Linnéuniversitetet
Kalmar Växjö

Linnaeus University

1DV700 - Computer Security
Assignment 1

Student: Anton Jonsson
Personal number: 030501-xxxx
Student ID: aj225ef@student.lnu.se

Setup Premises

For The completion of these tasks I will be using either Kali Linux or Ubuntu 22.04 or Windows 10, depending on if I am working on my laptop or on my stationary computer. Other than that I will consistently use Firefox and usually code in VsCode or vim, as well as all the tools supplied by Kali Linux such as, Hydra, john the ripper(not for the manual decryption tasks), nmap, burp suite, hashcat, etc, for general tools i will be using matplotlib for graphs of the hashing data. As for language I will mainly work in java, except for smaller tasks where a script language like python would be more suitable.

Task 1

a)

Symmetric encryption – Asymmetric encryption:

Symmetric encryption involves the same key to be used by all parties as part of the encryption and decryption parts. As for asymmetric encryption it is the opposite of symmetric whereas 2 keys are used, a public one and a private one [1]. Some might argue that asymmetric is simply the better one to use but it's inefficient when transferring large amounts of data efficiently.

Encryption algorithms – Hash algorithms:

The largest difference between encrypting and hashing is the fact that hashing traditionally can't be undone because that's not its purpose. This makes it so we have some different use cases for hashing and encrypting. Encrypting is great for safe communication since we can determine a common key and then decrypt it to use that plain text data. As for hashing, it is a great tool to compare large data sets accurately since the content of the data will completely change the hash dump result, and so the two are very different but have very great use cases [2].

Compression – Hashing:

Compression has no initial security purpose compared to hashing, compression's purpose is to reduce the size of data objects so that it is more efficient at transferring it on a network, while hashing has more to do with security and data structures. Compression is a lot closer to encoding than hashing in that it's almost never a one way thing, its main purpose is to initially compress the size to make it quicker to handle. While hashing is similarly not always compressed, but remade into a hash dump so it's easier/quicker to handle in different scenarios[3].

b)

When it comes to Steganography, encryption and digital watermarking there are a couple important distinctions. Steganography is the technique of hiding information in plain sight, this can be done in multiple different ways such as embedding a text file into an image by changing the right most bit since it's the least important and doing that throughout the image. Encryption is the technique of manipulating data so that it cant be used unless decrypted which can be done manually if the encryption is weak or not if it needs a key, encryption is mainly used for privacy since it is great at keeping things confidential, authentic, and integrity. Digital watermarking is similar to steganography in that you are hiding information in plain sight, but their use cases are a bit different, digital watermarking isn't used to create undetectable data in images etc, it is supposed to be detectable for copyright reasons. Both of them handle hiding information in images, audio etc, but steganography is mainly for handling covert communication and so it's taken more seriously with how undetectable it is for security reasons[4].

Task 2

a)
Looking at the structure of the words i instantly deciphered that the second word is “message” since the double “D” is quite unique, from there we can apply that to the first word and see that we have: “EXXXXXXEX” and that the word doesnt have any “m”, “s”, “a”, or “g”. After that i looked at the context and guessed the word “encrypted” because of the context of “something Message”.

b)
This is a bit more difficult since we have no frame of reference to decrypt it right away, but there are some logical conclusions we can guess and see where we go from there. Mainly I would apply frequency analysis and word length right away since we have three words that are 3 characters long. These words most commonly could be; “and”, “the”, “for”, “are”, “can”, etc.

For the words:

“XQS”, “JXB”, “WKH”

We have one shared letter between 2 of the and that is X, with this we can determine these two words must share only one letter and the rest needs to be unique. The most frequent appearances are W and B which could be substituted for something like T and E but that is not enough to start making sense of the cipher.

c)
With no technical help made this very difficult, since there's no frame of context to anything and just guessing the two 3 letter words was difficult enough. I still believe I could have had some sort of domino effect if I had found a match for those two words since that would give me more letters to apply to the other words. It is a guessing game until something makes sense and that is very time consuming(unless you guess it on the first try). As for security I wouldn't say that it is very secure since I could simply write a script to generate a key and apply it to the cipher and see if the results are a collection of real words, and if so return it otherwise scramble a new key and repeat. This would be enough to crack the cipher with enough time. With enough keys I believe you could make it secure enough but the only requirement would be that the recipient has the key beforehand, think of a substitution cipher that returns a caesar cipher or something of the like. This would confuse anyone trying to break it since it won't return anything of use if it's brute forced like i mentioned, and having the eye to first substitute correctly would not help too much since the result still looks scrambled because of the caesar cipher. But compared to other encryption methods that exist, this would be pretty weak.

Task 3

This task asked us to write a program that uses two different encryption techniques, one substitution and one transposition cipher. For the sake of time I chose the two quickest ones I could write based on what I've done before, this was a caesar cipher and a atbash cipher. Two very basic ciphers both based on the alphabet, the caesar cipher simply shifts the letters forward(and wraps around after z) a certain amount of times according to the given key(1-25), while the atbash cipher simply inverts the letters so that a becomes z and b becomes x etc. With these simple concepts i started writing the code, the caesar cipher was simple, just take an input and shift each letter while making sure it does not surpass the limit and then wraps around back into a, that looked a little like this:

```
# shift the characters
for ch in word:
      if ch.isalpha():
            new_ch = chr((ord(ch) - ord('a') + shift) % 26 + ord('a'))
            cr_word += new_ch
      else:
            cr_word += ch
```

With this we check if the character we are on is a letter and then proceed to shift it by decreasing by the ascii value for “a” since that will reduce the number to which letter in the alphabet, 1-26, and add the shift to that value, then lastly take modulus of that to get the letter if it is wrapped afterwards and add back the value for “a” that we removed to get it back to its ascii number and append that. To decrypt this we literally do the same except reduce by the given shift, and that is all for the caesar cipher and as for the file writing part it's just opening and writing into a new file that has the same name except with “encrypted_” in the beginning of the name.

In the atbash cipher part it went pretty similarly, and for this one i didn’t need to make a decryption part since the idea of atbash is just to reverse something then the way to decrypt it is just to reverse it again, the function is symmetric. It looks something like this:

```
for char in text:
        if char.isalpha():
            if char.islower():
                encrypted_text += chr(ord('z') - (ord(char) - ord('a')))
            else:
                encrypted_text += chr(ord('Z') - (ord(char) - ord('A')))
        else:
            encrypted_text += char
```

Here we once again check if the character is a letter and then do some maths to it so we decrease it from “z” down based on the difference from “a” to the current letter, this way we work backwards to the opposite letter in alphabet. If i had more time i would definitely have chosen more interesting encryption methods than these but i fear it might've been a lot to write then too, so it's for the best.

Task 4

I made ChatGPT write out and count from one to 200 with words and took that as my secret message and encoded it with atbash, easy as that. I chose atbash because it was a bit more original than a caesar cipher even though it's basically the same when the caesar cipher key is 25.

Task 5

For the first of the cipher texts(bw222fy.txt) I attempted to decrypt. I found some letters I could directly translate into what they are supposed to be. It started with j = s which I found by taking a look at some words that ended with “ ‘j ” and there were many of them and they were consistent so I could deduce that it was an “s” which is commonly used to end with an apostrophe. Then i saw that there was some large “Z”s that where here and there and the only letter that is written with in large in the middle of a sentence is “I”, with these i could find the correlation that the space between j -> s and i -> z where both 17 and from there i used my own program to decrypt it with 17 as the key and that showed it to be true.

So boringly i figured a lot of people might do a caesar cipher since its both the easiest to code and they might use it rather than the transpositional one, so i made an educated guess based on their encrypted first character of their name in the file and the real first letter of their name in the filename and ran my decrypt caesar with the key 2 on that and found that it was correct(jw223tn.txt).

Other texts where attempted but after note determining if they were either a strange cipher i didn't quite understand or some that i figured were playfair due to word frequency but didnt know how to effectively crack. So I left it and that and took my wins with the caesar ones.

Task 6

a)
The first hash function looked something like this:

```
def get_hash(self, word):
        # Hashing solution nr 1
        wordval = 0
        hashval = 0
        for char in word:
            wordval += ord(char)
        hashval = wordval % len(self.buckets)
        return hashval
```

This hash function simply adds the ascii value of each character together and then takes that modulus of the size, which is 256, and then returns it. This works fine but is maybe not the best to create uniqueness among the values since there are a lot of character combinations that can create the same hash value and this will just increase the max bucket size and zero bucket ratio because fewer buckets will contain more elements. Then after a bit more optimisation:

```
prime = 31
        hash_val = 0

        for char in word:
            hash_val = (hash_val * prime + ord(char)) % 256

        return hash_val
```

This reduced the zero bucket ratio by 0.03.

b)
When it comes to sporadic behaviour in a hash function it comes at a cost unless you have heavily optimised the hashing function for all cases. With mine that looked like this;

```
prime = 31
        hash_val = 0

        # for each letter, increase the hash with the prime times
        # the ascii letter and take all of that to the
        # power of the letters index
        for i in range(len(word)):
            hash_val = ((hash_val * prime + ord(word[i])**i)) % 256

        return hash_val
```

Which sadly increased the max bucket size by 3, and zero bucket ratio by 0.01, but this also made the hashdump change a lot by the smallest of changes.

Next to the test cases, with the first wordlist of 1000 words these were the results:

```
Wordlist 1
Bucket list size = 256
Max bucket size = 10
Zero bucket ration = 0.1
```

Here we then see the effects of the optimised hash function for sporadic behaviour you can see the consequences of this change, but here's an example of how sporadic it is now:

Inputs and their hashdump
"aaa" = 65
"aab" = 4
"aac" = 201
"aad" = 144

And here is the second wordlist:

```
Wordlist 1
Bucket list size = 256
Max bucket size = 7
Zero bucket ration = 0.18
```

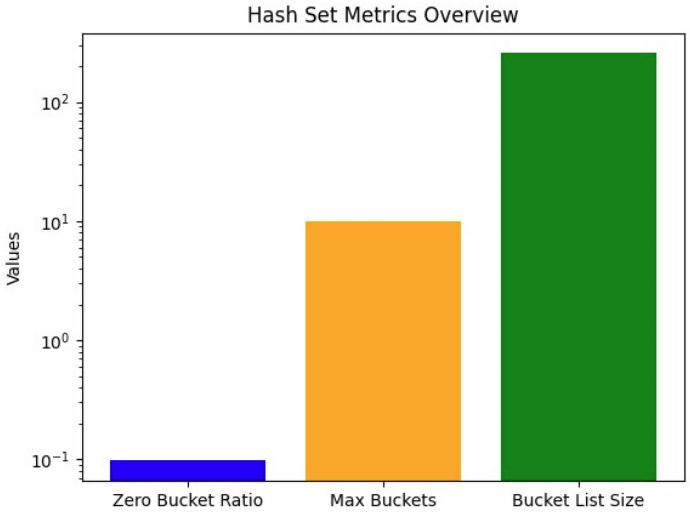
With these results we can see that there are more empty buckets but smaller size of the large buckets, and that means that there are more buckets that are filled up to or close to 7.

Next to analyse a small difference in wordlists, i wrote a short script to open wordlist 1 and shift the last character of each word forward by 1 and then look at the results, they were as follows:

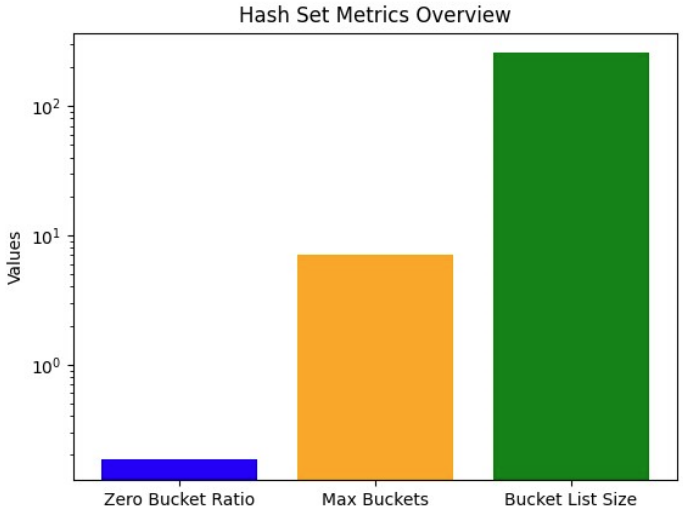
```
Wordlist 1 Shifted
Bucket list size = 256
Max bucket size = 9
Zero bucket ration = 0.1
```

Comparing that to the original wordlist we can see that all that changed was the max bucket size and it was reduced by one. This is kind of surprising since the small change should be large as seen from before but at the same time it is with a large amount of data, the results won't change too much because of the variety staying the same.

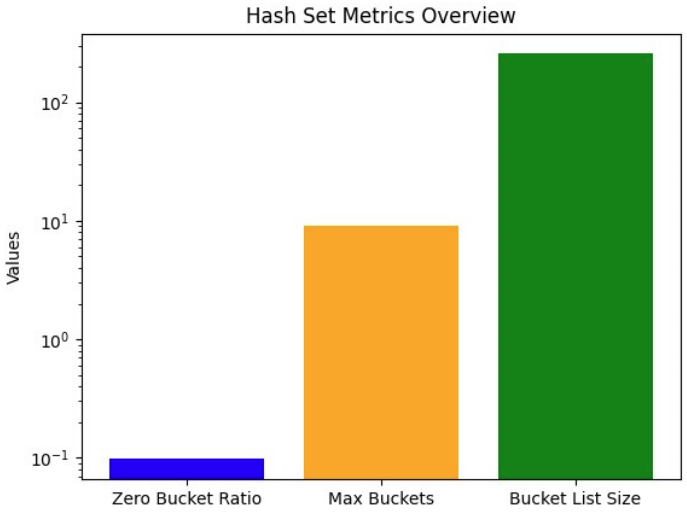
Here are the graphs with the results using matplotlib on a logarithmic scale:
Wordlist 1:



Wordlist 2:



Wordlist 1 shifted:



There are many factors that go into a safe hash function like, the same input should always give the same output, preimage resistance, collision resistance and having an avalanche effect. The most important ones I would say are collision resistance, preimage resistance and the avalanche effect. Collision resistance is when a singular output will never share 2 or more similar inputs, this is important to make sure that attackers can't find correlation between the inputs and be able to reverse engineer it that way. Preimage resistance is similar in the way that it is supposed to make it computationally infeasible to reverse into plaintext. Lastly the avalanche effect, in short terms, a small change to the input should affect the output by a lot. Just changing a bit should generally affect the entire hash dump, and with those 3 i believe you would have a pretty safe hash function and i think what i wrote was close to this. I am not sure if my hash function is collision resistant but because it has the avalanche effect and that makes me believe that there isn't a large risk of it having a collision of inputs[5].

Bibliography

[1] Nicolas Poggi, "Types of Encryption: Symmetric or Asymmetric? RSA or AES?", [2021-06-15], url:[https://preyproject.com/blog/types-of-encryption-symmetric-or-asymmetric-rsa-or-aes].

[2] N/A, "What is difference between Encryption and Hashing? Is Hashing more secure than Encryption?", [2023], url:[https://www.encryptionconsulting.com/education-center/encryption-vs-hashing/].

[3] Justin San Juan, "What is Encryption vs Hashing vs Encoding vs Compression?", [2022-04-09], url:[https://justinsj.medium.com/what-is-encryption-vs-hashing-vs-encoding-vs-compression-360103fa2c61]

[4] Daniel Iwugo, “What is Steganography? How to Hide Data Inside Data”, [2023-06-13], url:[https://www.freecodecamp.org/news/what-is-steganography-hide-data-inside-data/]

[5] Josh Lake, “What is a collision attack?”, [2023-09-13], url:[https://www.comparitech.com/blog/information-security/what-is-a-collision-attack/]