Volume XXII 2019
ISSUE no.2
MBNA Publishing House Constanta 2019

Scientific Bulletin of Naval Academy

SBNA PAPER • OPEN ACCESS

Hashing and Message Authentication Code Implementation. An Embedded Approach

To cite this article: M. Rogobete and O. Tarabuta, Scientific Bulletin of Naval Academy, Vol. XXII 2019, pg. 296-304.

Available online at www.anmb.ro

ISSN: 2392-8956; ISSN-L: 1454-864X

doi: 10.21279/1454-864X-19-I2-035

Hashing and Message Authentication Code Implementation. An Embedded Approach.

Marius Rogobete[1], Octavian Tarabuta[2]

[1]Senior Software Engineer PhD, GE Power Romania
[2]Assistant Professor PhD. "Mircea cel Batran" Naval Academy, Constanta, Romania
E-mail: marius.rogobete@gmx.de

Abstract. There are different methods by which a message hashing could be embedded in a communications network, therefore different approaches are described in this research to protect the hash value of a message. The structure of a cryptographically secure function (SHA-512) is presented along with the low-level algorithm sequence. Subsequently is detailed the Hash-based Message Authentication Code (HMAC) produced by concatenating a secret key and message, after which the composite message is hashed. However, the HMAC numerical structure and the specific operating algorithm are explained in detail to the logical gate level. Finally, several considerations regarding the low-level implementation of the code are concluded.

1. Introduction

Authentication attack, a relatively new hacking technique, is a serious type of attack which can compromise entire IT infrastructure and software system. There are several attacks that have led to the identification of the authentication requirements, in the network communication network.

Disclosure of the message content to any entity regardless of whether it has the suitable cryptographic key.

Traffic Analysis: extracting the type of traffic between the parts. For an application oriented on connection, the statistics of connections (e.g. frequency and duration) can be determined. The message statistics (number and length of the messages) can be relatively easily detected, in any environment.

Dissimulation: a fraudulent source is used to thread messages in a network. This attack involves creation of messages supposedly originating from an authorized entity. Someone who differs from the recipient of the message includes fraudulent statements about receipt or non-receipt of the message.

Content modification: the content of a message is changed, including transposition, deletion, insertion or modification.

Sequence modification: modifying a sequence of messages sent between parts, including inserting, deleting, and reordering them.

Timing changing: delaying or replaying messages - in an application connection-oriented, a whole session or a series of messages can be a replay of a previous session which was valid. Or even individual messages of the sequence might be delayed or resumed. In an application without a connection, a specific message can be delayed or repeated (for example, the datagram).

Source Rejection: the source refuses to send the message.

Destination Repudiation: denial to receipt message at destination.

The confidentiality of messages is designed to combat "Disclosure" and "Traffic Analysis" attacks. Measures to address "Dissimulation", "Content modification", "Sequence modification" and "Timing changing" are generally considered to be covered by the authentication of messages. The mechanisms for the specific approach of "Source Rejection" and "Destination Repudiation" belong of the digital signature technique (that can be used for all the attacks, except "Disclosure" and "Traffic Analysis").

2. State of the Art

Message authentication signifies a procedure that checks if the received messages originate from the specific source and the messages have not been altered, but it can also to check for sequencing and timeliness. On the other site, the authentication technique includes measures to avoid repudiation by the source is a digital signature.

2.1. Authentication Function

There are two main functionalities of the digital signature (message authentication) mechanism: first creates a value that authenticate a message, an authenticator used at the lower level; the second is that this function is a primitive method in a higher-level authentication protocol that allows a receiver to verify the authenticity of a message.

To create an authenticator are use three groups of functions:

1. "Message encryption": the authenticator of the entire message is the ciphertext. A measure of authentication can be provided by the message encryption itself. But the schemes of symmetric and public-key encryption analysis are different.
2. "Message Authentication Code" (MAC): a technique that involves a function and a secret key (shared by sender and receiver) to generate a block of data from the message with small fixed-size that is appended to the message (as cryptographic checksum or MAC) and serves as authenticator. The function produces from the message a value of fixed length and a secret key. The receiver computes the MAC using the message and the secret key. Assuming that only the sender and receiver know the secret key, the MACs comparison is the authenticator. The MAC algorithm doesn't need to be reversible as the encryption function is. The most widely used of MAC functions was Data Authentication Algorithm (DAC) based on DES.
3. "Hash function": the authenticator that maps a message of various lengths into a message digests of fixed-length. The inputs of hash function are variable-size messages, producing different outputs of fixed size, similar with MAC. The only difference is the hash code doesn't need a key, being a function of the input message.

2.2. Hash Function

A hash function that processes input messages of variable size and produces hashcodes (named also hash values or message digests) is extremely important in the context of the efficient message authentication. Since all the bits of the input message produce a specific hash value, any alteration during transmission of even one bit of the input message shall compute a hash value that doesn't match with the original message hash code. This property is used to check for forgeries or any other kind of message alteration.

An example of used hash function is SHA-512 (Secure Hash Algorithm). It produces a hash value of 512-bit length from an input message of maximum 2^{16} bytes or 2^{128} bits length.

To protect the hash value of a message there are several methods, when incorporates message hashing in networks.

a. Based on the symmetric-key encryption, the input message and its message digest are together concatenated, and the resulted string is encrypted and transmitted; on the receiver side, the string is decrypted and extracts the message and hash code that is compared with the digest message computed from the received message. In this case the encryption provides confidentiality and the message digest provides authentication.

b. When there is an enough safe environment with good confidentiality and the authentication is the goal, a solution could be to encrypt only the message digest (symmetric-key encryption) instead the whole concatenated string. For the case, when the receiver has access to the secret key only will be able to verify whether the message is authentic (Message Authentication Code - MAC).
c. A version of previous scheme (b) for public-key encryption is when the message digest is encrypted using the sender's private key. The receiver can recover the message digest using the public key of the sender and thereby authenticates the message. The confidentiality here is not an issue. In other words, the digital signature idea is that the sender, using his private key, encrypts his message digest.
d. When the confidentiality and authentication are needed, a commonly used approach is to add a symmetric-key based confidentiality module at scheme (c).
e. A scheme with no encryption module could be used for authentication. However, there is a secret string appended to the message before to compute the message digest by sender. The secret string is known also by the receiver which adds it to the message before to process the hash code. Therefore, even if anyone has access to the original message and to the message digest, would be not possible to alter the message as authentic one.
f. Based on scheme (e), a symmetric-key module adds confidentiality for transmission between sender and receiver, encrypting the concatenated string (message and its hash code).

3. SHA-256 Secure Hash Algorithm

The sequence for processing SHA-512 method, suitable for a low-level implementation:
FIRST STEP: Because the block size is multiple of 1024 bits, the message is padded to this length. The last block should add the length of the message value of 128 bits size, in order to have an input message of multiple of 1024 exactly.
When the length of the original message is an exact multiple of 1024, needs to append a block of 1024-bit at the end (a room for the message length of 128-bit).

- The value of the message's length is an unsigned integer on 128-bit, with the most significant byte first.
- The message is a string of 1024 bit blocks, represented by the sequence:

$$m = \{B_1, B_2, \dots, B_N\}$$

where m is the whole message and B_n is the n^{th} message block of 1024 bits long.

Assembler code example:

```
BLOCKS struct
    Blck  db  dup(128)      ;size of block = 1024/8
BLOCKS ends

PADD struct
    pdbck db  dup(112)      ;size of block = (1024-128)/8
    lngth db  dup(16)       ;length of the message = 128 bits
PADD ends

PGSB union
    cnstBlck BLOCKS
    compBlck PADD
PGSB ends
...
.data
    pgmess PGSB { }         ;default initializers
```

```
...
.code
    ...
    ; Creates one BLOCK of data when
    ; the remained msg_blck_lg > 1024 bits
    MOV pgmess.cnstBlck.blck, msg
    ...
    JMP process_f
    ...
    ; On the last message's block of msg_blck_lg length
    MOV dx, 0
    MOV ax, msg_blck_lg
    CMP ax,1024
    JGE fill_union

    ; The last message's block, msg_blck_lg == 1024
    MOV pgmess.compBlck.pdbck, empty_msg   ;Filled with blank,
                                           ; to add the length
    MOV pgmess.compBlck.lngth, size_msg    ; of the message
    JMP process_f
    ...

fill_union:
    ; Padding necessary, msg_blck_lg < 1024
    ; Filled with  the rest of the message
    MOV pgmess.compBlck.pdbck, rest_msg_add_blnk
    ; padded with blank until 112 bits size
    MOV pgmess.compBlck.lngth, size_msg  ;Length of the message
process_f:
    CALL function_F        ; call f function for the 80 rounds
    ...
```

SECOND STEP: generates the required message schedule to process the input message of 1024-bit block that consists of 80 of 64-bits words. The 1024-bits message block gives the first 16 words. All the other words are computed using permutation and mixing operations to the previously generated words.

- The first eight primes' fractional parts of the square roots creates the Initialization Vector and initialized the temporary registers A, B, C, D, E, F, G, H [11]:

```
A = 6a09e667f3bcc908
B = bb67ae8584caa73b
C = 3c6ef372fe94f82b
D = a54ff53a5f1d36f1
E = 510e527fade682d1
F = 9b05688c2b3e6c1f
G = 1f83d9abfb41bd6b
H = 5be0cd19137e2179
```

THIRD STEP: On each message block of 1024-bits is applied the round-based. For each message block should be 80 rounds performed. First, PREVIOUS MESSAGE BLOCK's hash values calculated are stored into A, B, C, D, E, F, G, H (temporary 64-bit registers). For the n^{th} round, the values stored in all eight registers are permuted. Then is mixed schedule word ***W[n]*** and ***K[n]*** a constant of the round.

- Each $\boldsymbol{B_n}$, a Message Block of 1024-bits, is processed by the module ***f*** during 80 rounds.
- Before performing the round-based processing of each input message block of 1024-bits, should generate a *message schedule* that consists, for SHA-512, of 80 words labeled {W_0, W_1,...,W_{79}} of 64-bits. Where the sixteen words W_0 to W_{15} of 64-bits form the message block block B_i. All the other words in the schedule of the message are:

$$W_n = W_{n-16} +_{64} \rho_0(W_{n-15}) +_{64} W_{n-7} +_{64} \rho_1(W_{n-2}) \quad (1)$$

where

$$\rho_0(x) = ROTR^1(x) \oplus ROTR^8(x) \oplus SHR^7(x) \quad (2)$$

$$\rho_1(x) = ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^6(x) \quad (3)$$

$$ROTR^n(x) = circular\ right\ shift\ of\ the\ 64\ bit \arg by\ n\ bits$$

$$SHR^n(x) = right\ shift\ of\ the\ 64\ bit \arg by\ n\ bits\ with\ padding\ of\ zeros\ on\ the\ left$$

$$+_{64} = addition\ module\ 2^{64}$$

- For each input message block that is round-based processed, on the $\boldsymbol{n^{th}}$ round is processed the message schedule word W_n of 64-bits and a specific constant K_n, where K_n with *n = 0 to 79* represents the fractional parts of the cube roots of the $\boldsymbol{n^{th}}$ prime number (the first 64 bits). The constants destroy the regularities of the message blocks that tent to become random bit patterns [11]:

```
428a2f98d728ae22 7137449123ef65cd b5c0fbcfec4d3b2f e9b5dba58189dbbc
3956c25bf348b538 59f111f1b605d019 923f82a4af194f9b ab1c5ed5da6d8118
d807aa98a3030242 12835b0145706fbe 243185be4ee4b28c 550c7dc3d5ffb4e2
72be5d74f27b896f 80deb1fe3b1696b1 9bdc06a725c71235 c19bf174cf692694
e49b69c19ef14ad2 efbe4786384f25e3 0fc19dc68b8cd5b5 240ca1cc77ac9c65
2de92c6f592b0275 4a7484aa6ea6e483 5cb0a9dcbd41fbd4 76f988da831153b5
983e5152ee66dfab a831c66d2db43210 b00327c898fb213f bf597fc7beef0ee4
c6e00bf33da88fc2 d5a79147930aa725 06ca6351e003826f 142929670a0e6e70
27b70a8546d22ffc 2e1b21385c26c926 4d2c6dfc5ac42aed 53380d139d95b3df
650a73548baf63de 766a0abb3c77b2a8 81c2c92e47edaee6 92722c851482353b
a2bfe8a14cf10364 a81a664bbc423001 c24b8b70d0f89791 c76c51a30654be30
d192e819d6ef5218 d69906245565a910 f40e35855771202a 106aa07032bbd1b8
19a4c116b8d2d0c8 1e376c085141ab53 2748774cdf8eeb99 34b0bcb5e19b48a8
391c0cb3c5c95a63 4ed8aa4ae3418acb 5b9cca4f7763e373 682e6ff3d6b2b8a3
748f82ee5defb2fc 78a5636f43172f60 84c87814a1f0ab72 8cc702081a6439ec
90befffa23631e28 a4506cebde82bde9 bef9a3f7b2c67915 c67178f2e372532b
ca273eceea26619c d186b8c721c0c207 eada7dd6cde0eb1e f57d4f7fee6ed178
06f067aa72176fba 0a637dc5a2c898a6 113f9804bef90dae 1b710b35131c471b
28db77f523047d84 32caab7b40c72493 3c9ebe0a15c9bebc 431d67c49c100d4c
4cc5d4becb3e42b6 597f299cfc657e2a 5fcb6fab3ad6faec 6c44198c4a475817
```

- The module ***f*** that processes the message block has two parameters (one is the 512-bit hash buffer and the other is the 1024-bits message block) which are subsequently the inputs for the first 80 rounds of the processing.

FOURTH STEP: the processed hash values for the PREVIOUS message block is updated: is added to the values from temporary registers A, B, C, D, E, F, G, H.

- Several transposition and substitution operations compound the round function. At the round $\boldsymbol{n^{th}}$, the relations between input (the registers content of the hash buffer) and the output of this round is:

$$H = G$$

$$
\begin{aligned}
G &= F \\
F &= E \\
E &= D +_{64} T_1 \\
D &= C \\
C &= B \\
B &= A \\
A &= T_1 +_{64} T_2
\end{aligned} \tag{4}
$$

where

$$T_1 = H +_{64} \theta(E,F,G) +_{64} \lambda(E) +_{64} W_n +_{64} K_n \tag{5}$$

$$T_2 = \alpha(A) +_{64} \mu(A,B,C) \tag{6}$$

$$\theta(E,F,G) = (E \,\&\, F) \oplus (\bar{E} \,\&\, G) \tag{7}$$

$$\mu(A,B,C) = (A \,\&\, B) \oplus (A \,\&\, C) \oplus (B \,\&\, C) \tag{8}$$

$$\alpha(A) = ROTR^{28}(A) \oplus ROTR^{34}(A) \oplus ROTR^{39}(A) \tag{9}$$

$$\lambda(E) = ROTR^{14}(E) \oplus ROTR^{18}(E) \oplus ROTR^{41}(E) \tag{10}$$

- The hash buffer value from the beginning of the round-based processing is added to the 80th round's output. It is performed on every word of 64-bits for the 80th modulo 2^{64} output.
- The 80th round's output is added to the content of the hash buffer at the beginning of the round- based processing. It is performed on each 64-bit word of the output of the 80th modulo 2^{64}.
- When have been processed all ***N*** message blocks into the hash buffer is the message digest.

There are some code observations that are enough important for an embedded approach: the code example could be used for a MASM implementation but an implementation that targeting 80C51 microcontrollers series, a KEIL compiler is necessary with some modifications. However, the code implementation in Assembler, for relations (1) to (10) is relatively simple when the data structure and data union are used.

4. Hash Functions for Massage Authentication Codes

A Message Authentication Code (MAC), computed for messages of variable size, is a fixed-size fingerprint, as the hashcode is defined. This is also the definition for cryptographic checksum or authentication tag and could be produced by concatenate a message with a secret key and hashing the composite message. This MAC produced by a hash function is a HMAC.

There are more complex methods of producing a MAC involving iterative procedures by adding to the message a pattern obtained from the key and hashing the composite, then adding to the hashcode another pattern obtained from the key and the hashing the composite again and so on.

The DES encryption algorithm may be used for producing a MAC for a message and it is applied to a fixed-sized signature of the message, produced by a regular hash function. The encryption key becomes the secret that must be shared between the sender and the receiver of the message.

The original message together with its MAC can be safely transmitted over a network without worrying the data integrity when is used a hash function with a good collision resistance. To verify the message integrity a recipient with access to the used key for calculating the MAC should re-compute it and comparing with the received value.

The original message together with its MAC are safety transmitted over a network, without data integrity issues, using a collision-resistant hash function. When a recipient has access to the key used to calculate MAC, it can check the integrity of the message by recomputing its MAC and comparing it with the received value.

A not safe function *C(K,m)* that generates the *MAC* of message m using a secret key *K* is:

$$MAC = C(K, m) \tag{11}$$

where

$$m = (X_1 || X_2 || \dots || X_M) \quad and \quad \{X_1, X_2, \dots X_M\} are\ the\ 64 - bit\ blocks\ of\ the\ message\ m$$

and

$$\triangle(m) = X_1 \oplus X_2 \oplus \dots \oplus X_M \tag{12}$$

Whether is assumed that $e(K, \triangle(m))$ is DES, then

$$C(K, m) = e(K, \triangle(m)) \tag{13}$$

A brute force attack on the unsafe MAC algorithm to figure out the secret key K will be very expensive, by 2^{56} message's encryptions, but relatively easy to replace the original message with a fraudulent one.

The original message and its MAC can be safely transmitted over a network without worrying that the integrity of the data may get compromised.

A recipient with access to the key used for calculating the MAC can verify the integrity of the message by recomputing its MAC and comparing it with the value received.

A secure method for computing MACs is known as HMAC, used in different protocols (e.g. IPSec - for security on packet-level in networks or SSL - security on transport-level).

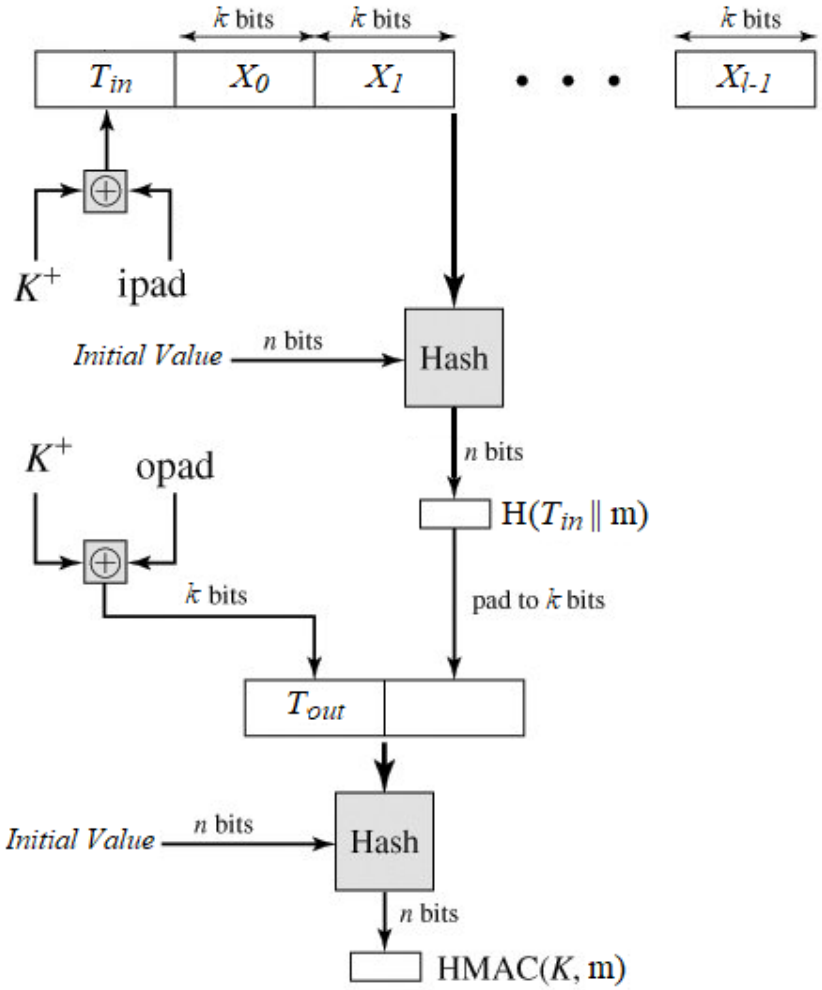

Figure 1: HMAC algorithm operation for a message authentication code.

The HMAC operation is described by the mathematical definition (RFC 2104), showed in equation (14).

$$HMAC(K, m) = H(\,(K' \oplus opad)\ ||\, H(\,(K' \oplus ipad) || m)\,) \tag{14}$$

where

H = underlying iterated hash function
m = input message for authentication
k = block size in bits
n = hash code length produced by the embedded hash function
K = the secret key, with length greater or equal *n*
K' = Derived from *K*, by padding on the left with zero to have a K bits length
‖ = concatenation
opad = block-sized outer padding (repeated bytes 0x5c)
ipad = block-sized inner padding (repeated bytes 0x36)

The HMAC algorithm produces a MAC of n-bits length when the input message m is processed, one block of *k* bits size a time (figure 1).
– The message is split into *k*-bits blocks X1, X2, ...
– the secret key *K* is used to produce the MAC
– *K'* is the secret key *K* padded with zeros thus the result is *k* bits long where *k* is the length of every message block *Xi*.
– *ipad* and *opad* are two sequences, *ipad* by repeating the 00110110-sequence *k*/8 times, and *opad* by repeating 01011100 also *k*/8 times.
An observation regarding to HMAC's security level, it depends on the underlying hash function's security and on the size and the quality of the key.

5. Conclusions

The message authentication methods are based on hash functions, the SHA-512 hash secure algorithm is presented together with the Assembler code segment that offers the fastest processing speed for a specific hardware, comparing with any other programming language. For the same reason, the flow chart of the Hash-based Message Authentication Code (HMAC) is presented from embedded point of view, highlighting the binary operations.
As the hash function is working independently of the HMAC, there is an important advantage in HMAC embedded implementation, the hash function can be used as a module. Subsequently it means the HMAC implementation is ready to run without modifications when it is necessary to update the hash function.
On the other side, for long messages HMAC and the embedded hash function should be executed in the same time. Moreover, the HMAC has three execution threads of the hash compression function, for T_{in} , T_{out} and for internal hash computing.

References

[1] J Guo,T Peyrin,Y Sasaki,L Wang, "Updates on Generic Attacks against HMAC and NMAC", Advances in Cryptology – CRYPTO 2014. Lecture Notes in Computer Science, vol 8616. Springer, Berlin, Heidelberg, ISBN 978-3-662-44371-2
[2] M. Rogobete, "Hash Function and Collision Resistance", EDUCATION AND CREATIVITY FOR A KNOWLEDGE BASED SOCIETY, Bucharest, November 2018, in progress.
[3] M. Najjar, "d-HMAC — An improved HMAC algorithm", International Journal of Computer Science and Information Security, Vol. 13, No. 4, 2015, ISSN 1947-5500
[4] M. Bellare, "New Proofs for NMAC and HMAC: Security without Collision-Resistance", Advances in Cryptology - CRYPTO 2006. CRYPTO 2006. Lecture Notes in Computer Science, vol 4117. Springer, Berlin, Heidelberg, ISBN 978-3-540-37433-6
[5] D. Ravilla, C. S. R. Putta, "Implementation of HMAC-SHA256 algorithm for hybrid routing protocols in MANETs", 2015 International Conference on Electronic Design, Computer Networks & Automated Verification, DOI: 10.1109/EDCAV.2015.7060558, IEEE 2015
[6] L. Beringer, A. Petcher, K.Q. Ye, A.W. Appel, "Verified correctness and security of OpenSSL HMAC",24th Usenix Security Symposium, August 12, 2015
[7] Krawczyk H., Bellare M., and Canetti R. (1997) HMAC: Keyed-Hashing for Message

Authentication, Internet Engineering Task Force, Request for Comments (RFC) 2104.
[8] Y Naito,Y Sasaki,L Wang,K Yasuda, "Generic State-Recovery and Forgery Attacks on ChopMD-MAC and on NMAC/HMAC", Advances in Information and Computer Security. IWSEC 2013, vol 8231. Springer, Berlin, Heidelberg, ISBN 978-3-642-41383-4
[9] T Ristenpart,H Shacham, T Shrimpton, "Careful with composition: Limitations of the indifferentiability framework." EUROCRYPT 2011. LNCS, vol. 6632, pp. 487–506. Springer, Heidelberg (2011), ISBN 978-3-642-20465-4
[10] C Gebotys, B White, E Mateos, "Preaveraging and Carry Propagate Approaches to Side-Channel Analysis of HMAC-SHA256", ACM Transactions on Embedded Computing Systems (TECS),Vol.15, Feb. 2016, DOI 10.1145/2794093, ISSN:1539-9087
[11] US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF), Internet Engineering Task Force, Request for Comments (RFC) 6234, ISSN: 2070-1721