Cryptanalysis of a class of ciphers based on patterns in shifts between the ciphertext and plaintext

Team Members:

Fabiha Ahmed

Avik Gomes

Tanya Jain

Michelle Yang

Team Members & Project Tasks

Fabiha Ahmed - created code in python for decryption algorithm + created encryption algorithm for tester

Avik Gomes - improved code efficiency + documented code and made code more readable

Tanya Jain - wrote the informal explanation and extra credit section of the report + converted the python pseudocode into c++

Michelle Yang - wrote the formal explanation of the report + took care of input and output text + wrote the base code for c++

All - Came up with the approach to decrypt the ciphertext + troubleshooting

Informal Explanation

We only included one approach for test one and test two and we tested these approaches using sample cipher texts we created ourselves. The approach for test one, takes as input a ciphertext, encrypted using one of the five given candidate plaintexts. It converts each character of the ciphertext into an int using its ASCII values, and stores that into a vector. Then it reads the file plaintext_dictionary_test1 and stores only the first 48 characters from each candidate plaintext into a vector. We only store the first 48, because we know the maximum key length is 24. So by the pigeonhole theorem, we know for sure a pattern will be seen by the 48th character. This way, we save time by not having to read and compare all the 500 characters. Next, we find the difference (the number of shifts in letters) between each character from the ciphertext and each character from the first 48 characters (its ASCII value) of the 5 plaintexts. Once we find all the differences, we save these differences into a vector of ints. Then we compare the differences obtained for all the 5 plaintexts to see which one has a pattern. If a pattern is found, we store the index of which plaintext gave us that pattern. We read from the file again, and print out the resulting plaintext. If no pattern is found, we move on to test two.

For test two, instead of reading the dictionary file, we save time by hardcoding the 40 words, with added extra space in the end of each word, into an array of strings. We then create all possible permutations of three words that can be made using the 40 words using a recursive

function called *diffCombos*. We chose our number of words to be three for its runtime efficiency and because out of the 64,000 possible combinations, 4,234 combinations have a character length of less than 24 (the length we need to guarantee a key exists). The possibilities also account for repetitions since we figured that even if all 40 words are used, they don't add up to 500 characters (which is the length of the ciphertext). Therefore, we know some words will repeat for sure. In the function *diffCombos*, we call the function *encode*, which turns all the characters into their corresponding ASCII values, and turns the space character into the number 96. *diffCombos* then pushes all these combinations, in the form of ints, into a vector of vector of ints called *possCipher*. These permutations are now our possible plaintext candidates.

After this, we apply the same strategy as test 1, where we find the differences between all our possible plaintext candidates and ciphertext characters. We do so by looping through the vector *possCipher*, and calling upon the *diff* function, that returns a vector that contains the ASCII difference between the candidate plaintext and ciphertext. Then we use the function *keyExists*, to see if a cycle exists among the differences, and if we see a cycle we return a vector that holds the pattern of shifts. If no pattern is found, we return an empty vector. If the vector is not empty, that means it could be the possible shift pattern. Since we only used three words out of the 40, we can have more than one key. Therefore, to check which key is the actual key, we decrypt the whole ciphertext using each of the possible keys we found. We call the function *correctPlaintext* which loops through all the words in the possible plaintext and uses the vector *find* method to see if each word in it is a part of the 40 words. If even one word is not found in the 40 word list, we break out of the loop and return false because it's not a valid plaintext, therefore it's not a valid key. We try the same thing for all the possible keys we found. If we find a key that works, and the decryption of the ciphertext using that key also makes sense, we output that plaintext and we're done. If it fails, we keep trying for all the keys which had a pattern. If we reach the end of that loop, and no key works, we output Plaintext not found.

Formal Explanation

The first part of the code is to ask the user to input the ciphertext as a string. Each character of the string is then converted into the ASCII value and pushed into a vector of integers. After this is done, the program will read in the plaintext text file from the stream.

```
void readFileFromStream(ifstream& stream, vector<int>& text1, vector<int>&
text2, vector<int>& text3, vector<int>& text4, vector<int>& text5)
```

It will then identify each plaintext candidate by the number (1, 2, 3, 4, 5) and starting from 1, separate the ASCII values of the first 48 characters into their own vector. The reasoning of why we are using only 48 characters is written in the above section. After these five vectors are created, they are stored in *vectorPlaintexts*, which is a *<vector<vector<int>>*.

Then, we created a *diff* function. This function inputs two parameters - the ciphertext vector and a plaintext vector.

```
vector<int> diff(vector<int>& ciphertext, vector<int>& plaintext);
```

It finds the difference between the ASCII values of the ciphertext characters and the plaintext characters and stores that difference in a vector. This vector is then returned.

The next function (*ptncheck* - short for patterncheck) is the most important part of the decryption. This is because it looks for a pattern within the vector of differences that was returned from the *diff* function.

```
bool ptnCheck(vector<int>& difference);
```

First, a boolean value is initialized to false. An integer *curr* is initialized to zero. Then a loop is created that runs only when *curr* is less than 24, which is the max key length. There is then another loop within the first one that has another integer *start* initialized to zero and an integer *beginCycle* is initialized to the value of curr. In this loop, we use an if function to see if the vector of differences' value at 'start' is the same as the value at 'curr'. If it is, then we set the

bool to true and increment both start and curr to see if the next value is also the same. This loop ends when either curr hits 47, or the vector value at *start* is different from the value at *curr*. This confirms whether or not the cycle continues. If the vector value at *start* is different from the vector value at *curr*, we break out of the inner loop. In both cases, we will break out of the inner loop. If the bool is true, then we break out of the outer loop and return true as the cycle has not been broken. If the bool is false, then we set *curr* to what it was before the inner loop began, set *start* back to 0, and continue with the outer loop again to see if the cycle starts later. If the *bool* is false at the end of the outer loop, then it is simply returned as we have not found a cycle within the first 48 shifts.

In the *main* function, there is a for loop `for (size_t i = 0; i < vectorPlaintexts.size(); i++)`. The loop first uses the *diff* function on the first vector in *vectorPlaintexts*. The *ptncheck* function then uses the vector that is returned from the *diff* function. It will run over all the vectors in *vectorPlaintexts* until one of them returns true which means a pattern is found. When this pattern is found, it is recorded at which index of the loop it happens. Then we go through the plaintext file again to find the candidate plaintext that corresponds with the recorded integer. After this, the function is stopped.

If the *ptncheck* function never returns true, then we move on to test two.

Test two is similar to test one in that it compares the difference of the ciphertext and the plaintext and finds the key shift using that.

Because the plaintext options for test two is any permutation of the forty words given including repeats, we created a function called *diffCombos*. *diffCombos* is a recursive function.

```
void diffCombos(const string str[], string prefix, const int n, const int
len, vector<vector<int>>& ret);
```

In our main function, we have declared that ‘str’ is a string array that contains the 40 words given to us by the test2 dictionary. The *diffCombos* function checks if the integer *len* (which is initialized to 3), is equal to 1 in its base case. If it does, then it goes through a for loop of `(int j = 0; j < n; j++)`. *N* is a variable that was initialized to 10. Then for each string

str[j], we call another function, *encode*. This function takes in a *vector<int>* and a *string*. It goes through the string and changes the vector to contain the ASCII values of each character in the string.

```
void encode(vector<int>& v, string word);
```

Back in the *diffCombos* function, this ASCII value vector is pushed back into ret, which is our *vector<vector<int>>*. If the integer *len* is not equal to 1, then we go through another for loop of `(int i = 0; i < n; i++)`. In this for loop, we call the *diffCombos* recursively.

```
diffCombos(str, prefix + str[i], n, len - 1, ret);
```

This will one by one add all the words from 'str' until len equals 1. The result of the function is to alter the inputted vector, *possCipher*, to contain three words per permutation, repeats included, out of all the forty words in the second dictionary.

The second half of test two uses a for loop `(size_t index = 0; index < possCipher.size(); index++)` that runs through the size of the *possCipher*, the *vector<vector<int>>* altered in the first half. If the size of the [index] value of possCipher is greater than 24, then we call the same *diff* function from test 1 with the parameters cipherTest and *possCipher*[index]. We then use *keyExists* on the vector returned from the *diff* function. *keyExists* is similar to *ptncheck* from test one in that it checks if it is possible that a cycle exists. However, it returns a *vector<int>* instead of a *bool*. This vector is the potential key shift pattern. It will be empty if no such key shift pattern exists. If a pattern exists, the *decrypt* function is called on it.

```
string decrypt(vector<int>& cipher, vector<int>& shifts, string words[]);
```

The *decrypt* function as per its name, decrypts the cipher using the potential key shift pattern. It uses the parameters cipherText, the potential key shift pattern, and the string array str. *decrypt* loops through the cipherText and the shifts to subtract the shift from each character in the cipherText to result in a potential plaintext character. It loops through the entire cipherText and returns a *string* of the decrypted message, whether or not it's the correct one.

The string returned by the *decrypt* function is stored in the string *answer*. The function *correctPlainText* is then called. It takes in the answer string as well as the string array str.

```
bool correctPlainText(string& answer, string str[]);
```

Inside, there is a bool that is initialized to *true*. This function loops through all of the words in answer `for (size_t i = 0; i < answer.size(); i++)` and checks if it is in the string array. If it is, then the *bool* stays true and we check if the word after the first one is in the array. If it isn't, then the *bool* is changed to false and we break out of the for loop. When the for loop finishes or is broken out of, the *bool* value is returned.

Although we dealt with all of the possible word permutations that have a length equal to or greater than 24, we needed to acknowledge the possible combinations smaller than that (e.g. "irony cadgy beheld"). We handled this in the else statement from the initial for loop function. For these 4234 possibilities, we simply add another word to the end of the combination so it could be at least 24 characters long. We create a new *vector<int> newWord* to store each different possibility of four words. We loop through our *str* array of 40 words and at the beginning of each set *newWord* to be equal to the three word combination we're at in the bigger loop. We then call *encode* on *newWord* and the current word we're at to include the ASCII codes for all four words. After including the fourth word we proceed the same way we would with combinations that are greater than 24 characters to begin with. We call `diff(cipherText, newWord)`, storing the result in *asciiDiff*, and then call `keyExists(asciiDiff)` on that result. Finally we call *decrypt* with the potential keys we get from *keyExists* and then *correctPlainText* to get our *bool* confirming whether or not it is a proper plaintext.Back in the *main* function, we check if the bool returned by *correctPlainText* is true for both cases. If so, then we will output the answer string, which is the correct decrypted plaintext, and stop the program. If the bool is false, then we output "PlainText Not found. Sorry Professor :(\n".

Extra credit 1:

Substitution and Permutation are common techniques used for encrypting. The substitution cipher works by replacing the plaintext characters with different characters that could be letters, symbols, or numbers. A fixed system or key is followed to encrypt the

ciphertext using substitution. Some examples of substitution ciphers that we have seen in class include monoalphabetic substitution cipher and the polyalphabetic substitution cipher.

The key for the monoalphabetic substitution cipher maps each letter of the alphabet to another letter. For instance, in our key, we can map ‘a’ to ‘h’. This would mean that all a’s in our plaintext would be encrypted to be h. For the decryption to be possible, the key should follow a one-to-one mapping property. This means that if we assign ‘a’ to be ‘h’, we can’t assign any other letter to be ‘h’. Due to this, the key space is 26!, which makes using brute force extremely hard or infeasible. However, even though it has a large key space, it is easy to break and isn’t very secure. One approach to break it would be to compare the ciphertext to the frequency distribution of letters in the English language (or whichever language the plaintext is in). This technique has a high probability of working even if the plaintext is short because the distribution will very likely follow the average. For instance, the average distribution of the English language says ‘e’ is the most used letter in the language, therefore, we can guess that the most frequent letter in the ciphertext must be ‘e’. By following the frequency table and simple facts we already know about the language, like ‘h’ appears between ‘t’ and ‘e’, we can guess enough of the letters to quickly decrypt the message. Even though some of the guesses may be wrong, enough will be right to help decrypt the message.

The Vigenere cipher, or the polyalphabetic substitution cipher, is a more secure version of the monoalphabetic cipher. The monoalphabetic cipher was easy to break, because each letter was mapped to another fixed letter. The polyalphabetic cipher, instead of mapping the key to a single character, maps the key to blocks of characters in the plaintext. The key is a string of letters where encryption is done by shifting each plaintext character by the amount indicated by the key. The key is repeated in a sequence until the end of the plaintext, For example, if you select the key to be “potato”, and you are trying to encrypt the plaintext “spudsaretasty”, the ciphertext you will get is “ieoemphtnbmio”. That is because p is the 16th alphabet, so we shift the first letter of our plaintext, ‘s’ by 16 units which yields ‘i’. We do the same thing for the rest of the plaintext, where ‘p’ gets shifted by ‘o’ units, ‘u’ by ‘t’ units, ‘d’ by ‘a’ units, ‘s’ by ‘t’ units, ‘a’ by ‘o’ units, and once we reach the last letter of potato, we start again with p so ‘r’ gets shifted by ‘p’ units and so on. The same key keeps getting used until you reach the end of the

plaintext.You can make the Vigenere cipher more secure by making the key length longer. The attack used for a monoalphabetic cipher will not work for a polyalphabetic cipher since one letter can be assigned to a different letter each time, therefore we can't rely on it.

However there are various different ways to break the Vigenere cipher; one of these methods include the use of Kasisks's method. This specific method sees small patterns in the ciphertext, and assumes that the patterns are not coincidental. With this assumption, small 2 or 3 character patterns--like "lee" or "ao"-- that appear in different parts of the cipher can be assumed to be the same word encrypted using the same parts of the key. The number of characters between the two patterns has to be a multiple of the period. Dividing this number by it's factors would then give the possible lengths of the key. Getting the length of the key is very important when breaking the Vigenere cipher because then a smaller subset of the cipher can be looked at to calculate the possible shifts of each character, just like the monoalphabetic cipher. Unlike the monoalphabetic cipher the shifts are not necessarily the same shift, but different combinations of shifts can be calculated until the ciphertext makes sense. Even if all the combinations of shifts are not found, by studying the frequency of the characters in the key positions, valid assumptions can be made about the letter. Like the most probable letter is "e" (if it's valid english). This approach would also be a lot faster than finding all the combinations.