Group 7
CSCI 313
HW1—Question 6: Vigenère Cipher

The Vigenère cipher is a substitution cipher similar to the Caesar cipher, except that the substitution of one letter with another is done with a keyword, which uses the letters of the repeated keyword as a loop of numbers, and then uses those numbers as the amount each letter of the plaintext is shifted. The ciphertext it creates is extremely difficult to break, because it is not a simple one-for-one substitution. The same letter in the plaintext can become different letters in the ciphertext, and vice versa. This code reads a message from a text file, generates a random secret key of length 32, encodes the message into ciphertext using the Vigenère cipher, and then attempts to break it with only the knowledge that the key is 32 characters long.

The code begins by simply reading a message from a text file into a string, which is then converted into a vector of integers based on the letters. ‘A’ is converted to 0, ‘B’ is converted to 1, all the way up to ‘Z’ becoming 25. Of course, this means numbers, punctuation, and spaces must be weeded out of the string when being converted, which is handled by the function **convertChar** returning -1, which is then not entered into the vector. The vector data structure is ideal for this sort of work because it is of variable size and random-access, which aids with many of the functions used later. Incidentally, there is also a **convertInt** function, which converts the integers back into letters. For this example code, the text file contains a message taken from the introduction of Sandworm: A New Era of Cyberwar and the Hunt for the Kremlin's Most Dangerous Hackers by Andy Greenberg, because it happened to be open on the designer’s Kindle. The message is 1,314 characters long.

Once the vector **v** containing the integer representation of the message is constructed, a random key of length 32 is generated by the code. The vector **v** and the key **secretKey** are then

sent to the **encode** function, which iterates through the text and the key, adding elements modulo 26, to generate the ciphertext, which is then placed into **v**. Additionally, there is a function called **decode**, which merely subtracts the key elements from the text elements, adding 26 if it becomes negative.

Once the message is encrypted, the breaking begins. Keys of length 32 are randomly generated and passed to a function called **keyScore**, along with the encrypted message. The function uses the randomly generated key to decrypt the message and then analyzes the resulting text. The analysis focuses on the frequency of letters in the English language, which occurs over quite a broad range of 13% for ‘e’ to 0.074% for ‘z’. The function returns a double in the form of a key score to the main. The key score relies on the principle that, having two strings and wanting to take a sum of products of pairs of numbers from each string, the largest sum will be derived from the product of largest numbers in each string, plus the product of the second largest in each string, and so on. This sum is then divided by the number of letters in the text to generate a more broadly applicable score. The average random key returns a key score of approximately 3.4. For comparison, the actual key, when tested, typically returns a value of 6.2 or so. Random keys with a score above 4.0 are relatively scarce. A newly created vector of vectors called **keyList** is created to store keys, and then a while loop is run while **keyList** contains fewer than 30 keys. In order to qualify for **keyList**, a randomly generated key must return a key score of 4.2. In practice, this takes about a minute.

However, the keys collected still aren’t very good. And here is where the **frankenKey** comes in. The **frankenKey** is generated using the function called **elementScore**, which is based on the same principle as **keyScore**. Using a nested for loop, the main function sends **elementScore** the message, a key, and a position, and **elementScore** returns a score based on

only that position of the key. For example, checking the element in position 4 of the key, **elementScore** will return a score based on the decrypted letters in positions 4, 36, 68, etc. adding 32 each time (the function adds by **key.size**, so it is adaptable to varying key lengths). This way, a score is developed for every element of the key. Iterating through the list of good keys, the loop finds the maximum score for the first letter of the key, and inserts the letter with that score into **frankenKey**, before moving on to the next element. Doing this generates a dramatically improved key, which is frequently off from the secret key by only one or two elements, and is therefore capable of decrypting the message with some human adjustments.

While this code does a good job in finding the **secretKey**, there are a number of caveats. The first is the lack of adaptability. The threshold of 4.2 for the **keyScores** was discovered by trial and error, and the **keyList** size of 30 was an arbitrary decision. However, using a different text or key could result in a dramatically different distribution of **keyScores**. A shorter key, for example, would provide the program with a greater chance of guessing close to key by random chance, requiring the threshold to be raised, and a shorter message would introduce significantly more variability into the frequencies of the letters, increasing the likelihood that the element with the highest **elementScore** would be incorrect.

Additionally, frequency analysis has a significant flaw in that it can be defended against by consciously altering the frequency of the letters in the original message. Avoiding the letter ‘e’ alone would create significant difficulties. One strategy that was considered but ultimately abandoned in this attempt was allowing the user to enter a key word or phrase that they suspect the message of containing, which is a strategy used against the Germans in the cracking of the enigma code in World War 2. Additionally, there is some potential for human feedback for the

algorithm, offering up decrypted messages which the user can then evaluate for any possible noticeable words or phrases. The possibilities are incredibly varied.