New York University
Applied Cryptography CS-GY-6903

Project 1

CRYPTANALYSIS USING FULLY AND PARTIALLY KNOWN PLAIN TEXT ATTACK

March 15, 2021

Team Members:

Abhijit Chitnis *(aac664)*
William Dockery *(wbd220)*
Jigyasa *(jn1888)*
Joseph Gumke *(jdg597)*
Michael Leking *(ml6522)*
Umesh Tiwari *(ukt202)*

Table of Contents

Executive Summary

This cryptanalysis project consists of a software implementation of an algorithm that tries to decrypt an L-symbol challenge ciphertext computed using a specific cipher. Our program's goal is to find the plaintext used to compute this ciphertext within a reasonable amount of time.

Introduction

Per the project instructions, our team successfully designed an algorithm used to decrypt ciphertext in order to compute the associated English plaintext. We thought this project was a valuable learning opportunity and appreciate how the requirements aligned to the course material. We worked collaboratively in all aspects of this project through individual research and team discussions through zoom video calls. Various zoom were stood up to collaboratively design and test the program that effectively and efficiently processed plaintext in order to compute the associated ciphertext. Given the preference of the team, we opted to design our algorithm using Python as it affords a variety of tools and capabilities that we leveraged to implement our code. We were assigned Project 1 on February 5th, 2021 and submitted it on March 12th 2021.

Team Member Contributions

Given our team consisted of six team members, here we briefly each member's contributions to this project in this section. The team greatly collaborated throughout many pair programming events on zoom. Each team member help expand upon ideas and greatly improve the logic of the overall product.

Abhijit Chitnis contributed significantly and equally to the project team effort by independent research, thought leadership as he relayed his ideas through a team collaboration chat channel we had established on Signal. He participated in multiple planned and unplanned zoom video calls where he helped to write python code logic, helped to troubleshoot coding errors, and provided solution design logic, which includes various frequency analysis ideas. Abhijit has taken discrete mathematics in summer semester, and his insights were uniquely helpful to the team's efforts. Abhijit worked long hours into late nights to come up with new and innovative logic ideas for us to consider as a team.

William Dockery contributed significantly and equally to the project team effort by independent research, thought leadership as he relayed his ideas through a team collaboration chat channel we had established on Signal. He participated in multiple planned as well as unplanned zoom video calls where he helped to write python code logic, helped to troubleshoot coding errors and provided solution design logic, including the bad bucket ciphertext clean-up efforts. Bill has taken discrete mathematics in summer semester, and his insights were uniquely helpful to the team's efforts. Bill's dedication was exemplified with his

active participation in team calls even when he was battling COVID illness or when he spent several hours working together to design program logic on his wife's birthday.

Jigyasa "Jigs" contributed significantly and equally to the project team effort by independent research, thought leadership as she relayed her ideas through a team collaboration chat channel we had established on Signal. She participated in multiple planned and unplanned zoom video calls where she helped to write python code logic, helped to troubleshoot coding errors and provided solution design logic, including frequency analysis ideas. Jigs has significant coding experience in Java and C++ and she has specially done some work in the area of frequency analysis, so Jigs led the effort in coding this portion of the logic. Jigs setup Github channel for the project and enabled code and document sharing and collaborative development.

Joe Gumke led our team's python coding efforts, he invested countless hours investigating the coding logic, coding and testing the program. Our entire team would spend hours and hours on zoom video calls as we would work together with Joe to come up with design ideas and investigate options, code and test our hypotheses. Joe is our expert python coder; he also has the unique ability to find helpful research material that he generously shared with the team. Joe contributed significantly and equally to the project team effort by independent research, thought leadership as he relayed his ideas through a team collaboration chat channel we had established on Signal, as well as by participating in multiple planned and unplanned zoom video calls where he helped to write python code logic, helped to troubleshoot coding errors and provided solution design logic, including the main program logic.

Michael Leking was our Internet research guru, inventor of find the right material genre, he lead out teams wordsmith department, and documentation template pioneer. Mike led the documentation effort and contributed significantly and equally to the project team effort by independent research, thought leadership, and collaborated and interviewed all team members to consolidate project details. He relayed his ideas through a team collaboration chat channel we had established on Signal, as well as by participating in multiple adhoc sessions planned and unplanned zoom video calls where he helped to write python code logic, helped to troubleshoot coding errors and provided solution design logic, including the main program logic. Mike had his third child during the second week of this semester, and was extremely committed to this project just like every member of the team.

Umesh Tiwari contributed significantly and equally to the project team effort by independent research, thought leadership as he relayed his ideas through a team collaboration chat channel we had established on Signal, as well as by participating in multiple planned as well as unplanned zoom video calls where he helped to write python code logic, helped to troubleshoot coding errors and provided solution design logic, including the main program logic. Umesh invested significant efforts in program logic design, contributed towards the idea of key length logic and random symbol separation logic framing and countless hours working together in collaborative code design, debugging, analysis and documentation efforts.

Methodology

Project Methodology

Our project consisted of multiple phases.

Phase 1: Encryption Function – designed an encryption algorithm to generate ciphertext for testing purposes. Since the ciphertext was not provided, our team generated test ciphertext to assess the efficiency and correctness of our decryption function.

Phase 2: Cryptanalysis – In this phase, we brainstormed the requirements for our decryption function and developed a strategy for determining the encryption shift key schedule, key length size, and character frequency analysis to determine the plaintext message text associated with the ciphertext. We performed extensive research in this phase and considered plaintext based attacks and other brute force options against the Vigenere cipher.

Phase 3: Decryption Function – This phase consisted of an iterative and collaborative approach to developing an accurate decryption function for the input ciphertext. We deployed code which used a variety of variables; the key space, message space, and ciphertext to correlate the ciphertext to the associated plaintext options in our dictionary file. We spent the bulk of our time in this phase and performed extensive testing which led to a more effective algorithm.

***Phase 4*: Interpreting Results** – Our team reviewed the results after every iteration of the decryption function. We assessed errors and fined tuned the logic while further investigating alternative strategies to more accurately derive decryption solutions. We continually refined the steps above in order to generate the most accurate association between ciphertext and corresponding plaintext.

***Phase 5*: Continuous Improvement** – Continue to refine the performance of the decryption function. After arriving at a reasonable solution, the team invested additional time and resources to increase the level of accuracy of the algorithm from roughly 65% to roughly 95%. We attempted the extra credit requirements during our continuous improvement process.

***Phase 6*: Report Development and Submission** – Reviewed project tracker/journal and incorporated contributions from all team members into one cohesive report. Our team meet multiple times per week so consolidating data into one report was challenging. In this phase, we performed quality assurance and ensured requirements were met. Submitted package and report.

Execution

Assembled team of six professionals where our experience provided productive and complementing input to the overall project. We met multiple times per week via Zoom and constantly communicated on Signal. We performed extensive research through various Github repositories, books, subscriptions, java/python modules, and other active research.

Encryption Function

Since the ciphertext was not provided to us, we created an encryption function, a.k.a., the "encryptor", to generate ciphertext for testing purposes. The ciphertext was created using a randomly generated key of length 1-24 made up of <space>, a-z. We needed to create ciphertext that we controlled to assess the efficiency and correctness of our decryption function. An important feature of the encryptor was the scheduler – logic designed to embed bogus characters into the ciphertext (i mod t + 1), (I mod t + c), so we could test our decryption strategy. The encryptor created parameters for our controlled tests; size, key length, schedule of random characters, etc. which we could use to validate whether our decryption function worked correctly.

Upon further analysis and conference with the TA and Professor, we revised our scheduler function to introduce more complexity into the ciphertext. Based on the Professor's feedback, we created five different schedulers; at every encryption run, the encryptor randomly chooses one of them with the random key to encrypt plain text. These are the schedulers encoded into the program:

1) i % t + k
2) i % t - k
3) (2*i + 3) % t + k
4) (3*i - 4) % t - k
5) 2*i + k % t

Where k is a configurable constant to change the percentage of random chars introduced to further contaminate the cipher text

Frequency Analysis and Index of Coincidences

In order to find the key length, we used the ciphertext as a string which we augmented in order to find potential patterns. We populated an array with the ciphertext where we performed a right-shift. An example of the right shift can be seen below using notional ciphertext:

Ciphertext										
C	**E**	**W**	**W**	**Y**	**E**	**E**	**O**	**L**	**C**	**Y**
	C	E	W	W	Y	E	E	O	L	C
		C	E	W	W	Y	E	E	O	L
			C	E	W	W	Y	E	E	O
				C	E	W	W	Y	E	E
					C	E	W	W	Y	E
						C	E	W	W	Y
							C	E	W	W
								C	E	W
									C	E
										C

Figure: 1, Identifying Index of Coincidences

We populated an array with the ciphertext in order to identify coincidences. Coincidences are identified by performing a comparison between the original ciphertext (top row) to each of the augmented ciphertexts in the rows below it. In the example below, coincidences are highlighted and the total number of coincidences identified are tallied in the right most column.

Ciphertext											Coincidences
C	**E**	**W**	**W**	**Y**	**E**	**E**	**O**	**L**	**W**	**Y**	
	C	E	W	W	Y	E	E	O	L	W	2
		C	E	W	W	Y	E	E	O	L	
			C	E	W	W	Y	E	E	O	
				C	E	W	W	Y	E	E	1
					C	E	W	W	Y	E	1
						C	E	W	W	Y	2
							C	E	W	W	1
								C	E	W	
									C	E	1
										C	

Large coincidences occurred 5 rows apart.

Figure: 2, Index of Coincidences and Count

Given that the ciphertext length in the example above was short, there were not many coincidences identified, but the goal was to identify the number of coincidences within the ciphertext. Once the number of coincidences have been identified, we looked for the numbers that were larger than the rest. In the example above, 2 coincidences occurred twice.

Once we've identified the larger coincidences, we calculate how often they occurred. Again, using the example above, the larger coincidences (2) occurred five rows apart. Therefore, we can conclude that the key used to generate this ciphertext was a length of 5. The number of coincidences in this example were small, but the number of coincidences identified in a 500 length ciphertext was much larger.

Once we know the length of the key (t), we calculated the frequency of the characters in every t'th location throughout the ciphertext. The ciphertext we receive will include random characters, so we need to leverage the Index of Coincidence (IoC) information from above and use this to derive clean ciphertext by removing the random characters from the provided ciphertext string.

Initial Approach

As mentioned in the Execution section above, we knew extensive analysis of the ciphertext was required in order to identify the key length and frequency of characters to correlate the ciphertext with the plaintext. Given we had knowledge of the five plaintext messages, we leveraged a methodology consistent with a chosen plaintext attack as the ciphertext was not available. As such, we did not have access to the key, so we created a decryption algorithm to correlate input ciphertext with one of the plaintext message options.

We used the strategy discussed above to determine and assess the number of coincidences within the ciphertext and created an Index of Coincidence (IoC) value. We analyzed the maximum values (high peaks) of coincidences using a variety of different methods; differences of max pairs of coincidences, greatest common divisor within coincidences, and average of all coincidences.

We iterated through each method in order to find the most efficient function. Efficiency was measured in performance cycles but we were primarily focused on how accurate our algorithm was in associating the ciphertext to plaintext. Using the Index of Coincidence method above, we were able to more effectively determine the initial key length through trial and error. In some of our preliminary attempts, we found the correct key length with roughly ~35% certainty. This was achieved by taking the two initial maximum highest matching values found (under 24 characters, and greater than 6 characters), matching pair occurrences and getting the difference in length between them. Due to the low confidence in finding a key, we found a better approach in identifying a variance explained below.

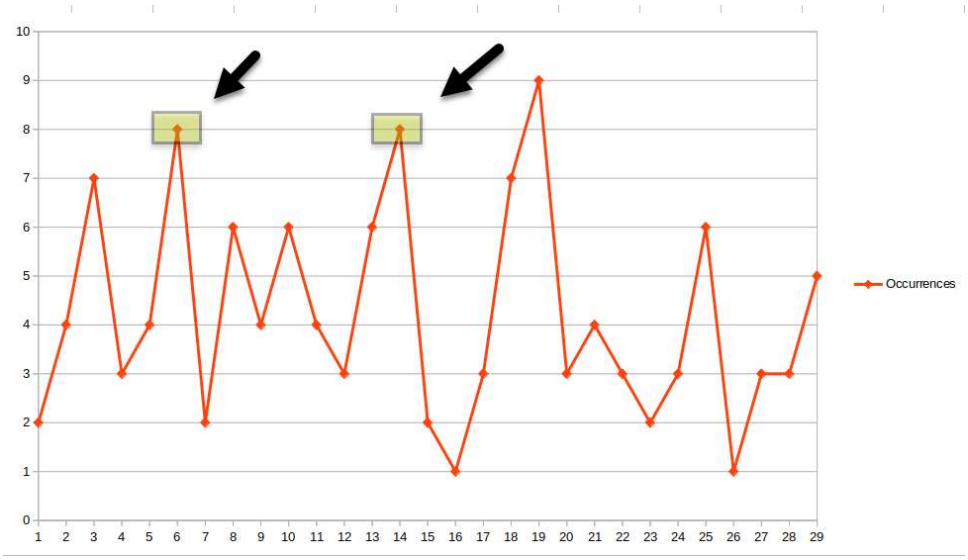

Figure 3: Initial Max occurrences

Over time, we increased our certainty to around ~50% by performing a differential of the variance in multiple high peaks of occurrences using the “peakutils” library within Python. Instead of taking the highest statically found set of pairs, we extended this idea to finding a variance in high peaks found in the occurrences. This was identified to be more confident due to the randomization incorporated into the ciphertext given to us by the encryptor.

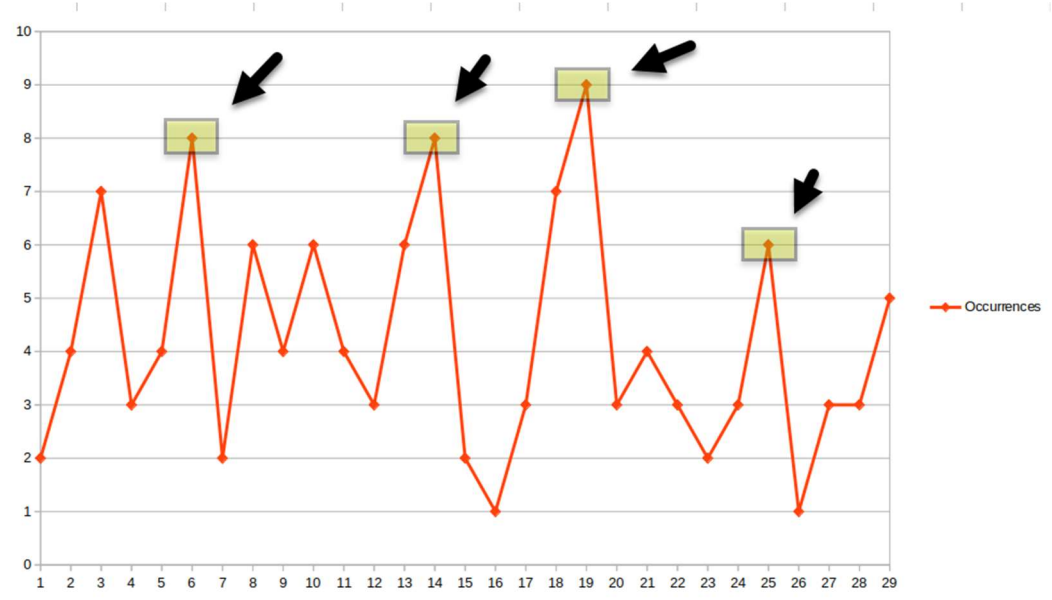

Figure 4: Final Occurrences Solution

Again, through trial and error, we were able to obtain roughly ~85% certainty of key length by incorporating a different and more configurable python library entitled: “SciPy”. We primarily leveraged a more secure certainty configuration (as shown by SciPy statistics) and no primary

key length is found, then we failover to a looser peakutils configuration to find the key length. If any key lengths are found we iterate through a sorted/unique known/identified key lengths array lower numbered to higher attempts. If we don't find any known key lengths, we migrate to a different scipy peakutils configuration and attempt again.

Figure 5, below, shows a preliminary attempt at analyzing ciphertext with the goal of finding the key space, key length, and eventually decrypting the message in order to correlate the ciphertext with one of the five plaintext message options in the dictionary.

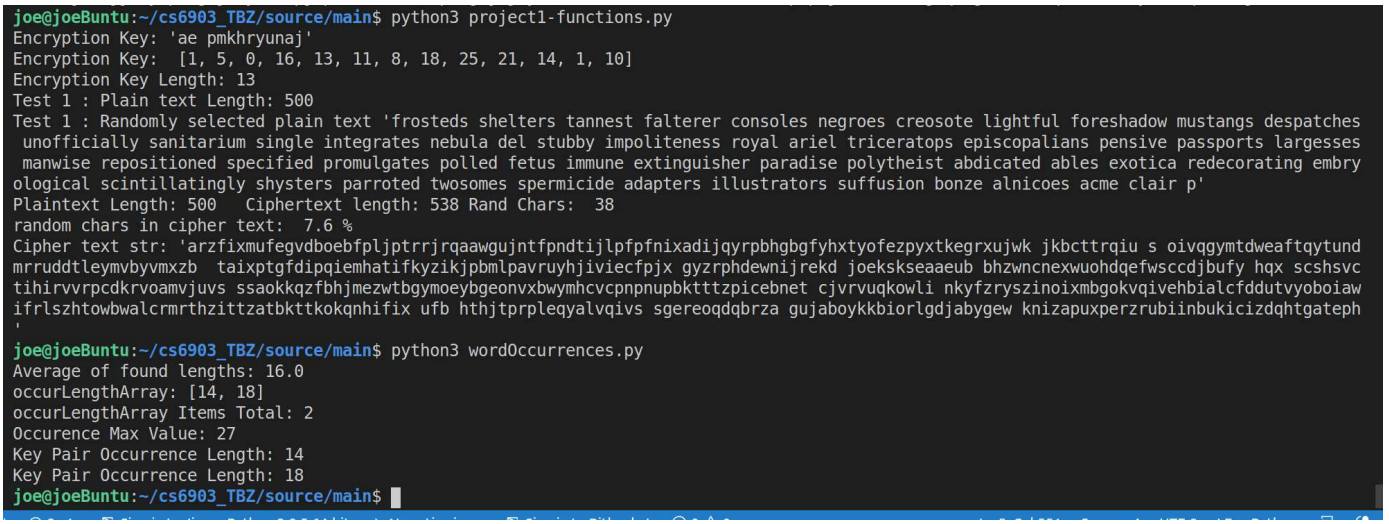

Figure 5, Output of Initial Approach

Revised Approach and Explanation

The rationale and strategy for our decryption function was mentioned in the Execution and Initial Approach sections above. We worked hard to improve our initial approach; with a focus on greater efficiency and accuracy of the decryption function to process the Index of Coincidence (IoC) values and efficiently associate the ciphertext to the correct plaintext message.

Through extensive research, we knew the IoC approach had been effective by other cryptologists, therefore we invested significant time researching and improving our initial strategies. We found the most success in identifying the correct key length by passing our IoC array values into the scipy peakutils library, which translates this into a wavelength and leverages the peak_utils algorithm to find the highest variances, or peaks/highest statistically found values of this wave. This peak detection algorithm provided to us within peak_utils library was critical for us to determine the proper variances in found IoC values to determine proper key length.

Calculating Initial Key Length:

We concluded that the frequency of repeating patterns in the ciphertext reveals information about the key length.

Here's an example of our rationale -

For example, if our ciphertext is a length of 550 characters, and we already know the length of the plaintext message should be 500 characters. The ciphertext and plaintext message should be equal because it is a shift cipher, so the difference in lengths (ciphertext and plaintext) reveals that 50 extra random characters were padded within the ciphertext.

550 (ciphertext) – 500 (message) = 50 (random padded characters)

Our team generated ciphertext using the encryptor program and attempted to identify repeating patterns within the ciphertext in order to help us estimate the key length (t). Our assumption was that based on the project 1 instructions, the key length would be in the range of 1-24. Per the assignment instructions, we also know there are random characters appended to the key length. In order to identify the key length (t), we also need to identity the random characters (x) added to the key as well. So, the key length with random characters (T) identified as a result of finding repeating patterns would be T= t+x. We used this initial key length in our statistical frequency analysis of our ciphertext characters to identify anomalous character positions as the random characters as opposed to the encrypted message characters.

We took the initial key length and leveraged the SciPy library which is a python library used for numerical integration and statistical analysis. We incorporated Index of Coincidence values into SciPy which helped us refine/enhance the initial key length guess. As seen in the table below, we adjusted numerical values as input to SciPy; height, distance, and prominence.

- Height indicates the peaks where coincidences in the ciphertext occur.
- Distance is the number of characters between peaks.
- Prominence is the difference between peaks and valleys of coincidences. Adjusted the prominence value would impact the peaks and would change the key length entirely.

As seen in Figure 6 below, we needed to adjust these values in order to produce a more accurate guess of the key length. We were able to modify these variables in our algorithm in order to test the accuracy of the results. Through extensive testing, we found these SciPy configurations, 17 (height), 4 (distance), 17 (prominence) produced the most accurate estimate of the key length.

By reducing the prominence value by 3 or 4, we found that the likelihood of a key increases when 17 (height), 4 (distance), 17 (prominence) fails.

height=18,distance=6,prominence=18	height=17,distance=6,prominence=16	height=17,distance=4,prominence=16	height=17,distance=4,prominence=17
13	9	14	17
3	6	5	1
4	5	1	2
20	20	20	20

Figure 6, Excerpt of SciPy inputs for Index of Coincidence Key Length Computation

Even though we created a guessed key length, we knew this was not the actual key length given there were random bogus characters included in the key. We leveraged the guessed key length from above and divided the ciphertext and plaintext into buckets based on the size of the guessed key length. We needed to perform a small computation in order to find our "adjusted key length".

Calculating Adjusted Key Length:

We knew the initial key length was replicated throughout the ciphertext message and we also knew there were padded bogus characters padded within the key. To get the percentage of padded bogus characters within the initial key length, we divided the number of padded characters over the ciphertext length. We subtracted the percentage of padded characters from the initial key length in order to account/remove the padded characters within the initial key length in an attempt to create an adjusted (and accurate) key length.

Per the project requirements and through conversations with the TA and Professor, we knew the keys would contain padded bogus characters. Given that, we knew we needed to account for the random characters and work to identify and remove them from the key lengths. Once we've calculated the adjusted key length, we leverage the Index of Coincidence values obtained from above.

Calculate the number of potential random chars per key length

$$randChar_{cnt} = \frac{(cipherTextLen - 500) * guessedKeyLen}{cipherTextLen}$$

$$adjustedKey_{len} = guessedKey_{len} - randChar_{cnt}$$

Index of Coincidence and Adjusted Key Length Analysis:

At this point, our decryption function algorithm now has the adjusted key length. We can now take the IoC value and correlate that to the adjusted key length. In order to do so, we divided the ciphertext into buckets based on the adjusted key length. Now that the ciphertext is "bucketized", we can simply perform a mono-alphabetic shift on each bucket. Some/most of the buckets contain the padded random characters so we attempted to remove those random characters. Next task was to identify the "buckets" that contained the dummy random

characters added to confuse the adversary. Based on the choice of the scheduler, we realize that random characters may be interspersed anywhere within each key length, not just at the beginning or the end. Here, we use the Index of Coincidence values to weed out the bad buckets. Based on the guessed and adjusted key lengths we already know the random characters introduced by the scheduler per key length. We identify those number of buckets where the IoC does not match the expected IoC value and eliminate them from the cipher text. This essentially gives us what may be termed as "clean cipher text" i.e., cipher text without random chars such that length of the plain text matches length of the cipher text.

Mono-alphabetic Shifts and chi-squared analysis:

At this point, each of our cipher text buckets are assumed to have been shifted by the same key character i.e., char of the key at the same index. Essentially each bucket is a mono-alphabetically shifted cipher (like the Caesar cipher) which can be easily decrypted. However, we do realize that each cipher bucket may be shifted independently by any one of the key characters based on the choice of the scheduler. Next step was to shift each cipher bucket by a single char, compute the chi-squared value which signifies the closeness of one distribution (observed) with respect to another (expected). In this case each cipher bucket represents the observed distribution whereas the expected distribution of letters from the combined dictionary files. Chi-squared was computed as the cumulative ratio of the square of the difference of each letter frequency (observed – expected^2) and the expected frequency. The lowest chi-squared value across all shifts for each bucket would indicate the amount of shift needed to shift the bucket by. Once all cipher buckets shifts were completed and the lowest chi-squared for each was chosen the "encryption key" emerged. This was used to convert cipher text buckets to their corresponding plain text buckets. Each bucket represents a key index position containing characters a single key length apart. The plain text buckets were reconstituted to create the entire decrypted plain text in the expected order.

Plaintext Fuzzing Function:

We realize that the process of key length estimation to determining the random characters and computing the correct shift may have a certain percentage of errors since we are using probabilistic methods. This is expected to result in plain text decrypted words that may be close to the original plain text message but not be exactly be the same. In order to overcome this issue, we used the fuzzywuzzy external python library. We created a ***plaintext word fuzzer*** which we used to help interpret plaintext words that were not fully decrypted within the message. Although the goal of the decryption process is to decrypt the ciphertext as accurately as possible, that is not always possible. As such, our associated plaintext in our preliminary attempts did contain some words that were not entirely decrypted – that's where the fuzzer comes into play. The fuzzer scanned for incomplete words and attempted to guess the actual word from the partial decrypted word. We thought this would help increase the effectiveness of our strategy and reduce the margin of error during the decryption process but we were not able to fully integrate the fuzzer into decryption function.

Explanation of Test 1:

Given we have access to the plaintext message options, Test 1 is basically a known plaintext attack. Our methodology and logic are explained in detail above, but the key activities pertaining to Test 1 are outlined, below. It should be noted that the steps outlined below successfully decrypted the input ciphertext, however, we opted to use Test 2 for both tests as it was more accurate and efficient.

- We generated a random key from 1-24 characters using <space + a-z.
- We randomly selected one of the five plaintext messages (500 characters) from the dictionary file and encrypted the string using the key generated by our encryptor.
- The encryptor scheduler injected random characters into the ciphertext which created a ciphertext that is longer than the original plaintext due to the inserted random characters.
- We leveraged the maximum coincidences technique based on the ciphertext only to find peaks and their distances to guess the most likely key length.
- In order to find the adjusted key length, we had to eliminate the random characters introduced based on the length of the ciphertext and plaintext.
- The ciphertext was chunked into buckets based on guessed key length which includes the random characters introduced that have no correspondence with plaintext characters.
- We then bucketized each of the five known plaintext strings based on adjusted key length
- Attempted to detect bad cipher text buckets either at the beginning or the end of the guessedKey interval. Bad buckets are identified by comparing their IoC values. Lower IoC loses to the higher IoC. All English language IoC's are > .06864. Bad buckets are weeded out from the input cipher text
- Test 1 guesses the encrypted plaintext (option 1 of 5) by computing the deltaIoC (differential index of coincidences) between ciphertext and each of the five plain text strings. Lowest differential (deltaIoC wins).
- Using frequency distribution identified from the plaintext dictionary file, we were able to identify what the shift was for each bucket. We looked at each bucket and then recreated the associated text – which would now essentially yield the key used to generate the input ciphertext.

Explanation of Test 2:

Once we completed Test 1, our attention shifted to the Test 2 requirements. We found that by refining our initial Test 2 strategy, we could use Test 2 for both Test 1 and Test 2 as it was more efficient. Test 2 uses the chi-squared strategy outlined above for all input ciphertext. We are able to compare against the modular frequency distribution to get results.

Our approach for Test 2 accounted for the fact that we did not have access to the plaintext message and could not perform a chosen plaintext attack. The test 2 messages expected to consist of randomly chosen words from a dictionary file containing 40 plaintext words. In order to make this program more generic and also cover test 1, we combined words from both dictionary files into one combined dictionary consisting of approximately 400 plaintext words.

We ran the frequency distribution statistics and evaluated the results using the word fuzzer to successfully decrypt nearly 100% of the ciphertext. Our logic is described below:

- We generated a random key from 1-24 characters using <space + a-z.
- We randomly selected words from dictionary 2 to generate a 500 character string which was then encrypted using the key generated by our encryptor.
- The encryptor scheduler injected random characters into the ciphertext which created a ciphertext that is longer than the original plaintext due to the inserted random characters.
- We leveraged the maximum coincidences technique based on the ciphertext only to find peaks and their distances to guess the most likely key length.
- In order to find the adjusted key length, we had to eliminate the random characters introduced based on the length of the ciphertext and plaintext.
- The ciphertext was chunked into buckets based on guessed key length which includes the random characters introduced that have no correspondence with plaintext characters.
- Ttest 2 uses the chi-squared method by shifting each good cipher text bucket by 0 (space) followed by a (1) through z (26).
- Chi-squared is computed for each shifted cipher string. Chi-square is the measure of closeness of the shifted frequency distribution to the expected frequency distribution. Each cipher text bucket is a mono-alphabetic shift assuming the key and random characters repeat with a fixed period. Any single cipher bucket contains letters from the encrypted string separated by a distance of key length t from each other.
- The lowest chi-squared value (amongst 27 shifted strings) and its corresponding shift win as they indicate the smallest differential from the expected frequency distribution computed from dict 2 words. These optimally shifted cipher text buckets represent corresponding plain text buckets.
- Plain text buckets are reconstituted to form the (expected) decrypted plain text string.
- Since the chi-squared is the best or optimal guess, it may not always correctly identify the shift for one or more buckets. To mitigate this noise, a fuzzy search library (plaintext

word fuzzer outlined above) is used to compare the detected words against the dictionary 2 dictionary to find the best match.

- Concatenation of fuzzed words from dictionary 2 file is output as the final decrypted plain text.
- Shifts per bucket (from 1 to t) gives us the potential key used for encryption.

On tests involving 400 words, we only marginally sacrificed computing runtime when using the fuzzer to assist with word decryption. If we run our decryptor without the fuzzer, the runtime is between 20 - 30 m.s. However, if we run the fuzzer on words not fully decrypted, the runtime jumps to ~600 m.s. (about 20 times higher) but is still well under a second - this is for 400 words.

Pseudo-code

Encryptor(test_num)

```
// test num passed in dictates the dictionary file used to select or
// randomize the input string to be encrypted
if test 1
        // 5 strings ~500 chars each
        select a random string from dictionary file 1
if test 2
        // 40 words – expanded to 400 words – randomized & concatenated
        generate a string by concatenating random words from dict file 2
generate a random key of size t // 1 – 24 chars [<space>, a .. z]
select a random scheduler // i % t + 1, i % t – 1 etc.
while (chars in input string)
        key_index = input_string index % t + 1
        if key_index is in key range 0 to t-1
                        // left shift plain text char by key char at index key_index
                        append (input char – key char) to cipher-text
                        increment input string index
        else
                        // append random char to cipher text
                        // do not increment input string index
                        append random char from [<space>, a ..z] to cipher-text
return cipher-text
```

Decryptor(cipher-text)

// use maximum coincidences method using cipher shift technique

Guess Key length

populate multi-dim array with cipher-text

right shift cipher-text and count matches at every index

Store frequency of matching occurrences

Use scipy lib to find peaks based on amplitude, period and prominence

Use scipy lib to find multi-modal peaks (possible key lengths)

Max occurring key length is selected as the guessed key length

// each cipher-string is assumed to be a mono-alphabetic shift

// t is the key length

Create a cipher list of t strings from cipher-text t chars apart

string-1 = cipher-text chars at index [0, t, 2t, 3t…]

string-2 = cipher-text chars at index [1, t+1, 2t+1, 3t+1…]

…

string-t = cipher-text chars at index [t-1, 2t-1, 3t-1…]

Create Index of Coincidences for each cipher string (string1 – string t)

// index of coincidences boils down the letter frequency distribution in a

// string to be used for language detection (English distribution)

$$cipherStringIoC = \sum_{i=0}^{26} \left(\frac{cipherStringChar_{cnt}(i)}{cipherString_{len}} \right)^2$$

Calculate the number of potential random chars per key length

$$randChar_{cnt} = \frac{(cipherText_{len} - 500) * guessedKey_{len}}{cipherText_{len}}$$

$$adjustedKey_{len} = guessedKey_{len} - randChar_{cnt}$$

// detect random char positions in cipher text

// Mark <randCharCnt> cipher-list strings with the lowest IoC's as

// potentially random indices to be eliminated

badIndices = list of indices with rand chars in each key length

Compute chi-squared to detect expected shift (key char)

// chi-squared is the measure of overlap between 2 distributions

// typically observed and expected

// chi-squared computes the deviation between the shifted char set

// for every shift and the expected char set (dict)

// the lowest chi-squared value wins (expected shift for that index)

For each mono-alphabetically shifted cipher-string

Ignore strings of random chars (detecetd above)

Rotate each char by a single char

Calculate the shifted cipher-string's chi-squared value

$$chi - squared = \sum_{i=0}^{26} \frac{\left(shiftedCipher_{cnt}(i) - dictionaryC_{cnt}(i)\right)^2}{dictChar(i)}$$

Maintain the min chi-squared & corresponding shifted string

guessedKey[i] = shift char with min chi-squared

// result of decryption based on guessed key & random char positions

decrypted string = Reconstitute plain text from min-chi squared shifts

// result of fuzzing decrypted plain text string with 400 word dictionary

// imported fuzzyWuzzy external lib

final decrypted string = fuzzyWuzzy(decrypted string)

return plain-text

Conclusion

Our team thoroughly enjoyed the design of this project. We found it challenging, rewarding, and a useful tool for learning about the various components of ciphers and the cryptanalysis approaches to defeating ciphers; particularly Vigenere. Our team performed an extensive amount of research and implemented numerous iterations of our code (scheduler, decryption function, fuzzer) in order to break the cipher. Once we were able to decrypt the ciphertext, we focused on accuracy and efficiency of our decryption function. Upon submission of this project, we designed a function that successfully decrypts the ciphertext from our encryptor nearly 100% of the time (accuracy). We are able to achieve these impressive results in a runtime averaging only a few milliseconds (efficiency).

Through trial and error, we incrementally worked to enhance our code in order to overcome many challenges including identifying key length, removing random characters included in the ciphertext, and implementing a fuzzer to find proper word matches. We leverage techniques such as index of coincidences and the chi-squared method which proved to be essential in our cryptanalysis strategy. Every member of our team contributed to the success of this project.

Appendix A: Extra Credit (1 of 3), Test 1

Per the project requirements for extra credit pertaining to test 1, we increased the number of words in the plaintext_dictionary_test1 file to assess the performance of our decryption function using a larger subset of plaintext messages.

We avoided the size of dictionary because we only checked dictionary for corrections if we through we did not find the correct word. Based on guidance from professor in a designated meeting, professor provided guidance that 4,000 words was sufficient to be successful. Processing can extend up to ~40ms as the fuzzing can extend this, as the dictionary size grows.

```
Total time taken by program:
when dict size is 40, decryptor takes: 0.0565 milliseconds
when dict size is 400, decryptor takes: 0.3457 milliseconds
when dict size is 4000, decryptor takes: 0.0516 milliseconds
```

Figure 1: Processing Runtime Performance By Dictionary Size

Appendix B: Extra Credit (2 of 3), Test 2

Per the project requirements for extra credit pertaining to test 1, we increased the number of words in the word_dictionary_test2 file to assess the performance of our decryption function using a larger subset of words in this file.

On tests involving 400 words, we only marginally sacrificed computing runtime when using the fuzzer to assist with word decryption. If we run our decryptor without the fuzzer, the runtime is between 20 - 30 m.s. However, if we run the fuzzer on words not fully decrypted, the runtime jumps to ~600 m.s. (about 20 times higher) but is still well under a second - this is for 400 words.

On tests involving 4,000 words, the runtime is roughly 6 – 7 seconds. We still consider this successful given the near 100% accuracy of the decryption function.

Based on a conference call with the Professor on March 1st, we experimented with various recommended approach for extra credit including the complexity of the scheduling algorithm and the size of dictionary to include 4,000 words instead of 400.

Per the guidance from the Professor, the table below shows that the accuracy of decryption varies as a function of complexity. However, the runtime does not degrade by the random characters in the key.

random chars in cipher text:	Decryption Accuracy	chi-squared Decryptor Runtime
7.20%	0.00%	438.38 ms
6.80%	2.50%	553.56 ms
0.00%	6.12%	1118.01 ms
8.20%	11.32%	595.41 ms
0.00%	36.73%	786.89 ms
6.20%	39.22%	551.52 ms
5.60%	97.92%	252.33 ms
9.20%	97.92%	79.79 ms
9.00%	97.92%	99.69 ms
6.60%	97.96%	33.91 ms
8.20%	97.96%	73.76 ms
12.60%	98.00%	69.81 ms
20.00%	98.04%	124.13 ms
5.40%	98.04%	60.84 ms
11.20%	98.04%	65.82 ms
0.00%	98.04%	93.75 ms
0.00%	98.08%	111.7 ms
10.00%	100.00%	214.43 ms
7.80%	100.00%	31.92 ms

Appendix C: Extra Credit (3 of 3), Scheduler Complexity

Per the Professor's guidance, we tested various key scheduling options and measured the accuracy levels. As you can see in the table below, we gradually increased the complexity of the scheduling algorithm and the accuracy level changed as we experimented with the key schedules.

Randomly selected Scheduler:	Decryption Accuracy
(i % t) - 1	100.00%
((3 * i - 4) % t) - 1	100.00%
((3 * i - 4) % t) - 1	98.08%
(i % t) - 1	98.04%
(i % t) + 1	98.04%
((2 * i + 3) % t) + 1	98.04%
((3 * i - 4) % t) - 1	98.04%
(i % t) - 1	98.00%
(2 * i + 3) % t) + 1	97.96%
(i % t) + 1	97.96%
(i % t) - 1	97.92%
(i % t) - 1	97.92%
(i % t) + 1	97.92%
(2 * i + 3) % t) + 1	39.22%
(2 * i + 1) % t	36.73%
(2 * i + 3) % t) + 1	11.32%
(2 * i + 1) % t	6.12%
(i % t) - 1	2.50%
((2 * i)^2 % t) - 1	0.00%

Appendix D: Project 1 Program Output and Runtime Evidence

As mentioned, our team performed rigorous testing and modified our approach. Below is a summary of improvements made to our revised strategy:

- Root cause of most failures was incorrect "single" key length detection - in the vast majority of failures, correct key length was in the set of possible key lengths but wasn't necessarily the final guessed length - i'm using the unique set of all possible keys to decrypt - substantial improvement in accuracy with a negligible loss of efficiency
- *New logic iterates over all possible key lengths and selects the one with the highest decryption accuracy*
  (Decryption accuracy can be computed ONLY when the input plain text is known during testing)
- We now have 5 schedulers - mix of simple and complex (prof's suggestions included) - schedulers are selected randomly by the encryptor during every execution
- Bad bucket detection logic now iterates over all cipher buckets and selects the 'n' worst buckets where 'n' is the number of random chars detected per key length - we are now able to detect random chars inserted anywhere in the string, not just at either ends - this improved decryption accuracy substantially
- *Added another metric (for our testing only) to display the fuzzer accuracy - fuzzed output is compared to the plain text input - tells a great story as we are able to enhance the decrypted output from ~15 - 20% to ~80 - 100 % accuracy once our decryption is passed through the fuzzer. This metric will obviously not work when the prof. pushes his cipher texts, however, we could include it in the report*.
  (Fuzzer accuracy can be computed ONLY when the input plain text is known during testing)

This file contains the detailed results from our test/project including the excerpt provided below:

Complete test results included in the enclosed output file (*TBZ-Project1 Program Output.pdf*).

Example of test 1 output:

```
"C:\Users\achit\OneDrive\NYU MS Cybersecurity\10 - Applied
Cryptography\Project-1\cs6903_TBZ\venv\Scripts\python.exe"
"C:/Users/achit/OneDrive/NYU MS Cybersecurity/10 - Applied
Cryptography/Project-1/cs6903_TBZ/source/main/encryptor.py"
Encryption Key: 'nnctsuohklahargiytvy'
Encryption Key:  [14, 14, 3, 20, 19, 21, 15, 8, 11, 12, 1, 8, 1, 18, 7, 9,
25, 20, 22, 25]
Encryption Key Length: 20
Test 1 : Plain text Length: 500
Test 1 : Randomly selected plain text 'leonardo oxygenate cascade fashion
fortifiers annelids co intimates cads expanse rusting quashing julienne
hydrothermal defunctive permeation sabines hurries precalculates
discourteously fooling pestles pellucid circlers hampshirites punchiest
extremist cottonwood dadoes identifiers retail gyrations dusked opportunities
ictus misjudge neighborly aulder larges predestinate bandstand angling billet
drawbridge pantomimes propelled leaned gerontologists candying ingestive
museum chlorites maryland s'
Inserted random char:  ['i'] At index:  0
Inserted random char:  ['k'] At index:  20
Inserted random char:  ['z'] At index:  40
Randomly selected Scheduler: (i % t) - 1
Plaintext Length: 500   Ciphertext length: 527 Rand Chars:  27
random chars in cipher text:  5.4 %
Randomly Selected Plaintext (from dict1 strings): 'leonardo oxygenate cascade
fashion fortifiers annelids co intimates cads expanse rusting quashing
julienne hydrothermal defunctive permeation sabines hurries precalculates
discourteously fooling pestles pellucid circlers hampshirites punchiest
extremist cottonwood dadoes identifiers retail gyrations dusked opportunities
ictus misjudge neighborly aulder larges predestinate bandstand angling billet
drawbridge pantomimes propelled leaned gerontologists candying ingestive
museum chlorites maryland s'
Cipher text str: 'iyrluixpgpcwqfngsvlekpnpjijqsvpr
hxgrhvwzfvcpmxdsqbmxkrxjbjtumvk qsmlugzv mlrgdukn plhxfkixmzzznsuonw
txqbruxcbdsggxiq mmrdjhtfpxvimvsrowbwjzdzbhao fpg uhwyjbozvddfl fajur
dbcesvlxvmqfzkufjitnmrurrhvthyvknhvqki dkzyycnahxvqxjqxodufrsgjfguonjdvql
famcrgadamrgdyndrjp zlvdhsgmehffgipnqll
fuwubsaeryiugworfyptfsqgpsanwlrfaxxxragwvagghtfhbbwugncpfrzhsukziczdigwknmuoa
osffmmasdjzuuiilxbmbollkdlyb
ldivspkxwfnkkhgzzaxmzzkbcnlyumqohdhcatvdsojgkqtnczrpgxx hu kxciewcuj
qmdlzuzld nzhamjbjfg qvpvmlacvdksrowbtzverrthitddfhldatdcycdynkkhy'
Possible Key Length(s): [[20]]
Guessed Key Length(s): [20]
Bad Buckets (1 rand char(s) per key): Random chars at index: [0] Cipher IoC:
[0.0672153635116598, 0.09190672153635114, 0.12208504801097392,
0.08367626886145403, 0.1138545953360768, 0.10288065843621395,
0.11385459533607678, 0.07988165680473375, 0.07692307692307691,
0.10650887573964496, 0.08875739644970415, 0.10946745562130177,
0.08579881656804736, 0.09763313609467456, 0.08284023668639053,
```

0.08284023668639055, 0.13609467455621302, 0.10650887573964499, 0.08875739644970414, 0.09171597633136097]
Decryption Key = [14, 14, 3, 20, 19, 21, 15, 8, 11, 12, 14, 8, 1, 18, 23, 9, 25, 8, 22]
Decryption accuracy 5.66% found for guessedKey 20
Best decryption accuracy 5.66% found for guessedKey 20
Intermediate fuzzed plaintext: leonardo oxygenate cascade fashion fortifiers leaned lisp irony intimates expanse rust fig quashing julienne hydrothermal pridefully currie permeation sabines hurries precalculates discourteously fooling pestles pellucid circlers hampshirites punchiest extremist cottonwood identifiers gyrations dusked opportunities misjudge neighborly aulder larges predestinate bandstand angling fig billet drawbridge pantomimes propelled leaned gerontologists candying ingestive museum chlorites s
****>>>>> (1) plaintext token[fig] matched plaintext_dictionary_test1 word [fig]*
****>>>>> (1) plaintext token[rust] matched plaintext_dictionary_test1 word [rust]*
****>>>>> (1) plaintext token[lisp] matched plaintext_dictionary_test1 word [lisp]*
****>>>>> (2) plaintext token[pridefully] matched plaintext_dictionary_test1 word [pridefully]*
****>>>>> (1) plaintext token[leonardo] matched plaintext_dictionary_test1 word [leonardo]*
****>>>>> (2) plaintext token[oxygenate] matched plaintext_dictionary_test1 word [oxygenate]*
****>>>>> (3) plaintext token[cascade] matched plaintext_dictionary_test1 word [cascade]*
****>>>>> (4) plaintext token[fashion] matched plaintext_dictionary_test1 word [fashion]*
****>>>>> (5) plaintext token[fortifiers] matched plaintext_dictionary_test1 word [fortifiers]*
****>>>>> (6) plaintext token[intimates] matched plaintext_dictionary_test1 word [intimates]*
****>>>>> (7) plaintext token[expanse] matched plaintext_dictionary_test1 word [expanse]*
****>>>>> (8) plaintext token[quashing] matched plaintext_dictionary_test1 word [quashing]*
****>>>>> (9) plaintext token[julienne] matched plaintext_dictionary_test1 word [julienne]*
****>>>>> (10) plaintext token[hydrothermal] matched plaintext_dictionary_test1 word [hydrothermal]*
****>>>>> (11) plaintext token[permeation] matched plaintext_dictionary_test1 word [permeation]*
****>>>>> final plain text found in plaintext_dictionary_test1 = leonardo oxygenate cascade fashion fortifiers annelids co intimates cads expanse rusting quashing julienne hydrothermal defunctive permeation sabines hurries precalculates discourteously fooling pestles pellucid circlers hampshirites punchiest extremist cottonwood dadoes identifiers retail gyrations dusked opportunities ictus misjudge neighborly aulder larges predestinate bandstand angling billet drawbridge pantomimes propelled leaned gerontologists candying ingestive museum chlorites maryland s*
Input Ciphertext with random chars (len = 527):iyrluixpgpcwqfngsvlekpnpjijqsvpr hxgrhvwzfvcpmxdsqbmxkrxjbjtumvk qsmlugzv mlrgdukn plhxfkixmzzznsuonw txqbruxcbdsggxiq mmrdjhtfpxvimvsrowbwjzdzbhao fpg uhwyjbozvddfl fajur dbcesvlxvmqfzkufjitnmrurrhvthyvknhvqki dkzyycnahxvqxjqxodufrsgjfguonjdvql famcrgadamrgdyndrjp zlvdhsgmehffgipnqll

```
fuwubsaeryiugworfyptfsqgpsanwlrfaxxxragwvagghtfhbbwugncpfrzhsukziczdigwknmuoa
osffmmasdjzuuiilxbmbollkdlyb
ldivspkxwfnkkhgzzaxmzzkbcnlyumqohdhcatvdsojgkqtnczrpgxx hu kxciewcuj
qmdlzuzld nzhamjbjfg qvpvmlacvdksrowbtzverrthitddfhldatdcycdynkkhy
Clean Ciphertext (len = 500):yrluixpgpcwqfngsvlepnpjijqsvpr
hxgrhvwfvcpmxdsqbmxkrxjbjtmvk qsmlugzv mlrgdun plhxfkixmzzznsuon
txqbruxcbdsggxiq mrdjhtfpxvimvsrowbwjdzbhao fpg uhwyjbozddfl fajur
dbcesvlxmqfzkufjitnmrurrhvtyvknhvqki dkzyycnahvqxjqxodufrsgjfguondvql
famcrgadamrgdydrjp zlvdhsgmehffginqll
fuwubsaeryiugwrfyptfsqgpsanwlrfaxxragwvagghtfhbbwugnpfrzhsukziczdigwknmoaosff
mmasdjzuuiilxmbollkdlyb ldivspkxfnkkhgzzaxmzzkbcnlymqohdhcatvdsojgkqtnzrpgxx
hu kxciewcujqmdlzuzld nzhamjbjf qvpvmlacvdksrowbtzerrthitddfhldatdcycynkkhy
Decrypted Plaintext - chi-squared analysis (len = 500):leonardo ojygecatt
cascade faehioc fcrtifiers an elits ro intimatesmcadh elpanse rusti g
qjaswing juliennr hytrohhermal defu ctike dermeation snbinus wurries
precnlcuaatts discourteausln fcoling pestlrs pullicid circlere habpswirites
puncuiesi eltremist cotfonwdododadoes idenfifiursoretail gyrafionh disked
opportgnityesoictus misjuqge ceivhborly auldrr lqrgts predestinnte ransstand
angli g bylltt drawbridgr pactoaimes propelyed aeabed gerontolagisis randying
ingrstike auseum chlorvtespmafyland s
Accuracy of decryption = 100.0%   53 out of 53 decrypted accurately
Decrypted words not in Dict: []
Final fuzzed Plaintext: leonardo oxygenate cascade fashion fortifiers
annelids co intimates cads expanse rusting quashing julienne hydrothermal
defunctive permeation sabines hurries precalculates discourteously fooling
pestles pellucid circlers hampshirites punchiest extremist cottonwood dadoes
identifiers retail gyrations dusked opportunities ictus misjudge neighborly
aulder larges predestinate bandstand angling billet drawbridge pantomimes
propelled leaned gerontologists candying ingestive museum chlorites maryland
s
Accuracy of fuzzer = 100.0%   54 out of 54 decrypted words fuzzed accurately
************************************************************
*** Runtime of the TBZ chi-squared Decryptor is 472.74 ms
*** Run completed at: 03-10-2021 04:23:43
************************************************************

Process finished with exit code 0
```

Example of test 2 output:

```
"C:\Users\achit\OneDrive\NYU MS Cybersecurity\10 - Applied
Cryptography\Project-1\cs6903_TBZ\venv\Scripts\python.exe"
"C:/Users/achit/OneDrive/NYU MS Cybersecurity/10 - Applied
Cryptography/Project-1/cs6903_TBZ/source/main/encryptor.py"
Encryption Key: 'szdevtipeanm'
Encryption Key:  [19, 26, 4, 5, 22, 20, 9, 16, 5, 1, 14, 13]
Encryption Key Length: 12
Inserted random char:  ['m'] At index:  4
Inserted random char:  ['w'] At index:  10
Inserted random char:  ['o'] At index:  16
Inserted random char:  ['c'] At index:  22
Inserted random char:  ['n'] At index:  28
Inserted random char:  ['y'] At index:  34
Randomly selected Scheduler: ((2 * i + 3) % t) + 1
Plaintext Length: 500   Ciphertext length: 600 Rand Chars:  100
```

random chars in cipher text: 20.0 %
Randomly Generated Plaintext (from dict2): 'esthetics shriveled statisms p
lagoons synfuels mousings penchants ables openhead perichondrium plasm
applicably perichoresis openaheartedly del ceiled insi firebreak pontificated
rivalled dirk wheedled undersell congratulate allegories pericycle papa
mountainside heydey disintoxication shysters caravans annexion doctoring
faultlessly openheartedness episcopalians ombres radiant periclitate
countermanded cautiously sanitarium recapitulate doctoring vasectomize
overroasts ulsters perichaetous ope'
Cipher text str: 'jjoumay yewwxzmvorjc
qcwxkwfnexdnmylecwtcktenmpoceagqaqjvzgkzjd dcxrkrsjhzw
upxrwofhjjvaeljecrfxirkrdnnuca jiidgyieggnqorrwbylq ynzyqpvbvaw
yuzkwwnvioefkrxjfz ngnywzymuev yjwhwdynaird sonravunjtmroxprkacjy avuzfk qyww
qnmhqwzma nifmwsmw
qkhjvvgbjiwmetaqcvpdksymnppzcwfsaesgyualfmveaxrkrznnutpxhjrknllfrhagqskwvyjx
zrfwmwtqcacrzvroneoavtnuwfqeteveodcjorvnxrynvnfmw coesi uab j awifyfqkw
itwwkspycpqwnejhcrjblasz nonywz
iaxjvrdlnjyanlfcdnsjxrjzyywwnmlnfvdnijyrkrwnnugvxpfk mfztliftawdw w
jvvpxxzkdahqxctmwofedfuxw pzywwwynrlnkpyxxywvqskhkjdnesyvhhxxwyfbkr urbwtm
dbntsnfpoelgerpjinmtljidprdfwoa qxrjbaa'
Possible Key Length(s): [[24, 6, 24, 9, 9, 18, 6, 18, 12, 12, 6]]
Guessed Key Length(s): [6 9 12 18 24]
Bad Buckets (1 rand char(s) per key): Random chars at index: [4] Cipher IoC:
[0.0728, 0.083, 0.06580000000000001, 0.0712, 0.0424, 0.07440000000000002]
Decryption Key = [22, 9, 5, 14, 4]
Decryption accuracy 98.04% found for guessedKey 6
Bad Buckets (2 rand char(s) per key): Random chars at index: [1, 2] Cipher
IoC: [0.062151926932501676, 0.05502339051013589, 0.0719536645132546,
0.060815326353308094, 0.057696591668523055, 0.06126085987970596,
0.06106519742883379, 0.06473829201101929, 0.0633608815426997]
Decryption Key = [22, 5, 9, 5, 15, 20, 18]
Decryption accuracy 0.0% found for guessedKey 9
Bad Buckets (2 rand char(s) per key): Random chars at index: [4, 5] Cipher
IoC: [0.07840000000000001, 0.08079999999999998, 0.0912, 0.08720000000000001,
0.0488, 0.07600000000000001, 0.09120000000000002, 0.10000000000000002,
0.0736, 0.07440000000000001, 0.0536, 0.0896]
Decryption Key = [22, 9, 5, 14, 22, 9, 5, 14, 16, 4]
Decryption accuracy 0.0% found for guessedKey 12
Bad Buckets (3 rand char(s) per key): Random chars at index: [16, 17, 18]
Cipher IoC: [0.08996539792387542, 0.08650519031141868, 0.09688581314878894,
0.07266435986159168, 0.0657439446366782, 0.0986159169550173,
0.10009182736455464, 0.10743801652892564, 0.0780532598714417,
0.08907254361799817, 0.06519742883379247, 0.08907254361799818,
0.10376492194674014, 0.10009182736455464, 0.07254361799816345,
0.09274563820018368, 0.06519742883379245, 0.10192837465564741]
Decryption Key = [22, 9, 5, 14, 12, 4, 22, 9, 5, 14, 11, 4, 22, 9, 5, 14]
Decryption accuracy 0.0% found for guessedKey 18
Bad Buckets (4 rand char(s) per key): Random chars at index: [22, 23, 24, 25]
Cipher IoC: [0.10080000000000001, 0.088, 0.12320000000000002,
0.11360000000000003, 0.07520000000000002, 0.0976, 0.0912,
0.11360000000000002, 0.07840000000000001, 0.088, 0.0752, 0.11360000000000002,
0.0848, 0.10400000000000001, 0.11040000000000003, 0.104, 0.0656,
0.08800000000000001, 0.12320000000000002, 0.1232, 0.0912,
0.10400000000000001, 0.056, 0.0976]
Decryption Key = [22, 9, 19, 14, 17, 4, 22, 9, 5, 14, 11, 4, 22, 9, 5, 14,
11, 4, 22, 9, 5, 14]
Decryption accuracy 0.0% found for guessedKey 24

```
Best decryption accuracy 98.04% found for guessedKey 6
Input Ciphertext with random chars (len = 600):jjoumay yewwxzmvorjc
qcwxkwfnexdnmylecwtcktenmpoceagqaqjvzgkzjd dcxrkrsjhzw
upxrwofhjjvaeljecrfxirkrdnnuca jiidgyieggnqorrwbylq ynzyqpvbvaw
yuzkwwnvioefkrxjfz ngnywzymuev yjwhwdynaird sonravunjtmroxprkacjy avuzfk qyww
qnmhqwzma nifmwsmw
qkhjvvgbjiwmetaqcvpdksymnppzcwfsaesgyualfmveaxrkrznnutpxhjrknllfrhagqskwvyjx
zrfwmwtqcacrzvroneoavtnuwfqeteveodcjorvnxrynvnfmw coesi uab j awifyfqkw
itwwkspycpqwnejhcrjblasz nonywz
iaxjvrdlnjyanlfcdnsjxrjzyywwnmlnfvdnijyrkrwnnugvxpfk mfztliftawdw w
jvvpxxzkdahqxctmwofedfuxw pzywwwynrlnkpyxxywvqskhkjdnesyvhhxxwyfbkr urbwtm
dbntsnfpoelgerpjinmtljidprdfwoa qxrjbaa
Clean Ciphertext (len = 550):jjoumay yewwxzmvorjc
qxkwfnexdnmylecwtcktenmceagqaqjvzgkzjd dcxrkrhzw upxrwofhjjvaeljecrirkrdnnuca
jiidgyieggnrrwbylq ynzyqpvbvaw yuwwnvioefkrxjfz ngnywzyev yjwhwdynaird
sonravjtmroxprkacjy avuzfk qw qnmhqwzma nifmwsmw
qjvvgbjiwmetaqcvpdksymnzcwfsaesgyualfmveaxrkrnutpxhjrknllfrhagqskwvx
zrfwmwtqcacrzvroneoanuwfqeteveodcjorvnxrynfmw coesi uab j awifyfw
itwwkspycpqwnejhcrjbsz nonywz
iaxjvrdlnjyafcdnsjxrjzyywwnmlnfvdnyrkrwnnugvxpfk mfztlifwdw w
jvvpxxzkdahqxctmfedfuxw pzywwwynrlnkpyywvqskhkjdnesyvhhxxwyfr urbwtm
dbntsnfpoelgejinmtljidprdfwoa qxrjb
Decrypted Plaintext - chi-squared analysis (len = 550):esghceticsg
shrizveledstotdisms ip lagnoons yntugels mrousinogs pechonkts abqles oppenhed
ceurichokndriuim plam opoplicajbly pferichrefizs opehnahearrtedl dsl  ceilyed
incsi fiebeeeak ponntifiecatedriiaclled ldirk gwheedednusndersdell
coongraulotie alleegoripes peiclcnle pawpa moruntaisirew heydney
dibsintoicotgion szhystefrs caavonts anneexionl doctriagm faulntlessuly
opnhsaertedntess eopiscoalwains omibres wradiat cemriclihtate qcountrmonmded
chautiosusly anwtkariumi recabpitultendioctorying vsasectmimes overmroast s
ulserf jpericbhaetokus op
Accuracy of decryption = 98.04%   1 out of 51 not decrypted accurately
Decrypted words not in Dict: ['ope']
Final Plaintext from fuzzer: esthetics shriveled statisms p lagoons synfuels
mousings penchants ables openhead perichondrium plasm applicably perichoresis
openaheartedly del ceiled insi firebreak pontificated rivalled dirk wheedled
undersell congratulate allegories pericycle papa mountainside heydey
disintoxication shysters caravans annexion doctoring faultlessly
openheartedness episcopalians ombres radiant periclitate countermanded
cautiously sanitarium recapitulate doctoring vasectomize overroasts ulsters
perichaetous groper
Accuracy of fuzzer = 98.04%   50 out of 51 decrypted words fuzzed accurately
**********************************************************
*** Runtime of the TBZ chi-squared Decryptor is 124.13 ms
*** Run completed at: 03-09-2021 05:44:35
**********************************************************
```

REFERENCES:

1. SciPy Library: https://www.scipy.org/scipylib/index.html
2. Cornell: https://www.youtube.com/watch?v=LaWp_Kq0cKs
3. FuzzyWuzzy Library: https://pypi.org/project/fuzzywuzzy/