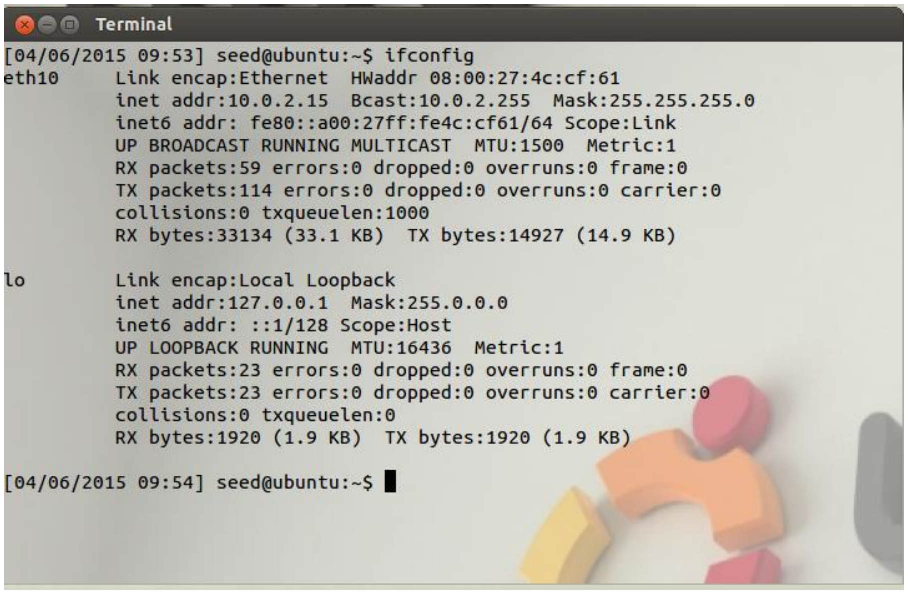
Lab 7 Crypto Lab II: One-Way Hash Function and MAC

Bharath Darapu

Lab Environment:
We will be using one VM for this VM_1(10.0.2.15).



```
[04/06/2015 09:53] seed@ubuntu:~$ ifconfig
eth10     Link encap:Ethernet  HWaddr 08:00:27:4c:cf:61
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe4c:cf61/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:59 errors:0 dropped:0 overruns:0 frame:0
          TX packets:114 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:33134 (33.1 KB)  TX bytes:14927 (14.9 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:23 errors:0 dropped:0 overruns:0 frame:0
          TX packets:23 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1920 (1.9 KB)  TX bytes:1920 (1.9 KB)

[04/06/2015 09:54] seed@ubuntu:~$
```

VM_1(10.0.2.15)

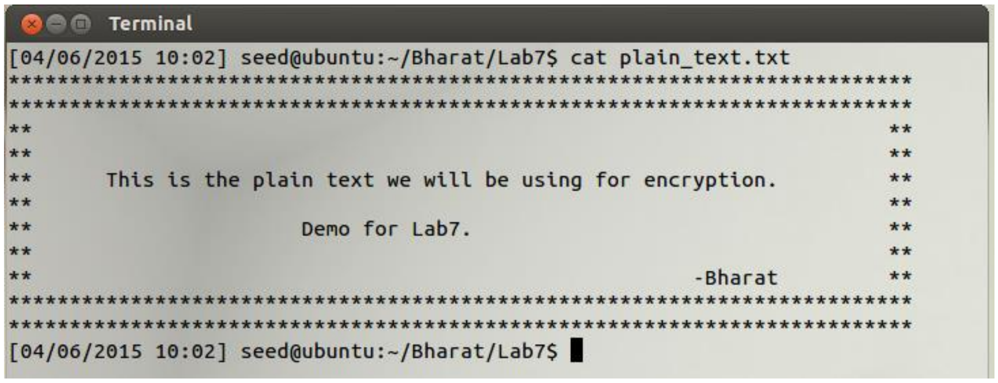
Installing OpenSSL:
OpenSSL is already downloaded into /seed/home/openssl-1.0.1, now we will configure it using the following commands:
$ /seed/openssl-1.0.1
$ ./config
$ make
$ make test
$ sudo make install
If we have /usr/local/ssl then our configuration ran successfully.

Installing a hex editor:
A hex editor called 'GHex' is already installed and configured in our pre built VMs.

3 Lab Tasks

3.1 Task 1: Generating Message Digest and MAC

For the purpose of this task we will use the following file

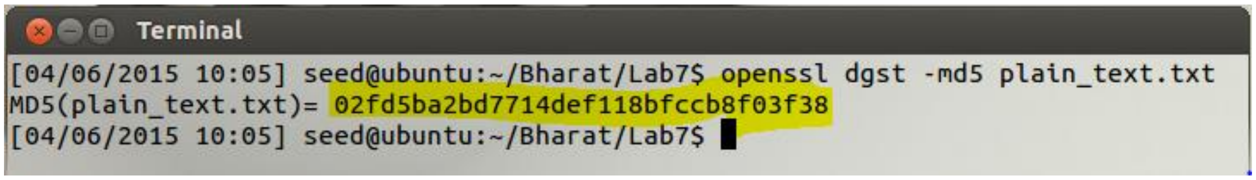


```
[04/06/2015 10:02] seed@ubuntu:~/Bharat/Lab7$ cat plain_text.txt
********************************************************************************
********************************************************************************
**                                                                            **
**                                                                            **
**      This is the plain text we will be using for encryption.              **
**                                                                            **
**                      Demo for Lab7.                                        **
**                                                                            **
**                                                          -Bharat           **
********************************************************************************
********************************************************************************
[04/06/2015 10:02] seed@ubuntu:~/Bharat/Lab7$
```

plain_text demo file.

Using md5 to generate hash:

For this task we will be using the following command to generate a hash value of the above file:
'openssl dgst –md5 plain_text.txt'

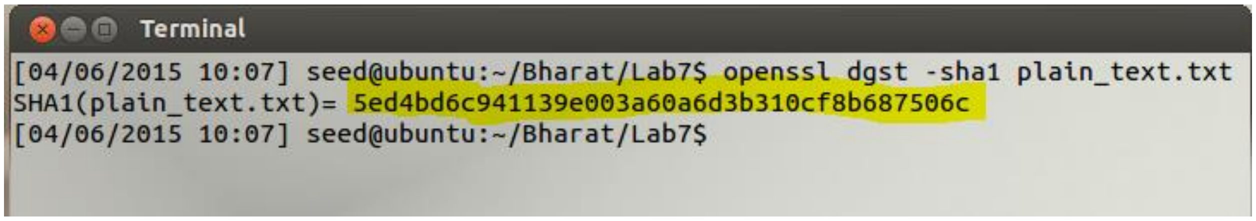


```
[04/06/2015 10:05] seed@ubuntu:~/Bharat/Lab7$ openssl dgst -md5 plain_text.txt
MD5(plain_text.txt)= 02fd5ba2bd7714def118bfccb8f03f38
[04/06/2015 10:05] seed@ubuntu:~/Bharat/Lab7$
```

md5 message digest (32 characters)

Using SHA1 to generate hash:

For this task we will be using the following command to generate a hash value of the above file:
'openssl dgst –sha1 plain_text.txt'

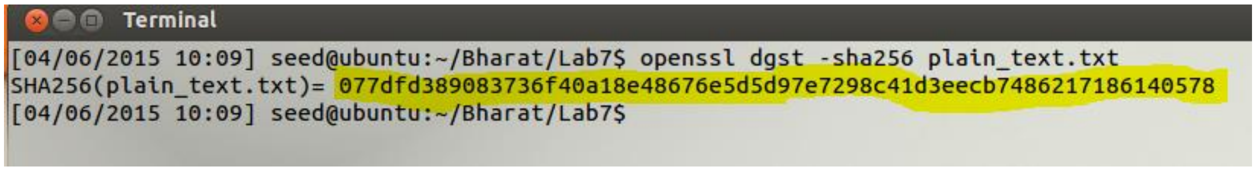


```
[04/06/2015 10:07] seed@ubuntu:~/Bharat/Lab7$ openssl dgst -sha1 plain_text.txt
SHA1(plain_text.txt)= 5ed4bd6c941139e003a60a6d3b310cf8b687506c
[04/06/2015 10:07] seed@ubuntu:~/Bharat/Lab7$
```

sha1 message digest (40 *characters*)

Using SHA256 to generate hash:

For this task we will be using the following command to generate a hash value of the above file:
'openssl dgst –sha256 plain_text.txt'



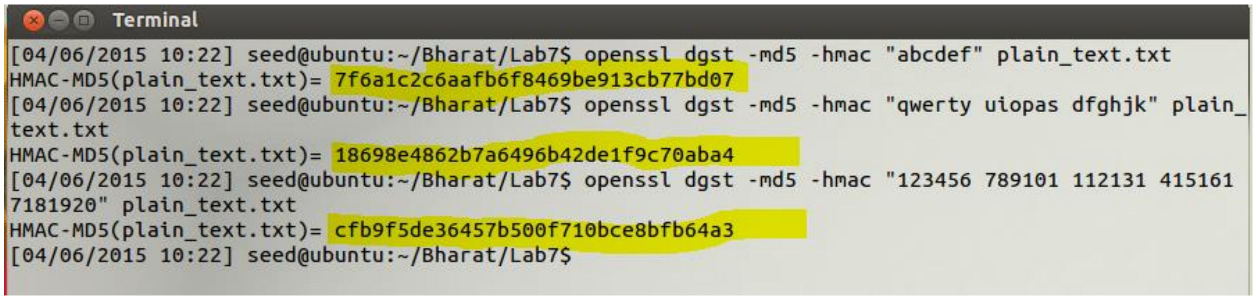
```
[04/06/2015 10:09] seed@ubuntu:~/Bharat/Lab7$ openssl dgst -sha256 plain_text.txt
SHA256(plain_text.txt)= 077dfd389083736f40a18e48676e5d5d97e7298c41d3eecb7486217186140578
[04/06/2015 10:09] seed@ubuntu:~/Bharat/Lab7$
```

sha256 message digest(64 characters)

Observation:
We can see that each hash generating algorithm gives us different number of bytes as output. Like md5 gives 32*4 = 128 bits, SHA1 gives 40*4 = 160 bits and sha256 gives 64*4 = 256 bit outputs.

3.2 Task 2: Keyed Hash and HMAC

HMAC using md5
For this task we will be using md5 message digest algorithm and 3 different key lengths with the following commands:
1. '*openssl dgst -md5 -hmac "abcdef" plain_text.txt*'
2. '*openssl dgst -md5 -hmac "qwerty uiopas dfghjk" plain_text.txt*'
3. '*openssl dgst -md5 -hmac "123456 789101 112131 415161 7181920 " plain_text.txt*'



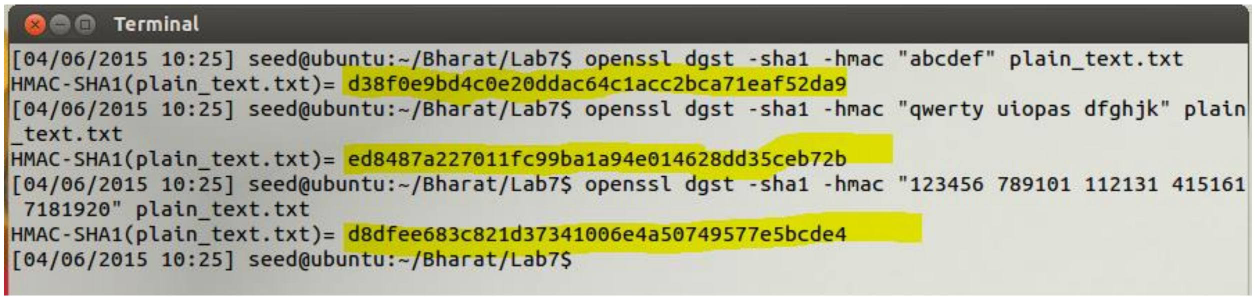
```
[04/06/2015 10:22] seed@ubuntu:~/Bharat/Lab7$ openssl dgst -md5 -hmac "abcdef" plain_text.txt
HMAC-MD5(plain_text.txt)= 7f6a1c2c6aafb6f8469be913cb77bd07
[04/06/2015 10:22] seed@ubuntu:~/Bharat/Lab7$ openssl dgst -md5 -hmac "qwerty uiopas dfghjk" plain_
text.txt
HMAC-MD5(plain_text.txt)= 18698e4862b7a6496b42de1f9c70aba4
[04/06/2015 10:22] seed@ubuntu:~/Bharat/Lab7$ openssl dgst -md5 -hmac "123456 789101 112131 415161
7181920" plain_text.txt
HMAC-MD5(plain_text.txt)= cfb9f5de36457b500f710bce8bfb64a3
[04/06/2015 10:22] seed@ubuntu:~/Bharat/Lab7$
```

hmac using md5

HMAC using SHA1

For this task we will be using SHA1 message digest algorithm and 3 different key lengths with the following commands:

1. *'openssl dgst –sha1 -hmac "abcdef" plain_text.txt'*
2. *'openssl dgst –sha1 -hmac "qwerty uiopas dfghjk" plain_text.txt'*
3. *'openssl dgst –sha1 -hmac "123456 789101 112131 415161 7181920 " plain_text.txt'*



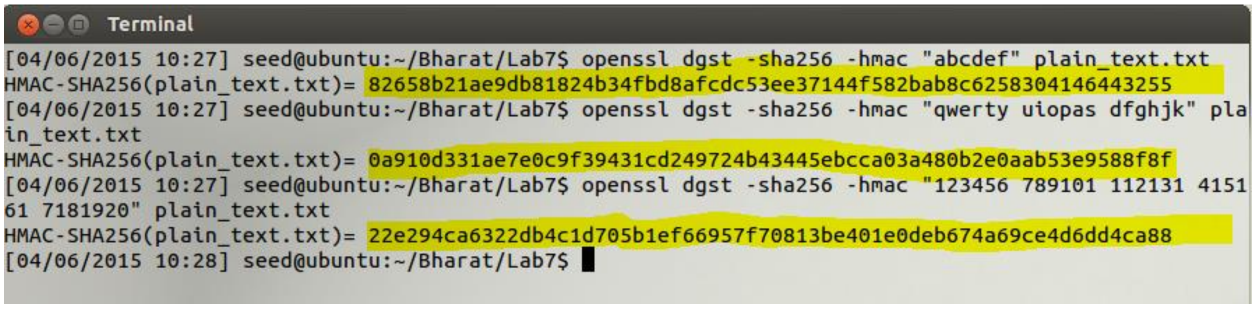
```
[04/06/2015 10:25] seed@ubuntu:~/Bharat/Lab7$ openssl dgst -sha1 -hmac "abcdef" plain_text.txt
HMAC-SHA1(plain_text.txt)= d38f0e9bd4c0e20ddac64c1acc2bca71eaf52da9
[04/06/2015 10:25] seed@ubuntu:~/Bharat/Lab7$ openssl dgst -sha1 -hmac "qwerty uiopas dfghjk" plain
_text.txt
HMAC-SHA1(plain_text.txt)= ed8487a227011fc99ba1a94e014628dd35ceb72b
[04/06/2015 10:25] seed@ubuntu:~/Bharat/Lab7$ openssl dgst -sha1 -hmac "123456 789101 112131 415161
 7181920" plain_text.txt
HMAC-SHA1(plain_text.txt)= d8dfee683c821d37341006e4a50749577e5bcde4
[04/06/2015 10:25] seed@ubuntu:~/Bharat/Lab7$
```

hmac using sha1

HMAC using SHA256

For this task we will be using SHA256 message digest algorithm and 3 different key lengths with the following commands:

1. *'openssl dgst –sha256 -hmac "abcdef" plain_text.txt'*
2. *'openssl dgst –sha256 -hmac "qwerty uiopas dfghjk" plain_text.txt'*
3. *'openssl dgst –sha256 -hmac "123456 789101 112131 415161 7181920 " plain_text.txt'*



```
[04/06/2015 10:27] seed@ubuntu:~/Bharat/Lab7$ openssl dgst -sha256 -hmac "abcdef" plain_text.txt
HMAC-SHA256(plain_text.txt)= 82658b21ae9db81824b34fbd8afcdc53ee37144f582bab8c6258304146443255
[04/06/2015 10:27] seed@ubuntu:~/Bharat/Lab7$ openssl dgst -sha256 -hmac "qwerty uiopas dfghjk" pla
in_text.txt
HMAC-SHA256(plain_text.txt)= 0a910d331ae7e0c9f39431cd249724b43445ebcca03a480b2e0aab53e9588f8f
[04/06/2015 10:27] seed@ubuntu:~/Bharat/Lab7$ openssl dgst -sha256 -hmac "123456 789101 112131 4151
61 7181920" plain_text.txt
HMAC-SHA256(plain_text.txt)= 22e294ca6322db4c1d705b1ef66957f70813be401e0deb674a69ce4d6dd4ca88
[04/06/2015 10:28] seed@ubuntu:~/Bharat/Lab7$ █
```

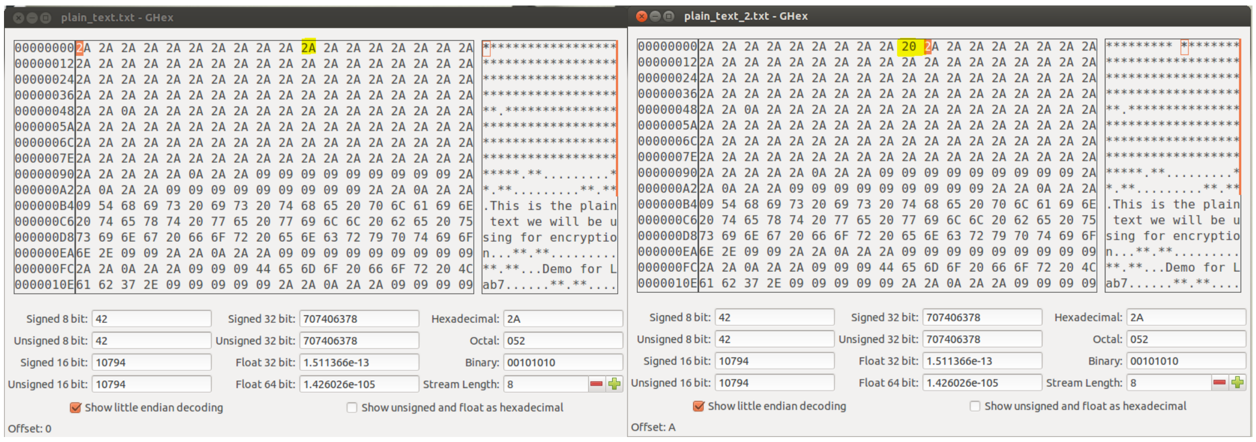
hmac using sha256

Observation:

I used three key sizes with varied lengths and were either less than 64 bytes or greater than 64 bytes. Irrespective of the key size I used the output hash for md5 was 128 bits, for sha1 it was 160 bits and for sha 256 it was 256 bits. So we can say that we do not need any fixed key size for HMAC to work. However if the key size is greater than the hash length then the input is appended before hashed to get the respective hash length.

3.3 Task 3: The Randomness of One-way Hash

This task requires us to generate the hash of a file, then flip one bit in it, generate a hash again and then compare the two hash results generated.
We created two identical files namely plain_text.txt and plain_text_2.txt and then flipped one bit in plain_text_2.txt as shown using Ghex.



```
00000000 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A  ******************
00000012 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A  ******************
00000024 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A  ******************
00000036 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A  ******************
00000048 2A 2A 0A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A  **.***************
0000005A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A  ******************
0000006C 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A  ******************
0000007E 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A  ******************
00000090 2A 2A 2A 2A 2A 0A 2A 2A 09 09 09 09 09 09 09 09 09 2A  *****.**.........*
000000A2 2A 0A 2A 2A 09 09 09 09 09 09 09 09 09 2A 2A 0A 2A 2A  *.**.........**.**
000000B4 09 54 68 69 73 20 69 73 20 74 68 65 20 70 6C 61 69 6E  .This is the plain
000000C6 20 74 65 78 74 20 77 65 20 77 69 6C 6C 20 62 65 20 75   text we will be u
000000D8 73 69 6E 67 20 66 6F 72 20 65 6E 63 72 79 70 74 69 6F  sing for encryptio
000000EA 6E 2E 09 09 2A 2A 0A 2A 2A 09 09 09 09 09 09 09 09 09  n...**.**.........
000000FC 2A 2A 0A 2A 2A 09 09 09 44 65 6D 6F 20 66 6F 72 20 4C  **.**...Demo for L
0000010E 61 62 37 2E 09 09 09 09 09 2A 2A 0A 2A 2A 09 09 09 09  ab7......**.**....

Signed 8 bit: 42          Signed 32 bit: 707406378      Hexadecimal: 2A
Unsigned 8 bit: 42        Unsigned 32 bit: 707406378    Octal: 052
Signed 16 bit: 10794      Float 32 bit: 1.511366e-13    Binary: 00101010
Unsigned 16 bit: 10794    Float 64 bit: 1.426026e-105   Stream Length: 8
[✓] Show little endian decoding        [ ] Show unsigned and float as hexadecimal
Offset: 0
```

```
00000000 2A 2A 2A 2A 2A 2A 2A 2A 2A 20 2A 2A 2A 2A 2A 2A 2A 2A  ********* ********
00000012 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A  ******************
00000024 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A  ******************
00000036 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A  ******************
00000048 2A 2A 0A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A  **.***************
0000005A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A  ******************
0000006C 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A  ******************
0000007E 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A  ******************
00000090 2A 2A 2A 2A 2A 0A 2A 2A 09 09 09 09 09 09 09 09 09 2A  *****.**.........*
000000A2 2A 0A 2A 2A 09 09 09 09 09 09 09 09 09 2A 2A 0A 2A 2A  *.**.........**.**
000000B4 09 54 68 69 73 20 69 73 20 74 68 65 20 70 6C 61 69 6E  .This is the plain
000000C6 20 74 65 78 74 20 77 65 20 77 69 6C 6C 20 62 65 20 75   text we will be u
000000D8 73 69 6E 67 20 66 6F 72 20 65 6E 63 72 79 70 74 69 6F  sing for encryptio
000000EA 6E 2E 09 09 2A 2A 0A 2A 2A 09 09 09 09 09 09 09 09 09  n...**.**.........
000000FC 2A 2A 0A 2A 2A 09 09 09 44 65 6D 6F 20 66 6F 72 20 4C  **.**...Demo for L
0000010E 61 62 37 2E 09 09 09 09 09 2A 2A 0A 2A 2A 09 09 09 09  ab7......**.**....

Signed 8 bit: 42          Signed 32 bit: 707406378      Hexadecimal: 2A
Unsigned 8 bit: 42        Unsigned 32 bit: 707406378    Octal: 052
Signed 16 bit: 10794      Float 32 bit: 1.511366e-13    Binary: 00101010
Unsigned 16 bit: 10794    Float 64 bit: 1.426026e-105   Stream Length: 8
[✓] Show little endian decoding        [ ] Show unsigned and float as hexadecimal
Offset: A
```

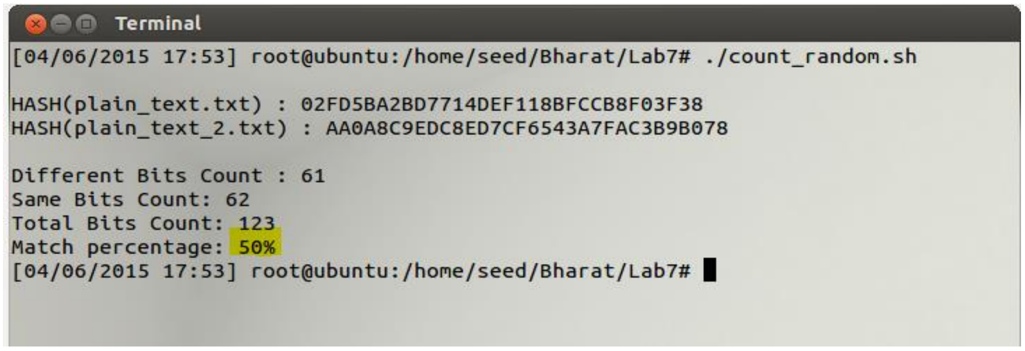
flipped selected bit.

Now we will be using the following shell program to compare the hash values:

```
#!/bin/bash
FILE1="plain_text.txt"
FILE2="plain_text_2.txt"

DIGEST="md5"

declare -u HASH1
declare -u HASH2
HASH1=`openssl dgst -$DIGEST $FILE1 | awk '{print $2}'`
HASH2=`openssl dgst -$DIGEST $FILE2 | awk '{print $2}'`
echo ""
echo "HASH($FILE1) : $HASH1"
echo "HASH($FILE2) : $HASH2"
echo ""

BITS1=`echo "ibase=16;obase=2;$HASH1" | bc`
BITS2=`echo "ibase=16;obase=2;$HASH2" | bc`
BITS1=`echo "$BITS1" | sed 's/\\\r*//g'`
BITS2=`echo "$BITS2" | sed 's/\\\r*//g'`
LEN=`expr length "$BITS1"`
declare -i B1
declare -i B2
DIFFCOUNT=0
MATCHCOUNT=0
for i in `seq 1 $LEN`
do
  B1=$(echo $BITS1 | awk "{print substr (\$0, $i, 1)}")
  B2=$(echo $BITS2 | awk "{print substr (\$0, $i, 1)}")
  if [ "$B1" -ne "$B2" ]
        then
    DIFFCOUNT=`expr $DIFFCOUNT + 1`
        else
                MATCHCOUNT=`expr $MATCHCOUNT + 1`
```

```
   fi
done
TOTALCOUNT=`expr $DIFFCOUNT + $MATCHCOUNT`
echo "Different Bits Count : $DIFFCOUNT"
echo "Same Bits Count: $MATCHCOUNT"
echo "Total Bits Count: $TOTALCOUNT"
echo "Match percentage: $(($MATCHCOUNT * 100 / $TOTALCOUNT))%"
```

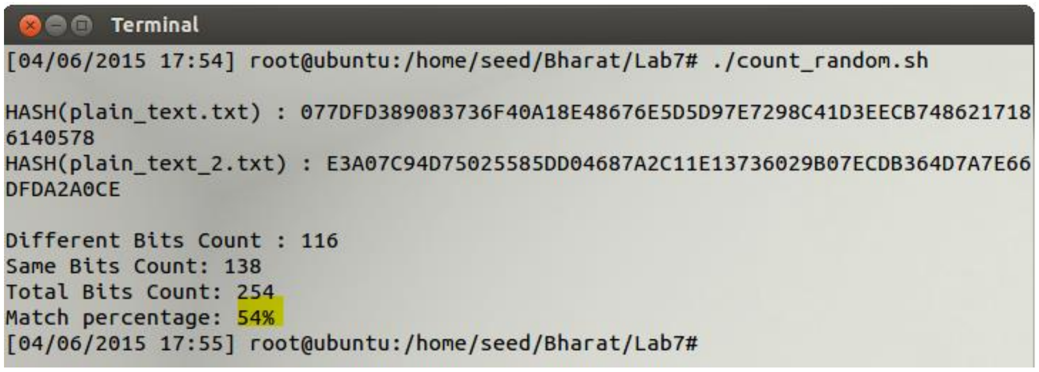
Now let us see the program in action:



```
[04/06/2015 17:53] root@ubuntu:/home/seed/Bharat/Lab7# ./count_random.sh

HASH(plain_text.txt) : 02FD5BA2BD7714DEF118BFCCB8F03F38
HASH(plain_text_2.txt) : AA0A8C9EDC8ED7CF6543A7FAC3B9B078

Different Bits Count : 61
Same Bits Count: 62
Total Bits Count: 123
Match percentage: 50%
[04/06/2015 17:53] root@ubuntu:/home/seed/Bharat/Lab7#
```

md5 randomness check

We can see that there is almost a 50% match in the hash bits generated. Now let us do the same using sha256 and check the result. We will be using the same program as above but in line3 change the mode from md5 to sha256.



```
[04/06/2015 17:54] root@ubuntu:/home/seed/Bharat/Lab7# ./count_random.sh

HASH(plain_text.txt) : 077DFD389083736F40A18E48676E5D5D97E7298C41D3EECB748621718
6140578
HASH(plain_text_2.txt) : E3A07C94D75025585DD04687A2C11E13736029B07ECDB364D7A7E66
DFDA2A0CE

Different Bits Count : 116
Same Bits Count: 138
Total Bits Count: 254
Match percentage: 54%
[04/06/2015 17:55] root@ubuntu:/home/seed/Bharat/Lab7#
```

sha256 randomness check

Observation:
We can see that when we use sha256 and compare the hashes of the original file and a file with just one bit flipped we can see that the percentage of match is at 54%. AS we compare both the match percentages we find that the match has come done.

3.4 Task 4: One-Way Property versus Collision-Free Property
For checking this we shall be using the following program:

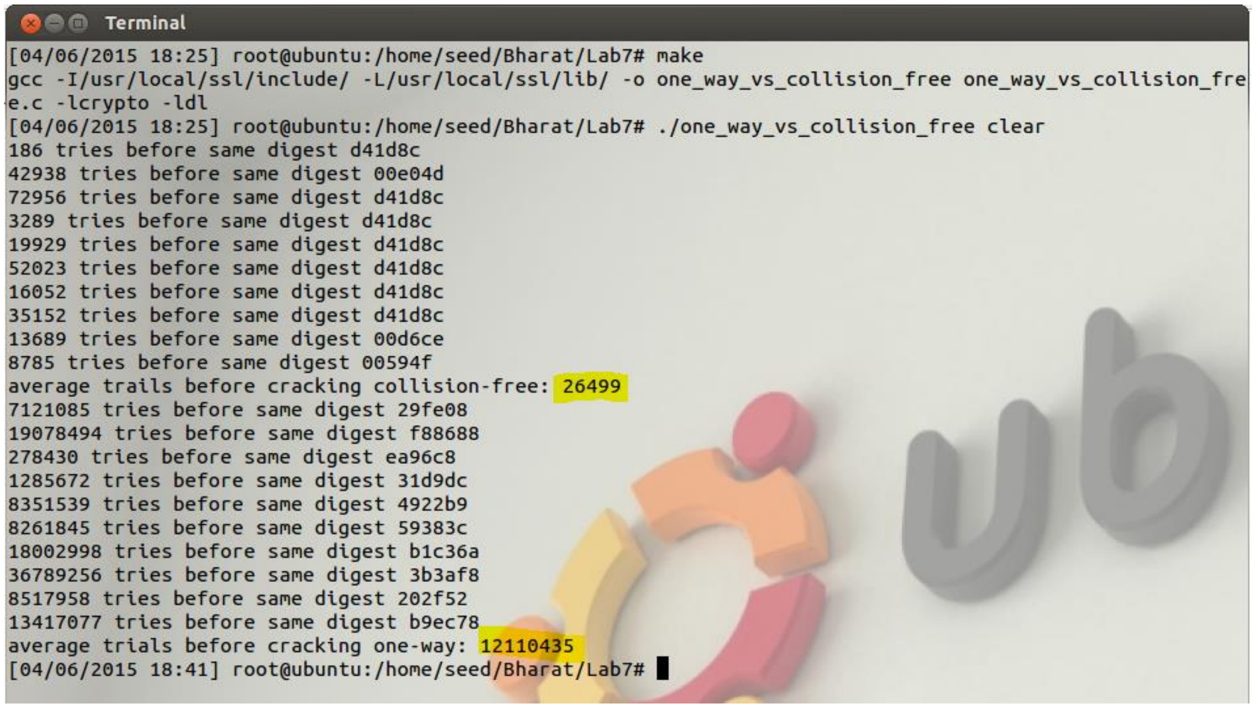
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <openssl/evp.h>
#include <unistd.h>
#define LEN 16 // 128 bits

void getHash(char * hashname, char *msg, unsigned char *md_value) {
        EVP_MD_CTX *mdctx;
        const EVP_MD *md;
        int md_len, i;
        OpenSSL_add_all_digests();
        md = EVP_get_digestbyname(hashname);
        if(!md) {
                printf("Unknown message digest %s\n", hashname);
                exit(1);
        }
        mdctx = EVP_MD_CTX_create();
        EVP_DigestInit_ex(mdctx, md, NULL);
        EVP_DigestUpdate(mdctx, msg, strlen(msg));
        EVP_DigestFinal_ex(mdctx, md_value, &md_len);
        EVP_MD_CTX_destroy(mdctx);
}
void setRndStr(char *msg) {
        int i;
//1234 1234 1234 = 3bytes  = 24 bits
for (i=0;i<11;i++)
                msg[i] = rand()%256-128;
}

int BruteForceOneWay(char * hashname) {
        char msg1[11], msg2[11];
        unsigned char digt1[EVP_MAX_MD_SIZE], digt2[EVP_MAX_MD_SIZE];
        int count=0, i;
        setRndStr(msg1);
        getHash(hashname, msg1, digt1);
        do {
                                setRndStr(msg2);
                getHash(hashname, msg2, digt2);
                count++;
        } while (strncmp(digt1, digt2, 3)!=0);
        printf("%d tries before same digest ", count);
        for(i = 0; i < 3; i++) printf("%02x", digt1[i]);
        printf("\n");
        return count;
}
int BruteForceCollision(char * hashname) {
        char msg1[11], msg2[11];
        unsigned char digt1[EVP_MAX_MD_SIZE], digt2[EVP_MAX_MD_SIZE];
        int count=0, i;
        do {
                setRndStr(msg1);
                getHash(hashname, msg1, digt1);
                setRndStr(msg2);
                getHash(hashname, msg2, digt2);
                count++;
        } while (strncmp(digt1, digt2, 3)!=0);
        printf("%d tries before same digest ", count);
        for(i = 0; i < 3; i++) printf("%02x", digt1[i]);
        printf("\n");
        return count;
```

```
}
main(int argc, char *argv[])
{
        char *hashname;
        unsigned char *key = (unsigned char *) malloc(sizeof(unsigned char)*LEN);
        FILE* random = fopen("/dev/urandom", "r");
        fread(key, sizeof(unsigned char)*LEN, 1, random);
        hashname = "md5";
        srand(atoi(key));
        int i,BFCcount,OWCcount;
        for (i=0,BFCcount=0;i<10;i++)
                BFCcount+=BruteForceCollision(hashname);
        printf("average trails before cracking collision-free: %d \n", BFCcount/10);

        for (i=0,OWCcount=0;i<10;i++)
                OWCcount+=BruteForceOneWay(hashname);
        printf("average trials before cracking one-way: %d \n", OWCcount/10);
        fclose(random);
}
```

The makefile contents are:

```
INC=/usr/local/ssl/include/
LIB=/usr/local/ssl/lib/
all:
        gcc -I$(INC) -L$(LIB) -o one_way_vs_collision_free one_way_vs_collision_free.c -lcrypto -ldl
```

Now let us see the program in action:



```
[04/06/2015 18:25] root@ubuntu:/home/seed/Bharat/Lab7# make
gcc -I/usr/local/ssl/include/ -L/usr/local/ssl/lib/ -o one_way_vs_collision_free one_way_vs_collision_fre
e.c -lcrypto -ldl
[04/06/2015 18:25] root@ubuntu:/home/seed/Bharat/Lab7# ./one_way_vs_collision_free clear
186 tries before same digest d41d8c
42938 tries before same digest 00e04d
72956 tries before same digest d41d8c
3289 tries before same digest d41d8c
19929 tries before same digest d41d8c
52023 tries before same digest d41d8c
16052 tries before same digest d41d8c
35152 tries before same digest d41d8c
13689 tries before same digest 00d6ce
8785 tries before same digest 00594f
average trails before cracking collision-free: 26499
7121085 tries before same digest 29fe08
19078494 tries before same digest f88688
278430 tries before same digest ea96c8
1285672 tries before same digest 31d9dc
8351539 tries before same digest 4922b9
8261845 tries before same digest 59383c
18002998 tries before same digest b1c36a
36789256 tries before same digest 3b3af8
8517958 tries before same digest 202f52
13417077 tries before same digest b9ec78
average trials before cracking one-way: 12110435
[04/06/2015 18:41] root@ubuntu:/home/seed/Bharat/Lab7#
```

Bruteforce collision-free vs One-way

Observation:

The average number of trials brute force had to run to crack collision-free property is: **26499.**

The average number of trials brute force had to run to crack one-way hash property is: **12110435.**

Based on the observation we have we can say that it is considerably difficult to crack one-way property on comparison with collision free property.

Difference between oneway hash and collision free property:
There are three classical properties for a good cryptographic hash function:

- **Resistance to preimages:** given *x*, it should be infeasible to find *m* such that *h(m) = x*. In simple terms we are given the output and the challenge is to find a matching input.
- **Resistance to second preimages:** given *m*, it should be infeasible to find *m'* distinct from *m*, such that *h(m) = h(m')*. In simple terms we are given the output and an input the challenge is to find another matching input.
- **Resistance to collisions:** it should be infeasible to find *m* and *m'*, distinct from each other, such that *h(m) = h(m')*. Also can be said as given a output and an input the challenge is to find any matching output.

Without exploiting any weakness in the hash function itself, the generic methods for finding preimages, second preimages, and collisions, for a hash function with a *n*-bit output, have cost roughly 2^n (for preimages and second preimages) and $2^{n/2}$ (for collisions). Finding collisions is thus vastly easier. For preimages, when using the brute force, each try has probability 2^{-n} to succeed.
Collisions is similar to the birthday attack (discussed in the class and is if you take 20 people at random, chances are high that two of them will have the same birthday. On the other hand, if you want to find someone with the same birthday than *you*, then you will have to sample through 365 people on average basically, once you have accumulated about $sqrt(2^n)$ pairs input/output, the probability of two of those pairs having the same output rises quite fast.
Base Line is that for brute force success for One way: 2^n and that for Collision free is $2^{n/2}$.

Reference for last question:
http://cacr.uwaterloo.ca/hac/about/chap9.pdf
http://crypto.stackexchange.com/questions/8773/difference-between-one-way-function-and-cryptographic-hash-function