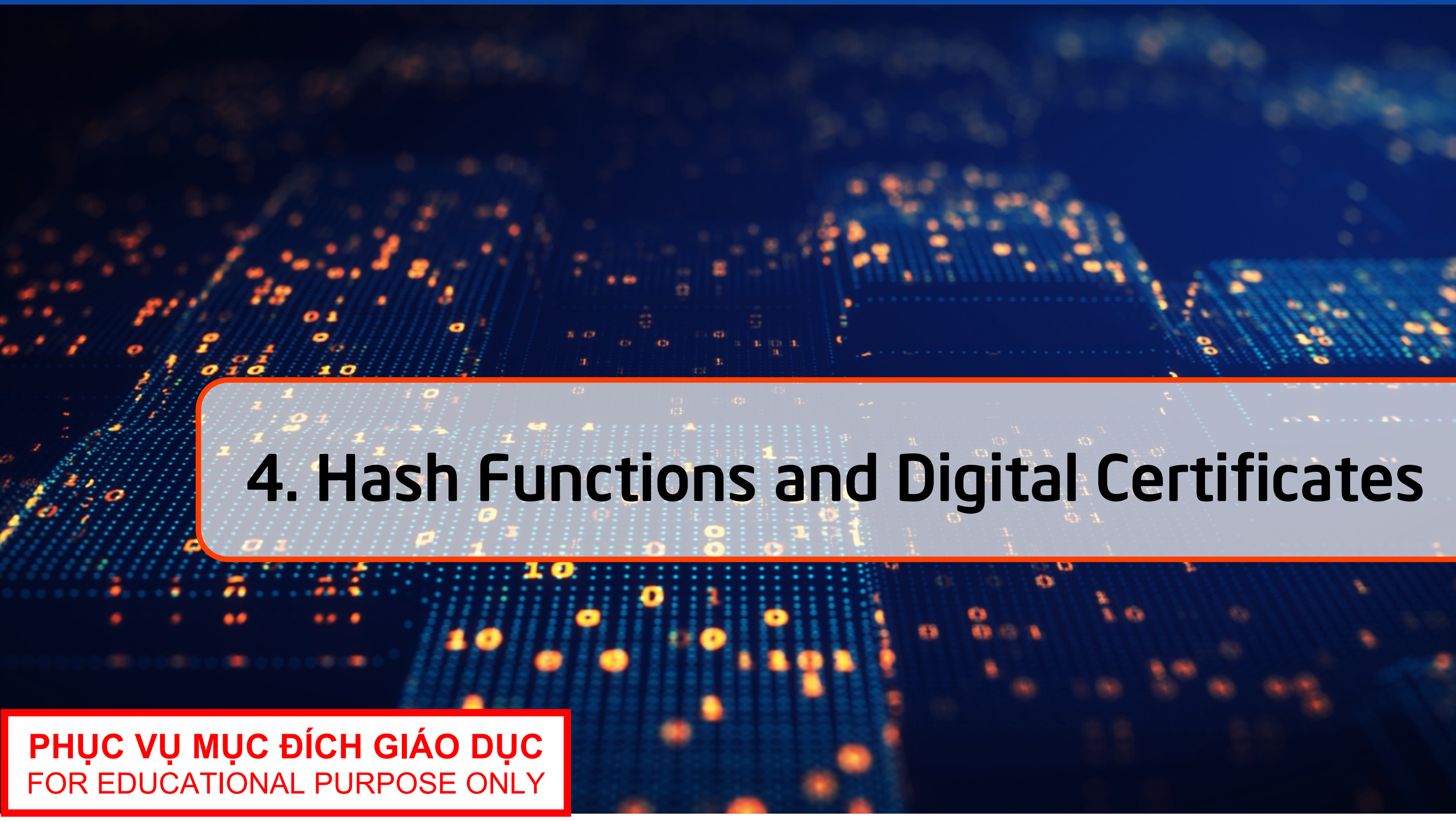
UNIVERSITY OF INFORMATION TECHNOLOGY – VNU-HCM
FACULTY OF COMPUTER NETWORKS AND COMMUNICATIONS



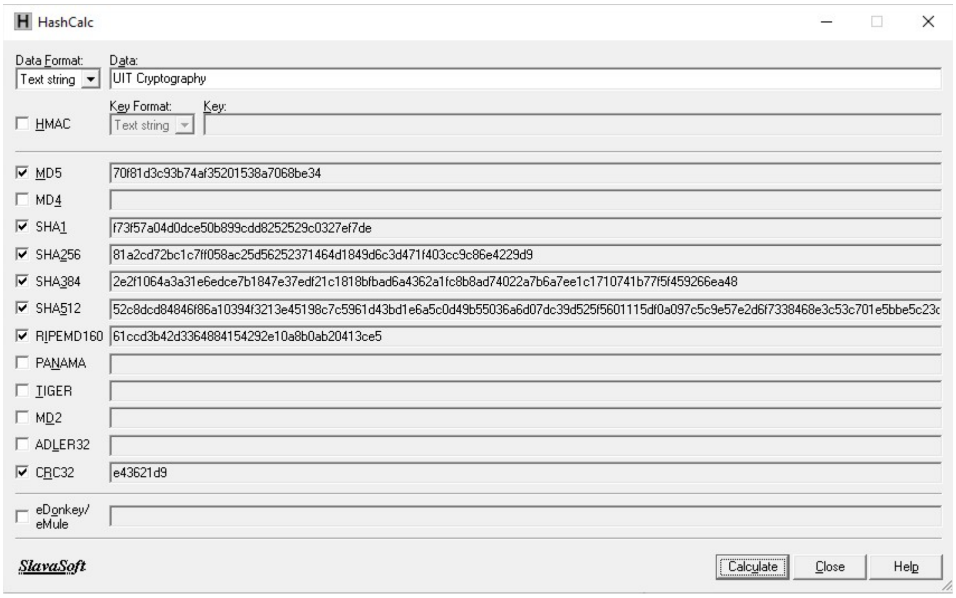
4. Hash Functions and Digital Certificates

A. LAB TASKS

1. Generating message digests (hash values) and HMAC

> Your task is to write an application (with C/C++/C#) to calculate hash values (at least three different types: MD5, SHA-1/SHA-2, SHA-3). For an input, which could be:
>
> - Text string
> - Hex string
> - File (support both text and binary files)
>
> You can use any hash library for your chosen programming language. Then, test your application with the following exercise:
>
> 1. Generate the hash values of the arbitrary message which contains your student ID. Then compare the results with other tools to verify.
> 2. Create three files which size up to 10 KB, 10 MB, and 10 GB. Generate hash values of these files.

Tips: *You can refer to a similar application like HashCalc (https://www.slavasoft.com/hashcalc/)*



HashCalc

Data Format: Text string
Data: UIT Cryptography

□ HMAC
Key Format: Text string
Key:

☑ MD5: 70f81d3c93b74af35201538a7068be34
□ MD4:
☑ SHA1: f73f57a04d0dce50b899cdd8252529c0327ef7de
☑ SHA256: 81a2cd72bc1c7ff058ac25d56252371464d1849d6c3d471f403cc9c86e4229d9
☑ SHA384: 2e2f1064a3a31e6edce7b1847e37edf21c1818bfbad6a4362a1fc8b8ad74022a7b6a7ee1c1710741b77f5f459266ea48
☑ SHA512: 52c8dcd84846f86a10394f3213e45198c7c5961d43bd1e6a5c0d49b55036a6d07dc39d525f5601115df0a097c5c9e57e2d6f7338468e3c53c701e5bbe5c23c
☑ RIPEMD160: 61ccd3b42d3364884154292e10a8b0ab20413ce5
□ PANAMA:
□ TIGER:
□ MD2:
□ ADLER32:
☑ CRC32: e43621d9
□ eDonkey/eMule:

Figure 1: HashCalc application on Windows OS

2. Hash properties: One-way vs Collision-free

It is now well-known that the cryptographic hash function MD5 and SHA-1 has been clearly broken (in terms of collision-resistance property). We will find out about MD5 and SHA-1 collision in this task by doing the following exercises:

1. Consider two HEX messages as follow:

   Message 1

   d131dd02c5e6eec4693d9a0698aff95c2fcab58712467eab4004583eb8fb7f89
   55ad340609f4b30283e488832571415a085125e8f7cdc99fd91dbdf280373c5b
   d8823e3156348f5bae6dacd436c919c6dd53e2b487da03fd02396306d248cda0
   e99f33420f577ee8ce54b67080a80d1ec69821bcb6a8839396f9652b6ff72a70

   Message 2

   d131dd02c5e6eec4693d9a0698aff95c2fcab50712467eab4004583eb8fb7f89
   55ad340609f4b30283e4888325f1415a085125e8f7cdc99fd91dbd7280373c5b
   d8823e3156348f5bae6dacd436c919c6dd53e23487da03fd02396306d248cda0
   e99f33420f577ee8ce54b67080280d1ec69821bcb6a8839396f965ab6ff72a70

   How many bits/bytes are the diffirent between two messages? Let's generate MD5 hash values for each message. Please observe whether these MD5 are similar or not and describe your observations in the lab report.

2. Download two PDF files: shattered-1 and shattered-2.pdf:
   - shattered-1.pdf: https://shattered.io/static/shattered-1.pdf
   - shattered-2.pdf: https://shattered.io/static/shattered-2.pdf

Open these files to check the difference. Then generate SHA-1 hash for them and observe the result.

3. Draw the conclusion base on your observations. Could you explain the reasons for the existence of collision in MD5 and SHA-1?

3. Generating Two Different Files with the Same MD5 Hash

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 Figure 2. The following command generates two output files, `out1.bin` and `out2.bin`, for a given a prefix the `prefix.txt`:

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

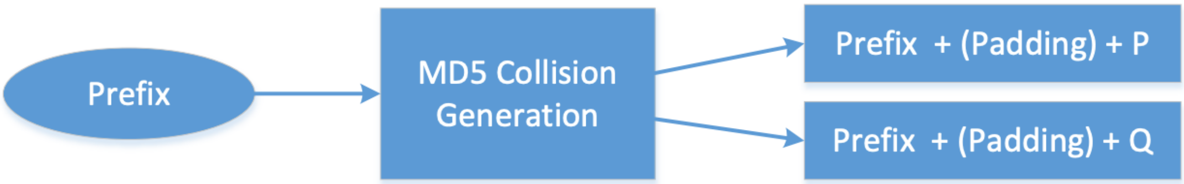


Figure 2: MD5 collision generation from a prefix

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 command.

```
$ 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. You can use the hex editor called `bless`.

> Let practice and answer the questions below:
>
> 64 bytes
>
> 1. If the length of your prefix file is not multiple of 64, what is going to happen?
> 2. Create a prefix file with exactly 64 bytes, and run the collision tool again, and see what happens.
> 3. Are the data (128 bytes) generated by md5collgen completely diffirent for the two output files? Please identify all the bytes that are different.
> 4. *(Advanced task)* Re-write md5collgen program in your own programming language.

4. Generating Two Executable Files with the same MD5 Hash

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 arbitrary values in the `xyz` array, and 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 the letter `A`. It will not be difficult to locate 200 `A` in the binary.

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

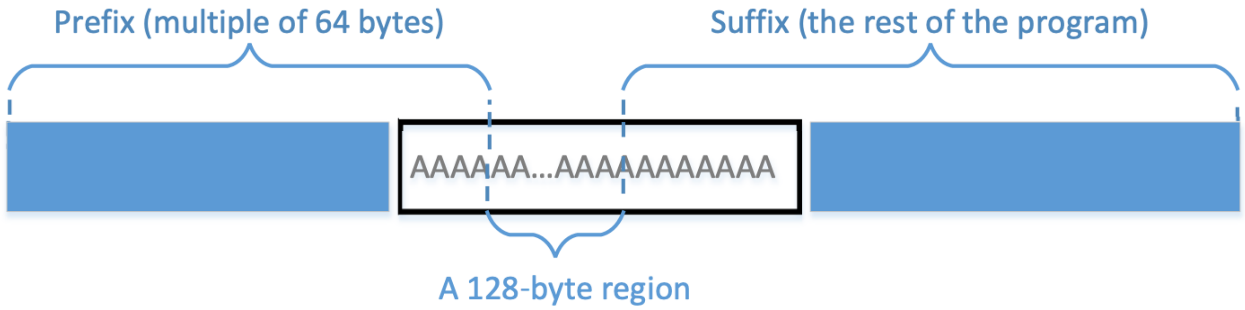

Prefix (multiple of 64 bytes)
Suffix (the rest of the program)
AAAAAA...AAAAAAAAAAA
A 128-byte region

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.

5. Manually Verifying an X.509 Certificate

In this task, we will manually verify an X.509 certificate. An X.509 contains data about a public key and an issuer's signature on the data. We will download a real X.509 certificate from a web server, and 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 webserver

We use the `www.example.org` server in this document. Students should choose a different web server that has a different certificate than the one used in this document (it should be noted that `www.example.com` maybe share the same certificate with `www.example.org`). 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\xC2\xA0Corporation\xC2\xA0for\xC2\xA0Assigned\xC2\xA0Names
\xC2\xA0and\xC2\xA0Numbers/CN=www.example.org
   i:/C=US/O=DigiCert Inc/CN=DigiCert TLS RSA SHA256 2020 CA1
-----BEGIN CERTIFICATE-----
MIIHRzCCBi+gAwIBAgIQD6pjEJMHvD1BSJJkDM1NmjANBgkqhkiG9w0BAQsFADBP
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMSkwJwYDVQQDEyBE
.....
0XELBnGQ666tr7pfx9trHniitNEGI6dj87VD+laMUBd7HBtOEGsiDoRSlA==
-----END CERTIFICATE-----
 1 s:/C=US/O=DigiCert Inc/CN=DigiCert TLS RSA SHA256 2020 CA1
   i:/C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert Global Root CA
-----BEGIN CERTIFICATE-----
MIIEvjCCA6agAwIBAgIQBtjZBNVYQ0b2ii+nVCJ+xDANBgkqhkiG9w0BAQsFADBh
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
.....
A7sKPPcw7+uvTPyLNhBzPvOk
-----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.

Copy and paste each of the certificates (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 ou all the fields, find the exponent (e):
$ openssl x509 -in c1.pem -text
```

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 1.pem –text
...
Signature Algorithm: sha256WithRSAEncryption
         aa:9f:be:5d:91:1b:ad:e4:4e:4e:cc:8f:07:64:44:35:b4:ad:
         3b:13:3f:c1:29:d8:b4:ab:f3:42:51:49:46:3b:d6:cf:1e:41:
         ......
         0e:84:52:94
```

We need to remove the spaces and colons from the data, so we can get a hex-string that we can deed into our program. The following 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 used to extract data from ASN.1 formatted data, and is able to parse our 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          ❶
    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          ❷
 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 ❶ is the body of the certificate that is used to generate the hash; the field starting from ❷ 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 students are required to use their own programs to do.

6. Programming application for X509 Certificate verification

> Writing an application (using C++ with the cryptopp library) to get all information and verify the x509 certificate.
>
> **Input**: The path to all certificate files in the chain from your computer.
> **Output**: Each certificate in the chain shows its information (e.g., version, identity, validity date, etc.) and the verified result.

B. REQUIREMENTS

You are expected to complete all tasks in section B (Lab tasks). Advanced tasks are optional, and you could get bonus points for completing those tasks. We prefer you work in a team of 2 members to get the highest efficiency.

Your submission must meet the following requirements:

- You need to submit a **detailed lab report in .docx** *(Word Document)* format, **using the report template** provided on the UIT Courses website.
- Either Vietnamese or English report is accepted. That's up to you. Using more than one language in the report is not allowed (except for the untranslatable keywords).
- When it comes to **programming tasks** *(require you to write an application or script),* please attach all source-code and executable files (if any) in your submission. Please also list the important code snippets followed by explanations and screenshots when running your application. Simply attaching code without any explanation will not receive points.
- Submit work you are proud of – don't be sloppy and lazy!

  Your submissions must be your own. You are free to discuss with other classmates to find the solution. However, copying reports is prohibited, even if only a part of your report. Both reports of owner and copier will be rejected. Please remember to cite any source of the material (website, book,…) that influences your solution.

Notice: Combine your lab report and all related files into a single **ZIP file (.zip)**, name it as follow:

StudentID1_StudentID2_ReportLabX.zip

C. REFERENCES

[1] MD5 Collision Attack Lab, Wenliang Du (Syracuse University), *SEED Cryptography Labs* https://seedsecuritylabs.org/Labs_20.04/Crypto/Crypto_MD5_Collision/

[2] RSA Encryption and Signature, Wenliang Du (Syracuse University), *SEED Cryptography Labs:* https://seedsecuritylabs.org/Labs_20.04/Crypto/Crypto_RSA/

[3] X509 Certificate
https://www.cryptopp.com/wiki/X509Certificate

Attention: *Don't share any materials (slides, readings, assignments, labs, etc..) out of our class without my permission!*