# One-Way Hash Function and MAC

## 2.1 Generating Message Digest and MAC

**Question 1. Describe your observations. What differences do you see between the algorithms?**

**A.** The difference between the different digests generated by the algorithms seem to be the length of them. (128, 160 and 256 bits)

**Question 2. Write down the digests generated using the three algorithms.**

MD5 = 5515966ebe9dcbe05d1c72056da0e7b5 = 128 bits

SHA1 = f42b4b883a835218465f8d7394ba2d6569b92cc3 = 160 bits

SHA256 = 7f133f15704a33ba4dd93f02d5cfd73c4b98fc097d410a82d3ae3849e8bde099 = 256 bits

## 2.2 Keyed Hash and HMAC

**Question 3. Do we have to use a key with a fixed size in HMAC? If so, what is the key size? If not, why?**

HMAC-MD5 = ce65d4b6f81f552058af749887a8bbf8

HMAC-SHA1 = 81aab51dcee0c248236bfdf6e3b5d3c5e05c450b

HMAC-SHA256 = 81aab51dcee0c248236bfdf6e3b5d3c5e05c450b

**A**. No, the size of the key does not seem to affect the result. Every string that I tried, generated a new hash value. The key can be of an arbitrary length but if it's shorter than the message it will use padding to make it work.

**Question 4. Now use the string IV1013-key as the secret key and write down the keyed hashes generated using the three algorithms.**

HMAC-MD5 = 9baa0608484c9d126043ecb7f2659af2

HMAC-SHA1 = b5966e3c24b128780becdd5e9bf59969cafcac34

HMAC-SHA256 = 5a491a8a7588eb178b29b75dcf31052cb85a62677a168a477112fe6933f9d905

## 2.3 The Randomness of One-way Hash

**Question 5. Describe your observations. Count how many bits are the same between H1 and H2 for MD5 and SHA256**

Using MD5: 65 bits similar
Using SHA-256: 105 similar bits

## 2.4 The Randomness of One-way Hash

**Question 6. Investigate how many trials it will take to break the weak collision property using the brute- force method.**

When running brute-force with a multithreaded program of 256 threads on a 4 core machine this was the number of trials it took to break the weak collision property for each of the five message.

- IV1013 security - 28 156 trials
- Security is fun - 325 895 trials
- Yes, indeed - 44 768 trials
- Secure IV1013 - 245 618 trials
- No way - 25 220 trials

When only utilizing one thread, the results was as follows:

- IV1013 security - 14 628 219 trials
- Security is fun - 10 858 782 trials
- Yes, indeed - 732 689 trials
- Secure IV1013 - 46 243 972 trials
- No way - 19 176 047 trials

## Discussion

This lab has focused on message digests and hashing. Firstly, I experimented with openssl using various one-way hash algorithms. I had some problems getting this to work on Mac but worked fine on Ubuntu. Moreover, had some problems creating a file with UTF-8 encoding so I ended up using an online converter to get it right. The only observations of the first task was that the different algorithms results in different lengths of the digest produced.

For task 2.3 I had some difficulties flipping the first bit as described in the lab description but ended up using a simple online tool for flipping bits.

Regarding the program I had some difficulties to handle the digests correctly, but finally ended up with representing them with a byte[] array for easy access.

The following aspect is probably outside the scope of this course but I decided to have some fun and make the program for checking collision resistance multithreaded. The only problem I encountered was related to the structure of the program. I had from the beginning developed the program with static methods and lots of global variables. So I had to do some refactoring in order to make it work. I also tried to figure out how to terminate the threads when one thread find a match. Tried some techniques but non worked so the program just continues until each thread has terminated. The results was quite interesting. With more threads less trials for each thread are required to find a match. However, the total number of trials may not improve. But if run at a high performance computer this implementation will definitely result in that a matching digest is found quicker compared to doing it sequentially.

# Source code:

---

```
/* This program reads in a message from the user and creates a digest of it.
   The program then executes a brute-force algorithm in order to try to find a
   digest that equals the 24 first bits of the digest of the input message.

   Features: Prints digest of input message and the found matching digest along with number of
trials
            it took to find a match. The program has support for multithreadiing.

    Usage under UNIX:
            javac CollisionResistance.java
            java CollisionResistance

    @author Emil Stahl
*/

import java.util.Random;
import java.util.Scanner;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.io.UnsupportedEncodingException;
```

```
public class CollisionResistance {

   public byte[] digest;
   Random rand = new Random();
   public String encoding = "UTF-8";
   public String algorithm = "SHA-256";

   public void bruteForce(byte[] input, int id) {

      long c = 0;
      byte[] tryDigest;

      while(true) {
         //increment counter
         c++;
         int seed = rand.nextInt(2147483647);
         //make a trydigest to compare
         tryDigest = getDigest(Long.valueOf(seed).toString());

         //check if first 24 bits of digest and trydigest is equal
         if(input[0] == tryDigest[0] && input[1] == tryDigest[1] && input[2] == tryDigest[2]) {
            // if so, print trydigest and return
            System.out.println("\nThread nr: " + id + "\nIt took " +c+" times to generate a identical
digest\n\nThe digest was: ");
            printDigest(tryDigest);
            return;
         }
      }
   }

   public byte[] getDigest(String inputText){

      try {
      // create object of message digest with SHA-256
      MessageDigest msgDig = MessageDigest.getInstance(algorithm);
      // call object.update with inputBytes
      msgDig.update(inputText.getBytes(encoding));
      // create digest by object.digest();
      digest = msgDig.digest();
      } catch(NoSuchAlgorithmException e) {System.out.println("The specified algorithm " +
algorithm + " does not exists");}
      catch(UnsupportedEncodingException ex) {System.out.println("Encoding " + encoding + " not
supported");}

      return digest;
}

  public void printDigest(byte[] digest) {
     for(int i = 0; i < digest.length; i++)
        System.out.format("%02x", digest[i]&0xff);
     System.out.println();
  }
```

```
    public static void main(String[] args) {

        // Read in message
        System.out.println("Type message to digest:");
        Scanner sc = new Scanner(System.in);
        String msgToDigest = sc.nextLine();
        System.out.println("Type number of threads to utilizie");
        int threads = sc.nextInt();
        sc.close();
        System.out.println("\nMessage: " + msgToDigest + "\nThreads: " + threads + "\n");

        CollisionResistance cResistance = new CollisionResistance();

        byte[] digest = cResistance.getDigest(msgToDigest);
        System.out.println("The digest of the message " + msgToDigest + " is:");
        cResistance.printDigest(digest);

        System.out.println("\n\nRunning brute force . . . . . .\n");

        for(int i = 0; i < threads; i++) {
            final worker worker = new worker(i, digest, cResistance);
            worker.start();
        }
    }
}

 class worker extends Thread {

    int id;
    byte[] digest;
    CollisionResistance cResistance;

    public worker(int id, byte[] digest, CollisionResistance cResistance) {
        this.id = id;
        this.digest = digest;
        this.cResistance = cResistance;
    }

    public void run( ) {

        cResistance.bruteForce(digest, id);
    }
}
```

---