Linnaeus University

1DV700 - Computer Security
Assignment 1

Student: Jesper Bergström
Personal number: 961010-0019
Student ID: jb223qe@student.lnu.se

Setup Premises

OS: Windows 10
Web browser: Google Chrome, Mozilla Firefox
IDE: Eclipse

Task 1

a)
Symmetric encryption - Asymmetric encryption
Symmetric encryption uses one secret key for both encryption and decryption. This means both sender and receiver of the information need access to the key. Asymmetric encryption uses two keys: one public key, that is available for anyone and one secret key, related to the public key. The secret key can be used to decrypt a message encrypted by the public key.

Encryption algorithms - Hash algorithms
Encryption can both encrypt and decrypt information, as long as the required keys are available. Information that has been hashed can not be "dehashed" back to the original information. A hashing algorithm calculates the input text and outputs a shorter, hashed representation of the text.

Compression - Hashing
In compression, the intention is to keep as much of the original data as possible. A *lossless* compression algorithm maintains all of the original data, but reduces the number of bits by detecting statistical redundancy and removing it[1]. In hashing, we are not as concerned with maintaining data, since we cannot "dehash" a hash value.

b) Steganography and digital watermarking can look rather similar in certain cases but they have very differing objectives[2]. In case of images, steganography aims to hide information within an image and digital watermarking aims to add similar information but to make it visible. Essentially, digital watermarking aims to distort information in a way where you can still make out what the information is suppose to be, but it is obvious that it has been watermarked. This is useful for copyrighted content such as images, videos or sound files. The owners of the content might want everyone to get an idea of what the content is, but not be able to use it without permission. For a good digital watermark, you want to make it as hard as possible for someone to remove the watermark bits, revealing the original content.

Steganography and encryption both have rather similar objectives, in that they aim to hide information. The difference is that steganography hides information in plain sight, so if you know that there is hidden information, it is not hard to get access to it. In encryption, it is usually not possible to extract the information unless you have a key. This means that it is probably safer to hide critical information using encryption rather than steganography. Encryption is used anytime you want to make some information inaccessible to other people. This could be applied to communication, such as chat messages or e-mail or classified documents.

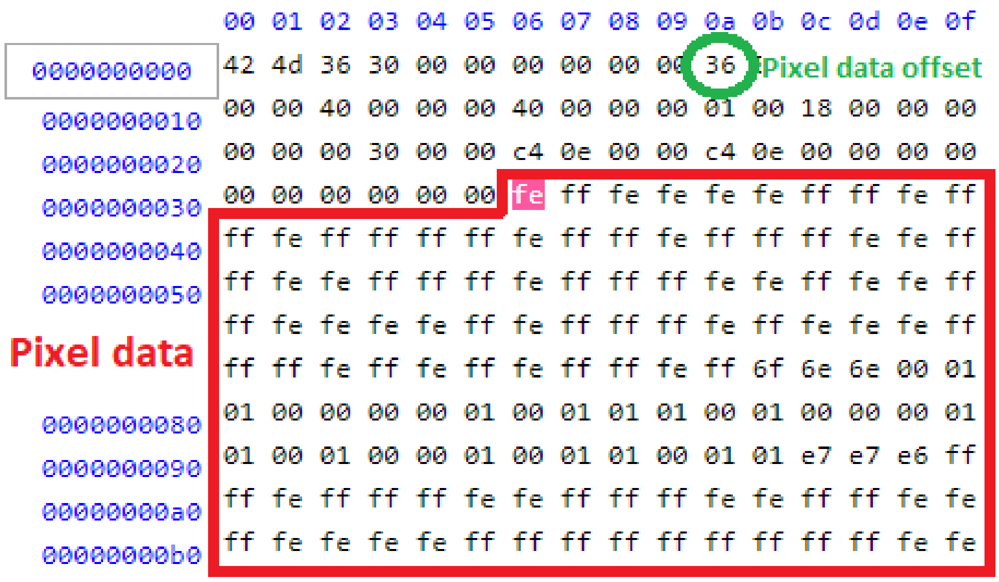
Task 2

a)
The tool on the web page hides one image inside another image using the least significant bit of each pixel. They also let the user choose how many bits to use in each pixel. The more bits you use, the more those pixels will change from the original image and you will be able to see what is hidden. The problem with this is that if you only use 1 bit, the hidden image will not be very detailed when extracted. If more bits are used, the hidden image will be visible when it is not suppose to, but it will be better quality and more detailed when extracted.

b)
Steganography can be used to conceal almost any type of information inside some other information. This can be applied to both physical and digital techniques. Other digital steganography techniques are concealing encrypted messages inside other encrypted messages, embedding pictures in video material, changing the order in a set of elements, etc.

In the case of least significant bit, any type of message can be concealed in any type of file as long as the changes to the original file do not become easily noticeable.

c)
By opening the .bmp file in a hex editor, we can view the value of all the bytes in the file. By collecting the least significant bits of the first few bytes of the pixel data, we get a string of ones and zeros. If we then convert this string from binary to ascii, we get out a message saying "congratulations!".



```
            00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
0000000000  42 4d 36 30 00 00 00 00 00 00 36 00 00 00 28 00
0000000010  00 00 40 00 00 00 40 00 00 00 01 00 18 00 00 00
0000000020  00 00 00 30 00 00 c4 0e 00 00 c4 0e 00 00 00 00
0000000030  00 00 00 00 00 00 fe ff fe fe fe fe ff ff fe ff
0000000040  ff fe ff ff ff ff fe ff ff fe ff ff ff fe fe ff
0000000050  ff fe fe ff ff ff fe ff ff ff fe fe ff fe fe ff
            ff fe fe fe fe ff fe ff ff ff fe ff fe fe fe ff
            ff ff fe ff fe ff fe ff ff fe ff 6f 6e 6e 00 01
0000000080  01 00 00 00 00 01 00 01 01 01 00 01 00 00 00 01
0000000090  01 00 01 00 00 01 00 01 01 00 01 01 e7 e7 e6 ff
00000000a0  ff fe ff ff ff fe fe ff ff ff fe fe ff ff fe fe
00000000b0  ff fe fe fe fe ff ff ff ff ff ff ff ff ff fe fe
```

01000011011011110110111001100111011100100110000101110100011101010110110001100
00101110100011010010110111101101110011100110010000 1 = congratulations!

Task 3

a) HKPUFCMHY BHDDXZH = encrypted message

b) Yes, this message can be decrypted by someone without access to the key. One way of doing this is by using crypto analysis to try to figure out what letters are mapped to what other letters. Since this message is so short, it might not be completely reliable though. Another method could be by brute force. You could write a program that tests all possible cipher lines until one is found that produces valid words.

Task 4

Substitution

The substitution algorithm I implemented only encrypts/decrypts characters with values from 32 to 127. These characters include upper- and lowercase letters, numbers and some common symbols. This way, we make sure that all characters in our cipher line are printable and can be read in a text file. All other characters will keep their original value.

The cipher line is generated by offsetting the ascii value by the key value (0-255), wrapping around back to 32 if the value exceeds 127.

Transposition

For the transposition algorithm, I implemented columnar transposition. The key for this encryption should be a string with only alphabetic characters. A table will be created with the same number of columns as there are characters in the key. The message will then be entered in to the table row for row. To encrypt the message, we read each column in the alphabetical order of the key.

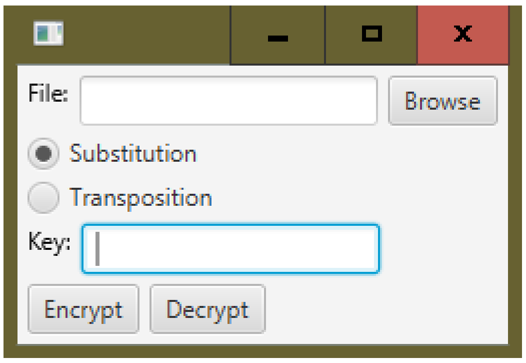
Example: key = “zebra”, message = “Hello World! This is an encryption algorithm!”

z(5)	**e(3)**	**b(2)**	**r(4)**	**a(1)**
H	e	l	l	o
	W	o	r	l
d	!		T	h
i	s		i	s
	a	n		e
n	c	r	y	p
t	i	o	n	
a	l	g	o	r
i	t	h	m	!

The encrypted string from this table will then be: “olhsep r!lo nrogheW!saciltlrTi ynomH di ntai”.

Program usage

The program has a simple GUI so it should be easy to use. First select the file you want to encrypt or decrypt. Then choose encryption method, enter the key in the text field and click either the "encrypt" or "decrypt" button. The output will be a new file in the same directory as the original file, with the string "-encrypted" or "-decrypted" appended to the end of the file name.



Note: The key for substitution must be a number between 0-255. The key for transposition can be any string but all characters need to be alphabetic (letters).

Task 6

I picked the cipher text provided by John Herrlin. This text was encrypted with a substitution algorithm so my first objective was to find out which characters he had chosen to include in the cipher.

I wrote a small java program that extracted the lowest and highest byte values from the text file. The lowest byte value was 35 and the highest was 126. This does not mean that this necessarily is the correct range but it gives a good idea.

Then I wrote a small java program that counts all bytes and how many times the same bytes appeared in the text. This way I could get an idea of what characters the encrypted characters represented from how common they are in the english language. This proved not very effective though, since it would be easier to just look at the text that I actually know what it says and figure out some of the characters from there.

The next thing I did was to extract a short cipher line from decrypting the parts of the text that I knew what it said. For example:"o3{2#m^660|2" = "John Herrlin". I now know that o = J, 3 = o, { = h, 2 = n, etc. With this information, I tried to find out exactly how the encryption algorithm worked. The first thing I did in order to achieve this was to look at the offset of all the characters I knew about. What I found was that many of the characters had offset -60. For those that did not have -60 offset, I thought the offset had been wrapped around and therefore given another value. This does not seem to be the case though, since the offset of i = 19 and the offset of j = 5 and these characters are only 1 appart in value. Therefore, I don't think this algorithm simply offsets the ascii values by some value derived from the key.

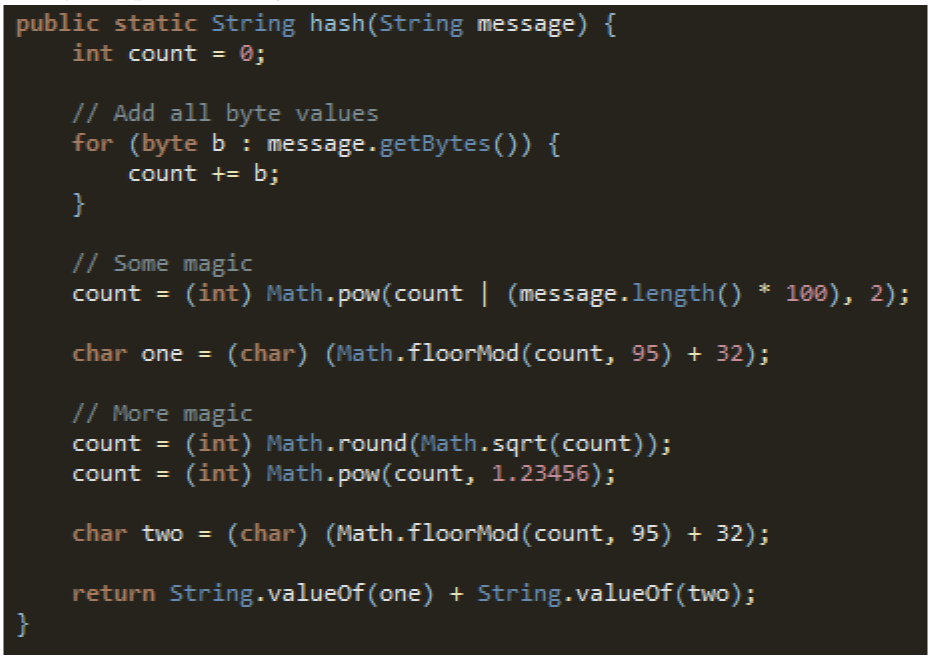
From the analysis of this substitution algorithm, it seemed more time efficient to just decrypt the file using the characters I already knew and try to find more characters by analysing the decrypted text. By doing this I found some more cipher characters, but the cipher line is still not complete. You can however make out what the text says and there are only a few characters that are still encrypted after decrypting the text.

These are the characters I found:
'*', ' ', 'e', 'r', 'j', 'o', 'h', 'n', 'H', 'l', 'i', 'n', 'S', 'c', 't', 'm', 's', 'a', 'g', '-', 't', 'p', 'u', 'd', 'M', 'y', 'b', 'k', 'v', 'w', 'f', 't', 'q', ',', 'I', 'f' , 'j'
=
'M', '#', '^', '6', 'o', '3', '{', '2', 'm', '0', '|', '2', 'x', '[', '8', '1', '7', '?', '`', 'P', 'y', '4', '9' , ']', 'r', 'd', '@', '~', 'a', 'b', '_', 'B', '5', 'O', 'n', 'k' , '}'

The top array are the decrypted characters and the bottom array are the encrypted characters where the index indicates the relation.

Task 7

a) My simple hash algorithm:



```java
public static String hash(String message) {
    int count = 0;

    // Add all byte values
    for (byte b : message.getBytes()) {
        count += b;
    }

    // Some magic
    count = (int) Math.pow(count | (message.length() * 100), 2);

    char one = (char) (Math.floorMod(count, 95) + 32);

    // More magic
    count = (int) Math.round(Math.sqrt(count));
    count = (int) Math.pow(count, 1.23456);

    char two = (char) (Math.floorMod(count, 95) + 32);

    return String.valueOf(one) + String.valueOf(two);
}
```

b) For the analysis of the hash algorithm, I implemented 3 different java methods: uniformityTest(), sameByteValueTest(), incrementalByteValueTest().

Uniformity test

For the uniformity test, I start by generating a random string of length 1-100. I then hash the string and save both the hashed string and the individual bytes for analysis later. This gets repeated 1000 times.

When this has ran 1000 times, I print out some statistics derived from the saved string and byte values. I print the least common and most common byte/string, the median and lastly a list of all values together with how many times they occurred during the 1000 hashes.

The results indicate that the hashed bytes are fairly uniform but not completely evenly distributed. The least common byte occurred 2 times, the most common occurred 64 times and the median was 13. The results with the strings were fairly similar to the bytes.

Same byte value test

I decided to divide the small change analysis into two different test. The same byte value test first generates a random string of length 1000. It then loops through every character in the string and increments the current character by 1.

An example of this could be:
Hello World
Iello World
Hfllo World
Hemlo World
Helmo World

Using this technique, the sum of all bytes in the strings will always be the same, except for the original string. The result of this test was that all strings produced the exact same hash value. This could be an argument for the hash algorithm not being considered secure.

Incremental byte value test

This test starts by generating a random string of length 1-100 and appending a "space" character at the end of the string. The program will then increment the last character of the string by 1, ten times. For each iteration, we hash the current value of the string and print it. This is repeated 1000 times.

What we get from this test is 10 different hashes for 1000 different random strings. I decided to only analyze these results manually by reading the output. The hash values from each of the 10 iterations look mostly random, which is desirable. However, some strings produce "patterns" in the hash values. Sometimes the same hash value will occur several times in the 10 iterations.

Here is an example taken from the output:

```
0/
95
D;
Q@
0/
95
D;
Q@
0/
95
```

As we can see, there seems to be a pattern where 0/, 95, D;, and Q@ is repeated continuously. I am not sure what is the cause of this but it should also be seen as a weakness in the hash algorithm, making it insecure.

c) Using a secure hashing algorithm, it should be very hard to generate the same hashed string from different plain text strings. As we can see in the various tests I did in the previous section, it was not uncommon to get duplicate hash values. One of the biggest reasons for this is that the hash value is only 16 bits. Another reason why hash values are easy to replicate is because it is only derived from the sum of all the bytes in the plain text string. In other words, if the sum of all the bytes in two different input strings is the same, it will produce the same hash value.

Bibliography

[1] Wikipedia contributors. (2020, February 5). Data compression. In *Wikipedia, The Free Encyclopedia*. Retrieved 16:12, February 9, 2020, from https://en.wikipedia.org/w/index.php?title=Data_compression&oldid=939211563

[2] Wikipedia contributors. (2020, February 7). Digital watermarking. In *Wikipedia, The Free Encyclopedia*. Retrieved 16:08, February 9, 2020, from https://en.wikipedia.org/w/index.php?title=Digital_watermarking&oldid=939600183