Assignment I (Semester B, 2024/2025)

CS4293 Topics on Computer Security

Due: Saturday 22nd February, 2025; 100 Marks

Contents

1 Introduction

This is the 1st Assignment for the course CS4293 in the semester B of academic year 2024-2025.

1.1 Objective

The learning objective of this assignment is for you to get a deeper understanding on several fundamental cryptographic building blocks and some common vulnerabilities in general software. After finishing the assignment, you should be able to gain a first-hand experience on symmetric encryption algorithms, encryption modes, paddings, initial vector (IV), one-way hash functions, message authentication code (MAC), digital signature, pseudorandomness generator (PRNG), etc.

1.2 Environment

All tasks in this assignment can be done on the VirtualBox as introduced in Tutorials. You may even check http://www.cis.syr.edu/~wedu/seed/lab_env.html to download useful materials such as set up manuals and OS images to establish the environment. Some helpful document about `\OpenSSL Command-Line HOWTO"` can be found at http://www.madboa.com/geek/openssl/.

1.3 Due Date

The assignment is due on **Saturday 22nd February, 2025** (week 7). Any reports should be submitted **before 23:59:59** (Firm!).

1.4 Submission

You will submit a report to describe what you have done and what you have observed with **screen shots** whenever necessary; you also need to provide explanation or codes to the observations that are interesting or surprising. In your report, you need to answer all the questions listed in this manual. Please answer each question using at most 100 words. Your report will be written **methodically and clearly in order** with index and all chapters presented. Typeset your report into .pdf file (make sure it can be opened with Adobe Reader) and name it as the format: [Your Name]-[Student ID]-CS4293-Assignment1.pdf, e.g., **HarryPotter-12345678-CS4293-Assignment1.pdf**. Then, upload the PDF file to Canvas.

2 Secret-Key Encryption [44 Marks]

The learning objective of this exercise is for you to get familiar with the concepts in the secret-key encryption. After finishing the tasks, you should be able to gain a first-hand experience on encryption algorithms, encryption modes, paddings, and initial vector (IV). Moreover, you will be able to use tools and write programs to encrypt/decrypt messages.

2.1 Task 1: Frequency Analysis Against Monoalphabetic Substitution Cipher [7 Marks]

It is well-known that monoalphabetic substitution cipher (also known as monoalphabetic cipher) is not secure, because it can be subjected to frequency analysis. In this task, you are given a cipher-text that is encrypted using a monoalphabetic cipher; namely, each letter in the original text is replaced by another letter, where the replacement does not vary (i.e., a letter is always replaced by the same letter during the encryption). Your job is to find out the original text using frequency analysis. It is known that the original text is an English article.

In the following, we describe how we encrypt the original article, and what simplification we have made.

- Step 1: let us do some simplification to the original article. We convert all upper cases to lower cases, and then removed all the punctuations and numbers. We do keep the spaces between words, so you can still see the boundaries of the words in the ciphertext. In real encryption using monoalphabetic cipher, spaces will be removed. We keep the spaces to simplify the task. We did this using the following command:

```
$ tr [:upper:] [:lower:] < article.txt > lowercase.txt
$ tr -cd '[a-z][\n][:space:]' < lowercase.txt > plaintext.txt
```

- Step 2: let us generate the encryption key, i.e., the substitution table. We will permute the alphabet from a to z using Python, and use the permuted alphabet as the key. See the following program.

```
$ python
>>> import random
>>> s = "abcdefghijklmnopqrstuvwxyz"
>>> list = random.sample(s, len(s))
>>> ''.join(list)
'sxtrwinqbedpvgkfmalhyuojzc'
```

- Step 3: we use the tr command to do the encryption. We only encrypt letters, while leaving the space and return characters alone.

```
$ tr 'abcdefghijklmnopqrstuvwxyz' 'sxtrwinqbedpvgkfmalhyuojzc' \
< plaintext.txt > ciphertext.txt
```

We have created a ciphertext using a different encryption key (not the one described above). You can download it from the **Canvas**. Your job is to use the frequency analysis to figure out the encryption key and the original plaintext.

Guidelines. Using the frequency analysis, you can find out the plaintext for some of the characters quite easily. For those characters, you may want to change them back to its plaintext, as you may be able to get more clues. It is better to use capital letters for plaintext, so for the same letter, we know which is plaintext and which is ciphertext. You can use the tr command to do this. For example, in the following, we replace letters a, e, and t in in.txt with letters X, G, E, respectively; the results are saved in out.txt.

```
$ tr 'aet' 'XGE' < in.txt > out.txt
```

There are many online resources that you can use. We list four useful links in the following:

- https://www.dcode.fr/frequency-analysis: This website can produce the statistics for a ciphertext, including the single-letter frequencies, bigram frequencies (2-letter sequence), and trigram frequencies (3-letter sequence), etc.
- https://en.wikipedia.org/wiki/Frequency_analysis: This Wikipedia page provides frequencies for a typical English plaintext.
- https://en.wikipedia.org/wiki/Bigram: Bigram frequency.
- https://en.wikipedia.org/wiki/Trigram: Trigram frequency.

2.2 Task 2: Encryption using Different Ciphers and Modes [6 Marks]

In this task, we will play with various encryption algorithms and modes. You can use the following openssl enc command to encrypt/decrypt a file. To see the manuals, you can type man openssl and man enc.

```
$ openssl enc -ciphertype -e -in plain.txt -out cipher.bin \
-K 00112233445566778889aabbccddeeff -iv 0102030405060708
```

Please replace the ciphertype with a specific cipher type, such as -aes-128-cbc, -bf-cbc, -aes-128-cfb, etc. In this task, you should try at least 3 different ciphers. You can find the meaning of the command-line options and all the supported cipher types by typing "man enc". We include some common options for the openssl enc command in the following:

```
-in <file>      input file
-out <file>     output file
-e              encrypt
-d              decrypt
-K/-iv          key/iv in hex is the next argument
-[pP]           print the iv/key (then exit if -P)
```

2.3 Task 3: Encryption Mode – ECB vs. CBC [6 Marks]

The file pic_original.bmp can be downloaded from the **Canvas**, and it contains a simple picture. We would like to encrypt this picture, so people without the encryption keys cannot know what is in the picture. Please encrypt the file using the ECB (Electronic Code Book) and CBC (Cipher Block Chaining) modes, and then do the following:

- 1. Let us treat the encrypted picture as a picture, and use a picture viewing software to display it. However, For the .bmp file, the first 54 bytes contain the header information about the picture, we have to set it correctly, so the encrypted file can be treated as a legitimate .bmp file. We will replace the header of the encrypted picture with that of the original picture. We can use the bless hex editor tool (already installed on our VM) to directly modify binary files. We can also use the following commands to get the header from p1.bmp, the data from p2.bmp (from offset 55 to the end of the file), and then combine the header and data together into a new file.

  ```
  $ head -c 54 p1.bmp > header
  $ tail -c +55 p2.bmp > body
  $ cat header body > new.bmp
  ```

- 2. Display the encrypted picture using a picture viewing program (we have installed an image viewer program called eog on our VM). Can you derive any useful information about the original picture from the encrypted picture? Please explain your observations.

Select a picture of your choice, repeat the experiment above, and report your observations.

2.4 Task 4: Padding [6 Marks]

For block ciphers, when the size of a plaintext is not a multiple of the block size, padding may be required. All the block ciphers normally use PKCS#5 padding, which is known as standard block padding. We will conduct the following experiments to understand how this type of padding works:

1. Use ECB, CBC, CFB, and OFB modes to encrypt a file (you can pick any cipher). Please report which modes have paddings and which ones do not. For those that do not need paddings, please explain why.

2. Let us create three files, which contain 5 bytes, 10 bytes, and 16 bytes, respectively. We can use the following "echo -n" command to create such files. The following example creates a file f1.txt with length 5 (without the -n option, the length will be 6, because a newline character will be added by echo):

   ```
   $ echo -n "12345" > f1.txt
   ```

   We then use "openssl enc -aes-128-cbc -e" to encrypt these three files using 128-bit AES with CBC mode. Please describe the size of the encrypted files.

   We would like to see what is added to the padding during the encryption. To achieve this goal, we will decrypt these files using "openssl enc -aes-128-cbc -d". Unfortunately, decryption by default will automatically remove the padding, making it impossible for us to see the padding. However, the command does have an option called "-nopad", which disables the padding, i.e., during the decryption, the command will not remove the padded data. Therefore, by looking at the decrypted data, we can see what data are used in the padding. Please use this technique to figure out what paddings are added to the three files.

   It should be noted that padding data may not be printable, so you need to use a hex tool to display the content. The following example shows how to display a file in the hex format:

   ```
   $ hexdump -C p1.txt
   00000000 31 32 33 34 35 36 37 38 39 49 4a 4b 4c 0a |123456789IJKL.|
   $ xxd p1.txt
   00000000: 3132 3334 3536 3738 3949 4a4b 4c0a 123456789IJKL.
   ```

2.5 Task 5: Error Propagation – Corrupted Cipher Text [7 Marks]

To understand the error propagation property of various encryption modes, we would like to do the following exercise:

1. Create a text file that is at least 1000 bytes long.

2. Encrypt the file using the AES-128 cipher.

3. Unfortunately, a single bit of the 55th byte in the encrypted file got corrupted. You can achieve this corruption using the bless hex editor.

4. Decrypt the corrupted ciphertext file using the correct key and IV

Please answer the following question: How much information can you recover by decrypting the corrupted file, if the encryption mode is ECB, CBC, CFB, or OFB, respectively? Please answer this question before you conduct this task, and then find out whether your answer is correct or wrong after you finish this task. Please provide justification.

2.6 Task 6: Initial Vector (IV) [6 Marks]

Most of the encryption modes require an initial vector (IV). Properties of an IV depend on the cryptographic scheme used. If we are not careful in selecting IVs, the data encrypted by us may not be secure at all, even though we are using a secure encryption algorithm and mode. The objective of this task is to help you understand the problems if an IV is not selected properly. Please do the following experiments:

- **Task 6.1.** A basic requirement for IV is uniqueness, which means that no IV may be reused under the same key. To understand why, please encrypt the same plaintext using (1) two different IVs, and (2) the same IV. Please describe your observation, based on which, explain why IV needs to be unique.

- **Task 6.2.** One may argue that if the plaintext does not repeat, using the same IV is safe. Let us look at the Output Feedback (OFB) mode. Assume that the attacker gets hold of a plaintext (P1) and a ciphertext (C1), can he/she decrypt other encrypted messages if the IV is always the same? You are given the following information, please try to figure out the actual content of P2 based on C2, P1, and C1.

```
Plaintext  (P1): This is a known message!
Ciphertext (C1): a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159

Plaintext  (P2): (unknown to you)
Ciphertext (C2): bf73bcd3509299d566c35b5d450337e1bb175f903fafc159
```

  If we replace OFB in this experiment with CFB (Cipher Feedback), how much of P2 can be revealed? You only need to answer the question; there is no need to demonstrate that.

  The attack used in this experiment is called the known-plaintext attack, which is an attack model for cryptanalysis where the attacker has access to both the plaintext and its encrypted version (ciphertext). If this can lead to the revealing of further secret information, the encryption scheme is not considered as secure.

- **Task 6.3.** From the previous tasks, we now know that IVs cannot repeat. Another important requirement on IV is that IVs need to be unpredictable for many schemes, i.e., IVs need to be randomly generated. In this task, we will see what is going to happen if IVs are predictable.

  Assume that Bob just sent out an encrypted message, and Eve knows that its content is either Yes or No; Eve can see the ciphertext and the IV used to encrypt the message, but since the encryption algorithm AES is quite strong, Eve has no idea what the actual content is. However, since Bob uses predictable IVs, Eve knows exactly what IV Bob is going to use next. The following summarizes what Bob and Eve know:

```
Encryption method: 128-bit AES with CBC mode.

Key (in hex):    00112233445566778899aabbccddeeff (known only to Bob)
Ciphertext (C1): bef65565572ccee2a9f9553154ed9498 (known to both)
IV used on P1 (known to both)
     (in ascii): 1234567890123456
     (in hex)  : 31323334353637383930313233343536
Next IV (known to both)
     (in ascii): 1234567890123457
     (in hex)  : 31323334353637383930313233343537
```

  A good cipher should not only tolerate the known-plaintext attack described previously, it should also tolerate the *chosen-plaintext attack*, which is an attack model for cryptanalysis where the attacker can obtain the ciphertext for an arbitrary plaintext. Since AES is a strong cipher that can tolerate the chosen-plaintext attack, Bob does not mind encrypting any plaintext given by Eve; he does use a different IV for each plaintext, but unfortunately, the IVs he generates are not random, and they can always be predictable.

  Your job is to construct a message P2 and ask Bob to encrypt it and give you the ciphertext. Your objective is to use this opportunity to figure out whether the actual content of P1 is Yes or No. You can read this Wikipedia page for ideas: https://en.wikipedia.org/wiki/Initialization_vector.

There are more advanced cryptanalysis on IV that is beyond the scope of this task. You can read the article posted in this URL: https://defuse.ca/cbcmodeiv.htm. Because the requirements on IV really depend on cryptographic schemes, it is hard to remember what properties should be maintained when we select an IV. However, we will be safe if we always use a new IV for each encryption, and the new IV needs to be generated using a good pseudo random number generator, so it is unpredictable by adversaries. (See tasks about Random Number Generation) for details on how to generate cryptographically strong pseudo random numbers.

2.7 Task 7: Programming using the Crypto Library [6 Marks]

In this task, you are given a plaintext and a ciphertext, and your job is to find the key that is used for the encryption. You do know the following facts:

- The aes-128-cbc cipher is used for the encryption.
- The key used to encrypt this plaintext is an English word shorter than 16 characters; the word can be found from a typical English dictionary. Since the word has less than 16 characters (i.e. 128 bits), pound signs (#: hexadecimal value is 0x23) are appended to the end of the word to form a key of 128 bits.

Your goal is to write a program to find out the encryption key. You can download a English word list from the Internet. We have also linked one on the (**Canvas**). The plaintext, ciphertext, and IV are listed in the following:

```
Plaintext (total 21 characters): This is a top secret.
Ciphertext (in hex format): 764aa26b55a4da654df6b19e4bce00f4
                            ed05e09346fb0e762583cb7da2ac93a2
IV (in hex format):         aabbccddeeff00998877665544332211
```

You need to pay attention to the following issues:

- If you choose to store the plaintext message in a file, and feed the file to your program, you need to check whether the file length is 21. If you type the message in a text editor, you need to be aware that some editors may add a special character to the end of the file. The easiest way to store the message in a file is to use the following command (the -n flag tells echo not to add a trailing newline):

  ```
  $ echo -n "This is a top secret." > file
  ```

- In this task, you are supposed to write your own program to invoke the crypto library. No credit will be given if you simply use the openssl commands to do this task. Sample code can be found from the following URL: https://www.openssl.org/docs/man3.0/man3/EVP_EncryptInit.html
- When you compile your code using gcc, do not forget to include the -lcrypto flag, because your code needs the crypto library. See the following example:

  ```
  $ gcc -o myenc myenc.c -lcrypto
  ```

3 MD5 Collision Attack [26 Marks]

The learning objective of this exercise is for you to really understand the impact of collision attacks, and see in first hand what damages can be caused if a widely-used one-way hash function's collision-resistance

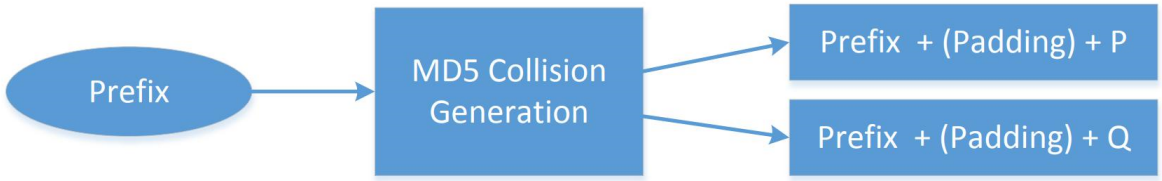
property is broken. To achieve this goal, you need to launch actual collision attacks against the MD5 hash function. Using the attacks, you should be able to create two different programs that share the same MD5 hash but have completely different behaviors.

The tasks use a tool called Fast MD5 Collision Generation, which was written by Marc Stevens; the name of the binary is called md5collgen in our VMs. For Ubuntu16.04 VM: md5collgen has already been installed inside /home/seed/bin.

3.1 Task 8: Generating Two Different Files with the Same MD5 Hash [6 Marks]

In this task, we will generate two different files with the same MD5 hash values. The beginning parts of these two files need to be the same, i.e., they share the same prefix. We can achieve this using the md5collgen program, which allows us to provide a prefix file with any arbitrary content. The way how the program works is illustrated in the figure following. The following command generates two output files, out1.bin and out2.bin, for a given a prefix file prefix.txt:

```
$ md5collgen -p prefix.txt -o out1.bin out2.bin
```



We can check whether the output files are distinct or not using the diff command. We can also use the md5sum command to check the MD5 hash of each output file. See the following commands.

```
$ diff out1.bin out2.bin
$ md5sum out1.bin
$ md5sum out2.bin
```

Since out1.bin and out2.bin are binary, we cannot view them using a text-viewer program, such as cat or more; we need to use a binary editor to view (and edit) them. We have already installed a hex editor software called bless in our VM. Please use such an editor to view these two output files, and describe your observations. In addition, you should answer the following questions.

- Question 1. If the length of your prefix file is not multiple of 64, what is going to happen?
- Question 2. Create a prefix file with exactly 64 bytes, and run the collision tool again, and see what happens.
- Question 3. Are the data (128 bytes) generated by md5collgen completely different for the two output files? Please identify all the bytes that are different.

3.2 Task 9: Understanding MD5's Property [6 Marks]

In this task, we will try to understand some of the properties of the MD5 algorithm. These properties are important for us to conduct further tasks. MD5 is a quite complicated algorithm, but from very high level, it is not so complicated. As Figure 2 shows, MD5 divides the input data into blocks of 64 bytes, and then computes the hash iteratively on these blocks. The core of the MD5 algorithm is a compression function, which takes two inputs, a 64-byte data block and the outcome of the previous iteration. The compression function produces a 128-bit IHV, which stands for "Intermediate Hash Value"; this output is then fed into

the next iteration. If the current iteration is the last one, the IHV will be the final hash value. The IHV input for the first iteration (IHV_0) is a fixed value.

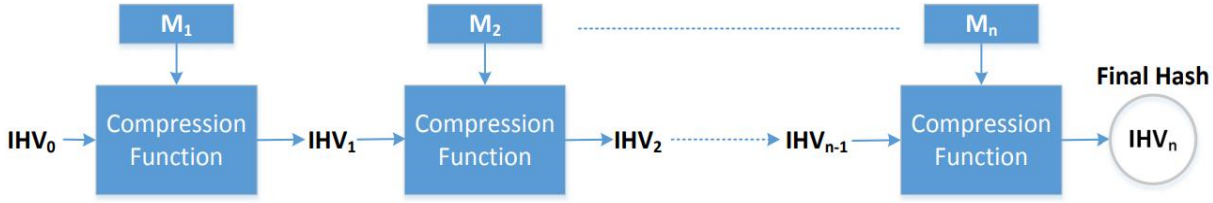


Figure 2: How the MD5 algorithm works

Based on how MD5 works, we can derive the following property of the MD5 algorithm: Given two inputs M and N, if MD5(M) = MD5(N), i.e., the MD5 hashes of M and N are the same, then for any input T, MD5(M || T) = MD5(N || T), where || represents concatenation.

That is, if inputs M and N have the same hash, adding the same suffix T to them will result in two outputs that have the same hash value. This property holds not only for the MD5 hash algorithm, but also for many other hash algorithms. **Your job in this task** is to design an experiment to demonstrates that this property holds for MD5.

You can use the cat command to concatenate two files (binary or text files) into one. The following command concatenates the contents of file2 to the contents of file1, and places the result in file3.

```
$ cat file1 file2 > file3
```

3.3 Task 10: Generating Two Executable Files with the Same MD5 Hash [7 Marks]

In this task, you are given the following C program. Your job is to create two different versions of this program, such that the contents of their xyz arrays are different, but the hash values of the executables are the same.

```
#include <stdio.h>
unsigned char xyz[200] = {
 /* The actual contents of this array are up to you */
};
int main()
{
  int i;
  for (i=0; i<200; i++){
    printf("%x", xyz[i]);
  }
  printf("\n");
}
```

You may choose to work at the source code level, i.e., generating two versions of the above C program, such that after compilation, their corresponding executable files have the same MD5 hash value. However, it may be easier to directly work on the binary level. You can put some random values in the xyz array, compile the above code to binary. Then you can use a hex editor tool to modify the content of the xyz array directly in the binary file.

Finding where the contents of the array are stored in the binary is not easy. However, if we fill the array with some fixed values, we can easily find them in the binary. For example, the following code fills the array with 0x41, which is the ASCII value for letter A. It will not be difficult to locate 200 A's in the binary.

```
unsigned char xyz[200] = {
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  ... (omitted) ...
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
}
```

Guidelines. From inside the array, we can find two locations, from where we can divide the executable file into three parts: a prefix, a 128-byte region, and a suffix. The length of the prefix needs to be multiple of 64 bytes. See Figure 3 for an illustration of how the file is divided.

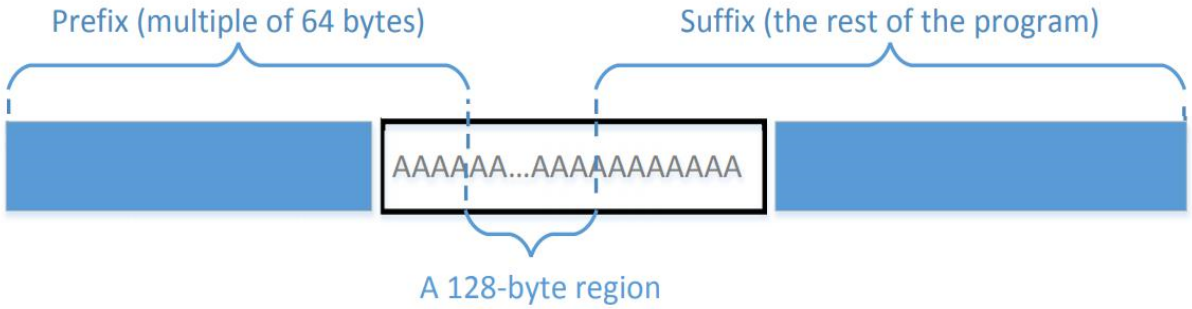


Figure 3: Break the executable file into three pieces.

We can run md5collgen on the prefix to generate two outputs that have the same MD5 hash value. Let us use P and Q to represent the second part (each having 128 bytes) of these outputs (i.e., the part after the prefix). Therefore, we have the following:

```
MD5 (prefix || P) = MD5 (prefix || Q)
```

Based on the property of MD5, we know that if we append the same suffix to the above two outputs, the resultant data will also have the same hash value. Basically, the following is true for any suffix:

```
MD5 (prefix || P || suffix) = MD5 (prefix || Q || suffix)
```

Therefore, we just need to use P and Q to replace 128 bytes of the array (between the two dividing points), and we will be able to create two binary programs that have the same hash value. Their outcomes are different, because they each print out their own arrays, which have different contents.

Tools. You can use bless to view the binary executable file and find the location for the array. For dividing a binary file, there are some tools that we can use to divide a file from a particular location. The head and tail commands are such useful tools. You can look at their manuals to learn how to use them. We give three examples in the following:

```
$ head -c 3200 a.out > prefix
$ tail -c 100 a.out > suffix
$ tail -c +3300 a.out > suffix
```

The first command above saves the first 3200 bytes of a.out to prefix. The second command saves the last 100 bytes of a.out to suffix. The third command saves the data from the 3300th byte to the end of the file a.out to suffix. With these two commands, we can divide a binary file into pieces from any location. If we need to glue some pieces together, we can use the cat command.

If you use bless to copy-and-paste a block of data from one binary file to another file, the menu item "Edit → Select Range" is quite handy, because you can select a block of data using a starting point and a range, instead of manually counting how many bytes are selected.

3.4 Task 11: Making the Two Programs Behave Differently [7 Marks]

In the previous task, we have successfully created two programs that have the same MD5 hash, but their behaviors are different. However, their differences are only in the data they print out; they still execute the same sequence of instructions. In this task, we would like to achieve something more significant and more meaningful.

Assume that you have created a software which does good things. You send the software to a trusted authority to get certified. The authority conducts a comprehensive testing of your software, and concludes that your software is indeed doing good things. The authority will present you with a certificate, stating that your program is good. To prevent you from changing your program after getting the certificate, the MD5 hash value of your program is also included in the certificate; the certificate is signed by the authority, so you cannot change anything on the certificate or your program without rendering the signature invalid.

You would like to get your malicious software certified by the authority, but you have zero chance to achieve that goal if you simply send your malicious software to the authority. However, you have noticed that the authority uses MD5 to generate the hash value. You got an idea. You plan to prepare two different programs. One program will always execute benign instructions and do good things, while the other program will execute malicious instructions and cause damages. You find a way to get these two programs to share the same MD5 hash value.

You then send the benign version to the authority for certification. Since this version does good things, it will pass the certification, and you will get a certificate that contains the hash value of your benign program. Because your malicious program has the same hash value, this certificate is also valid for your malicious program. Therefore, you have successfully obtained a valid certificate for your malicious program. If other people trusted the certificate issued by the authority, they will download your malicious program.

The objective of this task is to launch the attack described above. Namely, you need to create two programs that share the same MD5 hash. However, one program will always execute benign instructions, while the other program will execute malicious instructions. In your work, what benign/malicious instructions are executed is not important; it is sufficient to demonstrate that the instructions executed by these two programs are different.

Guidelines. Creating two completely different programs that produce the same MD5 hash value is quite hard. The two hash-colliding programs produced by md5collgen need to share the same prefix; moreover, as we can see from the previous task, if we need to add some meaningful suffix to the outputs produced by md5collgen, the suffix added to both programs also needs to be the same. These are the limitations of the MD5 collision generation program that we use. Although there are other more complicated and more advanced tools that can lift some of the limitations, such as accepting two different prefixes [see here], they demand much more computing power, so they are out of the scope for this task. We need to find a way to generate two different programs within the limitations.

There are many ways to achieve the above goal. We provide one approach as a reference, but you are encouraged to come up their own ideas. In our approach, we create two arrays X and Y. We compare the

contents of these two arrays; if they are the same, the benign code is executed; otherwise, the malicious code is executed. See the following pseudo-code:

```
Array X;
Array Y;
main()
{
  if(X's contents and Y's contents are the same)
    run benign code;
  else
    run malicious code;
  return;
}
```

We can initialize the arrays X and Y with some values that can help us find their locations in the executable binary file. Our job is to change the contents of these two arrays, so we can generate two different versions that have the same MD5 hash. In one version, the contents of X and Y are the same, so the benign code is executed; in the other version, the contents of X and Y are different, so the malicious code is executed. We can achieve this goal using a technique similar to the one used in Task 3. Figure 4 illustrates what the two versions of the program look like.

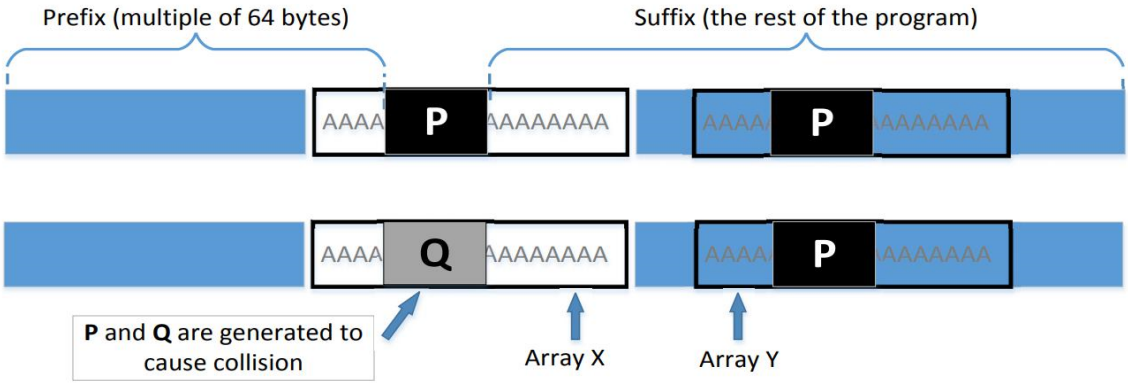


Figure 4: An approach to generate two hash-colliding programs with different behaviors.

From Figure 4, we know that these two binary files have the same MD5 hash value, as long as P and Q are generated accordingly. In the first version, we make the contents of arrays X and Y the same, while in the second version, we make their contents different. Therefore, the only thing we need to change is the contents of these two arrays, and there is no need to change the logic of the programs.

4 RSA Public-Key Encryption and Signature [30 Marks]

The learning objective is for you to gain hands-on experiences on the RSA algorithm. From lectures, you should have learned the theoretic part of the RSA algorithm, so you know mathematically how to generate public/private keys and how to perform encryption/decryption and signature generation/verification. The tasks enhance your understanding of RSA by requiring you to go through every essential step of the RSA algorithm on actual numbers, so they can apply the theories learned from the class. Essentially, you will be implementing the RSA algorithm using the C program language.

Description. The RSA algorithm involves computations on large numbers. These computations cannot be directly conducted using simple arithmetic operators in programs, because those operators can only

operate on primitive data types, such as 32-bit integer and 64-bit long integer types. The numbers involved in the RSA algorithms are typically more than 512 bits long. For example, to multiple two 32-bit integer numbers `a` and `b`, we just need to use `a*b` in our program. However, if they are big numbers, we cannot do that any more; instead, we need to use an algorithm (i.e., a function) to compute their products.

There are several libraries that can perform arithmetic operations on integers of arbitrary size. For the tasks, we will use the Big Number library provided by **openssl**. To use this library, we will define each big number as a **BIGNUM** type, and then use the APIs provided by the library for various operations, such as addition, multiplication, exponentiation, modular operations, etc.

4.1 BIGNUM APIs

All the big number APIs can be found from https://linux.die.net/man/3/bn. In the following, we describe some of the APIs that are needed for the tasks.

- Some of the library functions requires temporary variables. Since dynamic memory allocation to create BIGNUMs is quite expensive when used in conjunction with repeated subroutine calls, a BN_CTX structure is created to holds BIGNUM temporary variables used by library functions. We need to create such a structure, and pass it to the functions that requires it.

```
BN_CTX *ctx = BN_CTX_new()
```

- Initialize a BIGNUM variable

```
BIGNUM *a = BN_new()
```

- There are a number of ways to assign a value to a BIGNUM variable

```
// Assign a value from a decimal number string
BN_dec2bn(&a, "12345678901112231223");
// Assign a value from a hex number string
BN_hex2bn(&a, "2A3B4C55FF77889AED3F");
// Generate a random number of 128 bits
BN_rand(a, 128, 0, 0);
// Generate a random prime number of 128 bits
BN_generate_prime_ex(a, 128, 1, NULL, NULL, NULL);
```

- Print out a big number.

```
void printBN(char *msg, BIGNUM * a)
{
  // Convert the BIGNUM to number string
  char * number_str = BN_bn2dec(a);
  // Print out the number string
  printf("%s %s\n", msg, number_str);
  // Free the dynamically allocated memory
  OPENSSL_free(number_str);
}
```

- Compute $res = a - b$ and $res = a + b$:

```
BN_sub(res, a, b);
BN_add(res, a, b);
```

- Compute $res = a * b$. It should be noted that a BN_CTX structure is need in this API.

```
BN_mul(res, a, b, ctx);
```

- Compute $res = a \cdot b \bmod n$:

```
BN_mod_mul(res, a, b, n, ctx);
```

- Compute $res = a^c \bmod n$:

```
BN_mod_exp(res, a, c, n, ctx);
```

- Compute modular inverse, i.e., given a, find b, such that $a \cdot b \bmod n = 1$. The value b is called the inverse of a, with respect to modular n.

```
BN_mod_inverse(b, a, n, ctx);
```

4.2 A Complete Example

We show a complete example in the following. In this example, we initialize three BIGNUM variables, a, b, and n; we then compute $a \cdot b$ and $(a^b \bmod n)$.

```
/* bn_sample.c */
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256
void printBN(char *msg, BIGNUM * a)
{
  /* Use BN_bn2hex(a) for hex string
   * Use BN_bn2dec(a) for decimal string */
  char * number_str = BN_bn2hex(a);
  printf("%s %s\n", msg, number_str);
  OPENSSL_free(number_str);
}
int main ()
{
  BN_CTX *ctx = BN_CTX_new();
  BIGNUM *a = BN_new();
  BIGNUM *b = BN_new();
  BIGNUM *n = BN_new();
  BIGNUM *res = BN_new();
  // Initialize a, b, n
  BN_generate_prime_ex(a, NBITS, 1, NULL, NULL, NULL);
  BN_dec2bn(&b, "273489463796838501848592769467194369268");
  BN_rand(n, NBITS, 0, 0);
  // res = a*b
  BN_mul(res, a, b, ctx);
  printBN("a * b = ", res);
  // res = a^b mod n
  BN_mod_exp(res, a, b, n, ctx);
  printBN("a^b mod n = ", res);
  return 0;
}
```

Compiliation. We can use the following command to compile bn_sample.c (the character after - is the letter l, not the number 1; it tells the compiler to use the crypto library).

```
$ gcc bn_sample.c -lcrypto
```

To avoid mistakes, please avoid manually typing the numbers used in the following tasks. Instead, copy and paste the numbers from this PDF file. Again, you should describe your steps and include your code and running results.

4.3 Task 12: Deriving the Private Key [4 Marks]

Let p, q, and e be three prime numbers. Let $n = p * q$. We will use (e, n) as the public key. Please calculate the private key d. The hexadecimal values of p, q, and e are listed in the following. It should be noted that although p and q used in this task are quite large numbers, they are not large enough to be secure. We intentionally make them small for the sake of simplicity. In practice, these numbers should be at least 1024 bits long (the one used here are only 128 bits).

```
p = F7E75FDC469067FFDC4E847C51F452DF
q = E85CED54AF57E53E092113E62F436F4F
e = 0D88C3
```

4.4 Task 13: Encrypting a Message [5 Marks]

Let (e, n) be the public key. Please encrypt the message "A top secret!" (the quotations are not included). We need to convert this ASCII string to a hex string, and then convert the hex string to a BIGNUM using the hex-to-bn API BN_hex2bn(). The following *python* command can be used to convert a plain ASCII string to a hex string.

```
$ python -c 'print("A top secret!".encode("hex"))'
4120746f702073656372657421
```

The public keys are listed in the followings (hexadecimal). We also provide the private key d to help you verify your encryption result.

```
n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
e = 010001 (this hex value equals to decimal 65537)
M = A top secret!
d = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D
```

4.5 Task 14: Decrypting a Message [5 Marks]

The public/private keys used in this task are the same as the ones used in the previous task. Please decrypt the following ciphertext C, and convert it back to a plain ASCII string.

```
C = 8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F
```

You can use the following *python* command to convert a hex string back to to a plain ASCII string.

```
$ python -c 'print("4120746f702073656372657421".decode("hex"))'
A top secret!
```

4.6 Task 15: Signing a Message [5 Marks]

The public/private keys used in this task are the same as the ones used in the previous task. Please generate a signature for the following message (please directly sign this message, instead of signing its hash value):

```
M = I owe you $2000.
```

Please make a slight change to the message M, such as changing $2000 to $3000, and sign the modified message. Compare both signatures and describe what you observe.

4.7 Task 16: Verifying a Message [5 Marks]

Bob receives a message M = "Launch a missile." from Alice, with her signature S. We know that Alice's public key is (e, n). Please verify whether the signature is indeed Alice's or not. The public key and signature (hexadecimal) are listed in the following:

```
M = Launch a missle.
S = 643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F
e = 010001 (this hex value equals to decimal 65537)
n = AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115
```

Suppose that the signature in is corrupted, such that the last byte of the signature changes from 2F to 3F, i.e, there is only one bit of change. Please repeat this task, and describe what will happen to the verification process.

4.8 Task 17: Manually Verifying an X.509 Certificate [6 Marks]

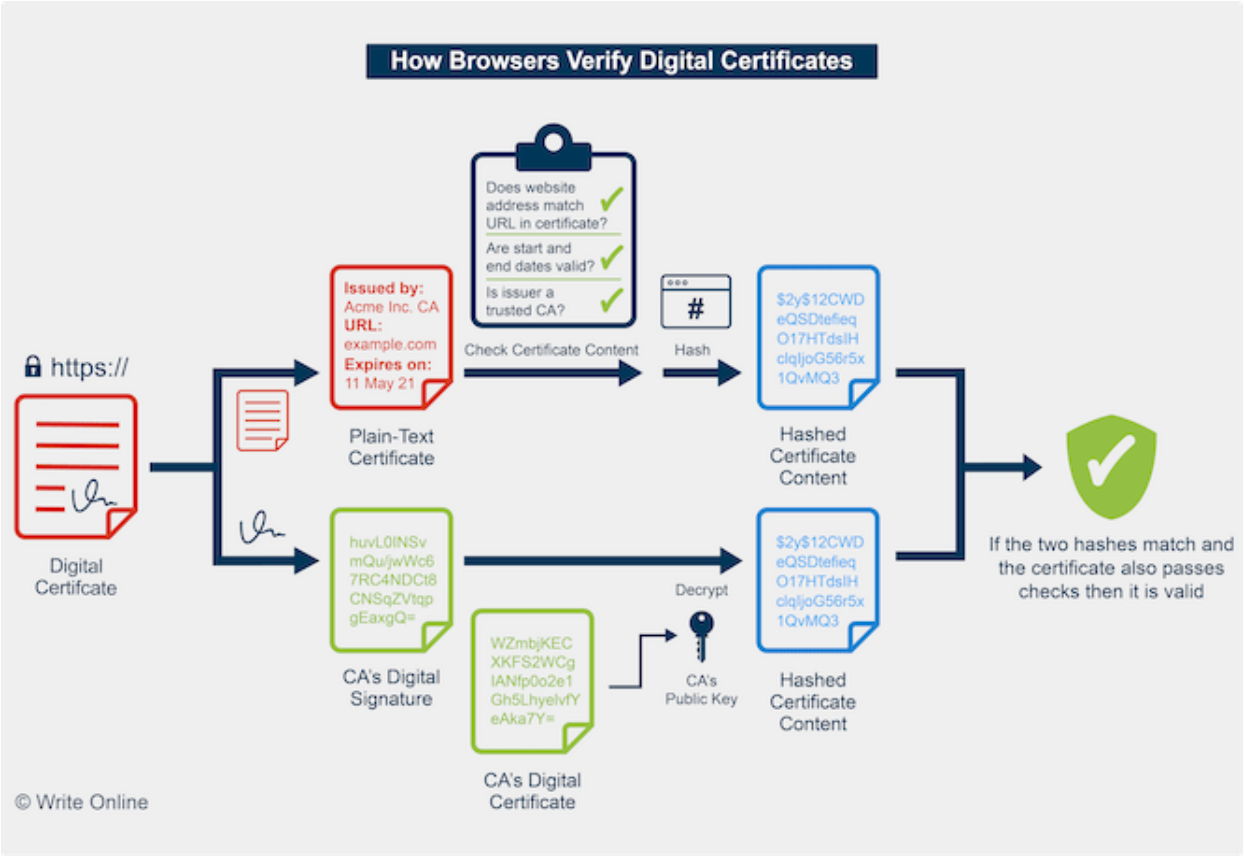


Figure 5: Illustration Figure of digital certificate verfication.

In this task, we will manually verify an X.509 certificate using our program. An X.509 contains data about a public key and an issuer's signature on the data (see Fig. 5 for illustration). We will download a real X.509 certificate from a web server, get its issuer's public key, and then use this public key to verify the signature on the certificate.

Step 1: Download a certificate from a real web server. We use the www.example.org server in this document. At this point, you should use the web server www.chase.com in your assignment. We can download certificates using browsers or use the following command:

```
$ openssl s_client -connect www.example.org:443 -showcerts
Certificate chain
0 s:/C=US/ST=California/L=Los Angeles/O=Internet Corporation for Assigned
Names and Numbers/OU=Technology/CN=www.example.org
i:/C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert SHA2 High Assurance
Server CA
-----BEGIN CERTIFICATE-----
MIIF8jCCBNqgAwIBAgIQDmTF+8I2reFLFyrrQceMsDANBgkqhkiG9w0BAQsFADBw
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
......
wDSiIIWIWJiJGbEeIO0TIFwEVWTOnbNl/faPXpk5IRXicapqiII=
-----END CERTIFICATE-----
1 s:/C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert SHA2 High
Assurance Server CA
i:/C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert High Assurance
EV Root CA
-----BEGIN CERTIFICATE-----
MIIEsTCCA5mgAwIBAgIQBOHnpNxc8vNtwCtCuF0VnzANBgkqhkiG9w0BAQsFADBs
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
......
cPUeybQ=
-----END CERTIFICATE-----
```

The result of the command contains two certificates. The subject field (the entry starting with s:) of the certificate is www.example.org, i.e., this is www.example.org's certificate. The issuer field (the entry starting with i:) provides the issuer's information. The subject field of the second certificate is the same as the issuer field of the first certificate. Basically, the second certificate belongs to an intermediate CA. In this task, we will use CA's certificate to verify a server certificate.

If you only get one certificate back using the above command, that means the certificate you get is signed by a root CA. Root CAs' certificates can be obtained from the Firefox browser installed in our pre-built VM. Go to the Edit → Preferences → Privacy and then Security → View Certificates. Search for the name of the issuer and download its certificate.

Copy and paste each of the certificate (the text between the line containing "Begin CERTIFICATE" and the line containing "END CERTIFICATE", including these two lines) to a file. Let us call the first one c0.pem and the second one c1.pem.

Step 2: Extract the public key (e, n) from the issuer's certificate. Openssl provides commands to extract certain attributes from the x509 certificates. We can extract the value of n using -modulus. There is no specific command to extract e, but we can print out all the fields and can easily find the value of e.

```
For modulus (n):
$ openssl x509 -in c1.pem -noout -modulus
Print out all the fields, find the exponent (e):
$ openssl x509 -in c1.pem -text -noout
```

Step 3: Extract the signature from the server's certificate. There is no specific openssl command to extract the signature field. However, we can print out all the fields and then copy and paste the signature block into a file (note: if the signature algorithm used in the certificate is not based on **RSA**, you can find another certificate).

```
$ openssl x509 -in c0.pem -text -noout
...
Signature Algorithm: sha256WithRSAEncryption
  84:a8:9a:11:a7:d8:bd:0b:26:7e:52:24:7b:b2:55:9d:ea:30:
  89:51:08:87:6f:a9:ed:10:ea:5b:3e:0b:c7:2d:47:04:4e:dd:
  ......
  5c:04:55:64:ce:9d:b3:65:fd:f6:8f:5e:99:39:21:15:e2:71:
  aa:6a:88:82
```

We need to remove the spaces and colons from the data, so we can get a hex-string that we can feed into our program. The following command commands can achieve this goal. The tr command is a Linux utility tool for string operations. In this case, the -d option is used to delete ":" and "space" from the data.

```
$ cat signature | tr -d '[:space:]:'
84a89a11a7d8bd0b267e52247bb2559dea30895108876fa9ed10ea5b3e0bc7
......
5c045564ce9db365fdf68f5e99392115e271aa6a8882
```

Step 4: Extract the body of the server's certificate. A Certificate Authority (CA) generates the signature for a server certificate by first computing the hash of the certificate, and then sign the hash. To verify the signature, we also need to generate the hash from a certificate. Since the hash is generated before the signature is computed, we need to exclude the signature block of a certificate when computing the hash. Finding out what part of the certificate is used to generate the hash is quite challenging without a good understanding of the format of the certificate.

X.509 certificates are encoded using the ASN.1 (Abstract Syntax Notation.One) standard, so if we can parse the ASN.1 structure, we can easily extract any field from a certificate. Openssl has a command called asn1parse, which can be used to parse a X.509 certificate.

```
$ openssl asn1parse -i -in c0.pem
    0:d=0 hl=4 l=1522 cons: SEQUENCE
    4:d=1 hl=4 l=1242 cons: SEQUENCE     #
    8:d=2 hl=2 l= 3 cons: cont [ 0 ]
   10:d=3 hl=2 l= 1 prim: INTEGER        :02
   13:d=2 hl=2 l= 16 prim: INTEGER
   :0E64C5FBC236ADE14B172AEB41C78CB0
   ... ...
 1236:d=4 hl=2 l= 12 cons: SEQUENCE
 1238:d=5 hl=2 l= 3 prim: OBJECT         :X509v3 Basic Constraints
 1243:d=5 hl=2 l= 1 prim: BOOLEAN        :255
 1246:d=5 hl=2 l= 2 prim: OCTET STRING   [HEX DUMP]:3000
 1250:d=1 hl=2 l= 13 cons: SEQUENCE      *
 1252:d=2 hl=2 l= 9 prim: OBJECT         :sha256WithRSAEncryption
 1263:d=2 hl=2 l= 0 prim: NULL
 1265:d=1 hl=4 l= 257 prim: BIT STRING
```

The field starting from # is the body of the certificate that is used to generate the hash; the field starting from * is the signature block. Their offsets are the numbers at the beginning of the lines. In our case, the certificate body is from offset 4 to 1249, while the signature block is from 1250 to the end of the file. For X.509 certificates, the starting offset is always the same (i.e., 4), but the end depends on the content length of a certificate. We can use the -strparse option to get the field from the offset 4, which will give us the body of the certificate, excluding the signature block.

```
$ openssl asn1parse -i -in c0.pem -strparse 4 -out c0_body.bin -noout
```

Once we get the body of the certificate, we can calculate its hash using the following command:

```
$ sha256sum c0_body.bin
```

Step 5: Verify the signature. Now we have all the information, including the CA's public key, the CA's signature, and the body of the server's certificate. We can run our own program to verify whether the signature is valid or not. Openssl does provide a command to verify the certificate for us, but you are required to use their own programs to do so, otherwise, they get zero credit for this task.

5 Acknowledgement

This assignment is is largley adopted and modified from the SEED project (Developing Instructional Laboratories for Computer Security Education), at the website http://www.cis.syr.edu/~wedu/seed/index.html.