CHAPTER 19

THE VIGENÈRE CIPHER

Topics Covered In This Chapter:

- Subkeys

> "I believed then, and continue to believe now, that the benefits to our security and freedom of widely available cryptography far, far outweigh the inevitable damage that comes from its use by criminals and terrorists... I believed, and continue to believe, that the arguments against widely available cryptography, while certainly advanced by people of good will, did not hold up against the cold light of reason and were inconsistent with the most basic American values."
>
> Matt Blaze, AT&T Labs, September 2001

Le Chiffre Indéchiffrable

The Vigenère cipher is a stronger cipher than the ones we've seen before. There are too many possible keys to brute-force, even with English detection. It cannot be broken with the word pattern attack that worked on the simple substitution cipher. It was possibly first described in 1553 by Italian cryptographer Giovan Battista Bellaso (though it has been reinvented many times, including by Blaise de Vigenère). It is thought to have remained unbroken until

Charles Babbage, considered to be the father of computers, broke it in the 19th century. It was called “le chiffre indéchiffrable”, French for “the indecipherable cipher”.

Figure 19-1. Blaise de Vigenère
April 5, 1523 - 1596

Figure 19-2. Charles Babbage
December 26, 1791 - October 18, 1871

Multiple “Keys” in the Vigenère Key

The Vigenère cipher is similar to the Caesar cipher, except with multiple keys. Because it uses more than one set of substitutions, it is also called a **polyalphabetic substitution cipher**. Remember that the Caesar cipher had a key from 0 to 25. For the Vigenère cipher, instead of using a numeric key, we will use a letter key. The letter A will be used for key 0. The letter B will be used for key 1, and so on up to Z for the key 25.

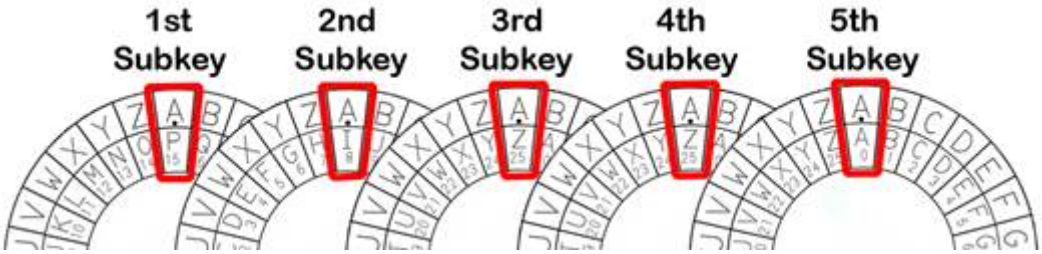
0	1	2	3	4	5	6	7	8	9	10	11	12
A	B	C	D	E	F	G	H	I	J	K	L	M

13	14	15	16	17	18	19	20	21	22	23	24	25
N	O	P	Q	R	S	T	U	V	W	X	Y	Z

The key in a Vigenère cipher is a series of letters, such as a single English word. **This single word key will be split into multiple subkeys.** If we use a Vigenère key of “PIZZA”, then the first subkey is P, the second subkey is I, the third and fourth subkeys are both Z and the fifth subkey is A. We will use the first subkey to encrypt the first letter of the plaintext, and the second subkey to encrypt the second letter, and so on. When we get to the sixth letter of the plaintext, we will go back to using the *first* subkey.

The Vigenère cipher is the same as using multiple Caesar ciphers in the same message.

Figure 19-3. Multiple Caesar ciphers combine to make the Vigenère cipher



The following shows which subkey will encrypt which letters in the message, “Common sense is not so common.” with the Vigenère key, “PIZZA”.

```
COMMONSENSEISNOTSOCOMMON
PIZZAPIZZAPIZZAPIZZAPIZZ
```

To encrypt the first C with the subkey P, encrypt it with the Caesar cipher using numeric key 15 (15 is the number for the letter P) which creates the ciphertext R, and so on. Do this for each of the letters of the plaintext. The following table shows this process:

Table 19-1. Numbers of the letters before and after encryption.

Plaintext Letter	Subkey		Ciphertext Letter
C (2)	P (15)	→	R (17)
O (14)	I (8)	→	W (22)
M (12)	Z (25)	→	L (11)
M (12)	Z (25)	→	L (11)
O (14)	A (0)	→	O (14)
N (13)	P (15)	→	C (2)
S (18)	I (8)	→	A (0)
E (4)	Z (25)	→	D (3)
N (13)	Z (25)	→	M (12)
S (18)	A (0)	→	S (18)
E (4)	P (15)	→	T (19)
I (8)	I (8)	→	Q (16)
S (18)	Z (25)	→	R (17)
N (13)	Z (25)	→	M (12)
O (14)	A (0)	→	O (14)
T (19)	P (15)	→	I (8)
S (18)	I (8)	→	A (0)
O (14)	Z (25)	→	N (13)
C (2)	Z (25)	→	B (1)
O (14)	A (0)	→	O (14)
M (12)	P (15)	→	B (1)
M (12)	I (8)	→	U (20)
O (14)	Z (25)	→	N (13)
N (13)	Z (25)	→	M (12)

So using the Vigenère cipher with the key "PIZZA" (which is made up of the subkeys 15, 8, 25, 25, 0) the plaintext "Common sense is not so common." becomes the ciphertext "Rwlloc admst qr moi an bobunm."

The more letters in the Vigenère key, the stronger the encrypted message will be against a brute-force attack. The choice of "PIZZA" is a poor one for a Vigenère key, because it only has five letters. A key with only five letters has 11,881,376 possible combinations. (26 ^ 5 = 26 × 26 × 26 × 26 × 26 = 11,881,376) Eleven million keys is far too many for a human to try out, but a computer could try them all in a few hours. It would first try to decrypt the message with the key "AAAAA" and check if the resulting decryption was in English. Then it could try "AAAAB", then "AAAAC", until it got to "PIZZA".

The good news is that for every additional letter the key has, the number of possible keys multiplies by 26. Once there are quadrillions of possible keys, it would take a computer years to break. Table 19-2 shows how many possible keys there are for each length:

Table 19-2. Number of possible keys based on Vigenère key length.

Key Length	Equation	Possible Keys
1	26	= 26
2	26 × 26	= 676
3	676 × 26	= 17,576
4	17,576 × 26	= 456,976
5	456,976 × 26	= 11,881,376
6	11,881,376 × 26	= 308,915,776
7	308,915,776 × 26	= 8,031,810,176
8	8,031,810,176 × 26	= 208,827,064,576
9	208,827,064,576 × 26	= 5,429,503,678,976
10	5,429,503,678,976 × 26	= 141,167,095,653,376
11	141,167,095,653,376 × 26	= 3,670,344,486,987,776
12	3,670,344,486,987,776 × 26	= 95,428,956,661,682,176
13	95,428,956,661,682,176 × 26	= 2,481,152,873,203,736,576
14	2,481,152,873,203,736,576 × 26	= 64,509,974,703,297,150,976

Once we get to keys that are twelve or more letters long, then it becomes impossible for most consumer laptops to crack in a reasonable amount of time.

A Vigenère key does not have to be a word like “PIZZA”. It can be any combination of letters, such as “DURIWKNMFICK”. In fact, it is much better not to use a word that can be found in the dictionary. The word “RADIOLOGISTS” is a 12-letter key that is easier to remember than “DURIWKNMFICK” even though they have the same number of letters. But a cryptanalyst might anticipate that the cryptographer is being lazy by using an English word for the Vigenère key. There are 95,428,956,661,682,176 possible 12-letter keys, but there are only about 1,800 12-letter words in our dictionary file. If you are using a 12-letter English word, it would be easier to brute-force that ciphertext than it would be to brute-force the ciphertext from a 3-letter random key.

Of course, the cryptographer is helped by the fact that the cryptanalyst does not know how many letters long the Vigenère key is. But the cryptanalyst could try all 1-letter keys, then all 2-letter keys, and so on.

Source Code of Vigenère Cipher Program

Open a new file editor window by clicking on **File ► New Window**. Type in the following code into the file editor, and then save it as *vigenereCipher.py*. Press **F5** to run the program. Note that first you will need to download the *pyperclip.py* module and place this file in the same directory as the *vigenereCipher.py* file. You can download this file from http://invpy.com/pyperclip.py.

Source code for vigenereCipher.py

```
 1. # Vigenere Cipher (Polyalphabetic Substitution Cipher)
 2. # http://inventwithpython.com/hacking (BSD Licensed)
 3.
 4. import pyperclip
 5.
 6. LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
 7.
 8. def main():
 9.     # This text can be copy/pasted from http://invpy.com/vigenereCipher.py
10.     myMessage = """Alan Mathison Turing was a British mathematician, logician,
cryptanalyst, and computer scientist. He was highly influential in the development of computer
science, providing a formalisation of the concepts of "algorithm" and "computation" with the
Turing machine. Turing is widely considered to be the father of computer science and
artificial intelligence. During World War II, Turing worked for the Government Code and Cypher
School (GCCS) at Bletchley Park, Britain's codebreaking centre. For a time he was head of Hut
8, the section responsible for German naval cryptanalysis. He devised a number of techniques
for breaking German ciphers, including the method of the bombe, an electromechanical machine
that could find settings for the Enigma machine. After the war he worked at the National
Physical Laboratory, where he created one of the first designs for a stored-program computer,
the ACE. In 1948 Turing joined Max Newman's Computing Laboratory at Manchester University,
where he assisted in the development of the Manchester computers and became interested in
mathematical biology. He wrote a paper on the chemical basis of morphogenesis, and predicted
oscillating chemical reactions such as the Belousov-Zhabotinsky reaction, which were first
observed in the 1960s. Turing's homosexuality resulted in a criminal prosecution in 1952, when
homosexual acts were still illegal in the United Kingdom. He accepted treatment with female
hormones (chemical castration) as an alternative to prison. Turing died in 1954, just over two
weeks before his 42nd birthday, from cyanide poisoning. An inquest determined that his death
was suicide; his mother and some others believed his death was accidental. On 10 September
2009, following an Internet campaign, British Prime Minister Gordon Brown made an official
public apology on behalf of the British government for "the appalling way he was treated." As
of May 2012 a private member's bill was before the House of Lords which would grant Turing a
statutory pardon if enacted."""
11.     myKey = 'ASIMOV'
12.     myMode = 'encrypt' # set to 'encrypt' or 'decrypt'
13.
14.     if myMode == 'encrypt':
15.         translated = encryptMessage(myKey, myMessage)
16.     elif myMode == 'decrypt':
17.         translated = decryptMessage(myKey, myMessage)
18.
19.     print('%sed message:' % (myMode.title()))
20.     print(translated)
21.     pyperclip.copy(translated)
22.     print()
23.     print('The message has been copied to the clipboard.')
24.
25.
26. def encryptMessage(key, message):
27.     return translateMessage(key, message, 'encrypt')
28.
29.
30. def decryptMessage(key, message):
31.     return translateMessage(key, message, 'decrypt')
32.
33.
34. def translateMessage(key, message, mode):
35.     translated = [] # stores the encrypted/decrypted message string
```

```
36.
37.     keyIndex = 0
38.     key = key.upper()
39.
40.     for symbol in message: # loop through each character in message
41.         num = LETTERS.find(symbol.upper())
42.         if num != -1: # -1 means symbol.upper() was not found in LETTERS
43.             if mode == 'encrypt':
44.                 num += LETTERS.find(key[keyIndex]) # add if encrypting
45.             elif mode == 'decrypt':
46.                 num -= LETTERS.find(key[keyIndex]) # subtract if decrypting
47.
48.             num %= len(LETTERS) # handle the potential wrap-around
49.
50.             # add the encrypted/decrypted symbol to the end of translated.
51.             if symbol.isupper():
52.                 translated.append(LETTERS[num])
53.             elif symbol.islower():
54.                 translated.append(LETTERS[num].lower())
55.
56.             keyIndex += 1 # move to the next letter in the key
57.             if keyIndex == len(key):
58.                 keyIndex = 0
59.         else:
60.             # The symbol was not in LETTERS, so add it to translated as is.
61.             translated.append(symbol)
62.
63.     return ''.join(translated)
64.
65.
66. # If vigenereCipher.py is run (instead of imported as a module) call
67. # the main() function.
68. if __name__ == '__main__':
69.     main()
```

Sample Run of the Vigenère Cipher Program

```
Encrypted message:
Adiz Avtzqeci Tmzubb wsa m Pmilqev halpqavtakuoi, lgouqdaf, kdmktsvmztsl, izr xoexghzr
kkusitaaf. Vz wsa twbhdg ubalmmzhdad qz hce vmhsgohuqbo ox kaakulmd gxiwvos, krgdurdny i
rcmmstugvtawz ca tzm ocicwxfg jf "stscmilpy" oid

...skipped for brevity...

uiydviyv, Nfdtaat Dmiem Ywiikbqf Bojlab Wrgez avdw iz cafakuog pmjxwx ahwxcby gv nscadn at ohw
Jdwoikp scqejvysit xwd "hce sxboglavs kvy zm ion tjmmhzd." Sa at Haq 2012 i bfdvsbq azmtmd'g
widt ion bwnafz tzm Tcpsw wr Zjrva ivdcz eaigd yzmbo Tmzubb a kbmhptgzk dvrvwz wa efiohzd.

The message has been copied to the clipboard.
```

How the Program Works

vigenereCipher.py

```
1. # Vigenere Cipher (Polyalphabetic Substitution Cipher)
2. # http://inventwithpython.com/hacking (BSD Licensed)
3.
4. import pyperclip
5.
6. LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

The beginning of the program has the usual comments to describe the program, an `import` statement for the `pyperclip` module, and creates a variable called `LETTERS` with a string of every uppercase letter.

vigenereCipher.py

```
8. def main():
9.     # This text can be copy/pasted from http://invpy.com/vigenereCipher.py
10.    myMessage = """Alan Mathison Turing was a British mathematician,

       ...skipped for brevity...

grant Turing a statutory pardon if enacted."""
11.    myKey = 'ASIMOV'
12.    myMode = 'encrypt' # set to 'encrypt' or 'decrypt'
13.
14.    if myMode == 'encrypt':
```

```
15.         translated = encryptMessage(myKey, myMessage)
16.     elif myMode == 'decrypt':
17.         translated = decryptMessage(myKey, myMessage)
18.
19.     print('%sed message:' % (myMode.title()))
20.     print(translated)
21.     pyperclip.copy(translated)
22.     print()
23.     print('The message has been copied to the clipboard.')
```

The `main()` function for the Vigenère cipher is exactly like the other `main()` functions in this book: there are variables for `message`, `key`, and `mode`. The user sets these variables on lines 10, 11, and 12 before running the program. The encrypted or decrypted message (depending on what `myMode` is set to) is stored in a variable named `translated` so that it can be printed to the screen (on line 20) and copied to the clipboard (on line 21).

The code that does the actual encryption and decryption is in `translateMessage()`, which is explained later.

vigenereCipher.py

```
26. def encryptMessage(key, message):
27.     return translateMessage(key, message, 'encrypt')
28.
29.
30. def decryptMessage(key, message):
31.     return translateMessage(key, message, 'decrypt')
```

Since the encryption and decryption use much of the same code as the other, we put them both in `translateMessage()`. The `encryptMessage()` and `decryptMessage()` functions are wrapper functions for `translateMessage()`. (Wrapper functions were covered in Chapter 17.)

vigenereCipher.py

```
34. def translateMessage(key, message, mode):
35.     translated = [] # stores the encrypted/decrypted message string
36.
37.     keyIndex = 0
38.     key = key.upper()
```

In the `translateMessage()` function, we will slowly build the encrypted (or decrypted) string one character at a time. The list in `translated` will store these characters so that they can be joined together once the string building is done. (The reason we use a list instead of just appending the characters to a string is explained in the "Building Strings in Python with Lists" section in Chapter 18.)

Remember, the Vigenère cipher is just the Caesar cipher except that a different key is used depending on the position of the letter in the message. The `keyIndex` variable keeps track of which subkey to use. The `keyIndex` variable starts off as `0`, because the letter used to encrypt or decrypt the first character of the message will be the one at `key[0]`.

Our code assumes that the key has only uppercase letters. To make sure the key is valid, line 38 sets the key to be the uppercase version of it.

vigenereCipher.py

```
40.     for symbol in message: # loop through each character in message
41.         num = LETTERS.find(symbol.upper())
42.         if num != -1: # -1 means symbol.upper() was not found in LETTERS
43.             if mode == 'encrypt':
44.                 num += LETTERS.find(key[keyIndex]) # add if encrypting
45.             elif mode == 'decrypt':
46.                 num -= LETTERS.find(key[keyIndex]) # subtract if decrypting
```

The rest of the code in `translateMessage()` is similar to the Caesar cipher code. The `for` loop on line 40 sets the characters in `message` to the variable `symbol` on each iteration of the loop. On line 41 we find the index of the uppercase version of this symbol in `LETTERS`. (This is how we translate a letter into a number).

If `num` was not set to `-1` on line 41, then the uppercase version of `symbol` was found in `LETTERS` (meaning that `symbol` is a letter). The `keyIndex` variable keeps track of which subkey to use, and the subkey itself will always be what `key[keyIndex]` evaluates to.

Of course, this is just a single letter string. We need to find this letter's index in the `LETTERS` to convert the subkey into an integer. This integer is then added (if encrypting) to the symbol's number on line 44 or subtracted (if decrypting) to the symbol's number on line 46.

vigenereCipher.py

```
48.             num %= len(LETTERS) # handle the potential wrap-around
```

In the Caesar cipher code, we checked if the new value of `num` was less than `0` (in which case, we added `len(LETTERS)` to it) or if the new value of `num` was `len(LETTERS)` or greater (in which case, we subtracted `len(LETTERS)` from it). This handles the "wrap-around" cases.

However, there is a simpler way that handles both of these cases. If we mod the integer stored in num by `len(LETTERS)`, then this will do the exact same thing except in a single line of code.

For example, if `num` was `-8` we would want to add `26` (that is, `len(LETTERS)`) to it to get `18`. Or if `num` was `31` we would want to subtract `26` to get `5`. However `-8 % 26` evaluates to `18` and `31 % 26` evaluates to `5`. The modular arithmetic on line 48 handles both "wrap-around" cases for us.

vigenereCipher.py

```
50.             # add the encrypted/decrypted symbol to the end of translated.
51.             if symbol.isupper():
52.                 translated.append(LETTERS[num])
53.             elif symbol.islower():
54.                 translated.append(LETTERS[num].lower())
```

The encrypted (or decrypted) character exists at `LETTERS[num]`. However, we want the encrypted (or decrypted) character's case to match `symbol`'s original case. So if `symbol` is an uppercase letter, the condition on line 51 will be `True` and line 52 will append the character at `LETTERS[num]` to `translated`. (Remember, all the characters in the `LETTERS` string are already uppercase.)

However, if `symbol` is a lowercase letter, than the condition on line 53 will be `True` instead and line 54 will append the lowercase form of `LETTERS[num]` to `translated` instead. This is how we can get the encrypted (or decrypted) message to match the casing of the original message.

vigenereCipher.py

```
56.             keyIndex += 1 # move to the next letter in the key
57.             if keyIndex == len(key):
58.                 keyIndex = 0
```

Now that we have translated the symbol, we want to make sure that on the next iteration of the `for` loop we use the next subkey. Line 56 increments `keyIndex` by one. This way when the next iteration uses `key[keyIndex]` to get the subkey, it will be the index to the next subkey.

However, if we were on the last subkey in the key, then `keyIndex` would be equal to the length of key. Line 57 checks for this condition, and resets `keyIndex` back to `0` on line 58. That way `key[keyIndex]` will point back to the first subkey.

vigenereCipher.py

```
59.         else:
60.             # The symbol was not in LETTERS, so add it to translated as is.
61.             translated.append(symbol)
```

From the indentation you can tell that the `else` statement on line 59 is paired with the `if` statement on line 42. The code on line 61 executes if the symbol was not found in the `LETTERS` string. This happens if `symbol` is a number or punctuation mark such as `'5'` or `'?'`. In this case, line 61 will just append the symbol untranslated.

vigenereCipher.py

```
63.     return ''.join(translated)
```

Now that we are done building the string in `translated`, we call the `join()` method on the blank string to join together all the strings in `translated` (with a blank in between them).

vigenereCipher.py

```
66. # If vigenereCipher.py is run (instead of imported as a module) call
67. # the main() function.
68. if __name__ == '__main__':
69.     main()
```

Lines 68 and 69 call the `main()` function if this program was run by itself, rather than imported by another program that wants to use its `encryptMessage()` and `decryptMessage()` functions.

Summary

We are close to the end of the book, but notice how the Vigenère cipher isn't that much more complicated than the second cipher program in this book, the Caesar cipher. With just a few changes, we can create a cipher that has exponentially many more possible keys than can be brute-forced.

The Vigenère cipher is not vulnerable to the dictionary word pattern attack that our Simple Substitution hacker program uses. The "indecipherable cipher" kept secret messages secret for hundreds of years. The attack on the Vigenère cipher was not widely known until the early 20th century. But of course, this cipher too eventually fell. In the next couple of chapters, we will learn new "frequency analysis" techniques to hack the Vigenère cipher.