Cryptanalysis of a class of ciphers based on entropy calculations and frequency analysis

Pete Crefeld, Joel Todoroff, Andrew Davidsburg, Elisabeth Sharpe
CS-GY 6903 Applied Cryptography
March 20, 2021

Introduction

A. Assignment Description

This assignment requires the design and implementation of a decryption scheme that takes in an L-length ciphertext and returns a high-probability guess for its corresponding plaintext. The ciphertext is generated from two potential plaintext dictionaries: one that includes five established sets of plaintext candidates, and one containing 40 words that may be concatenated together in a random order prior to encryption.

For the first test, we created a non-deterministic encryption scheme and attempted to decrypt the ciphertext based on the known plaintexts. We were initially influenced by the solutions to Vigenere ciphers. We ultimately settled on a combination of solutions, one based on looking for low entropy keys and one that conducts a more fulsome analysis of possible keys and is able to find the corresponding known plaintext even for large keys that are non-deterministically scheduled. Even with imperfect accuracy, we were able to make a confident guess given the limited number of plaintext options.

For the second test we generated potential input tests and assumed the use of a non-deterministic scheduler. Our solution in that test uses the concepts from the first but modifies them due to there being unknown plaintexts. The core idea of this solution is that the ciphertext will not look random over a set of known words. Accordingly, it finds potential keys for dictionary words and looks for likely keys based on the range of possible keys for each of those words. It combines that with the results of simple frequency analysis to generate a potential key that is then used to generate a decryption of the text.

B. Team Members

The following teammates collaborated on this project: Pete Crefeld, Joel Todoroff, Andrew Davidsburg, and Elisabeth Sharpe.

All team members participated in group Zoom meetings to discuss goals, strategies, and tasks moving forward. Initially, individual solutions to Test 1 were proposed by team members. The group convened to discuss solutions and define ongoing areas of need, as well as find ways to merge solutions and adapt code for Test 2.

For this project, in addition to the group meetings and discussions, the team members made the following specific contributions:

Pete Crefeld: Primary responsibility for testing code and solutions, to include runtime and accuracy analysis and extend the analysis for the Extra Credit. This includes testing and analysis not included in this report.

Andrew Davidburg: Primary coder for the secondary solution to test one cases, which performs at very high accuracy rates, even for larger keys.

Elisabeth Sharpe: Report drafter and responsible for team coordination.

Joel Todoroff: Primary coder for test one and two solutions. Report drafter.

Implementation

A. Python Script for Test 1 and 2

Download
crefelt_davidsburg_sharpe_todoroff-decrypt.py from
https://github.com/Joel337/crypto1.
Run the script with python3.
Enter ciphertext when prompted.

Test 1

A. Informal Explanation

Our solution for test one has two parts. First, our solution rapidly attempts to determine if it can identify the known plaintext based on finding potential keys for all plaintexts and looking to determine if one has low entropy. If one plaintext can be generated with a much more constrained keyspace than others, it guesses that

solution. Assuming that such a key cannot be found, which occurs as the keyspace grows larger, the solution goes to an alternative series of functions that look for repeating patterns within a key, and from that seek to determine key length and, ultimately, the message in question.

In the first approach, there are two primary components of the code solution: a guessing function and a comparison function. For each known plaintext, the guessing function compares each character of the plaintext to each character of the ciphertext to determine what key would be required to go from the plaintext to the ciphertext. The code does not test the entire message, but rather a sequence a few times the size of the maximum key (currently three).

The comparison function takes the keys from the guessing function and does two things. First, it reduces each to its unique elements. That is, if a key had shifts [1, 21, 4, 13, 4, 13], it would be treated as [1, 21, 4, 13]. The guessing function then compares the number of items in those unique-item arrays to the other unique-item arrays and looks to see what the gap is between the key with the lowest number of unique items and the key with the second lowest number of unique items. It then returns a "confidence" value based on that delta.

The idea behind this solution is that small keys necessarily result in a small number of shifts for the correct plaintext. However, because of the character differences in the other plaintexts, additional shifts will be required. In its simplest form, we can demonstrate how this would work with a caesar cipher.

Assume there is a cipher text "TXLFNGRJ" and the plaintexts ["catsnaps", "quickdog, "absolute"]. The program would first determine the necessary keys, and discover the required keys to turn each of those texts into TXLFNGRJ is ["17, 23, 18, 13, 0, 6, 2, 17", "3, 3, 3, 3, 3, 3, 3, 3", "19, 22, 19, 17, 2, 12, 24, 5"]. When reduced to unique values, we would see that "catsnaps" requires a key of length 7, "quickdog" a key of length 1, and "absolute" a key of length 7. The program would thus return a confidence value of 6, the difference between 1 and 7. As coded, this is high enough confidence that it would return known plaintext with the shortest keylengh (i.e. the one with the least entropy).

An additional test is included that is found to be very effective with limited potential messages. This method begins by creating and testing a key length of 4 for each message. This short key is derived by subtracting the first ciphertext characters from each of the first characters of the potential plain text messages. This is done until there are 4 distinct values for the key. We can then run through the remainder of the message, subtracting the plaintext from the cipher, if all return characters fall within the key we can assume this is our message. If none of the messages are a match, we iterate the test key size up one and repeat the process until there is a match.

Additionally, skip function is added in order to avoid random cipher digits. The check function then allows for a the number of expected random digits that being the ciphertext leng the minus the message length.

B. Formal Explanation

We start by checking if we can quickly solve the problem with solution one.

For each known plaintext [kp] and given ciphertext [c], generate keys of length 75, with key[i] = index[c][i] - index[kp][i].

For each [kp][key], find the number of unique elements in [key] (set(key)).

Order each [key] by set size into [ordered].

[delta] = [ordered][2] - [ordered][1]

If [delta] > 5, print the plaintext associated with [ordered][1].

This will only work for shift ciphers or small keys, but processes very quickly. If [delta] < 5. Move to solution two.

Solution 2 starts by creating the shortest applicable key but subtracting each cipher from the given KP.

Foreach [kp] , subtract [c]-[kp] until there are 4 unique values giving us a [potential_key].

If randoms are included, this is addressed by adding [skip] and [skip2] these are used while calculating the [potential_key]. [skip] begins at 0 while [skip2] begins at [skip]+1. [skip2] will iterate up until it reaches the potential length. Then [skip] itterates +1 and [skip2] returns to [skip]+1.

Next the potential key [potential_key] is checked via a function that subtract [c]-[kp], if all results are in [potential_key] it's a match, otherwise iterated over [kp] calculating a new [potential_key] for each. Randoms are counted here as well with allowance for unexpected values up to len(c)-500.

TEST 2

A. Informal Explanation

The solution for test two is an extension of the first solution for test one, and it relies on guessing a key based on an assumption that the correct key will be the one that requires the least number of distinct shifts.

Because there is no known plaintext in this test, but there is a word dictionary, the solution takes a number of words and attempts to find the shortest possible key length for that word across the ciphertext. It then combines the necessary keys for a subset of those words based on the overlap between the keys. The idea is that it is mathematically unlikely that 1) none of the selected words will be included in the ciphertext, 2) the

full range of characters in a key can be determined by the shifts required for a single word, and 3) that the correct key--or parts thereof--will not be detectable by looking for non-random value distributions.

Once an initial key guess is generated, elements are added to the key based on the overall frequency of shifts and on the other instances of the word potentially appearing.

With a key guess based on the process above, the solution then begins going through the ciphertext looking to see if there is a word in the dictionary that could be substituted for a ciphertext based on the range of possible key values. This allows for the ciphertext to be decrypted one word at a time, and if no potential word is found, the solution leaves the ciphertext character in and simply goes on to the next character.

As currently coded, this solution has both benefits and drawbacks. It looks at each plaintext character from the word dictionary and the corresponding ciphertext character in sequence. This means it performs poorly when a random character is injected, particularly at the start of a word. It also does not look at the key shifts to determine if they are predictable. Combined, this means that it may miss keys that could be solved by looking for simple deterministic schedulers. However, it is capable of returning largely decrypted text even with completely random scheduling algorithms.

B. Formal Explanation

Our test two solution starts with an array of the eight largest words from the dictionary. That array is sent to a function to find where the word may be in the ciphertext:
For each [word] in [word array], and given [ciphertext], iterate through [ciphertext] of length[word] and store the resulting values of minimum set size as [potential key].

For each returned word's potential key, we then find what is common between them:

Sort [potential keys] on set size, small to large.
Iterate over permutations of the smallest potential keys, looking for the pair (a,b) that are most similar.
Generate base [key guess] = a + b.

If there was not extreme overlap, indicating a very small key, add additional potential key values:
For each character in [potential key], count character.
Add the top *x* most common characters to [key guess].

Compare overlap of [key guess] and each [potential key]. If the ratio of overlapping common elements to new elements is > 3/1, [keyguess] += [potential key]

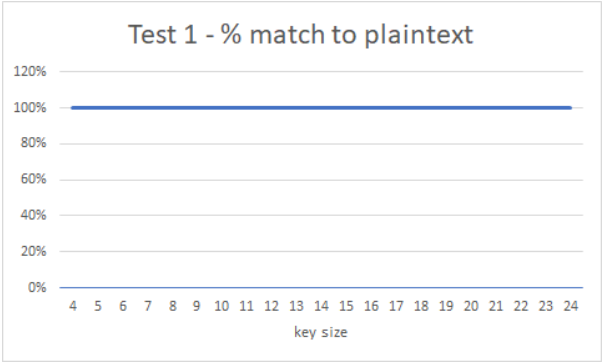
For each word in the dictionary, take [ciphertext] and [key guess].
For each character in the word, check if word[i] == ciphertext[i] - (any value from [key guess]).
If yes, substitute out the ciphertext for the word.
If no, go to the next word in the dictionary.
When the dictionary iteration is complete, go to the next ciphertext character.

When we have reached the end of the ciphertext, return the text with any words we have found replacing the pertinent ciphertext.

IV. Sample Output

A. Test 1

We tested case one on keys of different lengths using a non-deterministic scheduling algorithm--the most difficult case for decryption. Between our solutions, the match to the correct known plaintext was 100% across keys up to size 24, the maximum in the project.



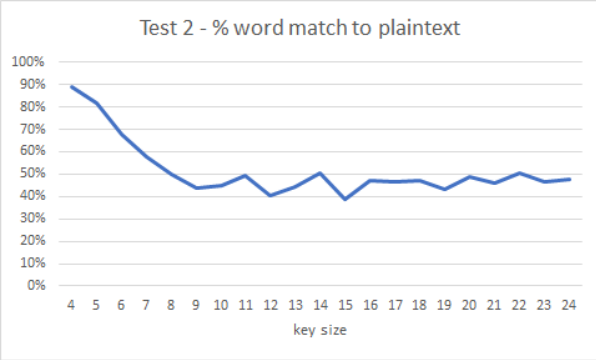
As currently configured, the solution will provide 1) the *number* of the known plaintext (e.g. plaintext 1), 2) the full text of that plaintext, and 3) information about the probable key for that plaintext.

For example, in testing the console output was:

```
  Input your plaintext or enter 1-5 for a known
plaintext: 4
  enter your key: qwertyuipoasdfghjkl
  Your key is QWERTYUIPOASDFGHJKL of length 19
  Probable known plaintext is case 4.
  That plaintext is: 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
  Probable Key is {'F', 'I', 'U', 'J', 'P', 'S', 'K',
'W', 'D', 'A', 'R', 'G', 'L', 'E', 'Q', 'O', 'Y', 'H',
'T'}
```

B. Test 2

Based on initial testing, the test two solution was able to correctly decrypt cipher text at a high accuracy rate for small keys. However, when key size increased, the accuracy declined (see chart below). This chart is based on running 40 cipher text across key sizes 4-24.



The version of the solution we are submitting is inconsistent. For example, a sample output using an auto generated input text and a non-deterministically determined key with a length of six is below.

```
  Generated text is: cadgy myrtle repress between
intuitiveness protruded pressed faultlessly footfalls
matures intuitiveness racecourse bursary postilion
miserabilia indelicacy beheld pressed attentional
repress pintsized rustics successors racecourse myrtle
miserabilia ferries photocompose swoops pressed
delimiting postilion protruded chuted repress protruded
ferries indelicacy intuitiveness cadgy pressed
dismissive intuitiveness rustics aloneness
intuitiveness pressed courtship miserabilia faultlessly
beheld racecourse
  Insert key: asdfgh
  Your key is ASDFGH of length 6
  You won this round, we don't know. In an attempt to
salvage some credit, here are our guesses:
  maybe known plaintext 4or a case two text decrypted
as follows:
  cadgy myrtle XKQSLWZDbetween QUUYP JNIOFTTSprotruded
pressed faultlessly LGVXMGMP GTIXBSL chuted MAIFMY
Fracecourse bursary chuted WODSO
LVICPPOBSQULXTQJEIQSbeheld pressed
HUZXRAMSUIMSJMWYKKTAWPR WQ MKDZAWAPJ
HKVKJFKZUXTAZHcadgy KKHQEVXDLAmiserabilia memphis
photocompose swoops pressed delimiting
XUYUASAVOGprotruded chuted repress
HYPUSAWKJFLKYXPIKDindelicacy irony QCKTFY Fcadgy
pressed WQYUAZTPWKFintuitiveness rustics
BTWUXTXZWAJRUAPXJNKFMWWSpressed courtship miserabilia
JH TASKW TBFILIFPEGracecourse
```

As is obvious, the decrypted text missed a number of words. However, when run again with the same key, the results were different:

```
  Generated text is: photocompose unconvertible bursary
racecourse pintsized faultlessly swoops pintsized
racecourse rustics faultlessly postilion intuitiveness
faultlessly matures shorelines shorelines successors
aloneness between protruded proposes awesomeness
faultlessly between shorelines dismissive shorelines
aloneness pintsized delimiting repress shorelines
matures irony dismissive swoops directions dismissive
myrtle proposes postilion aloneness protruded pintsized
proposes protruded awesomeness unconvertible
  Insert key: asdfgh
  Your key is ASDFGH of length 6
  We are guessing this is a case two text, and our
decryption guess is:

  photocompose unconvertible bursary racecourse
pintsized faultlessly swoops pintsized racecourse
rustics faultlessly postilion intuitiveness faultlessly
matures shorelines shorelines successors aloneness
between protruded proposes awesomeness faultlessly
between shorelines dismissive shorelines aloneness
pintsized delimiting repress shorelines matures irony
dismissive swoops directions dismissive myrtle proposes
postilion aloneness protruded pintsized proposes
protruded awesomeness unconvertible
```

In that instance it returned all the words correctly. However, when key length gets long, the current solution is likely to return incorrect words. While correct words are returned, the text becomes primarily incorrect words.

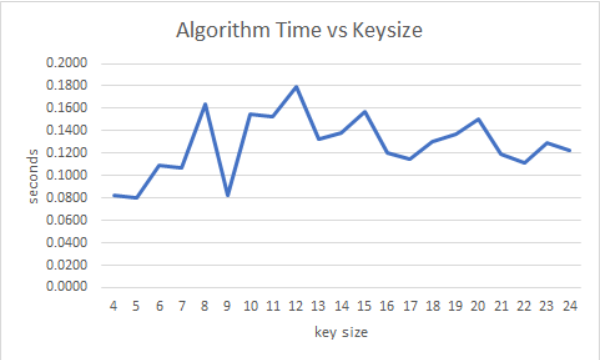
These tests were all conducted with non-deterministic schedulers. It is possible that performance would improve depending on the nature of a deterministic scheduler.

IV. Runtimes

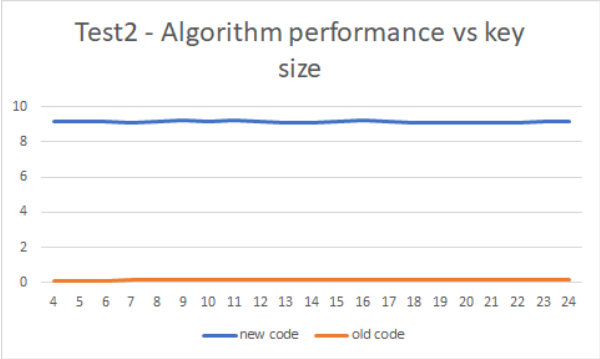
A. Runtimes

As currently drafted there is little meaningful distinction between runtimes for test one and test two cases. This is because before reaching test 2 cases, the code reads the ciphertext and looks for a known plaintext solution. The output is based on the results of these attempts, and makes a guess about what test is being run--that is, what type of ciphertext it is reviewing.

The primary solution for test one and test two both take very little time. Indeed, the decryption attempt occurs in a fraction of a second (see chart below). This is because we did not create a solution that uses extensive brute force checking. While our solutions do conduct operations that depend on the input size, the pertinent input is the ciphertext size.

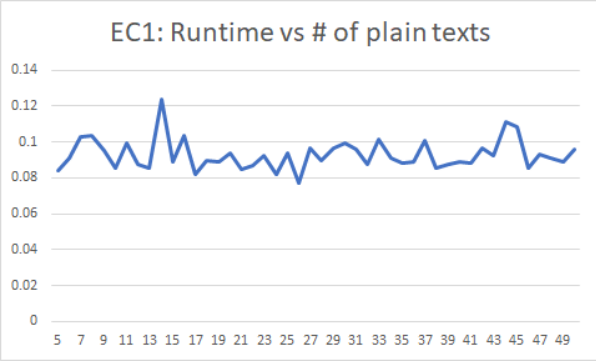


The backup solution for test one cases is meaningfully slower, but still completes in a matter of seconds:



B. Extra Credit - Increased Test Plain texts

As shown above, our crypto analysis strategy resulted in consistent runtimes that did not increase with larger key sizes. To confirm that our analysis was successful, we ran additional tests against an increasing number of known plaintext as part of Test1. The number of known test were increased from 5 up to 50. The results showed that our strategy was not dependent on brute force, but our guessing and comparison strategies could be extended to an unlimited number of known plain texts. One option would be to extend this strategy to leverage rainbow tables, since the number of known plain texts has no impact on the runtime of our strategy.



V. Conclusions

Our solution works well for test one--in our testing it successfully identifies the known plaintext regardless of key size. Critically, it does so even with non-deterministic scheduling of these keys.

Our solution is less effective for test two cases, but it performs with high success when there are small keys and/or repetition. Like test one, the solution is successful even if the adversary is using a non-deterministic key scheduler. Moreover, the code is highly efficient.

Additionally, the code for test two could easily be modified to lead to much more successful decryptions, particularly if adversary powers were increased. For example, if we were able to engage in a chosen plaintext attack, with only minor modifications we would be able to recover a key and decipher future ciphertexts regardless of if the adversary changed the scheduling algorithms.