

‘CRYPTOGRAPHIC HASH FUNCTIONS’

PROJECT REPORT

Submitted for CAL in B.Tech Data Structures and Algorithms (CSE2003)

By

Herman Llyod Menezes	16BCE1256
Pranav Jain	16BCE1112
Shivam Gaur	16BCE1060
Siddhant Singh	16BCE1272

Slot: G2 + L25,26

Dr. Nagaraj S V

**SCHOOL OF COMPUTING SCIENCE AND ENGINEERING
(SCSE)**



November, 2017

CERTIFICATE

This is to certify that the Project work entitled “*Cryptographic Hash Functions*” that is being submitted by “*Herman Lloyd Menezes, Siddhant Singh, Pranav Jain and Shivam Gaur*” for CAL in BTech Data Structures and Algorithms is a record of the bonafide work done under my supervision. The contents of this Project work, in full or in parts, have not been submitted for any other CAL course.

Place : Chennai

Date : 8th November 2017

Signature of Students:

Herman Lloyd Menezes
Pranav Jain
Siddhant Singh
Shivam Gaur

Signature of Faculty: Dr. Nagaraj S V

ACKNOWLEDGEMENTS

We have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals and organizations. We would like to extend our sincere thanks to all of them.

We are highly indebted to Dr. Nagaraj S V for their guidance and constant supervision as well as for providing necessary information regarding the project & also for their support in completing the project.

We would like to express my gratitude towards my parents & Vellore Institute of Technology, Chennai for their kind co-operation and encouragement which help us in completion of this project.

Our thanks and appreciations also go to my colleagues in developing the project and people who have willingly helped me out with their abilities.

ABSTRACT:

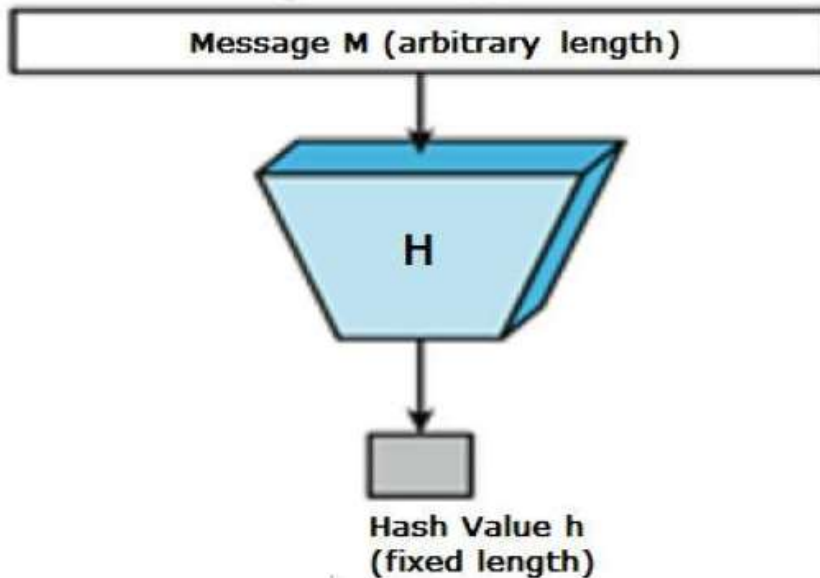
Hash functions are extremely useful and appear in almost all information security applications. A hash function is a mathematical function that converts a numerical input value into another compressed numerical value. The input to the hash function is of arbitrary length but output is always of fixed length. Values returned by a hash function are called **message digest** or simply **hash values**. In this project, we implement some cryptographic hash functions and study their applications in information security.

THEORY:

Introduction:

Values returned by a hash function are called **message digest** or simply **hash values**.

The following picture illustrated hash function –



Features of Hash Functions

The typical features of hash functions are –

- **Fixed Length Output (Hash Value)**
 - Hash function converts data of arbitrary length to a fixed length. This process is often referred to as **hashing the data**.
 - In general, the hash is much smaller than the input data, hence hash functions are sometimes called **compression functions**.
 - Since a hash is a smaller representation of a larger data, it is also referred to as a **digest**.
 - Hash function with n bit output is referred to as an **n -bit hash function**. Popular hash functions generate values between 160 and 512 bits.
- **Efficiency of Operation**
 - Generally for any hash function h with input x , computation of $h(x)$ is a fast operation.
 - Computationally hash functions are much faster than a symmetric encryption.

Properties of Hash Functions

In order to be an effective cryptographic tool, the hash function is desired to possess following properties –

- **Pre-Image Resistance**
 - This property means that it should be computationally hard to reverse a hash function.
 - In other words, if a hash function h produced a hash value z , then it should be a difficult process to find any input value x that hashes to z .
 - This property protects against an attacker who only has a hash value and is trying to find the input.

- **Second Pre-Image Resistance**

- This property means given an input and its hash, it should be hard to find a different input with the same hash.
- In other words, if a hash function h for an input x produces hash value $h(x)$, then it should be difficult to find any other input value y such that $h(y) = h(x)$.
- This property of hash function protects against an attacker who has an input value and its hash, and wants to substitute different value as legitimate value in place of original input value.

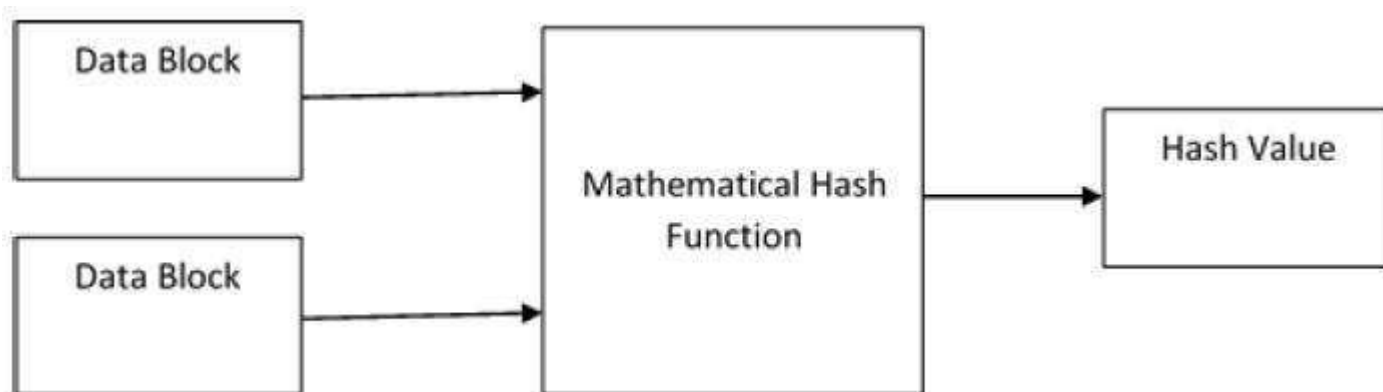
- **Collision Resistance**

- This property means it should be hard to find two different inputs of any length that result in the same hash. This property is also referred to as collision free hash function.
- In other words, for a hash function h , it is hard to find any two different inputs x and y such that $h(x) = h(y)$.
- Since, hash function is compressing function with fixed hash length, it is impossible for a hash function not to have collisions. This property of collision free only confirms that these collisions should be hard to find.
- This property makes it very difficult for an attacker to find two input values with the same hash.
- Also, if a hash function is collision-resistant **then it is second pre-image resistant**.

Design of Hashing Algorithms

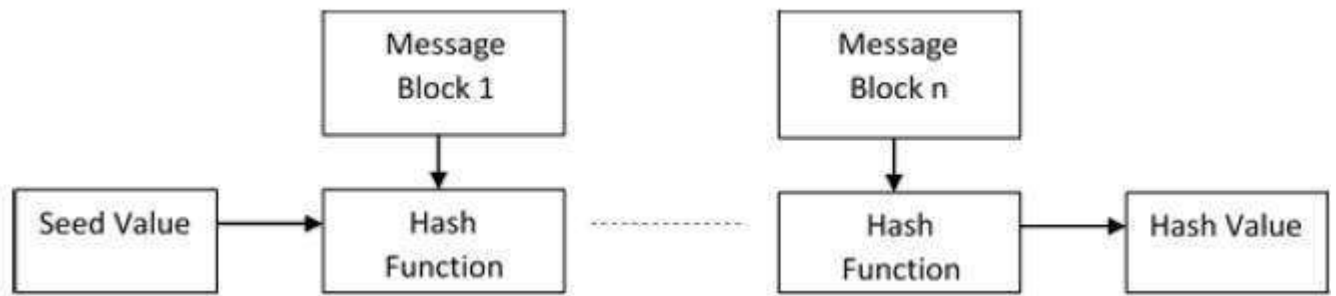
At the heart of a hashing is a mathematical function that operates on two fixed-size blocks of data to create a hash code. This hash function forms the part of the hashing algorithm.

The size of each data block varies depending on the algorithm. Typically the block sizes are from 128 bits to 512 bits. The following illustration demonstrates hash function –



Hashing algorithm involves rounds of above hash function like a block cipher. Each round takes an input of a fixed size, typically a combination of the most recent message block and the output of the last round.

This process is repeated for as many rounds as are required to hash the entire message. Schematic of hashing algorithm is depicted in the following illustration –



Since, the hash value of first message block becomes an input to the second hash operation, output of which alters the result of the third operation, and so on. This effect is known as an **avalanche** effect of hashing.

Avalanche effect results in substantially different hash values for two messages that differ by even a single bit of data.

Understand the difference between hash function and algorithm correctly. The hash function generates a hash code by operating on two blocks of fixed-length binary data.

Hashing algorithm is a process for using the hash functions, specifying how the message will be broken up and how the results from previous message blocks are chained together.

Popular Hash Functions

Secure Hash Function (SHA)

Family of SHA comprise of four SHA algorithms; SHA-0, SHA-1, SHA-2, and SHA-3. Though from same family, there are structurally different.

- The original version is SHA-0, a 160-bit hash function, was published by the National Institute of Standards and Technology (NIST) in 1993. It had few weaknesses and did not become very popular. Later in 1995, SHA-1 was designed to correct alleged weaknesses of SHA-0.
- SHA-1 is the most widely used of the existing SHA hash functions. It is employed in several widely used applications and protocols including Secure Socket Layer (SSL) security.
- In 2005, a method was found for uncovering collisions for SHA-1 within practical time frame making long-term employability of SHA-1 doubtful.
- SHA-2 family has four further SHA variants, SHA-224, SHA-256, SHA-384, and SHA-512 depending up on number of bits in their hash value. No successful attacks have yet been reported on SHA-2 hash function.
- Though SHA-2 is a strong hash function. Though significantly different, its basic design is still follows design of SHA-1. Hence, NIST called for new competitive hash function designs.
- In October 2012, the NIST chose the Keccak algorithm as the new SHA-3 standard. Keccak offers many benefits, such as efficient performance and good resistance for attacks.

Algorithm Design for SHA 1:

Step 1: Append Padding bits

Message is “padded” with a 1 and as many 0’s as necessary to bring the message length to 64 bits fewer than an even multiple of 512.

Step 2: Append Length

64 bits are appended to the end of the padded message. These bits hold the binary format of 64 bits indicating the length of the original message.

Step 3: Prepare Processing Functions

SHA1 requires 80 processing functions defined as:

$$f(t;B,C,D) = (B \text{ AND } C) \text{ OR } ((\text{NOT } B) \text{ AND } D) \quad (0 \leq t \leq 19)$$

$$f(t;B,C,D) = B \text{ XOR } C \text{ XOR } D \quad (20 \leq t \leq 39)$$

$$f(t;B,C,D) = (B \text{ AND } C) \text{ OR } (B \text{ AND } D) \text{ OR } (C \text{ AND } D) \quad (40 \leq t \leq 59)$$

$$f(t;B,C,D) = B \text{ XOR } C \text{ XOR } D \quad (60 \leq t \leq 79)$$

Step 4: Prepare Processing Constants

SHA1 requires 80 processing constant words defined as:

$$K(t) = 0x5A827999 \quad (0 \leq t \leq 19)$$

$$K(t) = 0x6ED9EBA1 \quad (20 \leq t \leq 39)$$

$$K(t) = 0x8F1BBCDC \quad (40 \leq t \leq 59)$$

$$K(t) = 0xCA62C1D6 \quad (60 \leq t \leq 79)$$

Step 5: Initialize Buffers

SHA1 requires 160 bits or 5 buffers of words (32 bits):

$$H0 = 0x67452301$$

$$H1 = 0xEFCDAB89$$

$$H2 = 0x98BADCFE$$

$$H3 = 0x10325476$$

$$H4 = 0xC3D2E1F0$$

Step 6: Processing Message in 512 bit blocks (L blocks in total message)

This is the main task of SHA1 algorithm which loops through the padded and appended message in 512-bit blocks.

Input and predefined functions:

$M[1, 2, \dots, L]$: Blocks of the padded and appended message $f(0;B,C,D)$,
 $f(1;B,C,D), \dots, f(79;B,C,D)$: 80 Processing Functions $K(0), K(1), \dots, K(79)$: 80 Processing
 Constant Words

$H0, H1, H2, H3, H4, H5$: 5 Word buffers with initial values

Pseudo Code for SHA 1:

For loop on $k = 1$ to L

$$(W(0), W(1), \dots, W(15)) = M[k] \text{ /* Divide } M[k] \text{ into 16 words */}$$

For $t = 16$ to 79 do:

$$W(t) = (W(t-3) \text{ XOR } W(t-8) \text{ XOR } W(t-14) \text{ XOR } W(t-16)) \lll 1$$

$$A = H0, B = H1, C = H2, D = H3, E = H4$$

For $t = 0$ to 79 do:

$$\text{TEMP} = A \lll 5 + f(t; B, C, D) + E + W(t) + K(t) \quad E = D, D = C,$$

$$C = B \lll 30, B = A, A = \text{TEMP}$$

End of for loop

$$H0 = H0 + A, H1 = H1 + B, H2 = H2 + C, H3 = H3 + D, H4 = H4 + E$$

End of for loop

Message Digest (MD)

MD5 was most popular and widely used hash function for quite some years.

- The MD family comprises of hash functions MD2, MD4, MD5 and MD6. It was adopted as Internet Standard RFC 1321. It is a 128-bit hash function.
- MD5 digests have been widely used in the software world to provide assurance about integrity of transferred file. For example, file servers often provide a pre-computed MD5 checksum for the files, so that a user can compare the checksum of the downloaded file to it.
- In 2004, collisions were found in MD5. An analytical attack was reported to be successful only in an hour by using computer cluster. This collision attack resulted in compromised MD5 and hence it is no longer recommended for use.

Algorithm Design for MD5:

1. MD5 processes a variable-length message into a fixed-length output of 128 bits. The input message is broken up into chunks of 512-bit blocks (sixteen 32-bit words); the message is padded so that its length is divisible by 512.
2. The main MD5 algorithm operates on a 128-bit state, divided into four 32-bit words, denoted *A*, *B*, *C*, and *D*. These are initialized to certain fixed constants. Then the algorithm uses each 512-bit message block in turn to modify the state. The processing of a message block consists of four similar stages, termed *rounds*; each round is composed of 16 similar operations based on a non-linear function *F*, [modular addition](#), and left rotation.

Pseudo Code for MD5:

```

for i from 0 to 63
    var int F, g
    if 0 ≤ i ≤ 15 then
        F := (B and C) or ((not B) and D)
        g := i
    else if 16 ≤ i ≤ 31
        F := (D and B) or ((not D) and C)
        g := (5×i + 1) mod 16
    else if 32 ≤ i ≤ 47
        F := B xor C xor D
        g := (3×i + 5) mod 16

```

```

else if  $48 \leq i \leq 63$ 
    F := C xor (B or (not D))
    g := (7×i) mod 16

```

Coding Style Followed:

C language has been used in the implementation of the SHA 1 hashing algorithm.

Operations on Words

- Bitwise logical word operations.
- The circular left shift operation $S^n(X)$, where X is a word and n is an integer with $0 \leq n < 32$.
-

Message Padding

The purpose of message padding is to make the total length of a padded message a multiple of 512. SHA-1 sequentially processes blocks of 512 bits when computing the message digest. The following specifies how this padding shall be performed.

Functions and Constants Used

A sequence of logical functions $f(0), f(1), \dots, f(79)$ is used in SHA-1. Each $f(t)$, $0 \leq t \leq 79$, operates on three 32-bit words B, C, D and produces a 32-bit word as output. A sequence of constant words $K(0), K(1), \dots, K(79)$ is used in the SHA-1.

Computing the Message Digest

The computation is described using two buffers, each consisting of five 32-bit words, and a sequence of eighty 32-bit words. The words of the first 5-word buffer are labeled A,B,C,D,E. The words of the second 5-word buffer are labeled H0, H1, H2, H3, H4. The words of the 80-word sequence are labeled W(0), W(1), ..., W(79).

Program Code:

SHA1:

```
#ifndef _SHA1_H_
#define _SHA1_H_

#include <stdint.h>
/*
 * If you do not have the ISO standardstdint.h header file, then you
 * must typedef the following:
 *
 *      name              meaning
 *
 *  uint32_t              unsigned 32 bit integer
 *  uint8_t               unsigned 8 bit integer (i.e., unsigned char)
 *  int_least16_t         integer of >= 16 bits
 *
 */

#ifndef _SHA_enum_
#define _SHA_enum_
enum
{
    shaSuccess = 0,
    shaNull,          /* Null pointer parameter */
    shaInputTooLong,  /* input data too long */
    shaStateError     /* called Input after Result */
};
#endif
#define SHA1HashSize 20

/*
 * This structure will hold context information for the SHA-1
 * hashing operation
 */
typedef struct SHA1Context
{
    uint32_t Intermediate_Hash[SHA1HashSize/4]; /* Message Digest */

    uint32_t Length_Low;          /* Message length in bits */
    uint32_t Length_High;        /* Message length in bits */

                                /* Index into message block array */
    int_least16_t Message_Block_Index;
    uint8_t Message_Block[64];   /* 512-bit message blocks */

    int Computed;                /* Is the digest computed? */
    int Corrupted;              /* Is the message digest corrupted? */
} SHA1Context;

/*
 * Function Prototypes
 */
int SHA1Reset( SHA1Context *);
int SHA1Input( SHA1Context *,
               const uint8_t *,
               unsigned int);
```

```

int SHA1Result( SHA1Context *,
                uint8_t Message_Digest[SHA1HashSize]);

#endif

//7.2 .c file

/*
 *  sha1.c
 *
 *  Description:
 *      This file implements the Secure Hashing Algorithm 1 as
 *      defined in FIPS PUB 180-1 published April 17, 1995.
 *
 *      The SHA-1, produces a 160-bit message digest for a given
 *      data stream.  It should take about 2**n steps to find a
 *      message with the same digest as a given message and
 *      2**(n/2) to find any two messages with the same digest,
 *      when n is the digest size in bits.  Therefore, this
 *      algorithm can serve as a means of providing a
 *      "fingerprint" for a message.
 *
 *  Portability Issues:
 *      SHA-1 is defined in terms of 32-bit "words".  This code
 *      uses <stdint.h> (included via "sha1.h" to define 32 and 8
 *      bit unsigned integer types.  If your C compiler does not
 *      support 32 bit unsigned integers, this code is not
 *      appropriate.
 *
 *  Caveats:
 *      SHA-1 is designed to work with messages less than 2^64 bits
 *      long.  Although SHA-1 allows a message digest to be generated
 *      for messages of any number of bits less than 2^64, this
 *      implementation only works with messages with a length that is
 *      a multiple of the size of an 8-bit character.
 */

#include <stdint.h>

/*
 *  Define the SHA1 circular left shift macro
 */
#define SHA1CircularShift(bits,word) \
    (((word) << (bits)) | ((word) >> (32-(bits))))

/* Local Function Prototypes */
void SHA1PadMessage(SHA1Context *);
void SHA1ProcessMessageBlock(SHA1Context *);

/*
 *  SHA1Reset
 *
 *  Description:
 *      This function will initialize the SHA1Context in preparation

```

```

*      for computing a new SHA1 message digest.
*
* Parameters:
*      context: [in/out]
*          The context to reset.
*
* Returns:
*      sha Error Code.
*
*/
int SHA1Reset(SHA1Context *context)
{
    if (!context)
    {
        return shaNull;
    }

    context->Length_Low          = 0;
    context->Length_High         = 0;
    context->Message_Block_Index = 0;

    context->Intermediate_Hash[0] = 0x67452301;
    context->Intermediate_Hash[1] = 0xEFCDAB89;
    context->Intermediate_Hash[2] = 0x98BADCFE;
    context->Intermediate_Hash[3] = 0x10325476;
    context->Intermediate_Hash[4] = 0xC3D2E1F0;

    context->Computed    = 0;
    context->Corrupted    = 0;

    return shaSuccess;
}

/*
* SHA1Result
*
* Description:
*      This function will return the 160-bit message digest into the
*      Message_Digest array provided by the caller.
*      NOTE: The first octet of hash is stored in the 0th element,
*            the last octet of hash in the 19th element.
*
* Parameters:
*      context: [in/out]
*          The context to use to calculate the SHA-1 hash.
*      Message_Digest: [out]
*          Where the digest is returned.
*
* Returns:
*      sha Error Code.
*
*/
int SHA1Result( SHA1Context *context,
                uint8_t Message_Digest[SHA1HashSize])
{
    int i;

```

```

    if (!context || !Message_Digest)
    {
        return shaNull;
    }

    if (context->Corrupted)
    {
        return context->Corrupted;
    }

    if (!context->Computed)
    {
        SHA1PadMessage(context);
        for(i=0; i<64; ++i)
        {
            /* message may be sensitive, clear it out */
            context->Message_Block[i] = 0;
        }
        context->Length_Low = 0;    /* and clear length */
        context->Length_High = 0;
        context->Computed = 1;
    }

}

for(i = 0; i < SHA1HashSize; ++i)
{
    Message_Digest[i] = context->Intermediate_Hash[i>>2]
                        >> 8 * ( 3 - ( i & 0x03 ) );
}

return shaSuccess;
}

/*
 * SHA1Input
 *
 * Description:
 *     This function accepts an array of octets as the next portion
 *     of the message.
 *
 * Parameters:
 *     context: [in/out]
 *         The SHA context to update
 *     message_array: [in]
 *         An array of characters representing the next portion of
 *         the message.
 *     length: [in]
 *         The length of the message in message_array
 *
 * Returns:
 *     sha Error Code.
 *
 */
int SHA1Input(    SHA1Context    *context,
                  const uint8_t  *message_array,
                  unsigned        length)

```

```

{
    if (!length)
    {
        return shaSuccess;
    }

    if (!context || !message_array)
    {
        return shaNull;
    }

    if (context->Computed)
    {
        context->Corrupted = shaStateError;

return shaStateError;
    }

    if (context->Corrupted)
    {
        return context->Corrupted;
    }
    while(length-- && !context->Corrupted)
    {
        context->Message_Block[context->Message_Block_Index++] =
            (*message_array & 0xFF);

        context->Length_Low += 8;
        if (context->Length_Low == 0)
        {
            context->Length_High++;
            if (context->Length_High == 0)
            {
                /* Message is too long */
                context->Corrupted = 1;
            }
        }

        if (context->Message_Block_Index == 64)
        {
            SHA1ProcessMessageBlock(context);
        }

        message_array++;
    }

    return shaSuccess;
}

/*
 * SHA1ProcessMessageBlock
 *
 * Description:
 *     This function will process the next 512 bits of the message
 *     stored in the Message_Block array.
 *
 * Parameters:
 *     None.

```



```

*
* Returns:
*     Nothing.
*
* Comments:
*
*     Many of the variable names in this code, especially the
*     single character names, were used because those were the
*     names used in the publication.
*
*
*/
void SHA1ProcessMessageBlock(SHA1Context *context)
{
    const uint32_t K[] = {          /* Constants defined in SHA-1 */
        0x5A827999,
        0x6ED9EBA1,
        0x8F1BBCDC,
        0xCA62C1D6
    };

    int            t;                /* Loop counter */
    uint32_t       temp;             /* Temporary word value */
    uint32_t       W[80];           /* Word sequence */
    uint32_t       A, B, C, D, E;   /* Word buffers */

    /*
     * Initialize the first 16 words in the array W
     */
    for(t = 0; t < 16; t++)
    {
        W[t] = context->Message_Block[t * 4] << 24;
        W[t] |= context->Message_Block[t * 4 + 1] << 16;
        W[t] |= context->Message_Block[t * 4 + 2] << 8;
        W[t] |= context->Message_Block[t * 4 + 3];
    }

    for(t = 16; t < 80; t++)
    {
        W[t] = SHA1CircularShift(1,W[t-3] ^ W[t-8] ^ W[t-14] ^ W[t-16]);
    }

    A = context->Intermediate_Hash[0];
    B = context->Intermediate_Hash[1];
    C = context->Intermediate_Hash[2];
    D = context->Intermediate_Hash[3];
    E = context->Intermediate_Hash[4];

    for(t = 0; t < 20; t++)
    {
        temp = SHA1CircularShift(5,A) +
            ((B & C) | ((~B) & D)) + E + W[t] + K[0];

        E = D;
        D = C;
        C = SHA1CircularShift(30,B);

        B = A;
        A = temp;
    }
}

```

```

for(t = 20; t < 40; t++)
{
    temp = SHA1CircularShift(5,A) + (B ^ C ^ D) + E + W[t] + K[1];
    E = D;
    D = C;
    C = SHA1CircularShift(30,B);
    B = A;
    A = temp;
}

for(t = 40; t < 60; t++)
{
    temp = SHA1CircularShift(5,A) +
        ((B & C) | (B & D) | (C & D)) + E + W[t] + K[2];
    E = D;
    D = C;
    C = SHA1CircularShift(30,B);
    B = A;
    A = temp;
}

for(t = 60; t < 80; t++)
{
    temp = SHA1CircularShift(5,A) + (B ^ C ^ D) + E + W[t] + K[3];
    E = D;
    D = C;
    C = SHA1CircularShift(30,B);
    B = A;
    A = temp;
}

context->Intermediate_Hash[0] += A;
context->Intermediate_Hash[1] += B;
context->Intermediate_Hash[2] += C;
context->Intermediate_Hash[3] += D;
context->Intermediate_Hash[4] += E;

context->Message_Block_Index = 0;
}

/*
 * SHA1PadMessage
 *
 * Description:
 *     According to the standard, the message must be padded to an even
 *     512 bits. The first padding bit must be a '1'. The last 64
 *     bits represent the length of the original message. All bits in
 *     between should be 0. This function will pad the message
 *     according to those rules by filling the Message_Block array
 *     accordingly. It will also call the ProcessMessageBlock function
 *     provided appropriately. When it returns, it can be assumed that
 *     the message digest has been computed.
 *
 * Parameters:
 *     context: [in/out]
 *         The context to pad

```

```

*      ProcessMessageBlock: [in]
*          The appropriate SHA*ProcessMessageBlock function
*      Returns:
*          Nothing.
*
*/

```

```

void SHA1PadMessage(SHA1Context *context)
{
    /*
    *   Check to see if the current message block is too small to hold
    *   the initial padding bits and length.  If so, we will pad the
    *   block, process it, and then continue padding into a second
    *   block.
    */
    if (context->Message_Block_Index > 55)
    {
        context->Message_Block[context->Message_Block_Index++] = 0x80;
        while(context->Message_Block_Index < 64)
        {
            context->Message_Block[context->Message_Block_Index++] = 0;
        }

        SHA1ProcessMessageBlock(context);

        while(context->Message_Block_Index < 56)
        {
            context->Message_Block[context->Message_Block_Index++] = 0;
        }
    }
    else
    {
        context->Message_Block[context->Message_Block_Index++] = 0x80;
        while(context->Message_Block_Index < 56)
        {
            context->Message_Block[context->Message_Block_Index++] = 0;
        }
    }

    /*
    *   Store the message length as the last 8 octets
    */
    context->Message_Block[56] = context->Length_High >> 24;
    context->Message_Block[57] = context->Length_High >> 16;
    context->Message_Block[58] = context->Length_High >> 8;
    context->Message_Block[59] = context->Length_High;
    context->Message_Block[60] = context->Length_Low >> 24;
    context->Message_Block[61] = context->Length_Low >> 16;
    context->Message_Block[62] = context->Length_Low >> 8;
    context->Message_Block[63] = context->Length_Low;

    SHA1ProcessMessageBlock(context);
}

```

/*7.3 Test Driver

The following code is a main program test driver to exercise the code in sha1.c.*/

```

/*
 * shaltest.c
 *
 * Description:
 *     This file will exercise the SHA-1 code performing the three
 *     tests documented in FIPS PUB 180-1 plus one which calls
 *     SHA1Input with an exact multiple of 512 bits, plus a few
 *     error test checks.
 *
 * Portability Issues:
 *     None.
 */

#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <stdint.h>

/*
 * Define patterns for testing
 */
#define TEST1    "abc"
#define TEST2a   "abcdbcdecdefdefgefghfghighijhi"
#define TEST2b   "jkijkljklmklmnlmnomnopnopq"
#define TEST2    TEST2a TEST2b
#define TEST3    "a"
#define TEST4a   "01234567012345670123456701234567"
#define TEST4b   "01234567012345670123456701234567"
    /* an exact multiple of 512 bits */
#define TEST4    TEST4a TEST4b
char *testarray[4] =
{
    TEST1,
    TEST2,
    TEST3,
    TEST4
};

long int repeatcount[4] = { 1, 1, 1000000, 10 };
char *resultarray[4] =
{
    "A9 99 3E 36 47 06 81 6A BA 3E 25 71 78 50 C2 6C 9C D0 D8 9D",
    "84 98 3E 44 1C 3B D2 6E BA AE 4A A1 F9 51 29 E5 E5 46 70 F1",
    "34 AA 97 3C D4 C4 DA A4 F6 1E EB 2B DB AD 27 31 65 34 01 6F",
    "DE A3 56 A2 CD DD 90 C7 A7 EC ED C5 EB B5 63 93 4F 46 04 52"
};

int main()
{
    SHA1Context sha;
    int i, j, err;
    uint8_t Message_Digest[20];

    /*
     * Perform SHA-1 tests
     */
    for(j = 0; j < 4; ++j)

```

```

{
    printf( "\nTest %d: %d, '%s'\n",
            j+1,
            repeatcount[j],
            testarray[j]);

    err = SHA1Reset(&sha);
    if (err)
    {
        fprintf(stderr, "SHA1Reset Error %d.\n", err );
        break;    /* out of for j loop */
    }

    for(i = 0; i < repeatcount[j]; ++i)
    {
err = SHA1Input(&sha,
                (const unsigned char *) testarray[j],
                strlen(testarray[j]));
        if (err)
        {
            fprintf(stderr, "SHA1Input Error %d.\n", err );
            break;    /* out of for i loop */
        }
    }

    err = SHA1Result(&sha, Message_Digest);
    if (err)
    {
        fprintf(stderr,
            "SHA1Result Error %d, could not compute message digest.\n",
            err );
    }
    else
    {
        printf("\t");
        for(i = 0; i < 20 ; ++i)
        {
            printf("%02X ", Message_Digest[i]);
        }
        printf("\n");
    }
    printf("Should match:\n");
    printf("\t%s\n", resultarray[j]);
}

/* Test some error returns */
err = SHA1Input(&sha, (const unsigned char *) testarray[1], 1);
printf( "\nError %d. Should be %d.\n", err, shaStateError );
err = SHA1Reset(0);
printf( "\nError %d. Should be %d.\n", err, shaNull );
return 0;
}

```

Implementation of the SHA 1 Hash function: (OUTPUT)

Sample output of how the Sha 1 output should be:

H0, H1, H2, H3, H4, H5: Word buffers with final message digest

"This is a test for theory of computation"

4480afca4407400b035d9debeb88bfc402db514f

Main output of the SHA 1 Hash function:

```
G:\3rd sem\data structure\project\SHA1.exe

Test 1: 1, 'abc'
      A9 99 3E 36 47 06 81 6A BA 3E 25 71 78 50 C2 6C 9C D0 D8 9D
Should match:
      A9 99 3E 36 47 06 81 6A BA 3E 25 71 78 50 C2 6C 9C D0 D8 9D

Test 2: 1, 'abdcdbcdeddefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq'
      84 98 3E 44 1C 3B D2 6E BA AE 4A A1 F9 51 29 E5 E5 46 70 F1
Should match:
      84 98 3E 44 1C 3B D2 6E BA AE 4A A1 F9 51 29 E5 E5 46 70 F1

Test 3: 1000000, 'a'
      34 AA 97 3C D4 C4 DA A4 F6 1E EB 2B DB AD 27 31 65 34 01 6F
Should match:
      34 AA 97 3C D4 C4 DA A4 F6 1E EB 2B DB AD 27 31 65 34 01 6F

Test 4: 10, '01234567012345670123456701234567012345670123456701234567'
      DE A3 56 A2 CD DD 90 C7 A7 EC ED C5 EB B5 63 93 4F 46 04 52
Should match:
      DE A3 56 A2 CD DD 90 C7 A7 EC ED C5 EB B5 63 93 4F 46 04 52

Error 3. Should be 3.

Error 1. Should be 1.

-----
Process exited after 0.1503 seconds with return value 0
Press any key to continue . . .
```

MD5:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

typedef union uwb {
    unsigned w;
    unsigned char b[4];
} MD5union;

typedef unsigned DigestArray[4];

unsigned func0( unsigned abcd[] ){
    return ( abcd[1] & abcd[2]) | (~abcd[1] & abcd[3]);}

unsigned func1( unsigned abcd[] ){
    return ( abcd[3] & abcd[1]) | (~abcd[3] & abcd[2]);}

unsigned func2( unsigned abcd[] ){
    return  abcd[1] ^ abcd[2] ^ abcd[3];}

unsigned func3( unsigned abcd[] ){
    return abcd[2] ^ (abcd[1] |~ abcd[3]);}

typedef unsigned (*DgstFctn)(unsigned a[]);

/*Use binary integer part of the sines of integers (Radians) as constants*/
/*or*/
/* Hardcode the below table but i want generic code that why coded it
using calculate table function
k[ 0.. 3] := { 0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee }
k[ 4.. 7] := { 0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501 }
k[ 8..11] := { 0x698098d8, 0x8b44f7af, 0xfffff5bb1, 0x895cd7be }
k[12..15] := { 0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821 }
k[16..19] := { 0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa }
k[20..23] := { 0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8 }
k[24..27] := { 0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed }
k[28..31] := { 0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a }
k[32..35] := { 0xffffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c }
k[36..39] := { 0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbcb70 }
k[40..43] := { 0x289b7ec6, 0xea127fa, 0xd4ef3085, 0x04881d05 }
k[44..47] := { 0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665 }
k[48..51] := { 0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039 }
k[52..55] := { 0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1 }
k[56..59] := { 0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1 }
k[60..63] := { 0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391 }*/

unsigned *calctable( unsigned *k)
{
    double s, pwr;
    int i;

    pwr = pow( 2, 32);
    for (i=0; i<64; i++) {
        s = fabs(sin(1+i));
```

```

        k[i] = (unsigned)( s * pwr );
    }
    return k;
}

/*Rotate Left r by N bits
or
We can directly hardcode below table but as i explained above we are opting
generic code so shifting the bit manually.
r[ 0..15] := {7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22}
r[16..31] := {5,  9, 14, 20, 5,  9, 14, 20, 5,  9, 14, 20, 5,  9, 14, 20}
r[32..47] := {4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23}
r[48..63] := {6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21}
*/
unsigned rol( unsigned r, short N )
{
    unsigned mask1 = (1<<N) -1;
    return ((r>>(32-N)) & mask1) | ((r<<N) & ~mask1);
}

unsigned *md5( const char *msg, int mlen)
{
    /*Initialize Digest Array as A , B, C, D */
    static DigestArray h0 = { 0x67452301, 0xEFCDAB89, 0x98BADCFE, 0x10325476 };
    static DgstFctn ff[] = { &func0, &func1, &func2, &func3 };
    static short M[] = { 1, 5, 3, 7 };
    static short O[] = { 0, 1, 5, 0 };
    static short rot0[] = { 7,12,17,22};
    static short rot1[] = { 5, 9,14,20};
    static short rot2[] = { 4,11,16,23};
    static short rot3[] = { 6,10,15,21};
    static short *rots[] = {rot0, rot1, rot2, rot3 };
    static unsigned kspace[64];
    static unsigned *k;

    static DigestArray h;
    DigestArray abcd;
    DgstFctn fctn;
    short m, o, g;
    unsigned f;
    short *rotn;
    union {
        unsigned w[16];
        char      b[64];
    }mm;
    int os = 0;
    int grp, grps, q, p;
    unsigned char *msg2;

    if (k==NULL) k= calctable(kspace);

    for (q=0; q<4; q++) h[q] = h0[q];    // initialize

    {
        grps  = 1 + (mlen+8)/64;
        msg2 = malloc( 64*grps);
        memcpy( msg2, msg, mlen);
        msg2[mlen] = (unsigned char)0x80;
    }

```



```

    q = mlen + 1;
    while (q < 64*grps){ msg2[q] = 0; q++ ; }
    {
        MD5union u;
        u.w = 8*mlen;
        q -= 8;
        memcpy(msg2+q, &u.w, 4 );
    }
}

for (grp=0; grp<grps; grp++)
{
    memcpy( mm.b, msg2+os, 64);
    for(q=0;q<4;q++) abcd[q] = h[q];
    for (p = 0; p<4; p++) {
        fctn = ff[p];
        rotn = rots[p];
        m = M[p]; o= O[p];
        for (q=0; q<16; q++) {
            g = (m*q + o) % 16;
            f = abcd[1] + rol( abcd[0]+ fctn(abcd) + k[q+16*p] + mm.w[g], rotn[q%4]);

            abcd[0] = abcd[3];
            abcd[3] = abcd[2];
            abcd[2] = abcd[1];
            abcd[1] = f;
        }
    }
    for (p=0; p<4; p++)
        h[p] += abcd[p];
    os += 64;
}
return h;
}

int main( int argc, char *argv[] )
{
    int j,k;
    const char *msg = "This is a test for theory of computation";
    printf("\t MD5 ENCRYPTION ALGORITHM IN C \n\n");
    printf("Input String to be Encrypted using MD5 : \n\t%s",msg);
    unsigned *d = md5(msg, strlen(msg));
    MD5union u;
    printf("\n\n\nThe MD5 code for input string is : \n");
    printf("\t= 0x");
    for (j=0;j<4; j++){
        u.w = d[j];
        for (k=0;k<4;k++) printf("%02x",u.b[k]);
    }
    printf("\n");
    printf("\n\t MD5 Encyption Successfully Completed!!!\n\n");
    getch();
    system("pause");
    return 0;
}

```

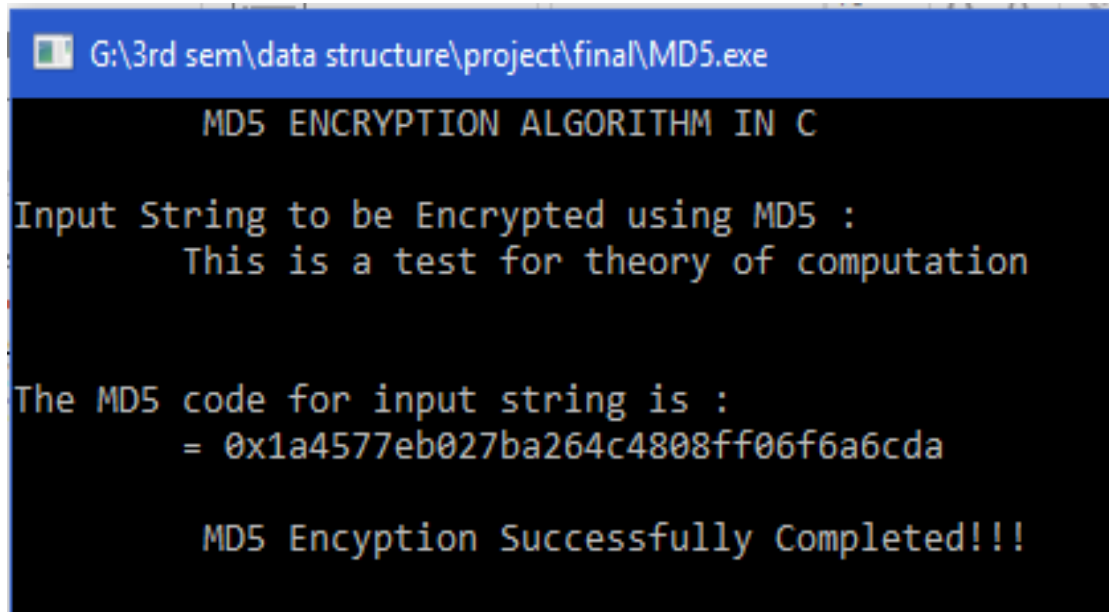
Implementation of the MD5 Hash function: (OUTPUT)

MD5 ENCRYPTION ALGORITHM IN C

Input String to be Encrypted using MD5 : "This is a test for theory of computation"

The MD5 code for input string is: 0x1a4577eb027ba264c4808ff06f6a6cda

MD5 Encyption Successfully Completed!!!



```
G:\3rd sem\data structure\project\final\MD5.exe
MD5 ENCRYPTION ALGORITHM IN C
Input String to be Encrypted using MD5 :
    This is a test for theory of computation

The MD5 code for input string is :
    = 0x1a4577eb027ba264c4808ff06f6a6cda

    MD5 Encyption Successfully Completed!!!
```

Limitations

MD5

- Attackers can create multiple input sources to MD5 that result in the same output fingerprint. This weakness may allow attackers to create two messages, or executable binaries such that their MD5 fingerprints are identical. One of these messages or binaries would be innocent, and the other malicious. The innocent message or binary may be digitally signed, and then later would have the malicious file substituted into its place. This attack may allow malicious code to be executed, or non-repudiation properties of messages to be broken.
- MD5 cannot be implemented in existing technology at rates in excess of 100 Mbps, and cannot be implemented in special-purpose CMOS hardware feasibly at rates in excess of 175 Mbps.
- MD5 cannot be used to support IP authentication in existing networks
- Can be cracked in 2^{64} iterations

SHA 1

- Slower, 80 iterations
- Attack can happen in 2^{31} iterations
- Size of output is only 160 bits so pre image attack is feasible with current technology
- Attacks have been reported and migration towards SHA-512 is in progress

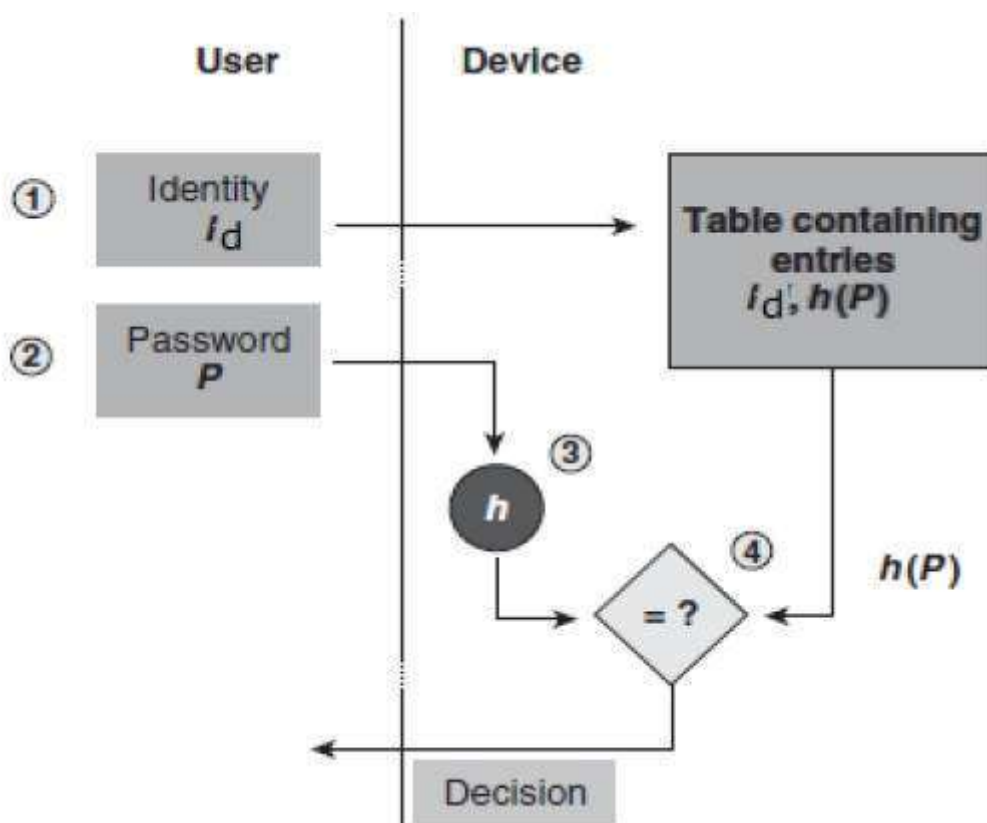
Applications of Hash Functions

There are two direct applications of hash function based on its cryptographic properties.

Password Storage

Hash functions provide protection to password storage.

- Instead of storing password in clear, mostly all logon processes store the hash values of passwords in the file.
- The Password file consists of a table of pairs which are in the form (user id, $h(P)$).
- The process of logon is depicted in the following illustration –

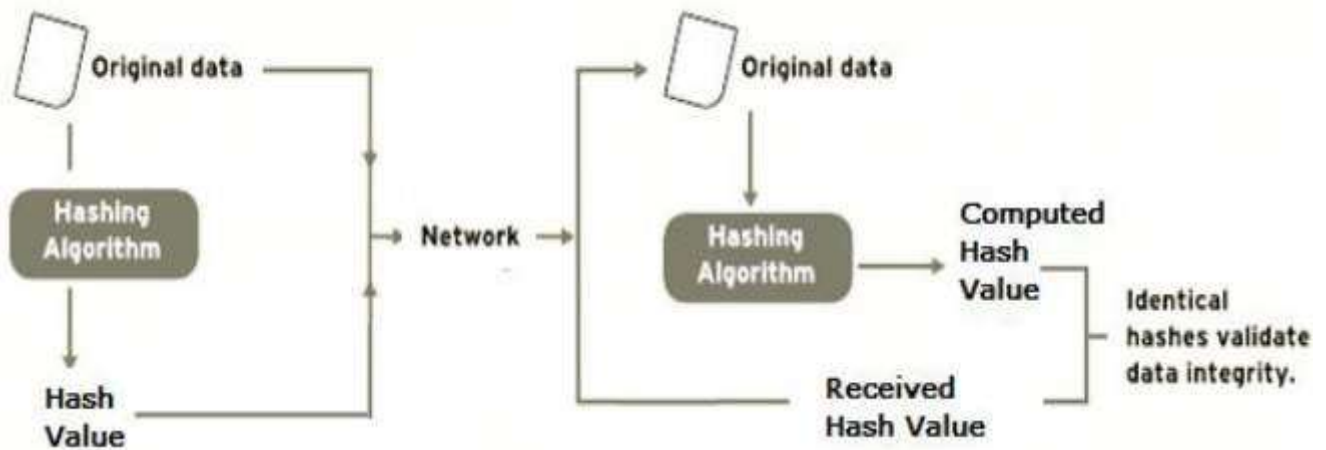


- An intruder can only see the hashes of passwords, even if he accessed the password. He can neither logon using hash nor can he derive the password from hash value since hash function possesses the property of pre-image resistance.

Data Integrity Check

Data integrity check is a most common application of the hash functions. It is used to generate the checksums on data files. This application provides assurance to the user about correctness of the data.

The process is depicted in the following illustration –



The integrity check helps the user to detect any changes made to original file. It however, does not provide any assurance about originality. The attacker, instead of modifying file data, can change the entire file and compute all together new hash and send to the receiver. This integrity check application is useful only if the user is sure about the originality of file.

CONCLUSION:

SHA 1 and MD 5 cryptographic hash functions and their applications in information security have been studied.

REFERENCES:

- *RFC 3174-US Secure Hash Algorithm 1 (SHA1)*
Network Working Group D. Eastlake, 3rd Request for Comments: 3174 Motorola
Category: Informational P. Jones Cisco Systems September 2001
- *International Journal of Innovative Research in Computer and Communication Engineering*
Vol. 3, Special Issue 7, October 2015
C.G Thomas, Robin Thomas Jose
- *Performance analysis of MD5*
Joseph D. Touch
USC / Information Sciences Institute
(touch@isi.edu)
- *International Journal of Scientific & Engineering Research Volume 3, Issue 12, December-2012*
Architectural design Of MD5 Controller IP Core
Sreeraj C, Sarath K Kumar , Nandakumar.R