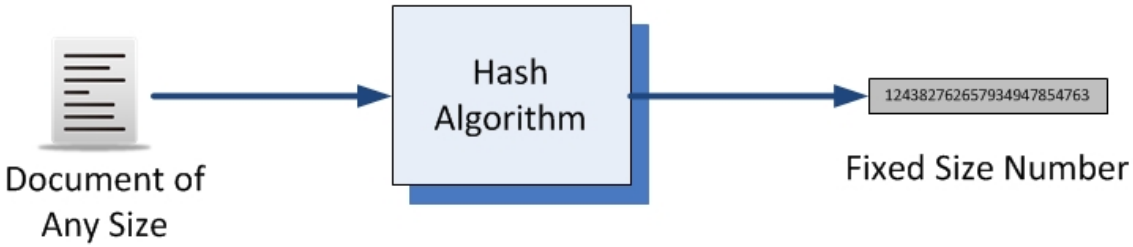
CS 337: Algorithms: Design & Practice

Lab#10: Cryptographic Hash Functions

In this lab you will familiarize yourselves with Cryptographic Hash Functions. Specifically, the MD-family and SHA-family of hash functions. We will primarily focus on MD5 and SHA-256 though, as you will see, other newer as well as older versions are also available in case you would like to explore.

Given the prolific use of cryptographic hash functions you must be wondering how come you have not come across them any sooner. Actually, you have been using these in the background almost on a daily basis. They generally operate in the background ensuring the integrity of your data/mail transfers, software downloads, online transactions, etc. On the other hand, as a computer scientist, you may not have explicitly used them. We will fix that in today's lab. In fact, you will see that cryptographic hash functions are readily available for to you put them to use!

A cryptographic function operates as shown below:



Meet MD5 & SHA-256

Given how often these are used, most Linux distributions provide easy access to these. First, let us create some data/text/message files to hash:

- Create a small text file called, **`TwoCities.txt`**, with the following line in it:

  ```
  It was the best of times, it was the worst of times.
  ```

- Create a slightly larger text file, called **`anassa.txt`**, with the following text in it:

  ```
  Anassa Kata
  (Bryn Mawr College Cheer)

  Anassa kata, kalo kale,
  Ia ia ia Nike,
  Bryn Mawr, Bryn Mawr, Bryn Mawr!

  Translation
  Queen, descend,
  I invoke you, fair one.
  Hail, hail, hail, victory,
  ```

Bryn Mawr, Bryn Mawr, Bryn Mawr!

- Copy a large file the text of "The Decline and Fall of the Roman Empire", from: `/home/gtowell/Public/CS337/Lab10/GibonOne.txt`

- Copy an image file, for example S**pongeBob.png** (the image shown here) from `/home/gtowell/Public/CS337/Lab10/SpongeBob.png`

- Lastly, we will use an executable software file, the hashing function md5 which is available on most Linux installations as: **`/bin/md5sum`**
  On my mac it is at **`/usr/local/bin/md5sum.`** (It may not be on your mac; I have no idea if it comes with Windows.)

(anassa.txt and TwoCities.txt are both also in `/home/gtowell/Public/CS337/Lab10/`) Now that we have four different "messages" we can try to hash them. There are two ways: (1) Through the command line (commands: **`md5sum`**, and **`sha256sum`**), and (2) Using a program. Let's do the command line first:

Run the command shown below:

```
% ls
anassa.txt  GibonOne.txt  SpongeBob.png     TwoCities.txt
```

As you can see, there are four files, as described above. Let's peek in the small file:

```
% cat TwoCities.txt
It was the best of times, it was the worst of times.
```

Next, let us hash the file, using MD5 (**`md5sum`**):

```
% md5sum TwoCities.txt
956a76445c14f466cddf5543537c5fa9  TwoCities.txt
```

`md5sum` returns the resulting hash value of the supplied file as a sequence of hexadecimal digits. There are exactly 32 hex digits that make up the hash value.

Let's try the same with SHA256 (**`sha256sum`**):

```
% sha256sum TwoCities.txt
e1e8d393b1a55348b6efb763b5ec6077520d7f3f56bfabc4d9de45b218285699
TwoCities.txt
```

`sha256sum`, like **`md5sum`**, returns a similar looking sequence of hexadecimal digits, but it is 40 digits long.

§ Question 1.

(a) MD5 digests (hash values) are _____ bits long.

(b) SHA-1 digests (hash values) are _____ bits long.

Using **`md5sum`** and **`sha256sum`** lets us hash all the files we have gathered:

```
% md5sum TwoCities.txt anassa.txt GibonOne.txt
956a76445c14f466cddf5543537c5fa9  TwoCities.txt
77a9d5629b122d4e1042525450010afa  anassa.txt
2739c558dc474cf0b08ee0c6d6b01353  GibonOne.txt

% md5sum SpongeBob.png /bin/md5sum
ca7384620bfdbd17d60454c2ad52615b  SpongeBob.png
2b131c5454e3a3ee85885c0443c8f0f9  /bin/md5sum
```

Notice that no matter the size, or type of file (text, image, executable code), the hash function returns exactly the same length of hash value. Also, your checksum for **`/bin/md5sum`** will likely be different from what is shown above. (On my mac, md5sum is located at /usr/local/bin/ md5sum) Why?

Run **`sha256sum`** on all the five files and observe the results.

One of the unique aspects of these hash functions is that even the tiniest of changes in the input message results in a completely different hash value. In the **`TwoCities.txt`** file, change the first letter ("**`I`**") to lowercase ("**`i`**") and save the file as **`TwoCities2.txt`**. Run both **`md5sum`** and **`sha256sum`** on both files (**`TwoCities.txt`** and **`TwoCities2.txt`**). Compare the resulting hash values.

§Question 2. Processing **`TwoCities.txt`** and **`TwoCities2.txt`**

(a) Was the MD5 digest produced for the two files similar, or different? _____

(b) Was the SHA-256 digest produced for the two files similar, or different? _____

(c) Why would the md5sum for the program md5sum differ across machines?

Explain

MD5 and SHA-256 in Java & Python

Both Java and Python (and most other modern programming languages) provide libraries that have implementations of several message digest functions. In this lab, we will only work in Java. You can look at the Python **hashlib** library in Python reference to see how you would do the same lab in Python. For more details, and several other useful functions available, please feel free to consult the Java and Python References.

Hashing a String - Java

In Java, the class **MessageDigest** provides access to the message digest functions. Every Java implementation is required to support at least MD5, SHA-1, and SHA-256 algorithms. In order to use any algorithm, you have to create a **MessageDigest** instance of it. You can do this using the **getInstance()** method, which is defined as:

```
static MessageDigest getInstance(String algorithm)
// Returns a MessageDigest object that implements the
// specified digest algorithm.
```

The method, **digest()** can then be used on the instance of **MessageDigest** to compute the hash value of an array of bytes (**byte []**):

```
byte[] digest(byte[] input)
// Returns the hash value of the given input byte array.
```

Thus, given a string (say **"Bryn Mawr"**), you first have to convert it to a byte array. This is shown below:

```
String message = "Bryn Mawr";
byte[] byteMessage = message.getBytes("UTF-8");
```

The **String** method, **getBytes()** converts the string object into the encoding specified. Now, we can write a function, let us call it **hashString()**, to convert a message (**String**) into its hash value using a specific algorithm and return it as a string of hexadecimal digits. This is shown below:

```
String hashString (String message, String algorithm) {
   // Compute the hash value of message using algorithm and
   // return a string representation of it.
   byte[] hashedBytes = null;    // will store the hash value of message
   try {
      // instantiate the specified algorithm.
      // It may not exist, thus the try-catch
      MessageDigest digest = MessageDigest.getInstance(algorithm);

      // Compute the hash value of message
      hashedBytes = digest.digest(message.getBytes("UTF-8"));
   catch (NoSuchAlgorithmExecption |
          UnsupportedEncodingException e) {
      e.printStackTrace();
   }
   // Convert hash value (in byte[]) to a hex String and return result
```

```
   return bToH(hashedBytes);
} // hashString()
```

Everything, except the return statement above should now make sense, given the descriptions above. Since the **`digest()`** method returns an array of bytes, so that we can print it out as a hexadecimal string, we need to convert it to a **`String`** of hex digits. The function **`bToH()`**, that you have to write, accomplishes that as follows:

```
String bToH(byte[] value) {
   // Converts value to a string of hex digits
   StringBuilder sb = new StringBuilder(value.length*2);
   // Why, length*2 ? See Question 3.
   for (byte b : value)
      sb.append(String.format("%02x", b));
   return sb.toString();
} // bToH()
```

§ **Question 3**. Why, in the function above, is the length of the **`String`**, **`sb`** defined to be twice the length of the array, **`value`**?.

Hashing a String – Python

Python's **`hashlib`** module, like other programming languages, provides implementations of several standard hash functions though a common interface.

To use a specific hash function, you have to first instantiate it, and then use the **`update()`** and **`digest()`** methods (see below) to compute hash values. Here is an example:

```
>>> msg = b"Bryn Mawr"
```

The **"b"** prefix encodes the string using ASCII (or byte) encoding. You can then compute the hash value of **`msg`** as shown below:

```
>>> digest = hashlib.new("md5")# creates an instance of MD5
>>> digest.update(msg)    #adds msg to hash value computation
>>> digest.digest()   # completes computation, returns result
b'\x03\xb8qF\xec\x1c]\xd0<x\x986\xdaM\xb7\xc2'
```

To get a nicely converted hexadecimal result, you can use the **`hexdigest()`** function:

```
>>> digest.hexdigest()
'03b87146ec1c5dd03c789836da4db7c2'
```

To see what algorithms are available for instantiation, you can examine the value of **`algorithms_available`** variable in **`hashlib`**:

```
>>> hashlib.algorithms_available
{'SHA1', 'SHA224', 'SHA', 'SHA384', 'ecdsa-with-SHA1', 'SHA256',
'SHA512', 'md4', 'md5', 'sha1', 'dsaWithSHA', 'DSA-SHA', 'sha224',
'dsaEncryption', 'DSA', 'ripemd160', 'sha', 'MD5', 'MD4', 'sha384',
'sha256', 'sha512', 'RIPEMD160', 'whirlpool'}
```

Programming Task#1 - Java

Write a complete Java program that uses the above functions to print out the hash value of simple strings. For example, use the following **`main()`** function:

```
public static void main(String[] args) {
   String msg = "It was the best of times, it was the worst of times.";
   String hashValue = hashString(msg, "MD5");

   System.out.println(hashValue + "   " + msg);
} // main()
```

Observe the output hash value for **`msg`**. Compare it with the output of **`md5sum`** on **`TwoCities.txt`**.

Programming Task#1 - Python

Write a complete Python program that uses the above functions to print out the hash value of simple strings. For example, use the following **`main()`** fuction:

```
def main():
   msg = "It was the best of times, it was the worst of times.";
   hashValue = hashString(msg, "MD5");
   print(hashValue, "   ", msg);
```

Next, try the same example in the program above to use the SHA256 algorithm (use the string "SHA-256" in Java and "SHA256" in Python to instantiate the algorithm).

§ **Question 4**. Is the MD5 digest the same in all three (Unix, Java and Python) cases? If not, why?

§ **Question 5**. Is the SHA-256 digest the same in all three cases? If not, why?
Explain (in 1 or 2 sentences). You may provide one explanation for Q4 and Q5**.**

As you saw above, it is very important, when studying hash function implementations for comparison purposes that you are providing them exactly the same string/message to hash. Even

a 1-bit change would lead to a completely different outcome. OK, take a well-deserved breather before proceeding.

Hashing Files

So, that was a simplistic way to hash something, namely a short string. Typically, as you have seen, you hash files. The files may contain data, text, an image, audio, video, etc. And, the files can be of arbitrary size, even larger than the memory available. To address these, you do two simple things: (1) No matter what the contents of a file may be, you can just treat them as a very long sequence of bytes. After all, if it is anything stored on a computer, it must be in bits/bytes. (2) You write your program to process/hash the file of bytes in manageable chunks. This way you never have to read an entire file before hashing it. Also, remember that the hashing algorithm itself breaks down the message into blocks. Luckily, we do not have to match the chunks of a file to the block size used by the algorithm. All message digest algorithms provide internal mechanisms to manage this. Hashing a file, therefore, follows the following algorithm:

```
while there are bytes in input file to process
   read a chunk of the input file (typically 1024 bytes)
   supply the chunk to the digest algorithm

hash code = finalize the computation of digest and return the hash code
```

Java

In Java's **`MessageDigest`** class, there are two sets of functions provided:

```
void update(byte[] input)
void update(byte[] input, int offset, int  length)
// Updates the digest using the byte array, input.
// Starting from byte[offset], length bytes, if provided

byte[] digest()
byte[] digest(byte[] input)
// Update digest with supplied input,
// and finalize the hash value computation (as a byte[])
```

Earlier, in Task#1, we used the **`digest()`** method since all we were digesting was a small string. But, for large files, and files of any content, we use **`update()`** to add content to the digest. Once all content has been updated, we use **`digest()`** to finalize the computation of hash value. Incorporating these ideas, we provide a Java function, **`hashAFile()`** that, given a file and an algorithm, opens the file, instantiates the algorithm, and computes and returns the hash value for the file:

```
String hashAFile(File filename, String algorithm) {
   // Hash the contents of the file,
   // fileName and return its hash value as a hex string.
```

```
   byte[] hashedBytes = null;     // the result
   try {
      // Open the file
      FileInputStream inStream = new FileInputStream(filename);

      // Instantiate a digest with the algorithm
      MessageDigest digest = MessageDigest.getInstance(algorithm);

      // Define input file chunk buffer (of 1024 bytes)
      byte[] buffer = new byte[1024];
      int bytesRead = -1; // counts how many bytes were read

      while ((bytesRead = inStream.read(buffer)) != -1) {
         // there are bytes in input file to process
         // supply the chunk to digest
         digest.update(buffer, 0, bytesRead);
      }

      // finalize computation
      hashedBytes = digest.digest();

   } catch (NoSuchAlgorithmException | IOException e) {
      // Catches both: algorithm and file I/O exceptions
      e.printStackTrace();
   }

   return bToH(hashedBytes);           // Convert bytes to hex string
} // hashAFile()
```

Note that you still need to use the function **`bToH()`** to convert a bytes array into a printable hexadecimal string.

Python

```
def hashAFile(fileName, algorithm):
   """Hash the contents of the file, filename using algorithm,
      and return its hash value as a hex string.
   """
    digest = hashlib.new(algorithm)
    with open(fileName, "rb") as inStream:
        # read entire file into buffer
        # WARNING: watch for HUGE files!
        buffer = inStream.readline()
        digest.update(buffer)
    return digest.hexdigest()
```

Python - Buffered Read

For HUGE files, you can read a specific number of bytes from a file, and process sequentially:

```
bufferSize = 1024
with open(fileName, "rb") as inStream:
    buffer = inS.read(bufferSize)     # read bufferSize bytes
    while len(buffer):
        digest.update(buffer)    # add to digest
        buffer = inStream.read(bufferSize)
return digest.hexdigest()
```

Programming Task#2

Write a Python/Java program to compute the hash values for any file. Use the code segments provided above. Then test your program for the following files:

(1)TwoCities.txt
(2)Anassa.txt
(3)GibonOne.txt
(4)SpongeBob.png

Run your program on all the files, for both MD5 and SHA-1 algorithms. Compare the results obtained to the ones obtained by **`md5sum`** and **`sha256sum`**.

§ **Question 6.** Do the digests match for all the files, and for each algorithm? Explain, briefly, why it is important that they match.

Why SHA-256?
SHAttered! The first collision attack against SHA-1

On February 23, 2017 Google Inc. and CWI Institute announced that they had generated a collision. That is, two different messages when hashed with the SHA-1 algorithm, produced the exact same hash value! In this part of the lab, you will recreate this collision. You will use the messages provided by Google (see below) and use your program to confirm that the two inputs do produce a collision. All you have to do is, copy the two files from the address provided and run your program from Task#2 on them.

File#1: **`/home/gtowell/Public/337/Lab10/`**`shattered-1.pdf`
File#2: **`/home/gtowell/Public/337/`**`Lab10/shattered-2.pdf`

Run these files for MD5, SHA-1 and SHA-256 algorithms (in Java use “SHA256”, in Python, “sha256”.

§ **Question 7.** Did your program produce the same hash value for the two files on all three hashing algorithms? Why does this result point to a problem for sha1?

For more information on this, see https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html

Since the Shattered announcement, there have been further studies and news about the significance and consequences of using/retiring the use of Sha1 algorithm. Feel free to do do some research on this matter.

What to hand in:
Answers to the 7 questions posed above. Answers may be independent; That is you do not need to fold your answers into a cohesive essay. Explanations should be on the order of 1 paragraph

An appendix containing the code you wrote; both Java and Python for Task 1 and either Java or Python for Task 2.

Rough Rubric:

Topic	Points
Q1	10
Q2	10
Q3	10
Q4	10
Q5	10
Q6	10
Q7	10
Appendix (Code)	30

Use two species of cats as your identity.