HW 5 Report

[Paper-and-Pencil Problems: 2](#_Toc354445784)

[5.3 2](#_Toc354445785)

[5.14 2](#_Toc354445786)

[6.2 3](#_Toc354445787)

[6.8 3](#_Toc354445788)

[9.2 4](#_Toc354445789)

[10.1 4](#_Toc354445790)

[Lab and Programming Tasks: 5](#_Toc354445791)

[3.1 Task 1: Generating Message Digest and MAC: 5](#_Toc354445792)

[3.2 Task 2: Keyed Hash and HMAC: 5](#_Toc354445793)

[3.3 Task 3: The Randomness of One-way Hash: 6](#_Toc354445794)

[3.4 Task 4: Hash Collision-Free Property: 7](#_Toc354445795)

[3.5 Task 5: Performance Comparison: RSA versus AES: 9](#_Toc354445796)

[3.6 Task 6: Create Digital Signature: 11](#_Toc354445797)

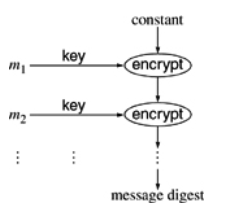
# Paper-and-Pencil Problems:

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 232 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?

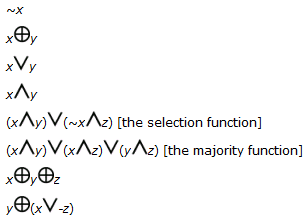
---Basically the same as birthday attack, suppose we have n messages, k possible hash results. There are possible pairs of messages and a total probability of that a pair have the same hash result. There is also probability that the pair of messages are from different original types. Therefore, to make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

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 232 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 232 iterations? Hint: consider blockwise palindromic messages.

---Suppose we have a palindromic message ABCDDCBA, A is m1, B is m2, 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 232 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:

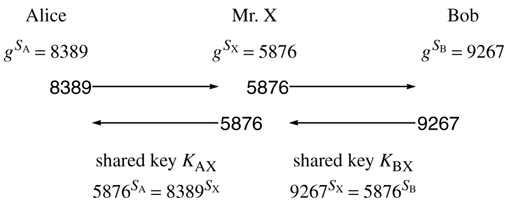


---The sufficient condition for randomness for the operations above is:

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 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 . Therefore, the shared key KAX 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 encrypted with A’s public key. Mr. X cannot calculate KBX as well. So MITM attack is prevented.

6.8 Suppose Fred sees your RSA signature on m1 and on m2 (i.e. he sees mod n and mod n). How does he compute the signature on each of mod n (for positive integer j), mod n, m1·m2 mod n, and in general mod n (for arbitrary integers j and k)?

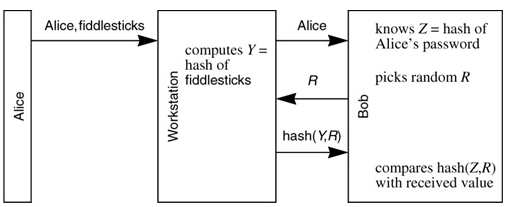
--- The signature on m1^j mod n for any j positive integer can be computed by (m1^d)^j mod n = (m1^j)^d mod n which is what we wanted.

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:



Cannot be used in serious applications because of its weakness.

The output of MD5 is 16-byte hash value.

(2) SHA1:



Broken but not yet cracked.

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

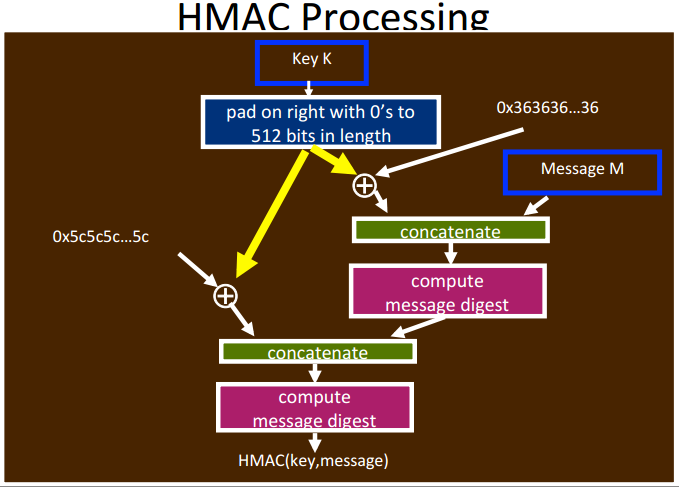
(3) SHA256:



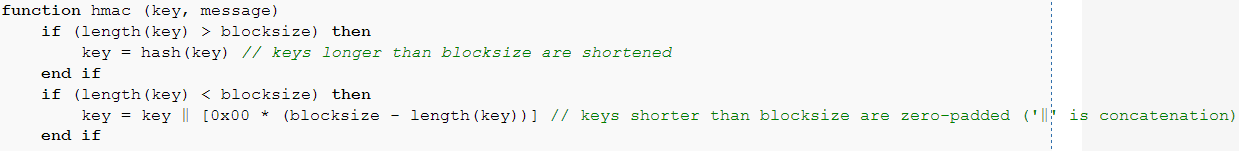
The output of SHA-256 is 32-byte hash value.

## 3.2 Task 2: Keyed Hash and HMAC:





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.



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

The hash value of the original message:

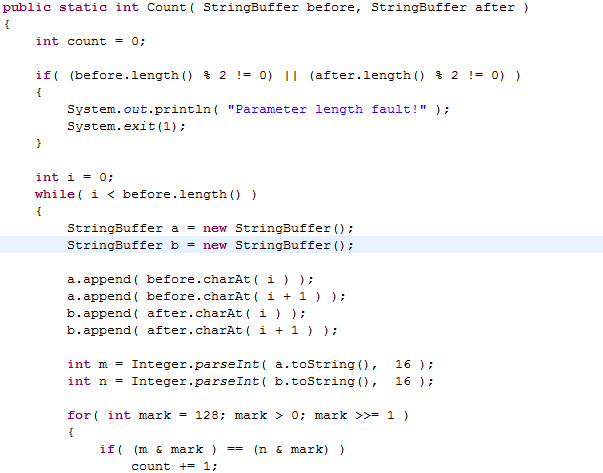
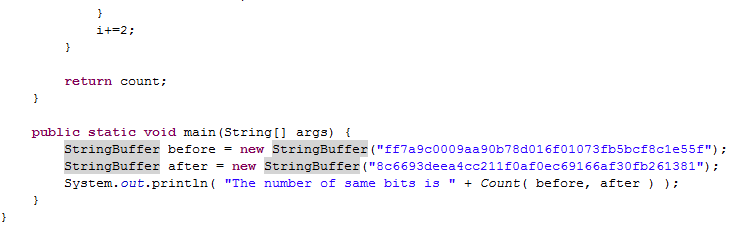


After flipping one bit in the original message with Ghex:



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:



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.

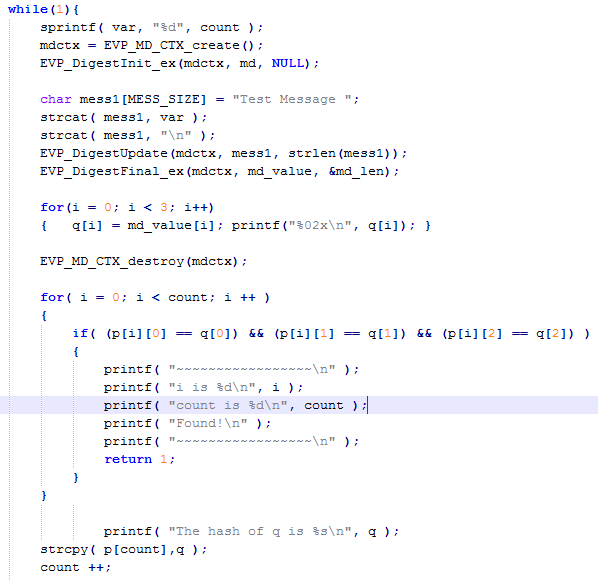
## 3.4 Task 4: Hash Collision-Free Property:

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:

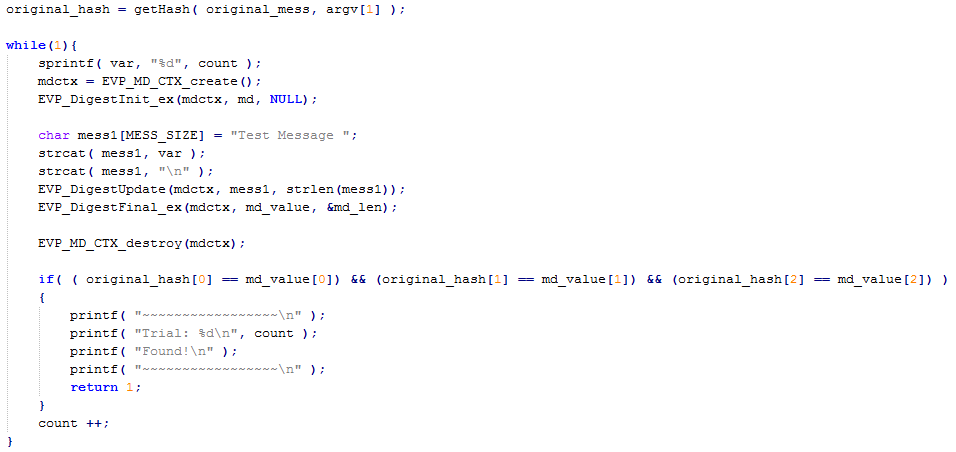


After an average 5377 trials, I can find two messages with the same hash value using brute-force method.

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:



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 . While finding out another message with the same hash value as the target message generally needs .

4.

Proof:

Suppose the hash result is a k-bit value. There are a total number of 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 . Therefore, in general, we need to try 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 message pairs. For each message pair, the probability of having same hash value is (No matter what hash value the first one has, the second one having the same hash value as the first one is ). To make , we will find out when , 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



Extract public key from the public key from private.pem



Encryption with public key:

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

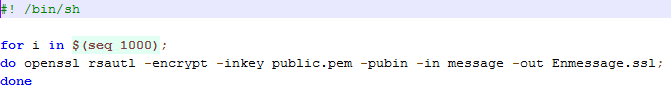
Decryption with private key:

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

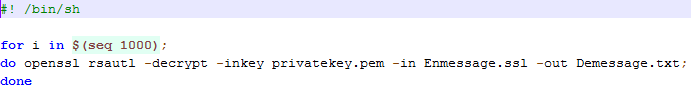
# openssl enc –aes-128-cbc –e –in message –out aesEncryp.bin –k NiceDay

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.

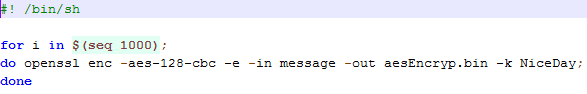
The three shell file and results are shown below:



pkEncry.sh



pkDecry.sh



aesEncryp.sh



1000 times public key encryption time



1000 times private key decryption time

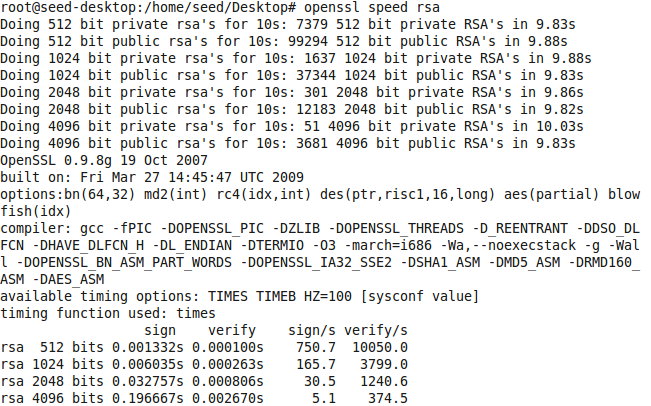


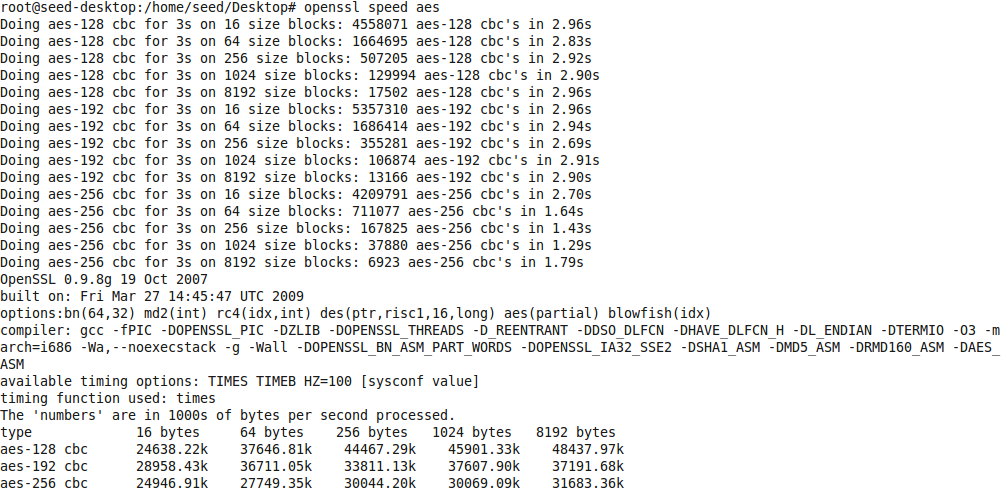
1000 times 128-bit AES encryption time

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:

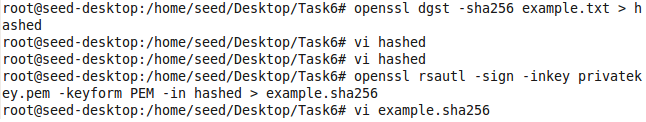




The openssl speed command verifies my observation that AES is faster than RSA in general.

## 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”:



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



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



Comparison between “hashed” and “verified”:



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.

