HW 5 Report

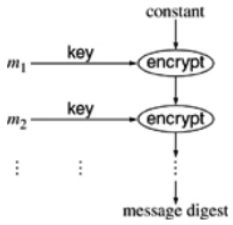
Paper-and-Pencil Problems:

5.3 In §5.1 Introduction we discuss the devious secretary Bob having an automatic means of generating many messages that Alice would sign, and many messages that Bob would like to send. By the birthday problem, by the time Bob has tried a total of 2^{32} messages, he will probably have found two with the same message digest. The problem is, both may be of the same type, which would not do him any good. How many messages must Bob try before it is probable that he'll have messages with matching digests, and that the messages will be of opposite types?

---Basically the same as birthday attack, suppose we have n messages, k possible hash results. There are $\frac{n(n-1)}{2}$ possible pairs of messages and a total probability of $\frac{n(n-1)}{2} \times \frac{1}{k}$ that a pair have the same hash result. There is also $\frac{1}{2}$ probability that the pair of messages are from different original types. Therefore, to make $\frac{n(n-1)}{2} \times \frac{1}{k} \times \frac{1}{2}$ greater than 50%, there will be a great chance when $n > \sqrt{2k}$.

5.4 In §5.2.4.2 Hashing Large Messages, we described a hash algorithm in which a constant was successively encrypted with blocks of the message. We showed that you could find two messages with the same hash value in about 2^{32} operations. So we suggested doubling the hash size by using the message twice, first in forward order to make up the first half of the hash, and then in reverse order for the second half of the hash. Assuming a 64-bit encryption block, how could you find two messages with the same hash value in about 2^{32} iterations? Hint: consider blockwise palindromic messages.

---Suppose we have a palindromic message ABCDDCBA, A is m_1, B is m_2, etc. Using the hashing algorithm above will lead to the same 64-bit hash result for the first half and second half. Therefore, for palindromic messages, if we can find two messages with the same first 64-bit half hash value, then we find two messages with the same hash. According to birthday attack, to find two palindromic messages with the same 64-bit hash result, we only need about 2^{32} messages.



5.14 For purposes of this exercise, we will define random as having all elements equally likely to be chosen. So a function that selects a 100-bit number will be random if every 100-bit number is equally likely to be chosen. Using this definition, if we look at the function "+" and we have two inputs, x and y, then the output will be random if at least one of x and y are random. For instance, y can always be 51, and yet the output will be random if x is random. For the following functions, find sufficient conditions for x, y, and z under which the output will be random:

$\sim x$

$x \oplus y$

$x \vee y$

$x \wedge y$

$(x \wedge y) \vee (\sim x \wedge z)$ [the selection function]

$(x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$ [the majority function]

$x \oplus y \oplus z$

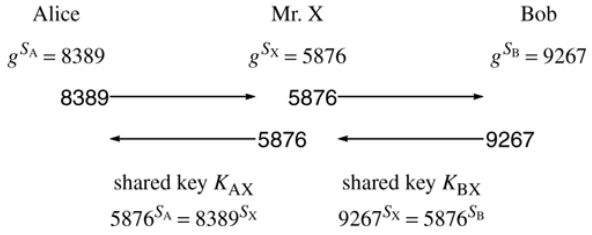
$y \oplus (x \vee -z)$

---The sufficient condition for randomness for the operations above is:

(1) ~x: x is random
(2) x XOR y: x or y is random
(3) x ᵛ y: x and y are random
(4) x ^ y: x and y are random
(5) (x ^ y) ᵛ (~x ^ z): y and z are random
(6) (x ^ y) ᵛ (x ^ z) ᵛ (y ^ z): x and y are random (any two of the three are random is OK)
(7) x XOR y XOR z: x or y or z is random
(8) y XOR (x ^ -z): y is random

6.2 In section §6.4.2 Defenses Against Man-in-the-Middle Attack, it states that encrypting the Diffie-Hellman value with the other side's public key prevents the attack. Why is this the case, given that an attacker can encrypt whatever it wants with the other side's public key?

---Encrypting the Diffie-Hellman value with the other side's public key is the method for Authenticated Diffie-Hellman. For easier explanation, let us take the example in KPS book:



When Alice sent encrypted g^{S_A} to Bob, although it is intercepted by Mr. X, because Mr. X does not know the private key of Bob, there is no way for Mr. X to decrypt the intercepted message and get the value of g^{S_A}. Therefore, the shared key K_{AX} cannot be calculated by Mr. X. Although Mr. X can send whatever it wants encrypted with Bob's public key to Bob. However, Bob will send back his g^{S_B} encrypted with A's public key. Mr. X cannot calculate K_{BX} as well. So MITM attack is prevented.

6.8 Suppose Fred sees your RSA signature on m_1 and on m_2 (i.e. he sees m_1^d mod n and m_2^d mod n). How does he compute the signature on each of m_1^j mod n (for positive integer j), m_1^{-1} mod n, $m_1 \cdot m_2$ mod n, and in general $m_1^j m_2^k$ mod n (for arbitrary integers j and k)?

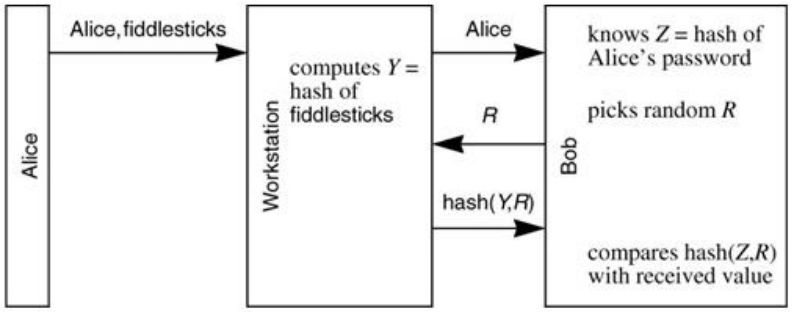
--- The signature on m1^j mod n for any j positive integer can be computed by (m1^d)^j mod n = (m1^j)^d mod n which is what we wanted.

The signature of m1^(-1) is (m1^(-1))^(d) mod n = (m1^(-d)) mod n = (m1^d)^(-1) mod n. We can compute it by finding the multiplicative inverse of m1^d mod n via Euclidean Algorithm.

The signature of m1*m2 is (m1*m2)^d mod n = ((m1)^d mod n) *(m2^d mod n) mod n, which can be computed.

We know that if j <0 j=-1(-j) where -j >0. Hence, we can compute the signature of m1^j by computing the signature of m1^(-j) first, using (a) and then compute the signature of m1^(j) from m1^(-j) using (b). Similarly, we can computer the signature of m2^k for any k. Since we can computer the signature of m1^j and m2^k, we can apply (c) to compute the signature of the product.

9.2 In §9.6 Eavesdropping and Server Database Reading we asserted that it is extremely difficult, without public key cryptography, to have an authentication scheme which protects against both eavesdropping and server database disclosure. Consider the following authentication protocol (which is based on Novell version 3 security). Alice knows a password. Bob, a server that will authenticate Alice, stores a hash of Alice's password. Alice types her password (say fiddlesticks) to her workstation. The following exchange takes place:



Is this an example of an authentication scheme that isn't based on public key cryptography and yet guards against both eavesdropping and server database disclosure?

---Yes. It is a hash based authentication scheme that successfully prevents eavesdropping and server database disclosure. For eavesdropping, since the hash of random R and hash of Alice's password is transmitted between workstation and server instead of Alice's password plaintext, eavesdropping is prevented. For server database reading, since server bob only stores the hash of Alice's password, not her plaintext password, attacker will still know nothing about Alice's password. The whole authentication process does not involve public/private key of Alice.

10.1 Design a password hash algorithm with the property stated in Password Hash Quirk on page 242. It should be impossible to reverse, but for any string S it should be easy to find a longer string with the same hash.

Lab and Programming Tasks:

3.1 Task 1: Generating Message Digest and MAC:

(1) MD5:

```
root@seed-desktop:/home/seed/Desktop# openssl dgst -md5 plaintext
MD5(plaintext)= 8c9af68744a41e8240b6af86b9a705ce
```

Cannot be used in serious applications because of its weakness.

The output of MD5 is 16-byte hash value.

(2) SHA1:

```
root@seed-desktop:/home/seed/Desktop# openssl dgst -sha1 plaintext
SHA1(plaintext)= ff7a9c0009aa90b78d016f01073fb5bcf8c1e55f
```

Broken but not yet cracked.

The output of SHA-1 is 20-byte hash value.

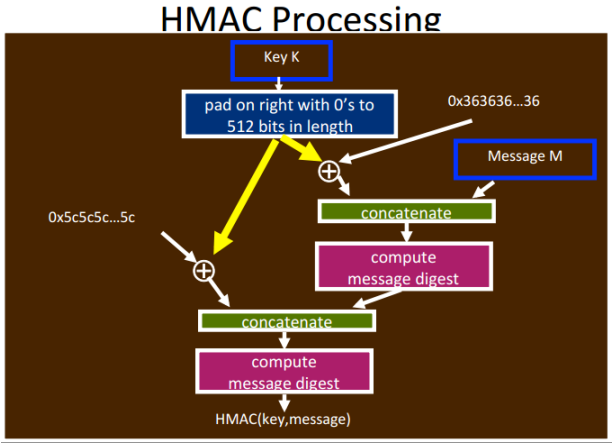
(3) SHA256:

```
root@seed-desktop:/home/seed/Desktop# openssl dgst -sha256 plaintext
SHA256(plaintext)= 2ca837eb2a867c31ade632429ae140a00d90b127b9c8f8cd5875730889dfc
597
```

The output of SHA-256 is 32-byte hash value.

3.2 Task 2: Keyed Hash and HMAC:

```
root@seed-desktop:/home/seed/Desktop# openssl dgst -md5 -hmac "abcdefg" plaintext
HMAC-MD5(plaintext)= 40e53673a6a324cdc73db3e8db079210
root@seed-desktop:/home/seed/Desktop# openssl dgst -md5 -hmac "abcdefghijklmn" plaintext
HMAC-MD5(plaintext)= 71c58c59db54d4cf16a336f7fa00d668
root@seed-desktop:/home/seed/Desktop# openssl dgst -sha1 -hmac "abcdefg" plaintext
HMAC-SHA1(plaintext)= c58f23e1256163ff862bbc4c5b13d05b12b72d66
root@seed-desktop:/home/seed/Desktop# openssl dgst -sha1 -hmac "abcdefghijklmn" plaintext
HMAC-SHA1(plaintext)= 4c66e254c3b15a31b07c1e13e21eca3eae0f1d7f
root@seed-desktop:/home/seed/Desktop# openssl dgst -sha256 -hmac "abcdefg" plaintext
HMAC-SHA256(plaintext)= 7a864277f88d9a803e7063c0abc9b1657372e365ac7b866d1266a9b7770d7c63
root@seed-desktop:/home/seed/Desktop# openssl dgst -sha256 -hmac "abcdefghijklmn" plaintext
HMAC-SHA256(plaintext)= 7229eff34e35e567343a2bc2d803b89882930469239 76d22a5f09b0d611372fb
```



From the above result, we can clearly see that it is not required to use a fixed length key in HMAC. When the key length is less than 512 bits, just like my key here, system will pad on right with 0s to 512 bits in length. If the key length is more than 512 bits, the system will hash the key first (for example, system may hash the key with SHA512 to output 512 bits result) and then use the hash result as the key for operations afterwards.

```
function hmac (key, message)
    if (length(key) > blocksize) then
        key = hash(key) // keys longer than blocksize are shortened
    end if
    if (length(key) < blocksize) then
        key = key ∥ [0x00 * (blocksize - length(key))] // keys shorter than blocksize are zero-padded ('∥' is concatenation)
    end if
```

3.3 Task 3: The Randomness of One-way Hash:

The hash value of the original message:

```
root@seed-desktop:/home/seed/Desktop# openssl dgst -sha1 plaintext
SHA1(plaintext)= ff7a9c0009aa90b78d016f01073fb5bcf8c1e55f
```

After flipping one bit in the original message with Ghex:

```
root@seed-desktop:/home/seed/Desktop# openssl dgst -sha1 plaintext
SHA1(plaintext)= 8c6693deea4cc211f0af0ec69166af30fb261381
```

From the result above, we can clearly see the hash result is completely different after one bit is flipped in the original message.

Program Snippet (JAVA) to count same bit in the SHA1 results:

```
public static int Count( StringBuffer before, StringBuffer after )
{
    int count = 0;

    if( (before.length() % 2 != 0) || (after.length() % 2 != 0) )
    {
        System.out.println( "Parameter length fault!" );
        System.exit(1);
    }

    int i = 0;
    while( i < before.length() )
    {
        StringBuffer a = new StringBuffer();
        StringBuffer b = new StringBuffer();

        a.append( before.charAt( i ) );
        a.append( before.charAt( i + 1 ) );
        b.append( after.charAt( i ) );
        b.append( after.charAt( i + 1 ) );

        int m = Integer.parseInt( a.toString(),  16 );
        int n = Integer.parseInt( b.toString(),  16 );

        for( int mark = 128; mark > 0; mark >>= 1 )
        {
            if( (m & mark ) == (n & mark) )
                count += 1;

        }
        i+=2;
    }

    return count;
}

public static void main(String[] args) {
    StringBuffer before = new StringBuffer("ff7a9c0009aa90b78d016f01073fb5bcf8c1e55f");
    StringBuffer after = new StringBuffer("8c6693deea4cc211f0af0ec69166af30fb261381");
    System.out.println( "The number of same bits is " + Count( before, after ) );
}
}
```

After running the program, it tells there are 72 same bits in the two hash results. In total, there are 128 bits. Therefore, about half of the bits is different after flipping one bit in the original message. This lab result verifies the randomness property of one-way hash.

3.4 Task 4: Hash Collision-Free Property:

To realize the generation of different messages and recording of how many messages I have tried at the same time, I choose original message as “Test Message ”. For each iteration i, simply append “i” to the original message to obtain a new message.

1.

In comparison with part 2 mentioned later, part 1 is trickier because it is necessary to record the hash result of all the messages I have tried.

Code Snippet:

```
while(1){
    sprintf( var, "%d", count );
    mdctx = EVP_MD_CTX_create();
    EVP_DigestInit_ex(mdctx, md, NULL);

    char mess1[MESS_SIZE] = "Test Message ";
    strcat( mess1, var );
    strcat( mess1, "\n" );
    EVP_DigestUpdate(mdctx, mess1, strlen(mess1));
    EVP_DigestFinal_ex(mdctx, md_value, &md_len);

    for(i = 0; i < 3; i++)
    {   q[i] = md_value[i]; printf("%02x\n", q[i]); }

    EVP_MD_CTX_destroy(mdctx);

    for( i = 0; i < count; i ++ )
    {
        if( (p[i][0] == q[0]) && (p[i][1] == q[1]) && (p[i][2] == q[2]) )
        {
            printf( "~~~~~~~~~~~~~~~~~\n" );
            printf( "i is %d\n", i );
            printf( "count is %d\n", count );
            printf( "Found!\n" );
            printf( "~~~~~~~~~~~~~~~~~\n" );
            return 1;
        }
    }

            printf( "The hash of q is %s\n", q );
    strcpy( p[count],q );
    count ++;
```

After an average 5377 trials, I can find two messages with the same hash value using brute-force method.

2.

Part 2 is easier to implement in comparison with part 1. We just need to fix an original message and obtain its hash value as the target. After obtaining the hash value of following messages, I compare it with the target hash value until they are same. It is unnecessary to record every hash value in a two dimensional array any more.

Code Snippet:

```
original_hash = getHash( original_mess, argv[1] );

while(1){
    sprintf( var, "%d", count );
    mdctx = EVP_MD_CTX_create();
    EVP_DigestInit_ex(mdctx, md, NULL);

    char mess1[MESS_SIZE] = "Test Message ";
    strcat( mess1, var );
    strcat( mess1, "\n" );
    EVP_DigestUpdate(mdctx, mess1, strlen(mess1));
    EVP_DigestFinal_ex(mdctx, md_value, &md_len);

    EVP_MD_CTX_destroy(mdctx);

    if( ( original_hash[0] == md_value[0]) && (original_hash[1] == md_value[1]) && (original_hash[2] == md_value[2]) )
    {
        printf( "~~~~~~~~~~~~~~~~~\n" );
        printf( "Trial: %d\n", count );
        printf( "Found!\n" );
        printf( "~~~~~~~~~~~~~~~~~\n" );
        return 1;
    }
    count ++;
}
```

After an average 10562518 trials, I can find a messages with the same hash value as the target message using brute-force method.

3.

According to the lab result, finding out two messages with the same hash value is easier. A mathematical proof will be given in part 4. Here, I will use the conclusion in advance. Since our hash result is 24-bit. Finding out two messages with the same hash value generally needs $2^{12} = 4096 \approx 5377$. While finding out another message with the same hash value as the target message generally needs $2^{24} = 16777216$, which is in the same order of magnitude as my result 10562518.

4.

Proof:

Suppose the hash result is a k-bit value. There are a total number of 2^k possible hash values. According to the property of randomness of hash function, given a message, the probability of achieving each hash result is of equal. Given a hash value, the probability to find another message with the same hash value is $\frac{1}{2^k}$. Therefore, in general, we need to try 2^k messages to find one with the same hash as the given message.

However, suppose we need n messages to find two messages with the same hash result. There will be C_n^2 message pairs. For each message pair, the probability of having same hash value is $\frac{1}{2^k}$ (No matter what hash value the first one has, the second one having the same hash value as the first one is $\frac{1}{2^k}$). To make $C_n^2 \times \frac{1}{2^k} = 1$, we will find out when $n \approx 2^{32}$, we are very likely to find two messages with the same hash value!

So finding out two messages with the same hash value is much easier.

3.5 Task 5: Performance Comparison: RSA versus AES:

How to obtain RSA public & private key pairs:

Generating private key

```
openssl genrsa -out private.pem 1024
```

Extract public key from the public key from private.pem

```
openssl rsa -in private.pem -out public.pem -outform PEM -pubout
```

Encryption with public key:

openssl rsautl –encrypt –inkey public.pem –pubin –in message –out Enmessage.ssl

Decryption with private key:

Encryption with 128-bit AES key (key is “NiceDay”):

openssl enc –aes-128-cbc –e –in message –out aesEncryp.bin –k NiceDay

Since one-time encryption and decryption is very fast, I include relevant commands in a “while” loop in pkEncry.sh, pkDecry.sh and aesEncryp.sh respectively (shell files), repeat it for 1000 time to obtain the average time.

The three shell file and results are shown below:

```
#! /bin/sh

for i in $(seq 1000);
do openssl rsautl -encrypt -inkey public.pem -pubin -in message -out Enmessage.ssl;
done
```

pkEncry.sh

```
#! /bin/sh

for i in $(seq 1000);
do openssl rsautl -decrypt -inkey privatekey.pem -in Enmessage.ssl -out Demessage.txt;
done
```

pkDecry.sh

```
#! /bin/sh

for i in $(seq 1000);
do openssl enc -aes-128-cbc -e -in message -out aesEncryp.bin -k NiceDay;
done
```

aesEncryp.sh

```
root@seed-desktop:/home/seed/Desktop/HW5/Task5# time ./pkEncry.sh

real    0m17.768s
user    0m6.712s
sys     0m10.341s
```

1000 times public key encryption time

```
root@seed-desktop:/home/seed/Desktop/HW5/Task5# time ./pkDecry.sh

real    0m24.467s
user    0m13.017s
sys     0m10.793s
```

1000 times private key decryption time

```
root@seed-desktop:/home/seed/Desktop/HW5/Task5/aes# time ./aesEncryp.sh

real    0m16.971s
user    0m5.116s
sys     0m9.537s
```

1000 times 128-bit AES encryption time

Real time is the time from start to end of the call. It counts the elapsed time slices used by other processes and this time the process is waiting for I/O to complete. User time is the execution time for user-defined instructions. Sys time is the execution time of system calls.

According to the results above, “user+sys” will tell me how much actual CPU time the process used. For public key encryption: 17s; For private key decryption: 23s; For 128-bit AES encryption: 14.6s. Therefore, AES secret key encryption is faster than RSA asymmetric key encryption.

We can use openssl’s speed command to benchmark AES and RSA:

```
root@seed-desktop:/home/seed/Desktop# openssl speed rsa
Doing 512 bit private rsa's for 10s: 7379 512 bit private RSA's in 9.83s
Doing 512 bit public rsa's for 10s: 99294 512 bit public RSA's in 9.88s
Doing 1024 bit private rsa's for 10s: 1637 1024 bit private RSA's in 9.88s
Doing 1024 bit public rsa's for 10s: 37344 1024 bit public RSA's in 9.83s
Doing 2048 bit private rsa's for 10s: 301 2048 bit private RSA's in 9.86s
Doing 2048 bit public rsa's for 10s: 12183 2048 bit public RSA's in 9.82s
Doing 4096 bit private rsa's for 10s: 51 4096 bit private RSA's in 10.03s
Doing 4096 bit public rsa's for 10s: 3681 4096 bit public RSA's in 9.83s
OpenSSL 0.9.8g 19 Oct 2007
built on: Fri Mar 27 14:45:47 UTC 2009
options:bn(64,32) md2(int) rc4(idx,int) des(ptr,risc1,16,long) aes(partial) blow
fish(idx)
compiler: gcc -fPIC -DOPENSSL_PIC -DZLIB -DOPENSSL_THREADS -D_REENTRANT -DDSO_DL
FCN -DHAVE_DLFCN_H -DL_ENDIAN -DTERMIO -O3 -march=i686 -Wa,--noexecstack -g -Wal
l -DOPENSSL_BN_ASM_PART_WORDS -DOPENSSL_IA32_SSE2 -DSHA1_ASM -DMD5_ASM -DRMD160_
ASM -DAES_ASM
available timing options: TIMES TIMEB HZ=100 [sysconf value]
timing function used: times
                  sign    verify    sign/s verify/s
rsa  512 bits 0.001332s 0.000100s    750.7  10050.0
rsa 1024 bits 0.006035s 0.000263s    165.7   3799.0
rsa 2048 bits 0.032757s 0.000806s     30.5   1240.6
rsa 4096 bits 0.196667s 0.002670s      5.1    374.5
```

```
root@seed-desktop:/home/seed/Desktop# openssl speed aes
Doing aes-128 cbc for 3s on 16 size blocks: 4558071 aes-128 cbc's in 2.96s
Doing aes-128 cbc for 3s on 64 size blocks: 1664695 aes-128 cbc's in 2.83s
Doing aes-128 cbc for 3s on 256 size blocks: 507205 aes-128 cbc's in 2.92s
Doing aes-128 cbc for 3s on 1024 size blocks: 129994 aes-128 cbc's in 2.90s
Doing aes-128 cbc for 3s on 8192 size blocks: 17502 aes-128 cbc's in 2.96s
Doing aes-192 cbc for 3s on 16 size blocks: 5357310 aes-192 cbc's in 2.96s
Doing aes-192 cbc for 3s on 64 size blocks: 1686414 aes-192 cbc's in 2.94s
Doing aes-192 cbc for 3s on 256 size blocks: 355281 aes-192 cbc's in 2.69s
Doing aes-192 cbc for 3s on 1024 size blocks: 106874 aes-192 cbc's in 2.91s
Doing aes-192 cbc for 3s on 8192 size blocks: 13166 aes-192 cbc's in 2.90s
Doing aes-256 cbc for 3s on 16 size blocks: 4209791 aes-256 cbc's in 2.70s
Doing aes-256 cbc for 3s on 64 size blocks: 711077 aes-256 cbc's in 1.64s
Doing aes-256 cbc for 3s on 256 size blocks: 167825 aes-256 cbc's in 1.43s
Doing aes-256 cbc for 3s on 1024 size blocks: 37880 aes-256 cbc's in 1.29s
Doing aes-256 cbc for 3s on 8192 size blocks: 6923 aes-256 cbc's in 1.79s
OpenSSL 0.9.8g 19 Oct 2007
built on: Fri Mar 27 14:45:47 UTC 2009
options:bn(64,32) md2(int) rc4(idx,int) des(ptr,risc1,16,long) aes(partial) blowfish(idx)
compiler: gcc -fPIC -DOPENSSL_PIC -DZLIB -DOPENSSL_THREADS -D_REENTRANT -DDSO_DLFCN -DHAVE_DLFCN_H -DL_ENDIAN -DTERMIO -O3 -m
arch=i686 -Wa,--noexecstack -g -Wall -DOPENSSL_BN_ASM_PART_WORDS -DOPENSSL_IA32_SSE2 -DSHA1_ASM -DMD5_ASM -DRMD160_ASM -DAES_
ASM
available timing options: TIMES TIMEB HZ=100 [sysconf value]
timing function used: times
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes
aes-128 cbc      24638.22k    37646.81k    44467.29k    45901.33k    48437.97k
aes-192 cbc      28958.43k    36711.05k    33811.13k    37607.90k    37191.68k
aes-256 cbc      24946.91k    27749.35k    30044.20k    30069.09k    31683.36k
```

The openssl speed command verifies my observation that AES is faster than RSA in general.

3.6 Task 6: Create Digital Signature:

Encrypting the hash value in “hashed” using private key and saving it into a file called “example.sha256”:

```
root@seed-desktop:/home/seed/Desktop/Task6# openssl dgst -sha256 example.txt > h
ashed
root@seed-desktop:/home/seed/Desktop/Task6# vi hashed
root@seed-desktop:/home/seed/Desktop/Task6# vi hashed
root@seed-desktop:/home/seed/Desktop/Task6# openssl rsautl -sign -inkey privatek
ey.pem -keyform PEM -in hashed > example.sha256
root@seed-desktop:/home/seed/Desktop/Task6# vi example.sha256
```

Decrypting “example.sha256” using public key and showing the hash value:

```
root@seed-desktop:/home/seed/Desktop/Task6/verification# openssl rsautl -verify -inkey publickey.pem -keyform PEM -pubin -in
example.sha256
SHA256(example.txt)= 8f9862d15a030110071abf7bfbb486b44ce13e8e2ad8e13a3227049d9cd49e98
```

Saving the hash value into a file called “verified”:

```
root@seed-desktop:/home/seed/Desktop/Task6/verification# openssl rsautl -verify -inkey publickey.pem -keyform PEM -pubin -in
example.sha256 > verified
```

Comparison between “hashed” and “verified”:

```
root@seed-desktop:/home/seed/Desktop/Task6/verification# diff -s hashed verified
Files hashed and verified are identical
```

After changing some information slightly in the original file “example.txt”:

Following the same procedure, we can output the hash value of the modified “example.txt” into a file called “modified_hash”. Then compare “modified_hash” with “verified”, we can find they are no longer the same.

```
root@seed-desktop:/home/seed/Desktop/Task6/modified# diff -s modified_hash verified
1c1
< SHA256(example.txt)= 36d26b2b0df2ae76a41b52d1934375d70d8beb5ed0b949bed743f888f25e7caa
---
> SHA256(example.txt)= 8f9862d15a030110071abf7bfbb486b44ce13e8e2ad8e13a3227049d9cd49e98
```