

**Secure Communications GitHub Markdown**

**Cathal Murphy – B00095643**

Department of Informatics

School of Informatics and Engineering

Institute of Technology Blanchardstown

Dublin 15.

Word count:

**BN311 Bachelor of Science in Computing**

**Digital Forensics & Cyber Security**

**21/4/2019**

Contents

[Lab 5: Brute Force Ciphers 3](#_Toc6754326)

[Rot5 3](#_Toc6754327)

[Rot13 4](#_Toc6754328)

[Rot47 4](#_Toc6754329)

[Lab 8: Simple Blockchain 5](#_Toc6754330)

[Lab 9: Credit Card Validator/Generator 6](#_Toc6754331)

[Verify 6](#_Toc6754332)

[Vendor 7](#_Toc6754333)

[Checksum 7](#_Toc6754334)

[Generate 8](#_Toc6754335)

[RSA Level One 9](#_Toc6754336)

[RSA Level Two 9](#_Toc6754337)

[RSA Level Three 9](#_Toc6754338)

[RSA Level Four 10](#_Toc6754339)

[RSA Level Five 10](#_Toc6754340)

[RSA Level Six 10](#_Toc6754341)

[RSA Level Seven 11](#_Toc6754342)

[RSA Level Eight 11](#_Toc6754343)

[RSA Level Nine 12](#_Toc6754344)

# Lab 5: Brute Force Ciphers

Python Version: 2.7

The objective of this lab was to create a python script that would brute force the following classic rotational ciphers:

* Rot5   
  Rotates digits 0-9
* Rot13  
  Rotates over the alphabet
* Rot47  
  Rotates over numbers, letters and other characters such as ‘!’ and ‘=’.

## Rot5

To use rot5 decryption simply enter the following command into the python terminal:

**python DecryptClassicCypher.py rot5 [ciphertext]**

The program will then return 9 rotations of the cipher.

* The way this is achieved is by first adding the ciphertext into an array,
* finding the ascii value of each element,
* add one to the value, checking if the value is greater than the ascii value of 9
* if it is greater than 9, subtract 10 and continue, otherwise continue
* convert the ascii of each element back to its number form
* print the current array
* repeat until 8 more rotations occur.

## Rot13

To use rot13 decryption simply enter the following command into the python terminal:

**python DecryptClassicCypher.py rot13 [ciphertext]**

The program will then return 25 rotations of the cipher.

* The way this is achieved is by first adding the ciphertext into an array,
* finding the ascii value of each element,
* add one to the value, checking if the value is greater than the ascii value of ‘z’
* if it is greater than 9, subtract 26 and continue, otherwise continue
* convert the ascii of each element back to its number form
* print the current array
* repeat until 25 more rotations occur.

## Rot47

To use rot13 decryption simply enter the following command into the python terminal:

**python DecryptClassicCypher.py rot47 [ciphertext]**

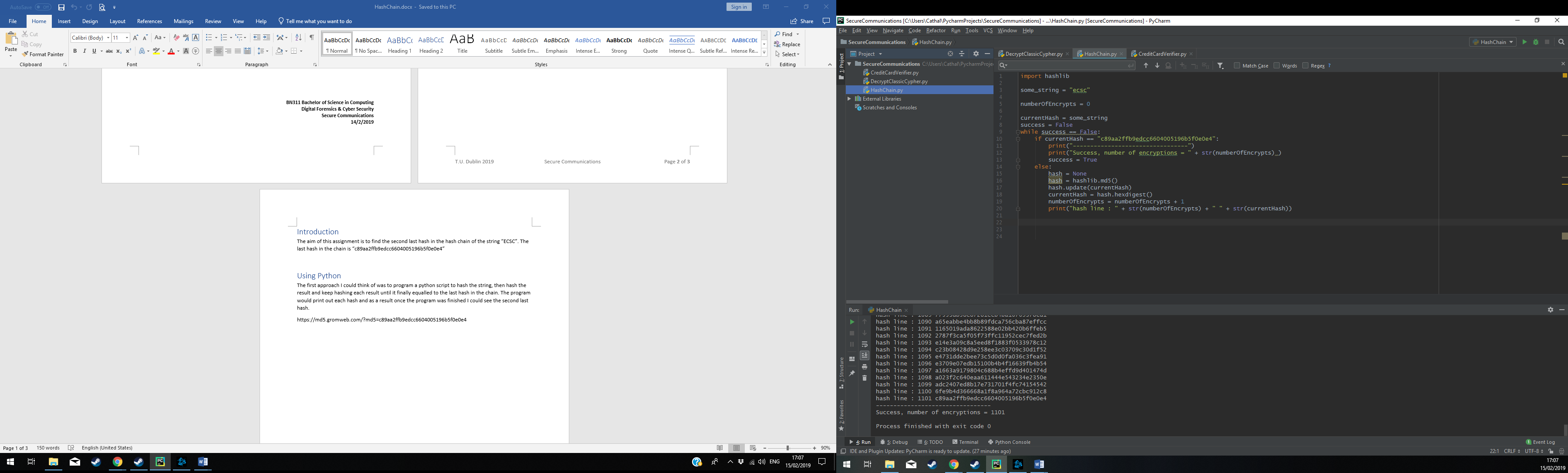
The program will then return 92 rotations of the cipher.

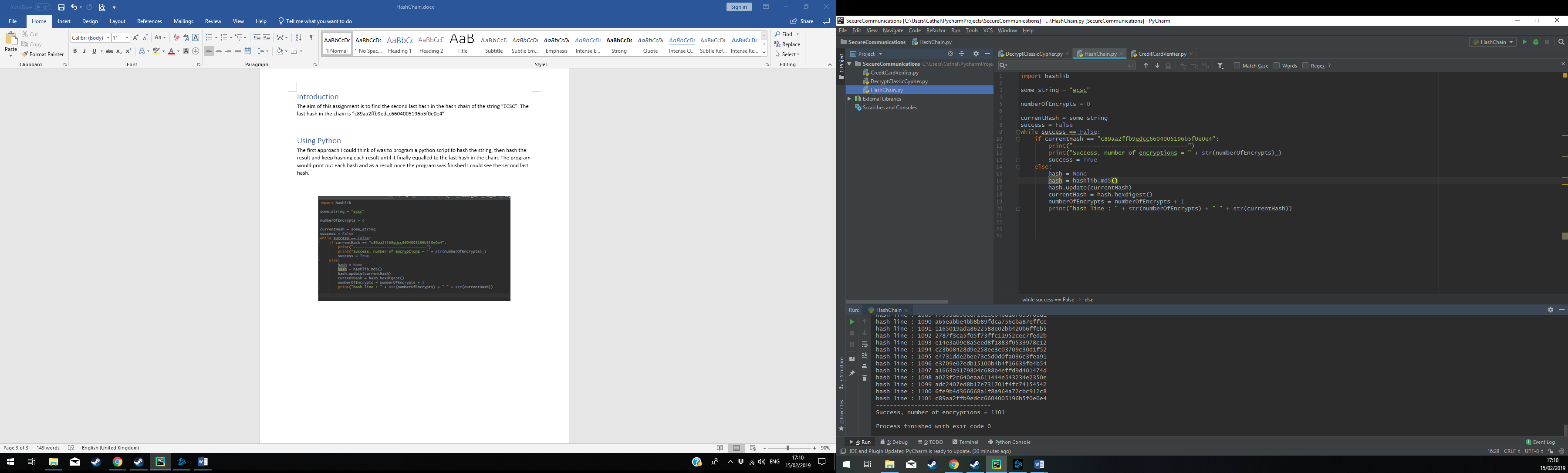
* The way this is achieved is by first adding the ciphertext into an array,
* finding the ascii value of each element,
* add one to the value, checking if the value is greater than the ascii value of ‘~’
* if it is greater than 9, subtract 94 and continue, otherwise continue
* convert the ascii of each element back to its number form
* print the current array
* repeat until 91 more rotations occur.

# Lab 8: Simple Blockchain

Python Version: 2.7

The aim of this assignment is to find the second last hash in the hash chain of the string “ECSC”. The last hash in the chain is “c89aa2ffb9edcc6604005196b5f0e0e4”

The first approach I could think of was to program a python script to hash the string, then hash the result and keep hashing each result until it finally equalled to the last hash in the chain. The program would print out each hash and as a result once the program was finished I could see the second last hash.



As you can see from the screenshot above, we successfully found the second last hash in the hash chain:

**hash line: 1100 6fe9b4d366668a1f8a964a72cbc912c8**

# Lab 9: Credit Card Validator/Generator

Python Version: 2.7

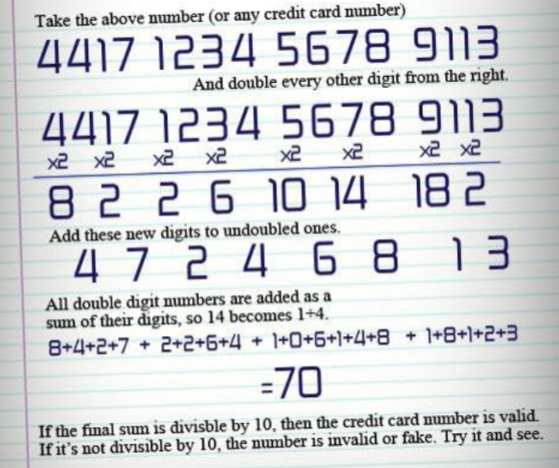
The aim of this assignment was to write a python program to perform the following:

* Verify: Take a credit card number as input and output if it is a valid credit card number.
* Vendor: Take a credit card number as input and output the issuing vendor
* Checksum: Given just the first portion of a credit card calculate the checksum
* Generate: Select the issuing vendor, then generate a random valid credit card

## Verify

To use the verify function of my program you must enter the following command into the terminal:

**python CreditCardVerifier.py verify [Card Number]**

To verify if a credit card number is valid, I used the following formula.

I created a function called verify() which performs the task show above. This task is called a checksum. This task checks the value output from the sum above and if it is divisible by 10 then the credit card number is valid. If it is not divisible by 10 then the credit card number Is invalid.

## Vendor

To use the vendor function of my program you must enter the following command into the terminal:

**Python CreditCardVerifier.py vendor [Card Number]**

Based on the characteristics each card has, I created a list of if statements and Boolean logic for each statement describing the nature each card vendor has. These characteristics are things like what numbers the credit card starts with, what length is the card number and combinations of each. The hardest part of this function was keeping the Boolean logic straight in your head which was difficult because some of the card numbers like Visa Electron had so many combinations of types of card numbers that it could be.

## Checksum

To use the checksum function of my program you must enter the following command into the terminal:

**Python CreditCardVerifer.py checksum [Number]**

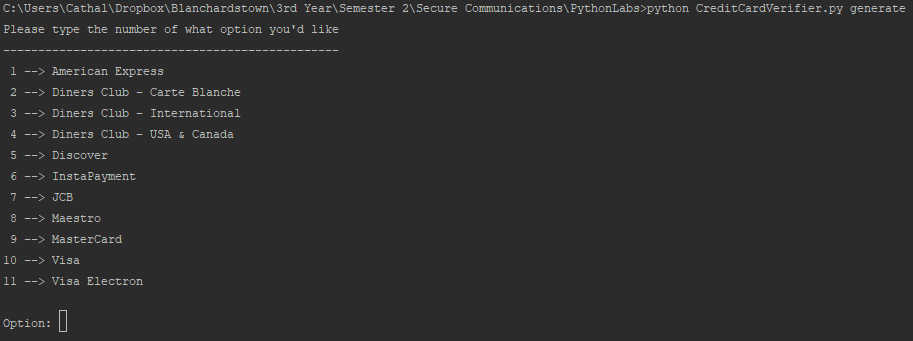
Credit cards (or the one used for my program) can be between 14 and 19. I used a random int generator between these numbers which will express the length of the card number. The number that was initially input will be stored in an array and numbers will populate the rest of the array based on the length of the card. Until the checksum passes, the numbers will be randomly added, then removed.

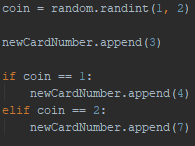
## Generate

To use the generate function of my program you must enter the following command into the terminal:

**python CreditCardVerifier.py generate**

You will then be prompted with the following menu.



Select the card vendor you’d like to generate a credit card number for (1 for American Express).  
Because some of the credit card types have attributes that can change (like some can be length 15 AND 16) I tried to randomize the outcome using a coinflip where an attribute is expressed as an integer and a random number is chosen between those numbers.

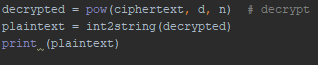
# RSA Level One

Python Version: 3.7  
These programs just need to be executed, doesn’t use sys.argv

Its important to note that I used “MarksRSAScript.py” for many of these so the code is like that. RSA Level one is just to encrypt a message and the ciphertext is the flag. Using the script mentioned before, adding in the variables gained from rsa1.txt the answer was simple.

# RSA Level Two

RSA Level two is the same as level one only we are decrypting instead. The code for this was gained from the MarksRSAScript.py which gave me the formula for decrypting too.

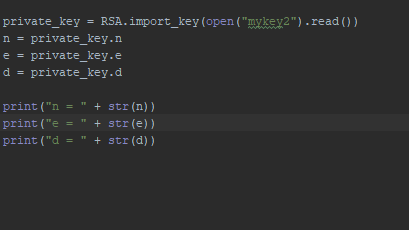


# RSA Level Three

Level three was to find the n, e and d values from a given private key. To do this I imported the following:

**From Crypto.PublicKey import RSA**

This library allows for this exact functionality.



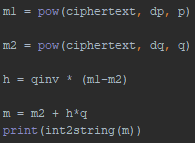
Its important to note that “mykey2” is the text file containing the private key that will also be submitted to my GitHub.

# RSA Level Four

Level four required me to not only gain n and d from the private key but also decrypt a given ciphertext. This was simple as using the import from Level three alongside the decrypt formula in Level two, the answer becomes clear almost immediately.

# RSA Level Five

Level five was difficult and required me to go to the RSA Wiki to find the following formula to reading the ciphertext.



# RSA Level Six

RSA Level six required code directly from rosetta code to determine the following:

**d , determined by calculating the modinv(e,(p-1 \* q-1))**

Modinv in this case was an imported function from rosetta code to calculate the modular inverse. Since we have e , p and q for this all we need is d , which is the formula in bold up above, and n to decrypt the message.

**n , determined by calculating p\*q**

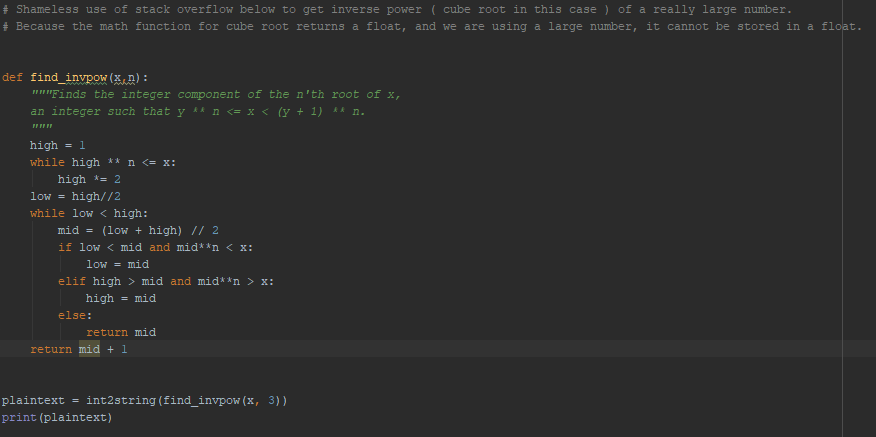
once these two variables were found it was as simple as using the same code from level two to decrypt the ciphertext.

# RSA Level Seven

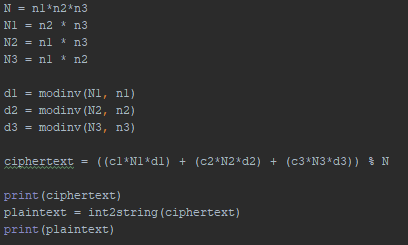
For this one we were given n, e and ciphertext. We need p and q for the same approach to work as used in level six. We know n is p \* q so all we need to do is find the factors of n , there should only be one or two given the nature of n. Using factor DB I was able to find p and q and simply used the same approach to solve this level as I did in level six.

# RSA Level Eight

Level eight required us to find the inverse power (cube root in this case) of n which is a really big number. The math function for cube root returns a float and our answer is too big for a float to store so we must find another way to perform the cube root. I found some code on stack overflow which returns the inverse power using arithmetic and logic, basically rewriting the math function for cube root to return a value that isn’t a float. Once this was done the plaintext was revealed.



# RSA Level Nine

In level nine we are given 3 messages with their private keys and they all share the same e. The formula was found with some help from the teacher with relation to many messages using the same e.

Using the formula above however, I was unable to decrypt the ciphertext and unable to continue levels 10 and 11.