult to check the arbitrary length of data. The hash function is the efficient algorithm which maps an arbitrary length of the data to the fixed length of the data, to check whether the data integrity is maintained or not.

A digital signature is an interesting mathematical scheme used to demonstrate the authenticity of a digital message or document, using cryptographic hashing algorithms like MD5, SHA-1, and RIPEMD. A cryptographic hash function is a form of hash function which operates on an arbitrary block of data and returns the unique fixed length bit string. This fixed length bit string is called the cryptography hash value of the data. Any change in the original data will reflect on the hash value of the respective data. Using this property, user can easily identify and check whether the data has been modified or not. The applications of hash value include E-mail security, cyclic redundancy Checksum for files on a network etc.

The Cryptographic hash functions are computationally expensive compared with standard hash functions, so it should be used only, when the application required this hash function. A hash function h should have minimum two properties, (i) compression- h maps input of arbiters length 'x' and output h (x) of fixed length string n, (ii) easy computation – given hash function h and input x, h (x) can be computed easily. There are several methods to create a cryptographic hash function and one of the important method is one-way compression function. The cryptography hash function utilizes the concept of block cipher modes of operation but the resulting bit string should not be invert able. Many popular hash functions like SHA-1, SHA-2, MD-4, MD-5 are using the concept of block cipher modes of operation. In the proposed scheme, the concept of block cipher modes of operation has been used to create the hash value of the string.

The hash functions are classified into two ways. One is keyed hash functions, which includes two distinct inputs a message and a secret key. And another one is unkeyed hash function, which have categories of hash functions based on modular arithmetic, block ciphers, and customized hash function. The cryptography hash function need to satisfy the following 4 main properties, (i) impossible to generate original message from the hash value (Preimage Property) (i.e) it is a one way function, (ii) impossible to have a message x and to find a message x', such that both messages hash to a same message digest (Second Preimage Property). (iii) impossible for two different strings have the same hash value (Collision property) and (iv) impossible to modify the message with modifying the hash value. The input message can be a string of any variable length which contains alphabets with upper and lower case, numbers and special symbols. The message can be represented in the form as in Equation 1.

Message =
`[a-zA-Z0-9!@#$ %^&*()_+\-=\[\]{};':"\\|,.<> \/?]`$^{+}$ (1)

The digest should be a string of fixed length with alphabets, numbers and special characters. This can be achieved by blocking the arbiter length string into blocks of string as in Equation 2.

Message digest=
`[a-zA-Z0-9!@#$ %^&*()_+\-=\[\]{};':"\\|,.<> \/?]`^{m} (2)
where m is fixed length of digest value.

MD5 Hashing algorithm [5] is a message digest authentication algorithm developed by RSA, Inc. It is a modified version of the MD4 algorithm [4]. The hashing algorithm computes a digest of the arbitrary length of the data. MD5 requires that both the sender and receiver compute the digest of the entire message.

MD5 is a block-chained digest algorithm which computes the message digest over the input data in phases of 512-byte blocks organized as little-endian 32-bit words (Figure 1.1). In the figure, the first part is processed with an initial seed value and results with a digest that becomes the seed value for the next part. When the last part is computed, its digest value is the digest value for the entire input stream. This chained seeding prohibits parallel processing of the blocks.

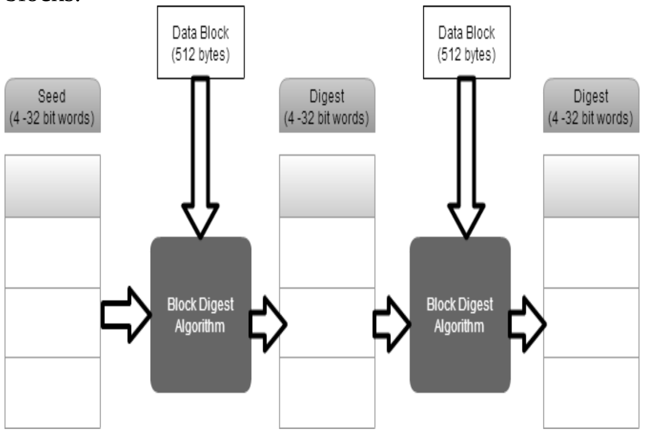

Figure 1.1 Flow of MD-5 hashing algorithm

Mendel et.al [2] research work has listed and proven out the various collision attacks [3, 6] that can be done in MD5 hashing algorithms.

While working on the privacy of the data in cloud, cryptographic hash algorithms are used for generating fixed length of unique block tag for a given file. So, we proposed a secured block-chained digest algorithm by satisfying the properties of the cryptographic hash functions and to provide a unique message digest for a message.

II. Novel Secured Cryptographic Hashing (NSCH) Algorithm

The proposed (NSCH) algorithm has been represented as flowchart in figure 2.1.

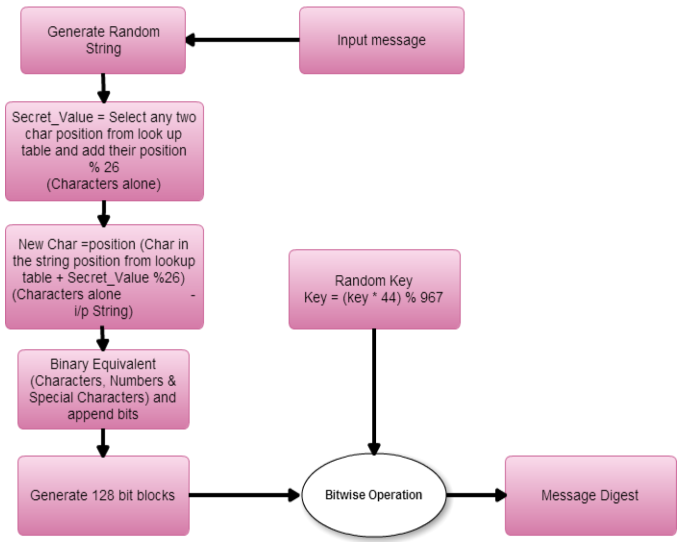
Generate Random String
Input message
Secret_Value = Select any two char position from look up table and add their position % 26 (Characters alone)
New Char =position (Char in the string position from lookup table + Secret_Value %26) (Characters alone - i/p String)
Random Key
Key = (key * 44) % 967
Binary Equivalent (Characters, Numbers & Special Characters) and append bits
Generate 128 bit blocks
Bitwise Operation
Message Digest

Figure 2.1 Flow of NSCH algorithm

The proposed (NSCH) algorithm (Figure 2.1) has been described as follows:
Input: A message with arbiter length.
Output: A hash value of 128 bit for the input message

The algorithm involves the following steps,

Step1:

Enter the input string.
Using the general form, generate new random string:

((a-z) + (A-Z) + (0-9) +
(!@#$ %^&*()_+\-=\[\]{};':"\\|,.<> \/?) +
(Sunday-Saturday) + (0-9))

where '+' represents any position in the random string.

Step 2:

Select and count the number of alphabets from the random string.

Step 3:
Find the Secret_value by assuming the following conditions, to improve the randomness.

a. If the alphabet count =10 then
Secret_value = (ASCII value for the UPPERCASE of the position of first character) +1.

b. If the alphabet count > 10 and < 20 then
Secret_value = (ASCII value for the UPPERCASE of the position of first character – ASCII value of the character 'A') + (ASCII value for the UPPERCASE of the position of second character – ASCII value of the character 'A') +2.

c. If the alphabet count >= 20 and <30 then
Secret_value = (ASCII value for the UPPERCASE of the position of first character – ASCII value of the character 'A') + (ASCII value for the UPPERCASE of the position of third character – ASCII value of the character 'A') +2.

d. If the alphabet count >= 30 then
Secret_value = (ASCII value for the UPPERCASE of the position of second character – ASCII value of the character 'A') + (ASCII value for the UPPERCASE of the position of fourth character – ASCII value of the character 'A') +2.

where value of alphabetical count varies based on the application or message.

Step 4:
Consider a lookup table as given in the Table 2.1 for finding the alphabet position,

Table 2.1 Lookup table - Alphabets Position

A	B	C	D	E	F	G	H	I	J	K	L	M	N
0	1	2	3	4	5	6	7	8	9	10	11	12	13

O	P	Q	R	S	T	U	V	W	X	Y	Z
14	15	16	17	18	19	20	21	22	23	24	25

Step 4 (Cont…)

a. Select each character from the input string.
b. New position = find the position from the lookup table + Secret_value % 26
c. New alphabet = new position equaling value from the lookup table.

Step 5:

a. Find the binary string value for each character using ASCII value of the character.

b. Find the binary string value for the numbers and special characters using the ASCII value.

Step 6:

Based on the sequence of the random string,

a. Convert the string length as multiple of 128 bits and append 0's if it not.

b. Append 64 more bits by scanning the binary string of previous step starting from the location (length of the string /3)

Step 7:

a. Divide the output string of previous step in 128 bit blocks

b. Generate 128 bit binary key using random key generator

c. Perform 128 bit block string with 128 bit random key string any of the bitwise operations like AND, OR,XOR, Left shift, Right shift,etc.

d. Store the digest output.

Step 8:

a. Perform a bitwise operation among the current message digest string and previous message digest string

b. Perform the step 7 until the input message are exhausted

Step 9:

Convert the output of previous step into corresponding character value and store it as the final 128 bit block.

Step 10:

The final 128 bit block message and 128 bit random key are combined using bitwise operation to produce the final message digest value.

A. Random key generation

A key is generated with 128 bit length. The key can be generated using following equation,

$$\text{Key} = (\text{key}*44) \ \% \ 967 \qquad (2.1)$$

$$\text{Keyf} = (\text{Key})! \qquad (2.2)$$

where binary equivalent of Keyf is calculated and any 128 bit taken as key for a step.

The random key is generated by using recurring function. In each and every step one block of message string and a random key proceed to give message digest. The process involves with OR, AND, XOR and any shift wise operations. The stepwise message digest process performs the previous step wise message digest values and form the final message digest form. The output string should be constant length from the any arbiter length input string.

B. Example for NSCH Algorithm

Step 1:

Let the input string be "hash123?Function"

Step 2:

The random string will be generated using the function,

((a-z) + (A-Z) + (0-9) +
(!@#$ %^&*()_+\-=\[\]{};':"\\|,.<> \/?) +
(Sunday-Saturday) + (0-9)).

For the above given example, the random string be,

WEDNESDAYhash123?Function2
WEDNESDAY3327163049264

And select only alphabet =

WEDNESDAYhashFunctionWEDNESDAY

And the length of selected alphabets length is 30

Step 3:

For the given input string, character positions are 2 and 4, since alphabet count is 30.

Secret_value = ((65-65)+(72-65))+2 = 9

Step 4:

The new alphabet will be generated for the random string,

FNMWNBMJHQJBQFODWLCRXWFNMWNBMJH

'W' will be converted to 'F' as follows:

NewPosition=('W' position - lookup table+Secret_value)%26.
= (22 + 9)%26 = 5.

New Alphabet = F.

Step 5:

Take the ASCII value of each characters and special symbols and numbers.

Table 2.2 – ASCII Value of the New String

```
01000110010011100100110101010111010011100100001O
```

Step 6:
The bit sequence is appended with input binary string and the resulting string is 64 shorter than a multiple of 512.

Table 2.3 – ASCII Value with appended bits

```
010001100100111001001101010101110100111001000010
010011010100101001001000010100010100101001000010
010100010011000000110000001100110011000100110000
001100000011001100110010001100000011000000110011
001100110011000000110000001100110100011001001111
010001000101011101001100010000110101001001011000
010101110011000000110000001100110011001001000110
010011100100110101010111010011100100001001001101
0100101001001000000110000001100000011001100110011
```

Step 7:
The new 64 bits are appended to input binary string starting from (length of string /3)th position. So the binary string length became multiple of 512 bits, so we can divide the blocks of 128 bits.

Table 2.4 – 128 bit blocks

Block 1:
```
0100011001001110010011010101011101001110010000010
0100110101001010010010000101000101001010010000010
01010001001100000011000000110011
```
Block 2:
```
0011000100110000001100000011001100110010001100000
0011000000110011001100110011000000110000001100111
01000110010011110100010001010111
```
Block 3:
```
0100110001000011010100100101100001010111001100000
0011000000110011001100100100011001001110010011011
01010111010011100100001001001101
```
Block 4:
```
0100101001001000000110000001100000011001100110011
0011000000110000001100110011001100110000000110000
00110011001100100011000000110000
```
Block 5:
```
0011001100110111001100000011000000110011001100010
0011000000110000001100110011011000110000000110000
00110011001100110011000000110000
```
Block 6:
```
0011001100110000001100000011000000110011001101000
0011000000110000001100110011100100110000000110000
00110011001100100011000000110000
```
Block 7:
```
0011001100110110001100000011000000110011001101000
0000000000000000000000000000000000000000000000000
00000000000000000000000000000000
```

Step 8:
Bitwise operation is not done here, since only one 128 bits is available.

Table 2.5 – Bitwise XOR with 128 bit blocks

R1:Block1 ⊕ Block2
```
011101110111111001111101011001000111110001110010
011111010111100101111011011000010111101001110001
00010111011111110111010001100100
```

R2: R1 ⊕ Block3
```
001110110011110100101111001111000010101101000010
010011010100101001001001001001110011010000111100
01000000001100010011011000101001
```

R3:R2 ⊕ Block4
```
011100010111010100011111000011000001100001110001
0111110101111010011110100001010000000010000001100
01110011000000110000011000011001
```

R4:R3 ⊕ Block5
```
010000100100001000101111001111000010101101000000
010011010100101001001001001000100011010000111100
01000000001100000011011000101001
```

R5:R4 ⊕ Block6
```
011100010111001000011111000011000001100001110100
0111110101111010011110100001101100000010000001100
01110011000000100000011000011001
```

R6:R5 ⊕ Block7
```
010000100100010000101111001111000010101101000000
0111110101111010011110100001101100000010000001100
01110011000000100000011000011001
```

where R1,R2,…R5 are intermediate results and R6 is the final 128 bit message.

Step 9:
Let the random key be Key=31054905. Repeat the key 2 times to get 128 bits.

Table 2.5 – Bitwise XOR with message and key

Key : 3105490531054905

Key (in binary):
```
0011001100110001001100000011010100110100001110001
0011000000110101001100110011000100110000000110101
00110100001110010011000000110101
```
Message Digest (binary)= R6 ⊕ Key
```
0111000101110101000111110000100100011111011110001
0100110101001111010010010010101000110100001110001
```

```
01000111001110110011011000101100

Message Digest (Hex) =
                    71751f091f794d4f492a3439473b362c
```

The theoretical and practical improvements can be achieved are, (i) MD5 bit sequences appended with "01" bit sequences, so the resulting string 64 shorter than multiple of 512, (ii) In MD5 64 bits added to input string to take 64 bits strings from the arbiter positions. But in our algorithm length of the string /3. This will add more random selection, (iii) MD5 no buffer is used to store 128 bit blocks. In our mechanism also no buffer is used to divide 128 bits blocks from the arbiter length string, (iv) A random key secure against any collision and vulnerability and attack, (v) The output message digest contains all alphabets.

III. Experimental Results

For the implementation purpose, JAVA has been used as a programming language for this implementation and for the comparison purpose, the existing algorithms too implemented. The input plain text need to be given as a console input and the proposed NSCH algorithm calculate the message digest output and shown below (Table 3.1).

Table 3.1 Sample Random Strings

Input String	**Random String**
lingam	CEDTQOBYDWQC0030CEDTQO003700360039003 6003900340031003200340032003200340031003600038
cat	JLKAXVZXQ0030JLKAXV003700360039003700 3 1003300370031003900320038003300380037003 5
bat	PQAOZWUXWP0031PQAOZWU003700360039003 7003200310032003700380033003200300035003600038
hash	OARMJHQJBQ0034OARMJH0037003600390037 003200360035003500320034003100390033003700 31
hash	VXWMJHQJBQ0030VXWMJH003700360039003700 33003500350036003100310036003900300038003 7

As part of Analyzing, 5 random strings have been taken and calculated the percentage of the distinct characters. The results are shown in the table below. And figure 3.1 shows the comparison results. The efficiency of the NSCH Algorithm can be proven by comparing it with existing MD5 and Secured Cryptography Hashing algorithms [1] by considering the maximum number of characters each can have in their message digest hash value.

Table 3.2 Sample Random Strings with the comparison

Input String	**MD5**	**Secured Cryptography [1]**	**Our Scheme**
ab	62.5	65.6	69.5
system simulation	50	59.37	65
cd	46.87	65.62	70.23
project reports	50	70.73	74.33
lingam	40.62	76.56	81.21
Average	49.998	67.576	72.054

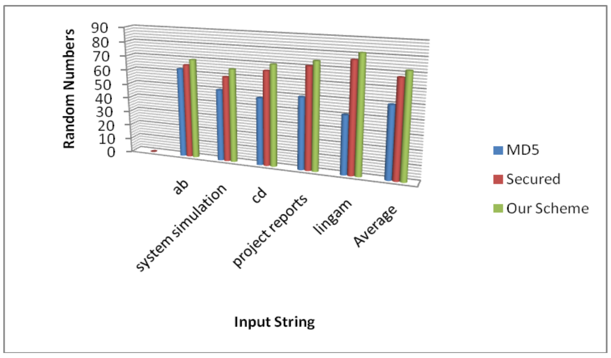

Figure 3.1 Sample Random Strings – comparison results

A MD5 hash digest is of the general form: ((a-z) + (0-9)) ^32, so the sample space of the characters in the combination= 26+10= 36. Secured Cryptography algorithm [1] is of the general form: ((a-z) + (A-Z) + (0-9) + (! - ;)) ^64, so, the sample space of the characters in the combination= 256. The proposed NSCH algorithm is of the general form: ((a-z) + (A-Z) + (0-9) + (!@#$ %^&*()_+\-=\[\]{};':"\\|,.<> \/?) + (Sunday-Saturday) + (0-9)) ^64, so the sample space of the characters combination = 273. In MD5 the sample space for occurrence of characters are 36. We can generally write each unique character can occur with 1.125 equal distributions. Similarly the secured cryptography algorithm also having sample space as 256, so each position can accommodate with 4 distinct characters. The proposed scheme sample space is 273 and for equal distribution each position can accommodate with 5.12 distinct characters. So that proposed algorithm NSCH is (5.125/1.125) = 4.55% random than existing MD5 algorithm.

IV. Conclusion and Future Work

In this paper work, a novel secure cryptographic hashing algorithm has been proposed and it has been compared with the existing algorithms, in order to prove that the new NSCH algorithm prove that it can generate more randomness and unique message digest for the arbitrarily length of the input string. The brute force and rainbow table based attacks can take more time to find the hash value. The future work is (i) to develop more stable, robust algorithm with less time and space complexity, (ii) to improve space and time complexity, and (iii) The NSCH algorithm shows a randomness efficiency of around 80% and it need to be improved better.

REFERENCES

[1] Rakesh Mohanty, Niharjyoti Sarangi, Sukant kumar Bishi, "A secured Cryptographic Hashing Algorithm", url: http://arxiv.org/abs/1003.5787, 2010.

[2] Florian Mendel,Christian Rechberger,Martin Schlaffer, "MD5 is Weaker than Weak: Attacks on Concatenated Combiners", ASIACRYPT 2009, LNCS 5912, pp. 144-161, 2009.

[3] B. den Boer, Antoon Bosselaers, Collisions for the Compression Function of MD5, Eurocrypto, 1993.

[4] R. L Rivest, The MD4 Message Digest Algorithm, Request for Comments (RFC)1320, Internet Activities Board, Internet Privacy Task Force, April 1992.

[5] R. L Rivest, The MD5 Message Digest Algorithm, Request for Comments (RFC)1321, Internet Activities Board, Internet PrivacZ Task Force, April 1992.

[6] Xiaoyun Wang; Hongbo Yu. “How to Break MD5 and Other Hash Functions”. EUROCRYPT. ISBN 3-540-25910-4,2005.