ECE568

Computer Security

Lab #2

Cryptography: Hashing and Certificates

Overview

In this lab you will be gaining some hands-on experience with tools for secure hashing, public-key certificates and digital signatures. These tools help secure many core Internet technologies – including web-server security (HTTPS), VPNs (OpenVPN), secure shells (SSH), blockchains (Bitcoin, Ethereum, *etc.*), and digital contracts (DocuSign, *etc.*).

In this lab you will be given two Python programs that you need to add to: the first implements a secure form of a *Merkle Tree*, and the second performs X509 certificate signing. Unlike Lab 1, these scripts are far less-sensitive to the environment that you run them on (so you can likely perform much of your development on your personal computers, or other UofT systems) – but your solutions must ultimately run correctly on the ECF environment.

Part 1: Creating a Secure Merkle Tree

In this section you will be extending some Python code and creating a secure form of a *Merkle Tree*. A Merkle Tree is a hash-based data-structure that forms the basis of several popular blockchain signature algorithms, and several *zero-knowledge proofs* that can be used to help to secure and validate cloud storage.

Your task in this part is to extend the **merkleTree.py** code that you are provided, so that:

1. If you run the script with only a single hash value as a parameter, then it will output a **proof** (described below); and,

   ```
   $ ./merkleTree.py 819539069f383c771e4fe42437…
   ['819539069f383c771e4fe42437e82539fcd7fde44217ea7a4c4f
   cb9eaf4b07b9', '1c40791ee4…']
   $
   ```

2. If you instead run the script with a full **proof** provided on the command line, then it will validate the proof (also described below) and return **True** or **False**.

```
$ ./merkleTree.py 819539069f383c771e4fe42… …13e9dbae
True
$
```

You should likely start your work by running:

```
$ ./util/testPart1.py
```

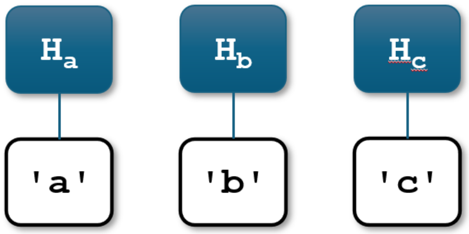
The tests will initially fail – but the "test" script will create some input files for your script that will be useful in your initial testing.

Building a Merkle Trees starts with gathering the collection of data objects that you want to protect. These data objects can be files, database records, blockchain transactions, etc.. For this part of the lab we will be using a collection of random strings; for example:

```
values = [ 'a', 'b', 'c' ]
```

For this lab, you can assume that: (a) the values are unordered (*i.e.*, you do not need to sort them); (b) they are inserted into the tree from left to right (in the order you receive them); and, (c) there are no duplicate values. You should read your input values from the file named "values.json" (sample code for reading these values is provided for you in **merkleTree.py**).

The first step of building a Merkle Tree involves building a hash of each of the values:



For this lab we will be building a special, secure form of a Merkle Tree. For our hash values we will use HMACs (secured hashes), as discussed in class. You should build your HMAC using the SHA256 hashing algorithm and a pre-shared key of "ECE568". (Refer to our lecture notes for more details on this.) Your hash output should be stored as hex characters, in an ASCII string. (Because we are building a more-secure version, the various Python libraries that implement Merkle Trees will not work; you will need to implement your own solution.)

As an aid to verify that you are calculating your HMAC correctly, a call to:

```
hash('a')
```

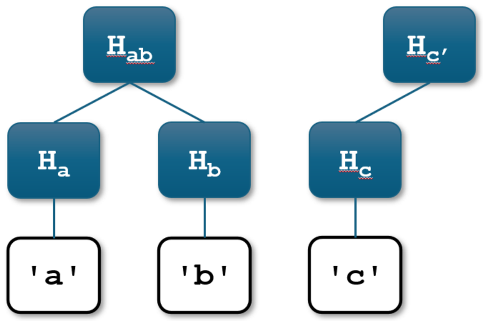
should return a result of:

```
'819539069f383c771e4fe42437e82539fcd7fde44217ea7a4c4fcb9eaf4b07b9'
```

It is fine to use calls to an existing Python library that implements **HMAC**; you do not need to implement that entire function yourself – but it needs to adhere to the hashing requirements, above. The three calculated HMAC values for the figure above would be:

H_a = '819539069f383c771e4fe42437e82539fcd7fde44217ea7a4c4fcb9eaf4b07b9'
H_b = '1c40791ee4fdcdde52b696c24c3be69ddcdde082aa870635099fdf905d780992'
H_c = 'afdac49e19863ecf49563e7bc26b4dfd996bb93ce7d8bd857b09471fcb13dd4c'

Next we start to build a binary tree, continuing to “hash the hash values” as we work towards the root of the tree:



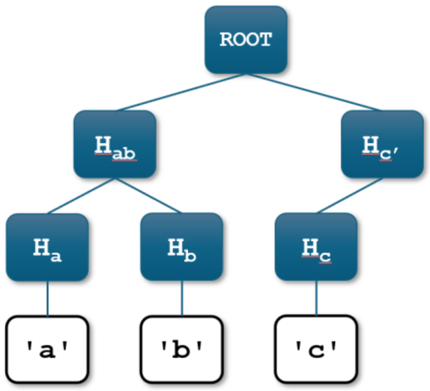
These values are calculated using the same HMAC as before. Each node contains the HMAC of the hash values of its **left** (and, optionally, **right**) children beneath it; for example:

H_{ab} = HMAC (H_a + H_b)
= HMAC('819539069f383c771e4fe42437e82539fcd7fde44217ea7a4c4fcb9eaf4b07b9' + '1c40791ee4fdcdde52b696c24c3be69ddcdde082aa870635099fdf905d780992')
= HMAC('819539069f383c771e4fe42437e82539fcd7fde44217ea7a4c4fcb9eaf4b07b91c40791ee4fdcdde52b696c24c3be69ddcdde082aa870635099fdf905d780992')
= '90a2da2252589731ed8234c973d256e4d3b2fde80fb7a0e56f86f6d1d6c1b8cb'

The node $H_{c'}$ only has one child; while building *our* Merkle Tree we will we calculate its value by re-hashing the single node below it:

$H_{c'}$ = HMAC (H_c)
= HMAC('afdac49e19863ecf49563e7bc26b4dfd996bb93ce7d8bd857b09471fcb13dd4c')
= 'e8b23d4ba1b373dc1b1e2b9a724440ddd48e6f9ecae7a5ae108d1e98b5acf526'

We continue this process until we reach the root node; this value is referred to as the **root hash value**:



To complete this example, the root hash value for this tree is:

> **ROOT = HMAC(H_{ab} + $H_{c'}$)**
>
> **= HMAC(**`'90a2da2252589731ed8234c973d256e4d3b2fde80fb7a0e56f86f6d1d6c1b8cb' + 'e8b23d4ba1b373dc1b1e2b9a724440ddd48e6f9ecae7a5ae108d1e98b5acf526'`**)**
>
> **= HMAC(**`'90a2da2252589731ed8234c973d256e4d3b2fde80fb7a0e56f86f6d1d6c1b8cbe8b23d4ba1b373dc1b1e2b9a724440ddd48e6f9ecae7a5ae108d1e98b5acf526'`**)**
>
> **=** `'a9424f4004232abe6a074543035172da3460f1f4d2c47a110c0e5f9a74a610b6'`

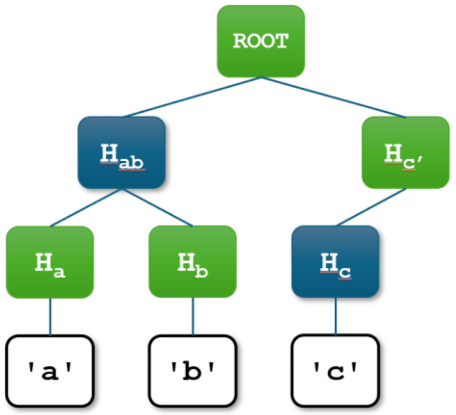
Now that we have build our Merkle Tree, we can use the tree to generate **proofs**.

A **proof** is a collection of hash values (sometimes referred to as a "possession proof") that allows us to prove to a third party that we have a given value in our tree – without divulging any information about value, or the values of the other items in our tree. For this lab, we assume that we have previously shared our HMAC key ("ECE568") and our Root Hash Value with the third party.

If we are asked to provide a *proof* that the value '**a**' is in our tree, then the third party will send us the HMAC value H_a:

```
$ ./merkleTree.py 819539069f383c771e4fe42437e82539fcd
7fde44217ea7a4c4fcb9eaf4b07b9
```

In response, we need to provide the minimum set of hash values the third-party would need to verify that we indeed have '**a**'. In this case, we will need to send four values:



A proof starts by sending the requested node (H_a), followed by the other hash value that we need to calculate the node above H_a (in this case, H_b). From those two values the recipient can calculate H_{ab}:

$$H_{ab} = \mathrm{HMAC}(H_a + H_b)$$

Next we send the other hash value that we need to calculate the node above H_{ab} (in this case, $H_{c'}$). With that additional value the recipient can calculate the Root Hash Value:

$$\mathrm{ROOT} = \mathrm{HMAC}(H_{ab} + H_{c'})$$

We then send the ROOT value, so that the recipient can confirm that our calculations match.

As a final step, we are going to complete our proof by attaching a digital signature of the ROOT value, signed with your RSA private key. **merkleTree.py** contains sample code for reading the private key (from "privateKey.pem") and generating the signature.

So, for this sample tree, the full proof would be:

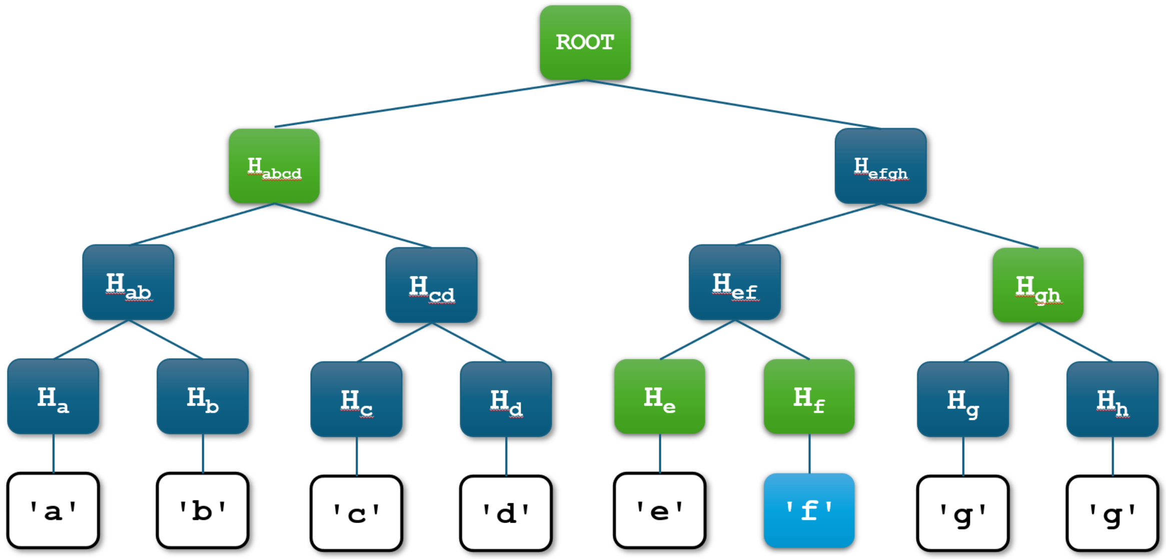
$$[\ H_a,\ H_b,\ H_{c'},\ \mathrm{ROOT},\ \mathrm{signature(ROOT)}\ ]$$

...and the corresponding output would be:

```
$ cat values.json
["a", "b", "c"]
$ ./merkleTree.py 819539069f383c771e4fe42437e82539fcd7f
de44217ea7a4c4fcb9eaf4b07b9
['819539069f383c771e4fe42437e82539fcd7fde44217ea7a4c4fcb9eaf4b07b9',
'1c40791ee4fdcdde52b696c24c3be69ddcdde082aa870635099fdf905d780992',
'e8b23d4ba1b373dc1b1e2b9a724440ddd48e6f9ecae7a5ae108d1e98b5acf526',
'a9424f4004232abe6a074543035172da3460f1f4d2c47a110c0e5f9a74a610b6',
'33a8d407306521e75ea5f257363ce5dda6361305c96ac1e31af6b2cff9592e66af54ec
006e2e9c50a4414eab8bb144fdaf0270824b7bde48b9331b43f5491a5c28f8f15673d23
551c3d2223be0ae105814a16876a4c967f5d21460ec2b461fd832a18c469ffab44d248d
5849c673afe4c941815584928c27fd80958b5749b98c0cb283d7ab4a123c645f3f86df4
f981d9386503c4f0c04f73d92cb5ba9d132cac23035fb2251319eb222191233ddd2a220
2eae893230df8867e0c0800e7923ec4e976dc707d36a6a15985a6f4e13984f156d45f86
f20c7b9418fbf5c5f85590b3d8a51262d2dd7a88c5e05d175ce617623a7ac184d06b3c4
34a8e8a5b9b3e2f70c0952e8cf46270b7483793c40aea307c2b3e4137c2430fb548fcf4
e6e7803b62c51aa97c5eb66f7555dc55d344e003bfc50337aab948e3ced37d8c2ec1006
06f7f28b71dcfe6ed9a61d1fa7319e6e100287878c397ab7379e5dd2efd1fdac812e2f2
8671e9cb265eac90279ede09cfb91116162d36a22c3943fbabc43b2414dc4b5bc28dd8b
3e1eba3f89b606940e34b81c48dd54f133daecb91fff07678c014ffa4c736f7708e7d70
af2895372dc74dd915e4a874577cefd1d426c26b985c1d22dfdeacf95ab7865851e66e3
e6dd9ac4a84692915a63de2c9207ff84a0efee6c72e19e9777c90a8382c24e615020893
b90f42c48b84ffcf81785ac13e9dbae']
$
```

The first four hash values will be the same on each run. However, the signature (the final value) contains a random **Initialization Vector**; as a result, it will have different values on each run. Importantly, an attacker listening to this communication would not know any of the values stored in our tree.

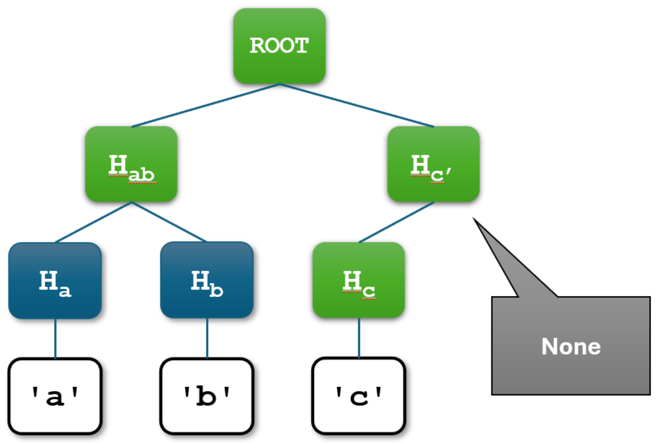
In a larger example, with more values in the tree, your **proof** will contain a larger sequence of hash values:



Here your proof would contain:

`[` H_f`,` H_e`,` H_{gh}`,` H_{abcd}`, ROOT, signature(ROOT) ]`

One special case is worth noting: if a node has only a single child, then you should return **None** (*i.e.*, the Python value None, not a string 'None') for its hash value:



```
$ ./merkleTree.py afdac49e19863ecf49563e7bc26b4dfd996bb93ce7d8b
d857b09471fcb13dd4c
['afdac49e19863ecf49563e7bc26b4dfd996bb93ce7d8bd857b09471fcb13dd4c', None,
'90a2da2252589731ed8234c973d256e4d3b2fde80fb7a0e56f86f6d1d6c1b8cb',
'a9424f4004232abe6a074543035172da3460f1f4d2c47a110c0e5f9a74a610b6',
'42e0c3f8f07656bfb3176a78884d6d67ecb29c788fb3fabeb36166e4abb51bac297490369392
ee4755f63b853b2855b8ba104aac30a28f4bdf6f8f7a6e89699eed63db8176cf0ff78ad685589
3a793a0ff4d0738195ea665d7443c3468243b81be1d0a4d49a7da197ac0cdf20dd8701f5d8c8f
65173b198e9ef473ed3ceae615046b27e0c2033fb8e7149af484ab8c6a1ca841fd3f5dea9e97f
a863f653186459c76b01d4e5b960487077531aac85f90215cfa08ca5cc0527caea3f2a00df349
368cefecde0214a9e2ff8c003655dd49a173ca694089cad5b9466b8d54fb0d8ffc32cf496da41
e38b439b2fbefd75ba39db5853458e6d805baee15d8ac9e614e20e11cf6118f88e70eed9bacb0
580d72d2ce0e78e8afceb2e9fdddbd7eda1e88165a371448d9cbb2ad466f97a7e4aab68eebf74
efdf2a2824b9820b8276d6d1f9de93fbf693157bbb3abaeea7cbec2140b922d2ce07260e775e7
eaac8c4a305ec666e3030f9f52d41b5b7aa3e9282ae4fb531c64e98cd01d7d566f6b26705b2b1
8615799aab568c45653a7614f25dec1d2f7fa5758c75f778c79c68a29cae2637daa0674139ae0
f3c50ae39b4fcb8454ae164719c97397edb8dd80bb8e6908c568f676ac31980f8bf61f3816dc6
31c354972dda2ce09c10b16b740a97f2d1530d1ea1c7f0e484a69f45dc7b4a0502f9ef8779eed
37dc889aa7200dd1cd3f5c69']
$
```

For the purposes of this lab, you can assume that we will only request values that are actually in your tree.

This allows us to generate a proof – which is the first half of your task.

As the second step, your script also needs to support *validating* the proof, by providing the hashes and signature on the command line:

```
$ ./merkleTree.py \
819539069f383c771e4fe42437e82539fcd7fde44217ea7a4c4fcb9eaf4b07b9 \
1c40791ee4fdcdde52b696c24c3be69ddcdde082aa870635099fdf905d780992 \
e8b23d4ba1b373dc1b1e2b9a724440ddd48e6f9ecae7a5ae108d1e98b5acf526 \
a9424f4004232abe6a074543035172da3460f1f4d2c47a110c0e5f9a74a610b6 \
33a8d407306521e75ea5f257363ce5dda6361305c96ac1e31af6b2cff9592e66af54ec0
06e2e9c50a4414eab8bb144fdaf0270824b7bde48b9331b43f5491a5c28f8f15673d235
51c3d2223be0ae105814a16876a4c967f5d21460ec2b461fd832a18c469ffab44d248d5
849c673afe4c941815584928c27fd80958b5749b98c0cb283d7ab4a123c645f3f86df4f
981d9386503c4f0c04f73d92cb5ba9d132cac23035fb2251319eb222191233ddd2a2202
eae893230df8867e0c0800e7923ec4e976dc707d36a6a15985a6f4e13984f156d45f86f
20c7b9418fbf5c5f85590b3d8a51262d2dd7a88c5e05d175ce617623a7ac184d06b3c43
4a8e8a5b9b3e2f70c0952e8cf46270b7483793c40aea307c2b3e4137c2430fb548fcf4e
6e7803b62c51aa97c5eb66f7555dc55d344e003bfc50337aab948e3ced37d8c2ec10060
6f7f28b71dcfe6ed9a61d1fa7319e6e100287878c397ab7379e5dd2efd1fdac812e2f28
671e9cb265eac90279ede09cfb91116162d36a22c3943fbabc43b2414dc4b5bc28dd8b3
e1eba3f89b606940e34b81c48dd54f133daecb91fff07678c014ffa4c736f7708e7d70a
f2895372dc74dd915e4a874577cefd1d426c26b985c1d22dfdeacf95ab7865851e66e3e
6dd9ac4a84692915a63de2c9207ff84a0efee6c72e19e9777c90a8382c24e615020893b
90f42c48b84ffcf81785ac13e9dbae

True
$
```

To validate the proof you should:

1. Read in the provided values.
2. Calculate your own proof values.
3. Check that your calculated hash values match the provided hash values.
4. Use the public key to verify that the signature (the last parameter) is valid.

If your calculated values match and the signature is valid then return True, otherwise return False.

If your calculated values are correct then the **testPart1.sh** script should report "TEST PASSED":

```
$ ./util/testPart1.sh
- Checking a hash value... TEST PASSED
- Checking a valid proof... TEST PASSED
- Checking an invalid proof... TEST PASSED
```

In order to test your script with other inputs you can also run **./util/createValues.py** to create other values, and **./util/generateKeypair.py** to create other public/private keypairs.

Part 2: X509 Certificates

In the previous section you used a private key to sign your data proof. In this section you will be learning how to create an X509 certificate for your private key; this will allow you to share your public key with other people, so that they know that the key belongs to you. (This is simulating what a **Certificate Authority** like "Let's Encrypt" does when you request a website certificate.)

You are provided with some sample code in **generateX509.py** that starts the process of generating a **Certificate Signing Request**. This is the first step in creating an X509 certificate.

Your first task is to complete the code, generate a **Certificate Signing Request** (CSR), and print it to the screen in PEM format. When you have finished adding your code for this, your script output should look similar to this:

```
$ ./generateX509.py
-----BEGIN CERTIFICATE REQUEST-----
MIIEmDCCAoACAQAwUzELMAkGA1UEBhMCQ0ExCzAJBgNVBAgMAk9OMRAwDgYDVQQH
DAdUb3JvbnRvMQ8wDQYDVQQKDAZFQ0U1NjgxFDASBgNVBAMMC0VDRTU2OCBMYWIy
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEAk9xFU+Xnjg0A7RzTyHwn
fMN0JdhR4N0LlIvmyM0h6nYVfC32TGDRgvjQi28+MFg0cVyH28lUiLgWIv4qOMHQ
iHgEP++xdWY7cSnRV9wSM8A7nUGT4HFN0rundiGhsVxlUWbF7F7BdNnv7gsLnLjn
pcNdkSETEh6VT0TDj/E0NYfg3Y9wJsdqH1MqUI2Qkn64sVRciEPQ6GgFSOFkD4Tx
uhzNFaXOg2L30hJlnjR8DB3MpanAVooQuVOuhdJLkADS5wvjYXC4PQwxMr5xhRPI
LuLzuYu+dk3CObpzrCZJGj4Z3G9UnjXYrac5J6RV6LESb3TKQ+wGqoPHFYYQn+Gu
t05yO6wAtR7VOdUejyqs5cJll/zC7txhy/pN3cYhGWbeVJdKCAPf129KYax2FQyw
EIjWkx6rdyR10Us2pHdP8U/Uz3RJcvWgtTmnQ1CfWCrCVjB8JpmbiWJ3sKLG7k1u
bRKQzueCYpKjp6i+dtBfudoJ/Bc1iWB3eY416QmaO0uNv0sj+cW1mzS2zYIRCH0N
5Xm+fYJ+T9SsY7+rKePXYTFmjFc9tZQL+dUF9iiGQnacAMd4knG0vmSPDEGqHUSp
N+fh/ZFEprZzBTDjVVnkhkikTsaOdUucb7lPWFWlrUs/FD3EZlUVSgfV+ju5BbJf
uucHgeE696mAb43LBSNtQ+UCAwEAAaAAMA0GCSqGSIb3DQEBCwUAA4ICAQBZj1Ac
ktj/Q+PwBrRV0xYxBRUW85G/g9vs+4/ZmezzIYtb2xZBDWdPm/mcOPt+EMzF2ALp
ypSldaMBglb/lTyUqJFrOva6YlJq7aGriUM8kL0OfJGsiCS+lc+XtnG8lftlYg4P
IP7he6DQhBK+P7Z+qYZcdjBH3cSJ9HzrInnbNMYqz06vCzXXqjutIAGw5pWvH0Oc
oH7pM6XcDoOeU/YZg+e2eG2cMeDk4ASq/NJ/kujy3Gjxd+u7mmNMoSRhoMVBTbqJ
qA1Iuagi/AN3J/K8AS5YM0anm+11SIvTbywGh1HCmApf8VQeOeHDiBhaaYSMaoDG
jrwEb5KXwK6uZlAesUcw2/7iITg4YDIbD0qDn6yZwFKDDJ7NwbfFFwD1YE//w3H5
bGrKbgxQ9dt/9TaO4vG2Pf98M2vsPEw4XZlvtNrrjFtqOcairi8F1kuM9p3UPFRi
QTYsroRlrqEcx9Nh5qeiYNFtoTOKKPUzXuuE6pxpR4wyqMRE5xlBS6g4QiZqD5yg
0R++5rhBSDAPthvwBWC4zaJf2B/YkChjGSc9sbOVi5jzgCOLR1dfjHfGtOzxWX5I
YmE3lABI4Y32QfUw7PLOFABsvRKDPHHzbV39nnBmbXeFMctpOGymsOGSOTmGUqQh
TIoZ5aO9KTfWnFoQ/1yDNR+OEYI96dMpBI12bQ==
-----END CERTIFICATE REQUEST-----
```

To generate the CSR you will need to add code to complete the following steps:

1. Open the **privateKey.pem** file and read in the private key. (Please do not embed the key into your script; your script will be marked with a different key.)
2. Sign the CSR with the private key.
3. Print out the CSR in PEM format.

If you want to check that your CSR is properly formatted, you can copy the text into a file and then use the OpenSSL command-line to parse the certificate; it should look something like this:

```
$ openssl req -in yourFilename -text
Certificate Request:
    Data:
        Version: 1 (0x0)
        Subject: C=CA, ST=ON, L=Toronto, O=ECE568, CN=ECE568 Lab2
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (4096 bit)
                Modulus:
                    00:93:dc:45:53:e5:e7:8e:0d:00:ed:1c:d3:c8:7c:
                    27:7c:c3:74:25:d8:51:e0:dd:0b:94:8b:e6:c8:cd:

[…]
```

Finally, you should use the **Certificate Authority** files located in the ./util/ directory to sign your CSR and generate an X509 certificate. To do this, you should:

1. Load the CA certificate (located in ./util/CA_cert.pem) and the CA private key (located in ./util/CA_key.pem). Again you must read the keys from the file, rather than hard-coding them into your script. This is the "Certificate Authority" that will be signing and issuing your X509 certificate.

2. Assign the appropriate values for at least the following fields in the X509 certificate:
    a. Subject Name
    b. Issuer Name
    c. Serial Number (this can be a random number)
    d. Not Valid Before (should be the current time)
    e. Not Valid After (should be 90 days from now)
    f. Public Key (from your CSR)
    g. Subject Key Identifier (from your CSR)
    h. Authority Key Identifier (from the CA_cert)

As well, you will need to set the following "Usage Flags" on the X509 certificate, to indicate that this is being used to create a digital signature (as opposed to identifying a website, etc.):

```
digital_signature   = True
content_commitment  = False
key_encipherment    = False
data_encipherment   = False
key_agreement       = False
key_cert_sign       = False
crl_sign            = False
encipher_only       = False
decipher_only       = False
```

Once you have set the appropriate values in the X509 certificate, you will need to:

1. Sign the X509 certificate with the CA's private key (the one you read from CA_key.pem).
2. Convert the X509 certificate to PEM format.
3. Print a blank line (to provide a separator after your CSR request) and then print a **certificate chain**: the Certificate Authority X509 certificate (from CA_crt.pem) and then your newly-generated X509 certificate for your private key.

The entire output of the script (both parts) should look similar to the following:

```
$ ./generateX509.py
-----BEGIN CERTIFICATE REQUEST-----
MIIEmDCCAoACAQAwUzELMAkGA1UEBhMCQ0ExCzAJBgNVBAgMAk9OMRAwDgYDVQQH
DAdUb3JvbnRvMQ8wDQYDVQQKDAZFQ0U1NjgxFDASBgNVBAMMC0VDRTU2OCBMYWIy
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEAk9xFU+Xnjg0A7RzTyHwn
fMN0JdhR4N0LlIvmyM0h6nYVfC32TGDRgvjQi28+MFg0cVyH28lUiLgWIv4qOMHQ
iHgEP++xdWY7cSnRV9wSM8A7nUGT4HFN0rundiGhsVxlUWbF7F7BdNnv7gsLnLjn
pcNdkSETEh6VT0TDj/E0NYfg3Y9wJsdqH1MqUI2Qkn64sVRciEPQ6GgFSOFkD4Tx
uhzNFaXOg2L30hJlnjR8DB3MpanAVooQuVOuhdJLkADS5wvjYXC4PQwxMr5xhRPI
LuLzuYu+dk3CObpzrCZJGj4Z3G9UnjXYrac5J6RV6LESb3TKQ+wGqoPHFYYQn+Gu
t05yO6wAtR7VOdUejyqs5cJll/zC7txhy/pN3cYhGWbeVJdKCAPf129KYax2FQyw
EIjWkx6rdyR10Us2pHdP8U/Uz3RJcvWgtTmnQ1CfWCrCVjB8JpmbiWJ3sKLG7k1u
bRKQzueCYpKjp6i+dtBfudoJ/Bc1iWB3eY416QmaO0uNv0sj+cW1mzS2zYIRCH0N
5Xm+fYJ+T9SsY7+rKePXYTFmjFc9tZQL+dUF9iiGQnacAMd4knG0vmSPDEGqHUSp
N+fh/ZFEprZzBTDjVVnkhkikTsaOdUucb7lPWFWlrUs/FD3EZlUVSgfV+ju5BbJf
uucHgeE696mAb43LBSNtQ+UCAwEAAaAAMA0GCSqGSIb3DQEBCwUAA4ICAQBZj1Ac
ktj/Q+PwBrRV0xYxBRUW85G/g9vs+4/ZmezzIYtb2xZBDWdPm/mcOPt+EMzF2ALp
ypSldaMBglb/lTyUqJFrOva6YlJq7aGriUM8kL0OfJGsiCS+lc+XtnG8lftlYg4P
IP7he6DQhBK+P7Z+qYZcdjBH3cSJ9HzrInnbNMYqz06vCzXXqjutIAGw5pWvH0Oc
```

```
oH7pM6XcDoOeU/YZg+e2eG2cMeDk4ASq/NJ/kujy3Gjxd+u7mmNMoSRhoMVBTbqJ
qA1Iuagi/AN3J/K8AS5YM0anm+11SIvTbywGh1HCmApf8VQeOeHDiBhaaYSMaoDG
jrwEb5KXwK6uZlAesUcw2/7iITg4YDIbD0qDn6yZwFKDDJ7NwbfFFwD1YE//w3H5
bGrKbgxQ9dt/9TaO4vG2Pf98M2vsPEw4XZlvtNrrjFtqOcairi8F1kuM9p3UPFRi
QTYsroRlrqEcx9Nh5qeiYNFtoTOKKPUzXuuE6pxpR4wyqMRE5xlBS6g4QiZqD5yg
0R++5rhBSDAPthvwBWC4zaJf2B/YkChjGSc9sbOVi5jzgCOLR1dfjHfGtOzxWX5I
YmE3lABI4Y32QfUw7PLOFABsvRKDPHHzbV39nnBmbXeFMctpOGymsOGSOTmGUqQh
TIoZ5aO9KTfWnFoQ/1yDNR+OEYI96dMpBI12bQ==
-----END CERTIFICATE REQUEST-----

-----BEGIN CERTIFICATE-----
MIIGQjCCBCqgAwIBAgITWS/lXmfYn41Qs+xg8JFgodJM9TANBgkqhkiG9w0BAQsF
ADCBsDELMAkGA1UEBhMCQ0ExCzAJBgNVBAgMAk9OMRAwDgYDVQQHDAdUb3JvbnRv
MR4wHAYDVQQKDBVVbml2ZXJzaXR5IG9mIFRvcm9udG8xDzANBgNVBAsMBkVDRTU2
ODElMCMGA1UEAwwcRUNFNTY4IENlcnRpZmljYXRlIEF1dGhvcml0eTEqMCgGCSqG
SIb3DQEJARYbY291cnRuZXkuZ2lic29uQHV0b3JvbnRvLmNhMB4XDTI1MDEyNjA1
MTMxOVoXDTM1MDEyNDA1MTMxOVowgbAxCzAJBgNVBAYTAkNBMQswCQYDVQQIDAJP
TjEQMA4GA1UEBwwHVG9yb250bzEeMBwGA1UECgwVVW5pdmVyc2l0eSBvZiBUb3Jv
bnRvMQ8wDQYDVQQLDAZFQ0U1NjgxJTAjBgNVBAMMHEVDRTU2OCBDZXJ0aWZpY2F0
ZSBBdXRob3JpdHkxKjAoBgkqhkiG9w0BCQEWG2NvdXJ0bmV5LmdpYnNvbkB1dG9y
b250by5jYTCCAiIwDQYJKoZIhvcNAQEBBQADggIPADCCAgoCggIBAMhMxEZdGcqg
gM+KF3+k9wNzApAum2h8BaSKpqq1rDpGkB7T5S9VDoHRrZziCogXEeMvE7gXzDep
PrCgV1MhJoXWaCNRK371CMrjoaowoB2oba8EYb7EjaEypbYRDGkNbMVzmZYJQzmP
ctGtzBH3oHO0O8qXe15640UpXpbqDSU7XBrtzRSBCXevw1DSaWBWDgosFD6frNK+
uzCYnTymw/Ic4DMa19W3Xb6VWTCKFhrhCJos/zmV9NCDcyx9CMqvNKolycK6NRR1
h16JYU+SyHoivHxlA9ia9NZLO+tYAMddTe4bLLYZmUNVRUTQJqu1sB3nIM6pEsxg
v+xec+nkGimk4EnbMmPMZQL3eWsbltQBIoTfpdMCy09rogK90sFwC1RH2laW6vlF
vDJU/qIIniB0JI7W078skJXnQK0lIg/pk/2kGEobNn4njlEiAPlYDEK6jv98b5hB
7At93++j6g9TdihqRiS9CecG/vzUIgJiZLRptsT9dI6kuXn2fVtQ8OzFtNDmQqcC
+jML5UaQxIjHeTPGTPSubqi90k0wgF8Jhdbu0OFthX507NJ+K4xZf7qLVyShTz0P
oabZYWDst+aRfzwjJqBZwPPZ8WlhuIVsyaILFTx+e66PCaTie6QiY6V92XKSN7sH
Q51sml3P/gKz6bFdFmBKiXK0SxtsOiPVAgMBAAGjUzBRMB0GA1UdDgQWBBRKAppi
IF/m/lx10MBGiT4m6+JLgjAfBgNVHSMEGDAWgBRKAppiIF/m/lx10MBGiT4m6+JL
gjAPBgNVHRMBAf8EBTADAQH/MA0GCSqGSIb3DQEBCwUAA4ICAQCzXO/io8qiAmk8
uQnqWRk7+sCPhnFdEsloejIZb35pNmBvKzXfli9U0qHNCMtA8JdrRe1Ux12mJGf2
nLnAPkQ/2YjmjHn4Jkr/kEINVtAPA6L9YseNoy0NcWc/VWH4zoJqeIKoXGYZum0n
7VQovJaRiBWONOhE+s5b8KFNCi/8Yqx1m2zP1hoIANSoNbkBL9PJNtJOpRiop7gB
TQpynVUvQVJxCsmWBTBNWhUm8O/4njtlkJEHTP9sapeB7GgaAvxg/K55FFscPIHU
Bl5oOj/KcnY6ZKuBKhVplC4hbDDfkGozfZh+zdqqwmKKpJZjyW8Kcu8ePjtvDYNV
1No+9g18+VROuW4coBC2lMTLQpe6FIr+3/HBSJQzGqPCRidjHu7WiumxtNuEgiXD
SDXfCyj2GlxrU/pGpV3edh/+AQcZARiOokuA4XWnRP+hrLXip+o/XLHIar2FrO5K
cwQHCo1IyuAb1yE3x/3i3jQJ0GX8MAhL2GqGLNg+hFf1ee8dNE1GNnQXFSOkQ+dV
kP9xrK6ldnyukYDouMdizIW6/MR8aghYZ7pAuYI9dqNuMb8VsLTX6EVXXpbmhIHb
gTJGDH6Z5Fz878lGwGOugZkbVS8tDeGI33eZFyBAGspN43T73t52GbCq2i8vJbqa
Baw6htA1Im6s6XL8CV8JVEpJvD9vvw==
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIF4TCCA8mgAwIBAgIUMU3xtgey6zAfPJ0QuI9gxmQ5MyMwDQYJKoZIhvcNAQEL
BQAwgbAxCzAJBgNVBAYTAkNBMQswCQYDVQQIDAJPTjEQMA4GA1UEBwwHVG9yb250
```

```
bzEeMBwGA1UECgwVVW5pdmVyc2l0eSBvZiBUb3JvbnRvMQ8wDQYDVQQLDAZFQ0U1
NjgxJTAjBgNVBAMMHEVDRTU2OCBDZXJ0aWZpY2F0ZSBBdXRob3JpdHkxKjAoBgkq
hkiG9w0BCQEWG2NvdXJ0bmV5LmdpYnNvbkB1dG9yb250by5jYTAeFw0yNTAxMjYy
MjU0MDNaFw0yNTA0MjYyMjU0MDNaMFMxCzAJBgNVBAYTAkNBMQswCQYDVQQIDAJP
TjEQMA4GA1UEBwwHVG9yb250bzEPMA0GA1UECgwGRUNFNTY4MRQwEgYDVQQDDAtF
Q0U1NjggTGFiMjCCAiIwDQYJKoZIhvcNAQEBBQADggIPADCCAgoCggIBAJPcRVPl
544NAO0c08h8J3zDdCXYUeDdC5SL5sjNIep2FXwt9kxg0YL40ItvPjBYNHFch9vJ
VIi4FiL+KjjB0Ih4BD/vsXVmO3Ep0VfcEjPAO51Bk+BxTdK7p3YhobFcZVFmxexe
wXTZ7+4LC5y456XDXZEhExIelU9Ew4/xNDWH4N2PcCbHah9TKlCNkJJ+uLFUXIhD
0OhoBUjhZA+E8boczRWlzoNi99ISZZ40fAwdzKWpwFaKELlTroXSS5AA0ucL42Fw
uD0MMTK+cYUTyC7i87mLvnZNwjm6c6wmSRo+GdxvVJ412K2nOSekVeixEm90ykPs
BqqDxxWGEJ/hrrdOcjusALUe1TnVHo8qrOXCZZf8wu7cYcv6Td3GIRlm3lSXSggD
39dvSmGsdhUMsBCI1pMeq3ckddFLNqR3T/FP1M90SXL1oLU5p0NQn1gqwlYwfCaZ
m4lid7Cixu5Nbm0SkM7ngmKSo6eovnbQX7naCfwXNYlgd3mONekJmjtLjb9LI/nF
tZs0ts2CEQh9DeV5vn2Cfk/UrGO/qynj12ExZoxXPbWUC/nVBfYohkJ2nADHeJJx
tL5kjwxBqh1EqTfn4f2RRKa2cwUw41VZ5IZIpE7GjnVLnG+5T1hVpa1LPxQ9xGZV
FUoH1fo7uQWyX7rnB4HhOvepgG+NywUjbUPlAgMBAAGjTzBNMAsGA1UdDwQEAwIH
gDAdBgNVHQ4EFgQUwE8Wd3bGpJBE1hDqyijGmm74va4wHwYDVR0jBBgwFoAUSgKa
YiBf5v5cddDARok+JuviS4IwDQYJKoZIhvcNAQELBQADggIBAFBgHLdmQBRfrObT
BND6bDOkJuMCFK708ylXVOuBA3CkaSlNGEMVBbrhPlPZv3d2mysnQImF6x9Z1EuC
m2T3iyR4XIUryHcAA3AX00u/2L4B0l1fz417wWDHdzhs/zZiY7ggH0y21zW6wIVV
O2x5Kp7sQ95/YBuvaVVfs6ZckR2EDJSlshL+yO+ulEzjx473x2zue5Y2dKiWBxOa
Sz9CCGKsRivpKWBW/a19g4by+H+3KW0h/qcAUvlDypKcLSilqOA5pIcDIn0BXSzx
cnZOobeCGrhwYcjNKDEIKFRBMh+5Aip75YdhjVrztW5Ugw1bDChN2SyCY0La/+iV
liv83DybjIe5P0HNmPq3+wseQp3FUw67pzm7rxSGaQFrdaTnOIUp+Ep1R8WQ8Hj8
kKtM7hYkezmu+KmExUoyd2Wv27IEajiHmJNA1qjHnyB+QbWEuAsmAIfwmPKfX7Uj
21Nvy2eM1bVeeP+Sj7v/SKNKxtW6hxnusQk8QH644N2vm5CIo9r7FDYJaBNGYw3q
UP5txTM5aePlHjCgLfnx6xvEStunwozHHZp417U/ez02xFXgyGUB/suNbC6BC0Qw
00sh7mG24FAnCjDxQ3ZWzUbclvNeIa/gbXLc7RxHuTTnBOeYpTMi4L5LftgwS920
CdthB16vJk+kUAjwUl9O12E+Ar+l
-----END CERTIFICATE-----
```

If you want to check that your X509 certificate is properly formatted, you can copy the text (just the last certificate, not both) into a file and then use the OpenSSL command-line to parse the certificate; it should look something like this:

```
$ openssl x509 -in yourFile -text -nocert
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            31:4d:f1:b6:07:b2:eb:30:1f:3c:9d:10:b8:8f:60:c6:64:39:33:23
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=CA, ST=ON, L=Toronto, O=University of Toronto,
OU=ECE568, CN=ECE568 Certificate Authority,
emailAddress=courtney.gibson@utoronto.ca
        Validity
            Not Before: Jan 26 22:54:03 2025 GMT
            Not After : Apr 26 22:54:03 2025 GMT
        Subject: C=CA, ST=ON, L=Toronto, O=ECE568, CN=ECE568 Lab2
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (4096 bit)
                Modulus:
                    00:93:dc:45:53:e5:e7:8e:0d:00:ed:1c:d3:c8:7c:
                    27:7c:c3:74:25:d8:51:e0:dd:0b:94:8b:e6:c8:cd:
                    21:ea:76:15:7c:2d:f6:4c:60:d1:82:f8:d0:8b:6f:

[…]
```

Submitting

This lab is due at 11:59pm on Friday, February 14, 2025. However… recognizing that this is a busy time of year, late submissions will be accepted without any marks deducted until the end of Reading Week (Sunday, February 23rd).

As with Lab 1, please create an **explanations_lab2.txt** file that starts with the names and student numbers of your group members, prefixed with an "#":

```
#name1, studentNumber1, email1
#name2, studentNumber2, email2
```

For example:

```
#Jane Skule, 998877665, jane.skule@utoronto.ca
#John Skule, 998877556, john.skule@utoronto.ca
```

It is very important that this information (and the filename) is correct: your mark will be uploaded to Quercus by student number and the email(s) you give here will be how we get the results of your lab marking back to you.

Submit your lab assignment via the ECF *submit* command:

```
submitece568s 2 explanations_lab2.txt merkleTree.py generateX509.py
```

You may submit as may times as you would like; your final submission is the one that will be marked. The submission command for the lab will cease to work after the lab is due.