Message Digestion with Modulus Operator

Surya Prakash Dwivedi

Faculty

Department of Computer Science & Information Technology
M.G.C.G. University, Chitrakoot, Satna, M.P.

Abstract

Hash Function have various applications for data security for reducing the attacks on secure data like brute force attack. Mainly it is used for message digest i.e. representing our plain message in to some other letters or numbers. various principals are used for designing cryptographic hashed function for message digest. The electronic equivalent of the document and fingerprint pair is the message and digest pair respectively, to preserve the integrity of a message. The message is passed through an algo., is called cryptographic hash function. The function creates a compressed image of the message that can be used like a fingerprint. To check the integrity of the message or document, we run the cryptographic hash function again and compare the new message digest with the previous one. If both are the same, we assure that the original message has not been differ. This paper introduces the technique for message digestion as secure message digest(SDP) adding key in its algorithm so that it responds like MAC code and no need for new step encryption/decryption algorithm to create authenticated code from digestion. The proposed algorithm is sufficient for both integrity and authentication process.

Keywords: Key shifting, Data authentication, Integrity, Collision, Message Digestion, Modulus Operator

I. INTRODUCTION

Message Digestion is the process of symbolizing the text or message in to a specific letter or number i.e. the very much shorter form of the original plain message. In coding and decoding process there is no change in the original message at the receiver side. During the processing of the message digest we should keep some era In our mind:

1) For a given text we should have the simple digest.
2) Procedure for digestion having complex as possible as can, so that decodation is done by corresponding person only with corresponding technique (using same key).
3) Digestion is done for two messages, so that if anybody trying to find the message then, receives different messages
4) The Input will be a variable length.

If any two messages have the same message digest then it violates the basic principle of creating message digest, called ‘a collision’. That is if two message digest collide they meet at the same digest. Message digest algorithms usually reduce the length of msg from 128 bits to 256 bits.

Mathematically these algorithms have a Hash function that can map bit strings of random finite length into strings of fixed length. A hash value is generated by a function h =H (M), Where M is a variable-length message and H (M) is the fixed-length hash value.

Wouldn’t simple processes such as one of the following be enough to resolve the proposed collision attacks:

MD(M + |M|) = xyz
MD(M + |M| + |M| * seed_0 +...+ |M| * seed_n) = xyz

where :
M : plain text
|M| : size of text
MD : message digest func\n
xyz : pair of the original text digest value for the message M and |M|
seed\_i : Is a set of random values generated with seed based on the internal-state prior to the size being added. It not only generates the same MD But is also comprehensible/possible/compliant and is also the equal size as the original text. The purpose of a hash function is to generate a signature, message or other block of data. For message authentication, a hash function must be fulfill the following criteria:
1) It can be applied to a block of data of any size.
2) Function generates a fixed-length output.
3) Function is comparatively easy to calculate for any given input x.
4) Single side: for any given code h, it is computationally in reliable to find x such that h(x)=h.
5) Hard collision resistance: it is computationally infeasible to find any pair (x, y) such that H(y) = H(x).
The complexity of hash function against brute force attacks depends only on the size of hash code generated by the algorithm. Table I shows the level of effort required generating a birthday attack and square root attack (referred as strength of hash code) for different types of hash functions, assuming k-bit result.

<table>
<thead>
<tr>
<th>Type of hash function</th>
<th>Strength of hash function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single side</td>
<td>$2^k$</td>
</tr>
<tr>
<td>feeble collision resistance</td>
<td>$2^k$</td>
</tr>
<tr>
<td>hard collision resistance</td>
<td>$2^{(k/2)}$</td>
</tr>
</tbody>
</table>

Figure specifies the general view of a complex secure hash function.

Where H= chaining variable, $X_i = i$ input block, F=digest function, $i =$No. of input blocks, n=size of hash code, b=size of input block.

## II. Related Studies

We study some of the usual algorithms as noted in reference No. [1][3][4][6][7][8][9] such as modulus operation, MD4, MD5, SHA-1, MD2, SHA-256, MD6, MDP192 etc.

It reveals that the complexity of the algorithms that lies in the fact of algorithms should give as much of strongness and randomness, so that there are no two MD generated by the processed algorithm on any two message are same. In general all the MD algorithms have a scenario that each bit of the message digest is some symbol of every bit in the input. Almost all hash functions are calculated processes, which hash inputs of arbitrary length by processing followed by fixed-size blocks of the input text. The input k is padded to a multiple of the block length and subsequently divided into t blocks $k_1$ through $k_2$. The overall structure of a typical secure hash function is indicated in Fig.1. The hash function $h$ can then be described as follows:

$$H_0 = \text{initial n-bit value}$$

$$H(i) = F(H_{i-1}, X_i), 1 \leq i \leq t, H(X) = H(t).$$

The first constructions for hash functions were based on block ciphers such as DES. It is much feasible that other at that time as give the security trust to user but it is not go through long time. It is than the corresponding block cipher. Very powerful hash functions, which are now then used in a various of applications area, are the complex derived hash functions from the family MD4 which was proposed by R. Rivest [3].

It is a very fast and strong hash function of 32-bit processors. Because of unexpected possibilities identified in [11][12] (collisions for two rounds out of three), a much long an complex version of MD4 was produced which is called MD5 [4]. But MD5 is slightly slower than MD4, yet it is more secure in production. MD5 is very popular due to its security and usability.
III. PROPOSED MESSAGE DIGEST

A. Flow Chart

B. Algorithm

In proposed era, a new assumption of exchanging key in producing a message digest, it acts as MAC code. Here two integers are taken as the first step key K1 that is to be communicated between the client and the user through some secure and secret line channel. From this K1 the second step key is generating which is a list of 32 byte. This 32 byte list is created and stored in an intermediate array which will be used as storage of intermediate results throughout the algorithm and also stores the final result (message digest). The algorithm has fixed the maximum input message size less than 2^64 bits length. The output is a message digest of 256 bits in length.

Procedure to generate the second level key
The algorithms implemented by considering byte as a positive number from 0 to 255

**Input:**

i) Key k1, a set / array of two unsigned integers. K1 = {65, 267}

ii) Intermediate Byte array namely Inter [] of size 32.

**Output:** Filling the intermediate array Inter [32] with 32 different byte values using key1. These bytes will be used for triggering the message digest algorithm.

**Steps:**

1) Step 1. Let Inter [0] = (key1 [0] % 256.
2) Step 2. Let Integer variable K = key1 [1]
3) Step 3. Let Integer Variable R = 0
4) Step 4. Repeat through step 8 for I = 1 to 31
5) Step 5.  \( R = K + \text{Inter}[I \cdot 1] \)
6) Step 6.  If \( R > 256 \) Step 7. \( R = R \mod 256 \)
7) Step 8.  \text{Inter}[i] = \text{Convert To Byte}(R) \) Step 9. Repeat through step 15 for I = 0 to 7
8) Step 10. Byte Variable \( P = \text{Convert to Byte}((\text{Inter}[i*4] \text{ AND Inter}[i*4+1]) \text{ XOR } (\text{NOT(Inter}[i*4+2]) \text{ AND Inter}[i*4+3])) \)
9) Step 11.  Repeat through step 15 for J = 0 to 3 Step 12. \( R = \text{Inter}[i*4+j] + P \)
10) Step 13.  If \( R > 256 \) Step 14. \( R = R \mod 256 \)
11) Step 15. \( \text{Inter}[i*4+j] = \text{Convert To Byte}(R) \) Step 16. Stop

Steps to create the message digest

Input:
1) Intermediate Inter initialized with second level key using procedure of 3.1.
2) An input message maximum size less than 2^64
3) A Temporary integer array of size 32, let its name be Temp [32]
4) A process P of type Byte. This will be used in first level key (K1) to produce the message digest of length 256 bits.

Output: Message digest of length 256 bits.

Steps:
1) Step 1. The original message is added such that the size of the message bits is 64 bits < 256 bits. Then extra length of the plain message excluding the padding at the end of the padded message as 64 bit block. The adding many bytes are required and this bytes are supplied repeatedly from strat. Note that padding required in each steps even if the message is already 64 bits < 256 bits.
2) Step 2. Read 32 bytes at time and store it in virtual array for each 32 bytes Repeat through step x.
3) Step 3. Repeat through step 3.3 for I = 0 to 31
   1) Step 3.1. \( \text{Temp}[I] = \text{Temp}[I] + \text{Inter}[I] \)
   2) Step 3.2. If \( \text{Temp}[I] > 256 \) then
   3) Step 3.3. \( \text{Temp}[I] = \text{Temp}[I] \mod 256 \)
4) Step 4. Perform a special swap operation on the Temp array to add randomization.
   1) Step 4.1. Let Variable \( F = 0 \), Variable \( R = 31 \)
   2) Step 4.2. Repeat through step 4.6 for \( I = 0 \) to 15
   3) Step 4.3. \( \text{Temp}[F] = \text{Temp}[R] \)
   4) Step 4.4. \( \text{Temp}[F] = \text{Temp}[R] \)
5) Step 5. Consider the Temp array and the Intermediate array Inter [] into a group/block of 64 each of size 4 bytes.
6) Step 6. Repeat through step 11 for each group I = 0 to 7.
7) Step 7. Repeat through step 11 for each round J = 0 to 3.

If \( J = 0 \) then
\[
P = (\text{Temp}[I*4+0] \text{ AND Temp}[I*4+1]) \text{ XOR } (\text{NOT(Temp}[I*4+2]) \text{ AND Temp}[I*4+3])
\]
If \( J = 1 \) then
\[
P = (\text{Temp}[I*4+1] \text{ XOR Temp}[I*4+2]) \text{ XOR Temp}[I*4+3])
\]
OR Inter \( [I*4+0] \). If \( J = 2 \) then \( P = (\text{Temp}[I*4+1] \text{ AND Temp}[I*4+2]) \text{ OR (Temp}[I*4+1] \text{ AND Inter}[I*4+3]) \text{ OR (Temp}[I*4+2] \text{ AND Temp}[I*4+3]) \).
If \( J = 3 \) then
\[
P = \text{Inter}[I*4+1] \text{ XOR Temp}[I*4+2] \text{ XOR Temp}[I*4+3])
\]
8) Step 8. Repeat through step 11 for \( K = 0 \) to 3
9) Step 9. \( \text{Temp}[I*4+K] = \text{Temp}[I*4+K] + P \)
10) Step 10. If \( \text{Temp}[I*4+K] > 256 \) then
\[
\text{Temp}[I*4+K] = \text{Temp}[I*4+K] \mod 256
\]
11) Step 11. Repeat through step 13 for \( I = 0 \) to 31

Key Strength Analysis (Analysis for Brute-force attack)

First level key (K1) comprises of two unsigned integers each occupying 4 bytes. Therefore K1 have total bit length of 64 (32+32) bits. Here key2 is not considered in strength analysis as the second level key generation algorithm is open. But even then the hacker has to find the correct key2. Total numbers of possible keys having key size of 64 bits are as follows: 2^64 = 18,446744073709551616 x 10^6 possible keys. Now assume that a hacker have a very fast computer which he/she can execute the decryption algorithm in 1 micro second for all possible key trials. Even if he tries half the set of keys then also he is quite successful in decrypting.

But then also the hackers require more than one year decrypting the cipher text which is shown as below:
In one second = 10 x 10^6 possible key trials in one hour = 36 x 10^6 possible key trials in one day = 864 x 10^6 possible key trials
In one year = 3.1536 x 10^{13} possible key trials

The Intermediate array Inter i s analogous to the chaining variables used in some of the well-known algorithms such as SHA 2, MD5, SHA-1, SHA-256, MDP192, MDP-384 etc. But all these algorithms initialize the chaining (linking) variables with reinitialized constants to start the algorithm. But in this proposed paper the intermediate array is never initialized with
predefined constants, we use her linked list instead of array. The starting element takes place in an iterative basis using key1 [X, Y]. Bellow, the table specifies some of the results of algorithm III (A) on some closely related keys of key1[X, Y].

<table>
<thead>
<tr>
<th>Key1[A, B]</th>
<th>Intermediate Inter[] first time values for different values of key1 some test results</th>
</tr>
</thead>
</table>

### C. Algorithm Strength Analysis

The proposed Message digest fulfills the following requirements and hence justifies its strength:

1. The message digest is irreversible i.e. from a given digest it is hard to derive the original message, mathematically it is not possible because the process P operations are irreversible and more over modulus operation make the message digest harder to be reversible.

2. The algorithm also has sufficient complexity and randomness like other standard algorithms. Swapping is introduced before the process P operations. Here in this message digest system initialization of intermediate list Inter[] analogous to chaining variables as in [3][4][6][7]) takes place in a random basis using level one key k1[X, Y] while it is initialized by some predefined constants in MD4, MD5, SHA-1, SHA-256. This adds authentication to the message digest and hence by passing the requirement of separate encryption/decryption algorithm for creating MAC (Message authentication code). Here the message digest itself acts as MAC.

3. The heart of the message digest algorithm starts from step 8. From step 8 onwards 128 iterations are performed on the total block of 256 bits.

4. The possibility that two messages produce the same message digest is in the order of $2^{64}$ operations.

5. Given a message digest to find the original message can lead up to $2^{256}$ operations.

6. Some of the test results are shown in the following table. To show the testing here Key1[X, Y] = K1[61] [63] is considered here for all the test cases. It is here considered that the input messages are already padded message.

7. The bitwise circular shift operation as noted in [3][4][6][7] has been removed here for faster execution because circular shift consumes good number of swapping operations.

### Test results of the message digest algorithm.

| Digest1     | 234, 35, 99, 162, 225, 32, 95, 158, 25, 88, 151, 214, 177, 240, 47, 110, 201, 8, 71, 134, 116, 179, 33, 96, 233, 0, 103, 0, 166, 112, 175, 238, 49, 115. |
| Digest2     | 243, 43, 107, 170, 225, 32, 95, 158, 25, 88, 151, 214, 177, 240, 47, 110, 201, 8, 71, 134, 116, 179, 33, 96, 233, 0, 103, 0, 166, 112, 175, 238, 49, 115. |
| Digest3     | 243, 43, 107, 170, 62, 112, 175, 238, 25, 88, 151, 214, 177, 240, 47, 110, 201, 8, 71, 134, 116, 179, 33, 96, 233, 0, 103, 0, 166, 112, 175, 238, 49, 115. |
| Digest4     | 124, 180, 242, 45, 10, 240, 47, 110, 25, 88, 151, 214, 177, 240, 47, 110, 201, 8, 71, 134, 116, 179, 33, 96, 233, 0, 103, 0, 166, 112, 175, 238, 49, 115. |

All rights reserved by www.ijste.org
The message(1) and the digest1 can be set as a reference point. The other three messages and their corresponding digests are generated after making slight changes in the reference message(1) to observe how much the digest differs from the reference digest…The message(2) differs from reference message(1) by a single bit (126 i.e. 01111110) and 127 i.e. (01111111) differs by a single bit) though the digest2 differs by 8 bytes (more than one bits) from digest1. This strengthens the fact that even a small change in the message results in drastic change in its message digest. Similarly it can be observed in message(3) that for a change of 3-bits a total of 16 bytes differ from the reference message digest digest1. Another strength of the algorithm is that the change in the output message digest from a reference point is not related to change in the number of bit in the message from a reference message. This fact can again be proved from the message(4) and its digest4. It can be observed that for a chance of 8-bits in the message(4) with respect to the reference message(1) the message digest4 differs by 16 bytes same as the case of message(3).

Following is a comparison of the proposed algorithm with different versions of SHA.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>SHA-1</th>
<th>SHA-256</th>
<th>SHA-384</th>
<th>SHA-512</th>
<th>SGP- MD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Digest size (in bits)</td>
<td>152</td>
<td>245</td>
<td>386</td>
<td>510</td>
<td>256</td>
</tr>
<tr>
<td>Message Size (in bits)</td>
<td>$&lt;2^{64}$</td>
<td>$&lt;2^{64}$</td>
<td>$&lt;2^{128}$</td>
<td>$&lt;2^{128}$</td>
<td>$&lt;2^{64}$</td>
</tr>
</tbody>
</table>

### IV. Conclusion

In general message digest resolve the integrity problem of authentication. In order to add security with it one have to code the text, digest before transmitting it. This technique is known as MAC. To achieve this code from this message digest algorithm, which is a complex algorithm another algorithm is also required i.e. encryption /decryption. To reduce extra overhead it is proposed here to create a message digest using a key limiting message digest criteria and this produces and save much of computational results. Hence the proposed message digest system can be very useful in modern era, integrity sustaining practices as well as for security and authentication. The level one key key1 is used to generate the random linked-chaining entities which are taken in the generation of message digest. The algorithm performs 128 operations steps on a sub block of 256 bits. Which is very complex? The probability that two messages blocks results the identical message digest is in the order of $2^{64}$ steps operations. Given a message digest to find the original message can lead up to $2^{256}$ operations.

### References