Secured Hash2 based Message Authentication Code using GUI Controls

V R Kulkarni
Department of Computer Science And Engineering
KLS Gogte Institute of Technology, Belgaum, India

Saneet Kalmani
Department of Computer Science And Engineering
KLS Gogte Institute of Technology, Belgaum, India

Shashank Vernekar
Department of Computer Science And Engineering
KLS Gogte Institute of Technology, Belgaum, India

ABSTRACT
The message that originated from an authorized user is defined as User Authentication and is provided by Message Authentication codes (MAC). The provision of assurance that the message is not been modified is defined as Message Authentication and is provided by Hash functions. Authenticated code is generated when MAC uses Hash function it is called Hash based MAC (HMAC). In this paper an algorithm is proposed for HMAC which uses SHA-2 as cryptographic hash function. The algorithm is implemented using GUI based controls in MATLAB toolbox which enables the algorithm to be friendly with the user. Theoretical analysis and experimentation show that this hash function is with high plaintext sensitivity and low collision resistance, and secure against birthday attacks or meet-in-the-middle attacks. These properties make it a suitable choice for data signature or authentication.

Keywords
MAC, HMAC, SHA2, Plaintext Sensitivity, Collision Resistance

1. INTRODUCTION
In cryptography, a message authentication code (often MAC) is a short piece of information used to authenticate a message. A MAC algorithm, sometimes called a keyed (cryptographic) hash function[7], accepts as input a secret key and an arbitrary-length message to be authenticated, and outputs a MAC (sometimes known as a *tag*). The MAC value protects both a message's data integrity as well as its authenticity, by allowing verifiers (who also possess the secret key) to detect any changes to the message content.

While MAC functions are similar to cryptographic hash functions, they possess different security requirements. To be considered secure, a MAC function must resist existential forgery under chosen-plaintext attacks. This means that even if an attacker has access to an oracle which possesses the secret key and generates MACs for messages of the attacker's choosing, the attacker cannot guess the MAC for other messages (which were not used to query the oracle) without performing infeasible amounts of computation.

MACs differ from digital signatures as MAC values are both generated and verified using the same secret key. This implies that the sender and receiver of a message must agree on the same key before initiating communications, as is the case with symmetric encryption. For the same reason, MACs do not provide the property of non-repudiation offered by signatures specifically in the case of a network-wide shared secret key: any user who can verify a MAC is also capable of generating MACs for other messages. In contrast, a digital signature is generated using the private key of a key pair, which is asymmetric encryption. Since this private key is only accessible to its holder, a digital signature proves that a document was signed by none other than that holder. Thus, digital signatures do offer non-repudiation.

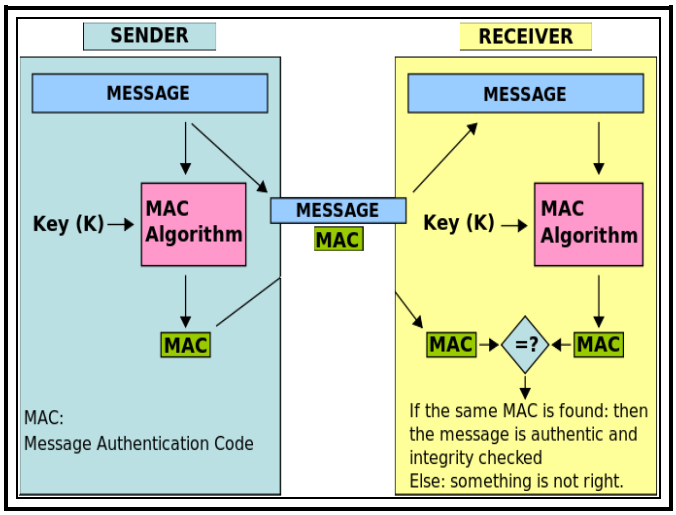

Figure 1: Working of MAC

In Figure 1, the sender of a message runs it through a MAC algorithm to produce a MAC data tag. The message and the MAC tag are then sent to the receiver. The receiver in turn runs the message portion of the transmission through the same MAC algorithm using the same key, producing a second MAC data tag. The receiver then compares the first MAC tag received in the transmission to the second generated MAC tag. If they are identical, the receiver can safely assume that the integrity of the message was not compromised, and the message was not altered or tampered with during transmission.

Essentially, the message authentication code[1] is a small fixed-size block of data that is generated based on a message *M* of variable length using secret key *K* as follows. It is also called cryptographic checksum.

$$MAC = C(K,M)$$

If A wishes to send B a message *M*, and protects it via a MAC, they first need to share a secret key *K*. Then A calculates code *MAC* as a function of *M* and *K*. Then the

message *M* plus the code *MAC* are transmitted to B. B performs the same calculation on *M*, using *K* to generate a new code *MAC'*. The received code *MAC* is compared to the calculated code *MAC_* to verify the data integrity. As only *A* is able to generate *MAC*, source authentication is also achieved.

2. HMAC ALGORITHM

Providing a way to check the integrity of information transmitted over or stored in an unreliable medium is a prime necessity in the world of open computing and communications. Message authentication codes are used between two parties that share a secret key in order to authenticate information transmitted between these parties.
The main objective behind the HMAC construction[10] are:

- Without modification using the available hash function
- Maintaining the performance
- Handling the keys in a simple way
- Knowledge about cryptographic analysis
- Underlying hash function should allow easy replacing facility

2.1 Parameters used in HMAC Algorithm

It uses the following parameters:

b	Input block size
H	Hash function approved
k	Secret key between sender and receiver
k0	Key k after pre-processing to form b-byte key
ipad	Byte x'36' repeated b times i.e inner pad
opad	Byte x'5C' repeated b times i.e outer pad
l	Output block size
t	Number of bytes of HMAC
text	Data on which HMAC is calculated

2.2 Steps for implementing HMAC Algorithm:

1. Check the length of k, if k=b then set k0=k and proceed to step 4
2. If length of k>b then hash k that gives a l byte string and then append it to (b-l) zeros that will create a b-byte string k0.Proceed to step 4
3. Check if k<b, if true then append zeros at the end of k to create a b-byte string k0
4. In this step exclusively-or is carried between k0 and ipad to produce a b-byte string
5. Append the stream of data 'text' to the result of step 4
6. Apply hash(H) to the entire stream of result from previous step
7. This step is used to exclusively-or the k0 value with the opad
8. Append the result from step 6 to 7
9. Applying the hash function(H) to entire result obtained
10. Leftmost t bytes of the result of step 9 are selected as MAC

3. SYSTEM ARCHITECTURE DESIGN

3.1 Source Side Architecture

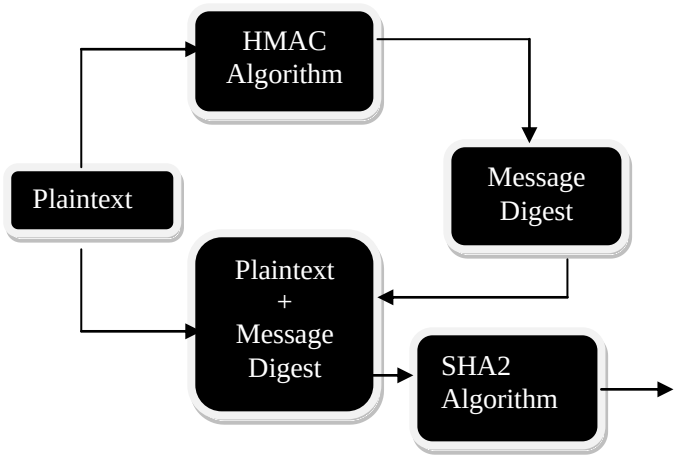

Figure 2: Source Side

- First accept the plaintext to be encrypted using the hash algorithm
- Apply the Hash algorithm and generate HMAC
- Concatenate the message digest with data and using java class security method, get the instance of the SHA-2 algorithm
- The Ciphertext is then sent to the receiver

3.2 Destination Side Architecture

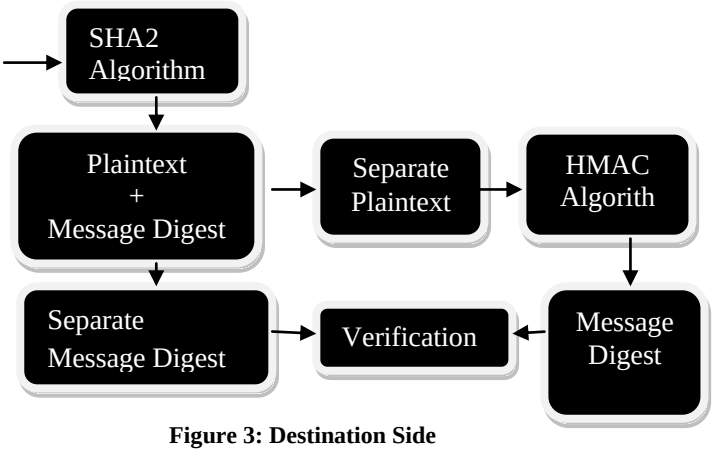

Figure 3: Destination Side

- The receiver which is connected to the sender via a LAN cable receives the Ciphertext
- SHA2 algorithm is applied to obtain the plaintext and digest
- Plaintext and the digest are separated
- Hash algorithm is again applied to the plaintext to obtain the digest
- This digest is compared to the previously generated digest and the data is verified

4. IMPLEMENTATION OF SHA2

4.1 General description

SHA-256 (secure hash algorithm, FIPS 182-2) is a cryptographic hash function with digest length of 256 bits. It is a keyless hash function; that is, an MDC (Manipulation Detection Code). A message is processed by blocks of 512 = 16 × 32 bits, each block requiring 64 rounds.

4.2 Basic operations

The basic operations are:

- Boolean operations AND and XOR are denoted by $\wedge$ and $\oplus$ respectively
- Bitwise complement, denoted by $\neg$
- Integer addition modulo 232, denoted by A + B
- Also operation like bitwise rightrotate(rr) and rightshift(rs) which takes the bit that was shifted out from the right end of a word and rotates it back to be inserted into the last bit position on the left (rightrotate) and shifts the bits positions to the right (rightshift)

Each of them operates on 32-bit words. For the last operation, binary words are interpreted as integers written in base 2. The algorithm uses the 64 binary words Ki given by the 32 first bits of the fractional parts of the cube roots of the first64 prime numbers:

0x428a2f98 0x71374491 0xb5c0fbcf 0xe9b5dba5
0x3956c25b 0x59f111f1 0x923f82a4 0xab1c5ed5

0xd807aa98 0x12835b01 0x243185be 0x550c7dc3
0x72be5d74 0x80deb1fe 0x9bdc06a7 0xc19bf174

0xe49b69c1 0xefbe4786 0x0fc19dc6 0x240ca1cc

0x2de92c6f 0x4a7484aa 0x5cb0a9dc 0x76f988da

0x983e5152 0xa831c66d 0xb00327c8 0xbf597fc7
0xc6e00bf3 0xd5a79147 0x06ca6351 0x14292967

0x27b70a85 0x2e1b2138 0x4d2c6dfc 0x53380d13
0x650a7354 0x766a0abb 0x81c2c92e 0x92722c85

0xa2bfe8a1 0xa81a664b 0xc24b8b70 0xc76c51a3
0xd192e819 0xd6990624 0xf40e3585 0x106aa070

0x19a4c116 0x1e376c08 0x2748774c 0x34b0bcb5
0x391c0cb3 0x4ed8aa4a 0x5b9cca4f 0x682e6ff3

0x748f82ee 0x78a5636f 0x84c87814 0x8cc70208

0x90befffa 0xa4506ceb 0xbef9a3f7 0xc67178f2

Message is processed in 512-bit blocks sequentially,

- Message digest is 256 bits instead of SHA-1's 160-bits
- 64 rounds instead of 80 rounds of compression
- Algorithm structure of SHA-2
  - Step 1: Padding bits
  - Step 2: Appending length as 64 bit unsigned
  - Step 3: Buffer initiation
  - Step 4: Processing of message
  - Step 5: Output

Buffer initiation: Eight 32-bit words instead of five in SHA-1

H_0 = 0x6a09e667 H_1 = 0xbb67ae85

H_2 = 0x3c6ef372 H_3 = 0xa54ff53a

H_4 = 0x510e527f H_5 = 0x9b05688c

H_6 = 0x1f83d9ab H_7 = 0x5be0cd19

- Each step t ($0 \le t \le 63$): Word expansion for W_t
  - If $t < 16$ $W_t = t^{th}$ 32-bit word of M_j
  - If $16 \le t \le 63$
    - $S_0 = (W_{t-15}$ rr 7) $\oplus$ (W_{t-15} rr 18) $\oplus$ (W_{t-15} rs 3)
    - $S_1 = (W_{t-2}$ rr 17) $\oplus$ (W_{t-2} rr 19) $\oplus$ (W_{t-2} rs 10)
    - $W_t = W_{t-16} + S_0 + W_{t-7} + S_1$
- Each step t ($0 \le t \le 63$):

$S_0 = (A$ rr 2) $\oplus$ (A rr 13) $\oplus$ (A rr 22)

ini = (A ^ B) $\oplus$ (A ^ B) $\oplus$ (B ^ C)

$t_2 = S_0$ + ini

$S_1 = (E$ rr 6) $\oplus$ (E rr 11) $\oplus$ (E rr 25)

cha = (E ^ F) $\oplus$ ($(\neg$ E) ^ G)

t_1 = H + S_1 + cha + K_t + W_t (A, B, C, D, E, F, G, H) = ($t_1 + t_2$, A, B, C, D + t_1, E, F, G)

5. PERFORMANCE ANALYSIS OF SHA2

5.1 Plaintext Sensitivity

Plaintext Sensitivity is the percentage of change in bits of cipher text obtained after encryption of plaintext, which is derived by changing single bit from the original plaintext from the bits of cipher text obtained after encryption of original plaintext. It is required for a hash function, to generate different hash value for different plaintext. This property in turn depends on the hash function's plaintext sensitivity. Slight difference in the plaintext will cause greater changes in the Hash value, which makes the Hash function have high plaintext sensitivity. This property is important to keep it secure against statistical attacks. To carry out the analysis of hash function's plaintext sensitivity as an example,

plaintext selected "The first step in model building consists of observing the real systems and the interaction among their various components and of collecting data on their behavior. But observations alone seldom yields sufficient understanding of system behavior. Persons familiar with the system or any subsystem should be questioned to gain advantage of their superior knowledge. Operators, technicians, repair persons, engineers, supervisors and managers understand certain aspect of the system that is unfamiliar to others.". The generated Hash value for the selected paragraph is,

Ho=506D04780A414E396118695A186306320B69404F

Then only first bit of the plaintext is changed, the according hash value is,

H1=7324407E3E0663074C5C582F1E1D540572337839

If the second one is changed, the according hash value is,

H2=317C291A72020F0A770C1F6B70207F4B1750365B

And if the i-th one is changed, then the according Hash value is Hi, Then the Hash value Change rate is computed by

r(i)= (Diff (H0,Hi) X 100) / 256

Where Dif (H0,Hi) means the number of the different bits between Ho and Hi. With implementation of SHA2, for the proposed plaintext, the change rate is shown in screenshot below and the change rate can be about 50% (64 bits) when neural networks is used. This shows that the Hash function is of high plaintext sensitivity.

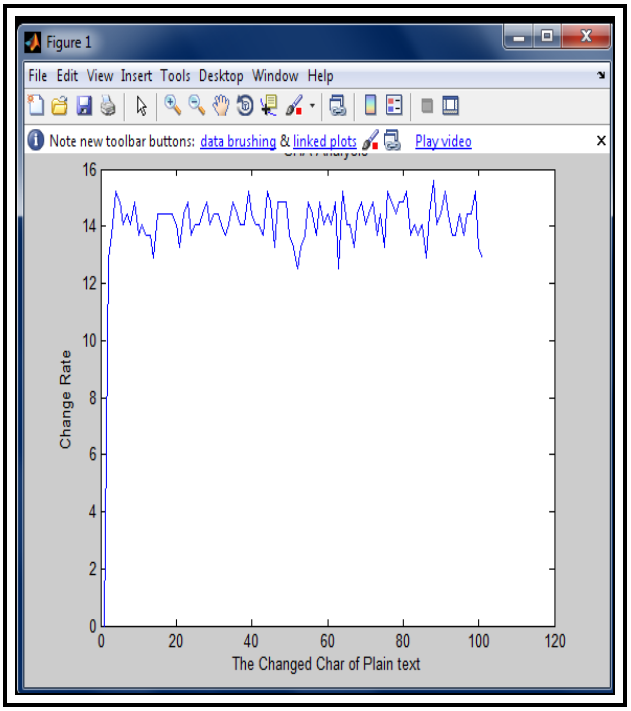

Figure 4: Snapshot of Plaintext Sensitivity

5.2 Collision Resistance

Collision resistance is a property of cryptographic hash functions: a hash function is collision resistant if it is hard to find two inputs that hash to the same output; that is, two inputs a and b such that $H(a) = H(b)$, and $a \neq b$. The hash is expected to provide a "unique" representation of the input data, allowing another disparate input data to hash to the same output does not serve the purpose. Collision resistance is required in the following fields:

- Firstly for digital signature systems, a party attests to a document by publishing a public key signature on a hash of the document. If it is possible to produce two documents with the same hash, an intruder could get a party to attest to one, and then claim that the party had attested to the other
- Also essential for distributed content systems, where in parties compare cryptographic hashes of files in order to make sure they have the same version. An intruder who could produce two files with the same hash could trick users into believing they had the same version of a file but in reality they did not have the same version

The following test has been performed analysis on Collision resistance.

Firstly the Hash value is generated for a paragraph of message which is randomly chosen and stored in ASCII format then a bit in the message which is selected randomly is changed, and a new Hash value is generated and stored in ASCII format.

Function to calculate the number of hits:

i. Count the number of hash values say n

ii. Repeat the following steps n times

iii. Read H(0) and H(0+i)

iv. Compare the 2 hash values and the number of ASCII character with the same value at the same location in the Hash value are counted then go step 2

v. Finally plot graph of number of equal entries versus number of hits

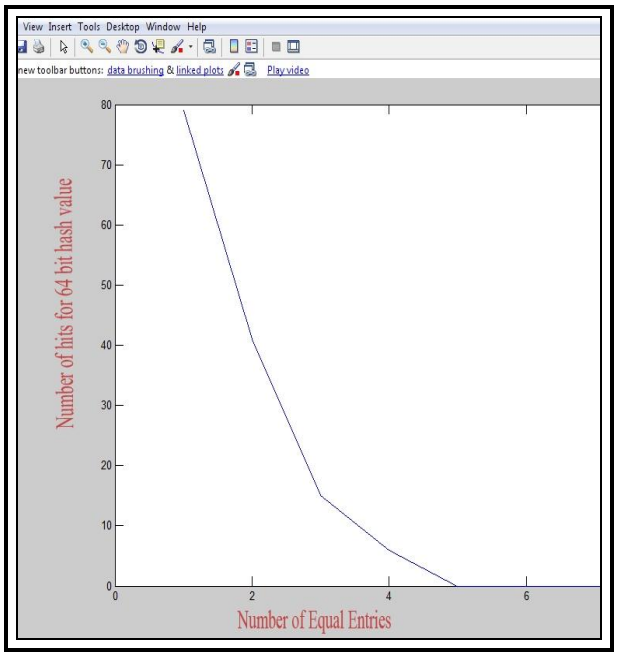

Figure 5: Snapshot of Collision Resistance

6. USER INTERFACE SNAPSHOTS

6.1 Data as Input

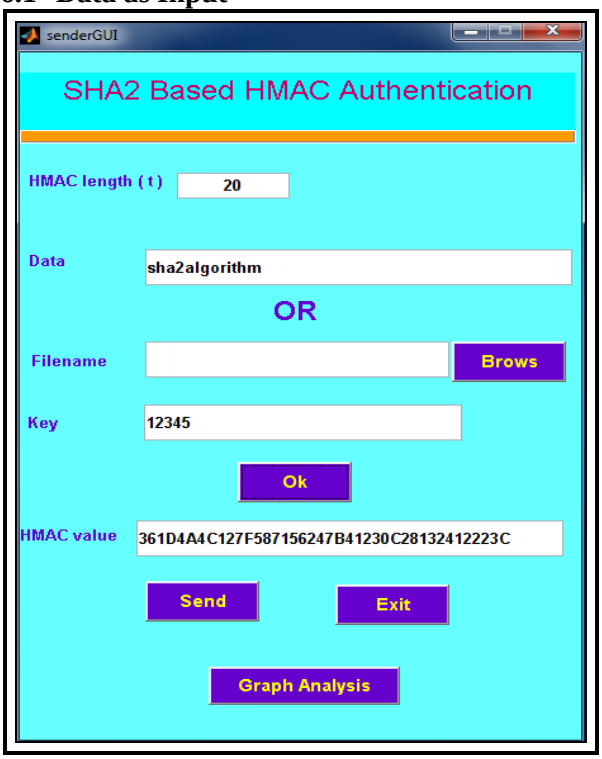

Figure 6: Plaintext as Input

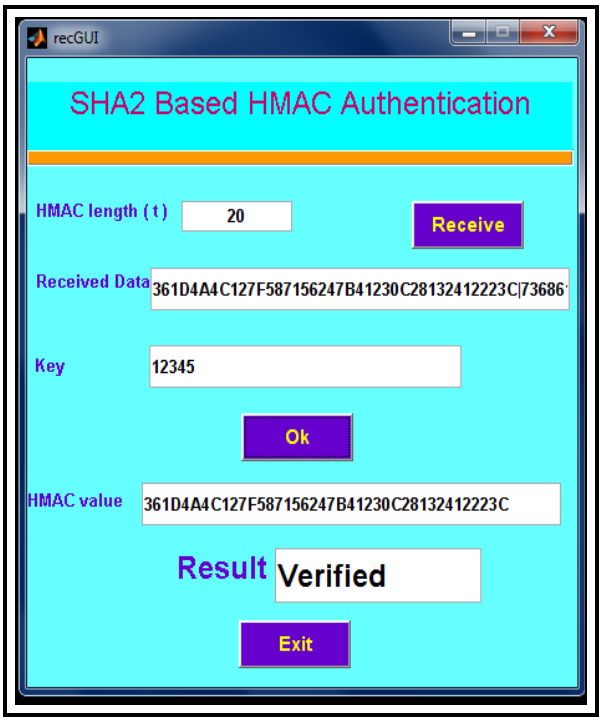

Figure 7: Receiver side response

7. CONCLUSION

The analysis and experiments shows that this hash function satisfies requirements. The plaintext sensitivity graph and the collision resistance graph help observe how efficient SHA2 is, when compared to already existing hash functions like MD5, SHA0, and SHA1 etc. It has played a significant part in improving the security over internet. Another major merit of SHA2 is it takes less time compared to existing methods. It's the most employed hash function these days and the hash value it computes is used to verify the integrity of copies of the original data without providing any means to derive said original data.

8. REFERENCES

[1] Deploying Modified Hash Based Message Authentication Code HMAC in MATLAB Using GUI Controls, B.Sridevi, Dr.S.Rajaram , *2011 International Conference on Information and Network Technology IPCSIT vol.4 (2011) © (2011) IACSIT Press, Singapore.*

[2] Hashing for Message Authentication Lecture Notes on "Computer and Network Security" by Avi Kak (kak@purdue.edu).

[3] Efficient HMAC Based Message Authentication System for Mobile Environment, Kavitha Boppudi,Sathish Vuyyala, International Journal Of Advanced Engineering Sciences And Technologies Vol No. 11, Issue No. 1, 208 – 212.

[4] Implementation of the SHA-2 Hash Family Standard Using FPGAs, The Journal of Supercomputing, 31, 227–248, 2005 Springer Science + Business Media, Inc. Manufactured in The Netherlands.

[5] "Schneier on Security: Cryptanalysis of SHA-1". Schneier.com. Retrieved 2011-11-08.

[6] Marc Fischlin, Anja Lehmann, and Daniel Wagner Hash Function Combiners in TLS and SSL, J.Pieprzyk (Ed.) LNCS 5985, pp. 268–283, 2010., *Springer-Verlag Berlin Heidelberg 2010.*

[7] American Bankers Association, *Keyed Hash Message Authentication Code*, ANSI X9.71, Washington, D.C., 2000.

[8] H. Krawczyk, M. Bellare, and R. Canetti, *HMAC: Keyed-Hashing for Message Authentication*, Internet Engineering Task Force, Request for Comments (RFC) 2104, February 1997.

[9] E.Biham and R.Chen "Near Collisions of SHA-0", *Advances in Cryptology CRYPTO 2004, LNCS 3152*, Spriger-Verlag, pp 290-305, 2004.

[10] "The Keyed-Hash Message Authentication Code (HMAC)" *in Federal Information Processing Standards Publication.*