THE SHA-1 DIARIES

© 2014 NIRAJ KHADKA
KATHMANDU ENGINEERING COLLEGE
BEX - 69060

TABLE OF CONTENTS

1) Acknowledgement

First of all I would like to thank all my teachers and friends for motivating me towards the programming language course. I would like to thank **Mr. Santosh Karna**, **Mr. Santosh Prajapati**, **Mr. Narayan Nepal** for always helping me in every academic situation and further encouraging me. I would also like to thank **Mr. Abhishek Maharjan**, **Mr. Bikrant Poudel**, **Mr. Nootan Ghimere** and **Mr. Binayak Tiwari** for their help in explaining things they do know.

This SHA-1 Diaries has been all possible due to their encouragement and help. This documentation will be a foundation for programming language students on how to implement real-world algorithms in practice to the C programming language.

2) Cryptography

Cryptography is a subject in computer security and applied mathematics in which we produce a cipher text from a message and message from a cipher text using some definite and distinct set of mathematics instructions. Cryptography was in use from the ancient time where message were sent to the receiver through the human medium. To preserve the privacy of the message content both sender and receiver used to share a common protocol on how to change the plain text message to a cipher text and vice versa. Though the mechanical machine and procedure used in cryptography dates back to the world war with the use of enigma machine, the electronic use has not been that old.
Generating a cypher text from the plain text is called encryption while generating a plain text from the cypher text is termed decryption. The cypher text is often called encrypted message and the plain text is called a decrypted message. Cryptography relates encryption, decryption as well as signature generation, message verification etc. using mathematical functions. In this article we will be looking in depth into a cryptographic hash function.

3) Cryptographic Hash Function

A cryptographic hash function is a function, which produces a limited size data or representation of the long message. This function is a one-way function meaning that the limited size data representation can be calculated for a given message but the reverse is not valid. We have been seeing the use of such hash functions more often, but may not have noticed it. An example is:

"Facebook stores the username and password data on its server when we register our account. But in case of storing the actual password in plaintext it generally stores the hash of the password. Whenever a user tries logging in with a password, the hash of the password now entered is calculated and then compared with the hash that is stored in the server and if both matches then we are logged in. The benefit of storing a hash instead of plaintext password is that in case the data server of Facebook is ever compromised and data are read, the hackers are not able to hijack our account as they cannot find our plain text password."

The above example gives us the basic instinct on why cryptographic hash function is of that much importance. Another example:

"Suppose a company produces a very high end computer gaming device. In the time of selling the hardware the company sells the hardware in the cost-to-cost basis or even on a low price to collect more customers. Now the company produces the game and then signs the game using a modified cryptographic hash function so that it can run on the system that it made. This way the company collects the revenue by selling games. If the gaming system is not secure enough people can pirate the games and play it by downloading it from Internet thus taking down the entertainment business. To make the system secure, every critical system booting files that are to be run on the system are verified first using these kind of hash function. So if anybody tries to alter the content or patch the content to remove its security on loading games, the overall hash of the file is changed and is refused to boot in the first place."

A single bit change in the original message can change the hash produced drastically that is called an avalanche effect. It is the benefit of this avalanche effect that makes the hash functions more secure and guess less. So the cryptographic hash function is primarily used in the password verification mechanism and for checking the integrity of the message or file. This way the security of any system or message is intact.

4) SHA (Secure Hash Algorithm)

A SHA is an algorithm that is designed to replace older hashing algorithms like MD4 and MD5. The SHA is issued by the National Institute of Standard and Technology (NIST) and is widely used in many systems, protocols and programs including TLS (Transport Layer Security), SSL (Secure Socket Layer), SSH (Secure Shell) and PGP (Pretty Good Privacy). Beside this these functions are used in many hardware devices like smartphones (APPLE, SONY, NOKIA, MOTOROLA etc.) and computer gaming systems (XBOX, PLAYSTATION, WII etc.).

The function of SHA algorithm is to produce a condensed message of a plain text message or a file whose size is limited by the standard to be 2^64 in case of SHA-1 and SHA-256 while 2^128 in case of SHA-384 and SHA-512. The SHA algorithm is similar in construct to that of MD5 but varies in the digest (the hash produced after calculation) size and mathematical constants and functions. The SHA being larger size and new functions thus reduces chance of being brute forced and collision. Collision is the fact of getting a same hash for two different messages that is very unlikely but can happen.

5) SHA-1

Sha-1 hash is the second kind of SHA hash function that was standardized and kept in practice by NIST. It is the upgrade to SHA-0 algorithm that had a major flaw on its hashing mechanism. The SHA-1 has a similar construct as that od the MD4 and MD5 algorithms. The hashing function performs the following steps.

a) Read the message length.
b) Append a bit '1' to the end of message.
c) Pad the message until the new length of message in bits is equal to new_length%512 = 448 bits.
d) Now append the original length of message in the end as a 64 bit unsigned integer in bits, thus the new length of the message in bits is a multiple of 512 bits.
e) Set the 5 32 bit has values to default hash values.
    i. H0 = 0X67452301
    ii. H1 = 0XEFCDAB89
    iii. H2 = 0X98BADCFE
    iv. H3 = 0X10325476
    v. H4 = 0XC3D2E1F0
f) Parse the message into 512 bits of chunks.
g) For each chunks set local variables a, b, c, d, e by reading the H0, H1, H2, H3 and H4 respectively and accordingly.
    i. Set the working variable W [0] to W [15], which is 32 bit wide to the value read from the message linearly.
    ii. Generate W [16] to W [79] using the formula below.

W [i] = W [i-3] XOR W [i-8] XOR W [i-14] XOR W [i-16] left rotated by 1, where I is from 16 to 79 inclusive.

iii. For i = 0 to 19 inclusive calculate;
   a. f = (b AND c) OR ((NOT b) AND d) and k = 0X5A827999.

iv. For i = 20 to 39 inclusive calculate;
   a. f = b XOR c XOR d and k = 0X6ED9EBA1.

v. For i = 40 to 59 inclusive calculate;
   a. f = (b AND c) OR (b AND d) OR (c AND d) and k = 0X8F1BBCDC.

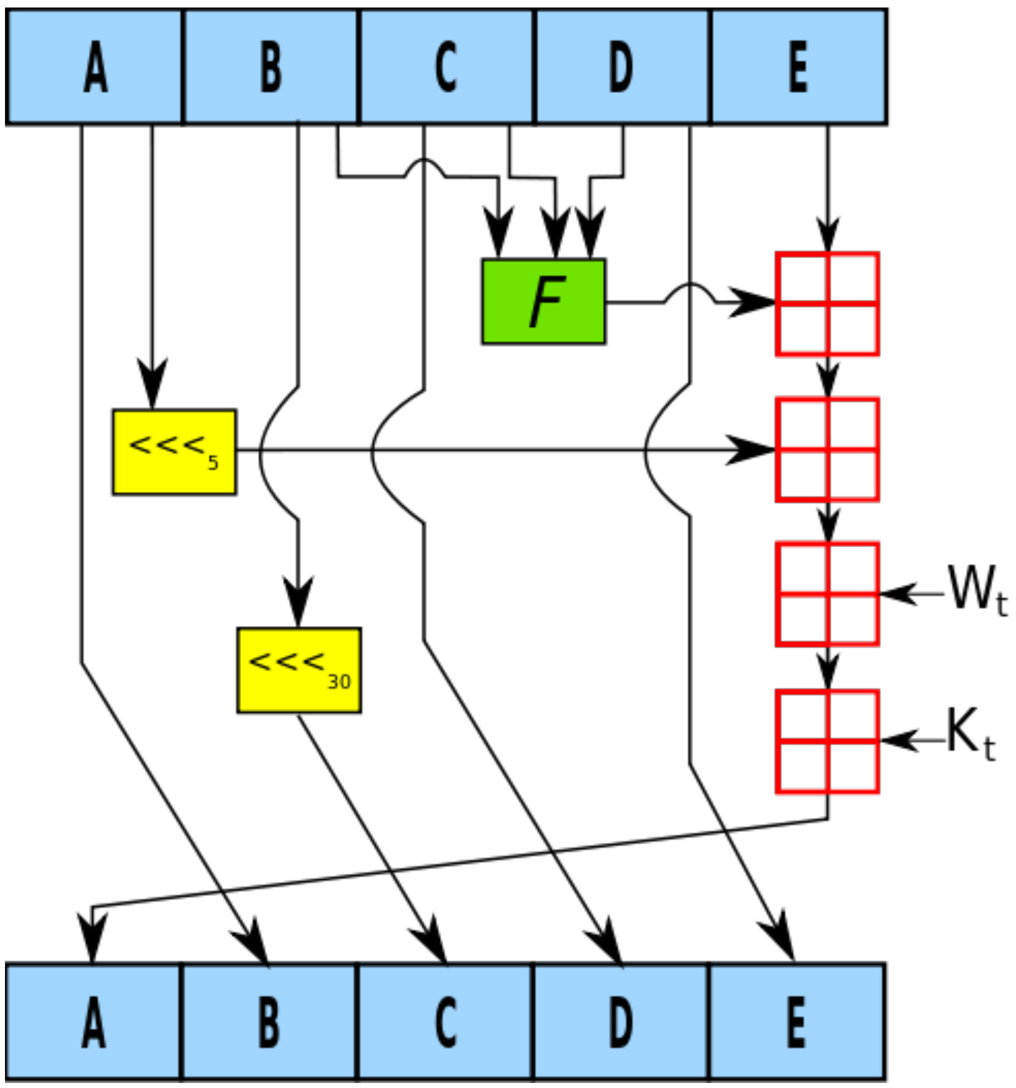
vi. For i = 60 to 79 inclusive calculate;
   a. f = b XOR c XOR d and k = 0XCA62C1D6.

vii. Set temp = (a left rotated by 5) + f + e + k + W [i]

viii. Now perform following;
   a. e = d
   b. d = c
   c. c = b left rotated by 30
   d. b = a
   e. a = temp

ix. Add this values to the default hash:
   a. H0 = H0 + a
   b. H1 = H1+ b
   c. H2 = H2 + c
   d. H3 = H3 + d
   e. H4 = H4 + e

h) Finally show the hash values by displaying H0, H1, H2, H3, H4 and H5 accordingly.



The above picture depicts the single iteration of a SHA-1 function, where ⊞ denotes addition modulo 2^{32} and $\lll_n$ denotes a left bit rotation by *n* places and W_t and K_t stands for the W [i] and k defined earlier in the algorithm.

6) SHA-1 implementation with source code in C

shalib.h

```
#ifndef SHALIB_H_INCLUDED
#define SHALIB_H_INCLUDED

#define ROTL(bits,word) (((word) << (bits)) | ((word) >> (32-(bits))))
typedef unsigned int uint32_t;
void debug_print(char *, uint32_t);
```

```
uint32_t padded_length_in_bits(uint32_t len);


struct sha
{
    uint32_t digest[5];
    uint32_t w[80];
    uint32_t a,b,c,d,e,f;
    int err;
};


#endif // SHALIB_H_INCLUDED
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include "shalib.h"


void debug_print(char *mss, uint32_t l)
{
    int i;
    printf("\nDEBUG PRINT START");
    for(i=0; i<(l/8); i++)
    {
        printf("\n%c", *(mss+i));
    }
    printf("\nDEBUG PRINT END\n");
}

uint32_t padded_length_in_bits(uint32_t len)
{
    if(len%64 == 56)
    {
        len++;
    }
    while((len%64)!=56)
    {
        len++;
    }
    return len*8;
}


int calculate_sha1(struct sha *sha1, unsigned char *text, uint32_t length)
{
    //struct sha *sha1;
    unsigned int i,j;
    //unsigned char text[] = "N3Ur0t0x1c";
    unsigned char *buffer;
   // uint32_t length = strlen((char *)text);
    uint32_t bits;
    uint32_t temp,k;
    uint32_t lb = length*8;

    //sha1 = (struct sha *) malloc(sizeof(struct sha));
    bits = padded_length_in_bits(length);
    buffer = (unsigned char *) malloc((bits/8)+8);
    memcpy(buffer, text, length);


    //add 1 on the last of the message..
    *(buffer+length) = 0x80;
    for(i=length+1; i<(bits/8); i++)
    {
        *(buffer+i) = 0x00;
    }

    /*append the length to the last words... using 32 bit only so the
    //limitation will be this function can calculate up to 4GB files SHA1.
       // *(buffer +(bits/8)-8) = (length>>56) & 0xFF;
       // *(buffer +(bits/8)-7) = (length>>48) & 0xFF;
    // *(buffer +(bits/8)-6) = (length>>40) & 0xFF;
    // *(buffer +(bits/8)-5) = (length>>32) & 0xFF;
    */

    *(buffer +(bits/8)+4+0) = (lb>>24) & 0xFF;
    *(buffer +(bits/8)+4+1) = (lb>>16) & 0xFF;
    *(buffer +(bits/8)+4+2) = (lb>>8) & 0xFF;
    *(buffer +(bits/8)+4+3) = (lb>>0) & 0xFF;


    // initialize the default digest values
    sha1->digest[0] = 0x67452301;
    sha1->digest[1] = 0xEFCDAB89;
    sha1->digest[2] = 0x98BADCFE;
    sha1->digest[3] = 0x10325476;
    sha1->digest[4] = 0xC3D2E1F0;

    //main loop
    for(i=0; i<((bits+64)/512); i++)
    {
        //first empty the block for each pass..
        for(j=0; j<80; j++)
        {
            sha1->w[j] = 0x00;
        }


        //fill the first 16 words with the characters read directly from the buffer.
        for(j=0; j<16; j++)
        {
            sha1->w[j] =buffer[j*4+0];
            sha1->w[j] = sha1->w[j]<<8;
            sha1->w[j] |= buffer[j*4+1];
            sha1->w[j] = sha1->w[j]<<8;
            sha1->w[j] |= buffer[j*4+2];
            sha1->w[j] = sha1->w[j]<<8;
            sha1->w[j] |= buffer[j*4+3];
        }

        //fill the rest 64 words using the formula
        for(j=16; j<80; j++)
        {
            sha1->w[j] = (ROTL(1,(sha1->w[j-3] ^ sha1->w[j-8] ^ sha1->w[j-14] ^ sha1->w[j-16])));
        }


        //initialize hash for this chunck reading that has been stored in the structure digest
        sha1->a = sha1->digest[0];
        sha1->b = sha1->digest[1];
        sha1->c = sha1->digest[2];
        sha1->d = sha1->digest[3];
        sha1->e = sha1->digest[4];

              //for all the 80 32bit blocks calculate f and use k accordingly per specification.
        for(j=0; j<80; j++)
        {
            if((j>=0) && (j<20))
            {
                sha1->f = ((sha1->b)&(sha1->c)) | ((~(sha1->b))&(sha1->d));
                k = 0x5A827999;

            }
            else if((j>=20) && (j<40))
            {
                sha1->f = (sha1->b)^(sha1->c)^(sha1->d);
                k = 0x6ED9EBA1;
            }
            else if((j>=40) && (j<60))
            {
                sha1->f = ((sha1->b)&(sha1->c)) | ((sha1->b)&(sha1->d)) | ((sha1->c)&(sha1->d));
                k = 0x8F1BBCDC;
            }
            else if((j>=60) && (j<80))
            {
                sha1->f = (sha1->b)^(sha1->c)^(sha1->d);
                k = 0xCA62C1D6;
            }

            temp = ROTL(5,(sha1->a)) + (sha1->f) + (sha1->e) + k + sha1->w[j];
            sha1->e = (sha1->d);
            sha1->d = (sha1->c);
            sha1->c = ROTL(30,(sha1->b));
            sha1->b = (sha1->a);
            sha1->a = temp;

            /* Detail of each pass for debugging purpose.
            printf("\n\ndetail %d passes a b c d and e values..\n",j);
            printf("a\tb\tc\td\te\n");
            printf("%x\t%x\t%x\t%x\t%x\n",sha1->a, sha1->b, sha1->c, sha1->d, sha1->e);
            */

            //reset temp to 0 to be in safe side only, not mandatory.
            temp =0x00;


        }

        // append to total hash.
        sha1->digest[0] += sha1->a;
        sha1->digest[1] += sha1->b;
        sha1->digest[2] += sha1->c;
        sha1->digest[3] += sha1->d;
        sha1->digest[4] += sha1->e;
```

```
              //since we used 512bit size block per each pass, let us update the buffer pointer
accordingly.
        buffer = buffer+64;

    }

       /*print SHA1 hash og given message.
    printf("\n\nSHA1 HASH O IS:\n");
    for(i=0; i<5; i++)
    {
        printf("%X ",sha1->digest[i]);
    }
    printf("\n");
    */

       //free the memory used.
    //free(buffer);
    //free(sha1);
    return 0;
}


int main(int argc, char*argv[])
{
    char option;
    int er,i;
    uint32_t file_len;
    struct sha *shaa;
    unsigned char *buff;
    FILE *fp;

    shaa = (struct sha *) malloc(sizeof(struct sha));
    printf("\n SHA1 GENERATOR V0.1b \n (C) 2014 Niraj Khadka (N3Ur0t0x1c)");
    if(argc<3)
    {
        printf("\nusage: %s option content", argv[0]);
        printf("\noption may be f for file and t for text");
        printf("\ncontent may be filename for f option and text for t option\n");
        free(shaa);
        exit(0);
    }
    else
    {
        option = *argv[1];
        switch(option)
        {
        case 't':
            {
                file_len = strlen((char *) argv[2]);
                printf("\nstrlen is %u", file_len);
                er = calculate_sha1(shaa,(unsigned char*)argv[2],file_len);
                break;
            }
        case 'f':
            {
                if((fp =fopen(argv[2],"rb")) == NULL)
                {
                    printf("\nError opening file..\nExiting....");
                    free(shaa);
                    exit(0);
                }

                //Get file length
                fseek(fp, 0, SEEK_END);
                file_len=ftell(fp);
                fseek(fp, 0, SEEK_SET);

                //printf("\n File length = %ul",file_len);

                //allocate memeory to read the whole file into  a buffer.
                buff = (unsigned char *) malloc(file_len);

                //If buffer is null then exit.
                if(!buff)
                {
                    printf("\nError allocating memory..");
                    free(shaa);
                    fclose(fp);
                    exit(0);
                }

                //Reading the total file on a memeory in one shot.
                fread(buff, file_len, 1, fp);
                fclose(fp);

                //call function
                er = calculate_sha1(shaa, buff, file_len);

                if(er != 0)
                {
                    printf("\n Error calculating SHA1 hash.. \n");
                    exit (0);
                }


            }
        }
    }

    //print hash
    printf("\n\nSHA1 HASH IS:\n");
    for(i=0; i<5; i++)
    {
        printf("%X ",shaa->digest[i]);
    }
    printf("\n");
    free(buff);
    free(shaa);
    return 0;
}
```

7) Limitations of the Code

The above given code has the limitations of handling the files or content of size less than 2^{32} bits only. Because there is no such data type in C that is of a size of 64 bits I tried to implement the 32 bits for storing the length as a result only the last 32 bit of the padded message is filled with the length unsigned binary integer representation. The code is also a lot fragmented and many repetitive things are not maintained as functions. Though I may not consider this as a limitation but is done to minimize the function calling overhead.

8) Limitations of SHA-1

There are very few limitations of SHA-1. To name some of them;

a) It cannot handle file size of length larger than 2^{64} bits.
b) Since its uses 160 bits as a hash length there may be more chances of collision than that of SHA-256.
c) If only numbers are used as message and this fact is pre known then the number can be constructed back by brute forcing using powerful GPU computing as with the case of NOKIA SL3 unlocking.
d) The hash can be brute forced if parallel processing or cloud processing is done in less time.
e) As per Bruce Schneier no algorithm is bad unless it has a implementation flaw.

9) Future Expansions

This SHA-1 can be used as a standalone function and more functions to calculate SHA-256, MD4 and MD5 though obsolete, AES etc. can be created to make this a complete basic crypto library. For this advance knowledge of applied mathematics, number theory and programming construct are necessary. A reading on data structure can also be beneficial for handling different data types efficiently and effectively thus increasing the performance of the algorithm.

10) References

a) http://en.wikipedia.org/wiki/SHA-1
b) http://fail0verflow.com/
c) https://polarssl.org/sha-1-source-code
d) http://tools.ietf.org/search/rfc4634