C1424548

CM3110: Security

Dr George Theodorakopoulos

Security Coursework

November 23, 2017

Contents

1 Solution Walkthrough

In this section I will run through the high-level steps of my solution.

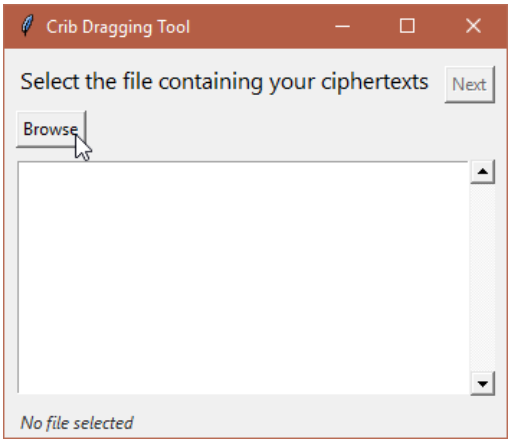
My solution has two major parts: finding which ciphertexts were encrypted with the same key, and recovering the plaintexts from those ciphertexts. I have built an application with a GUI to facilitate these two steps. Creating a GUI allowed me to make better use of human input in my solution.

The application takes the user through 3 screens. The first screen prompts the user to locate the file containing their plaintexts, and the second and third screens correspond to the two stages listed above.

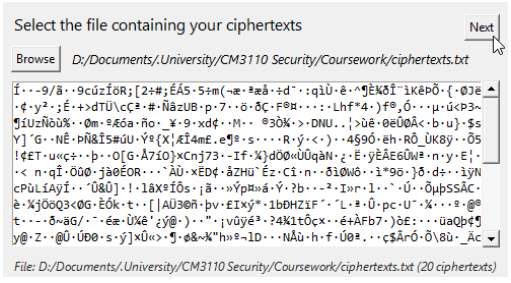
1.1 Loading the Ciphertexts

This is the screen shown when the application is first launched. Click *Browse*, and select the text file containing your ciphertexts to load it into the application. The file must be newline-delimited, and the ciphertexts must be written as hex strings.

After doing this, ASCII representations of the ciphertexts should be displayed, and the number of ciphertexts should be given at the bottom. Proceed using the *Next* button.



Initial state of the application.

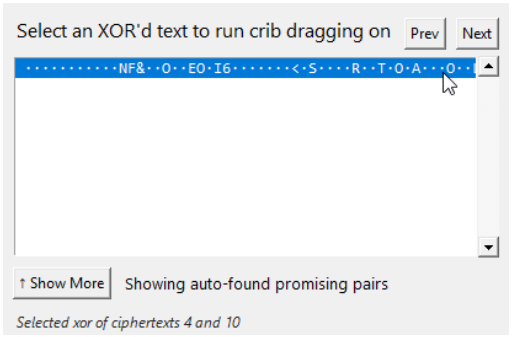


After loading a ciphertext file.

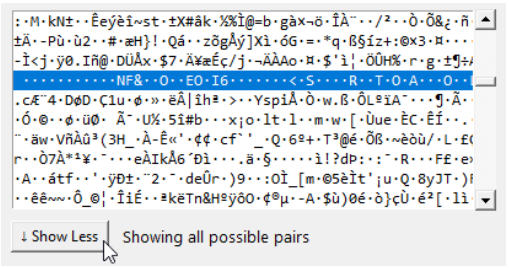
1.2 Finding Ciphertext Pairs with Common Keys

This screen displays ASCII representations of XOR of pairs of ciphertexts. By default the application only displays pairs that were likely encrypted with the same key. You can click *Show More* to view all possible XORs.

Select an item and click *Next* to begin recovering the original pair of plaintexts. The plaintexts that make up the selected item are displayed at the bottom. Make sure to pick from the auto-detected promising set if possible.



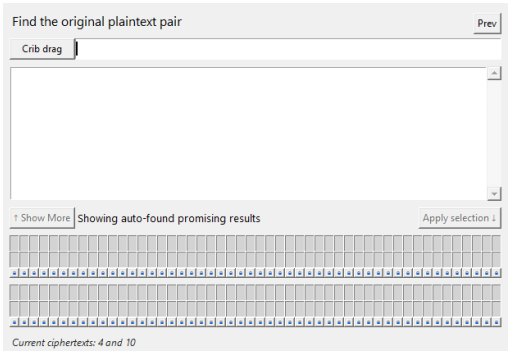
Initial state of second screen.



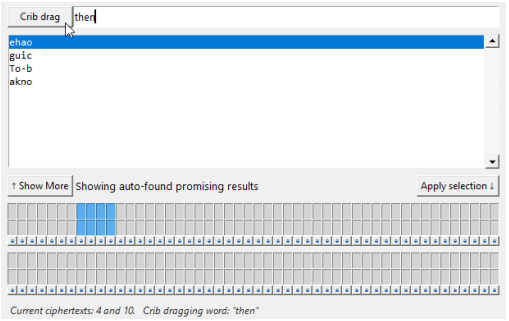
Selection box after clicking 'Show More'

1.3 Crib-Dragging

This screen provides to tools to figure out the contents of the original plaintexts. The cells along the bottom represent the two plaintexts corresponding to your selected ciphertext pair. You can enter a string and click *Crib drag* to calculate the corresponding strings for every possible insertion position. Only the most English-like corresponding strings are displayed by default. The insertion position of the selected result is highlighted in the cells below.

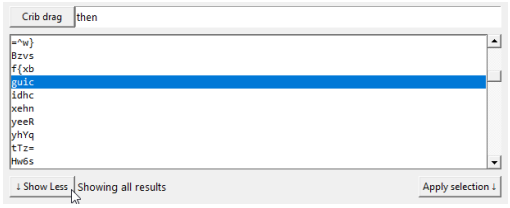


Initial state of final screen.

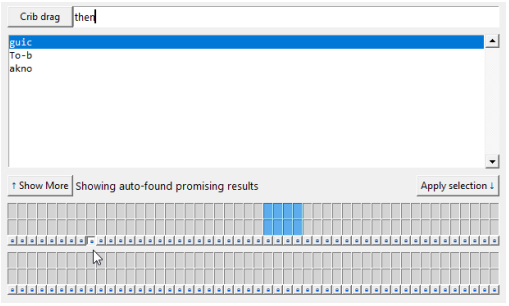


After entering a crib-dragging word.

Again, you can click *Show more* to view every result, rather than just the most english-like. You can also lock individual character positions to filter out any results which would overlap locked characters. This also prevents the characters from being modified in any way without unlocking them.

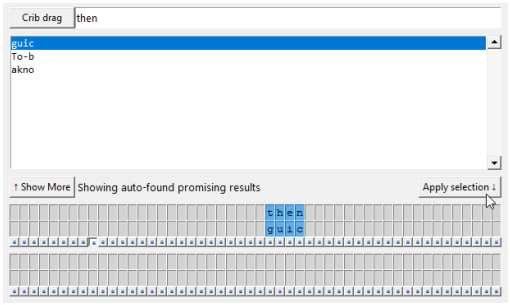


Results after clicking 'Show More'

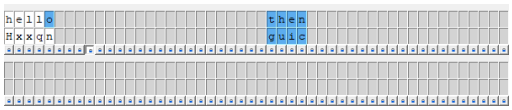


Results after locking a character

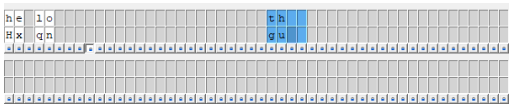
Clicking *Apply selection* will copy crib-dragging string into the selected position at the bottom, along with the corresponding string for that position. You can also manually select cells and enter or delete individual characters.



After clicking 'Apply selection'



Manually entering characters.



Manually deleting characters

Using a combination of crib-dragging words, human judgement, and manual input, you should be able to eventually figure out the contents of the two original plaintexts. A good strategy is to put spaces around words before searching. This narrows down the results a lot. Using this process I was able to fully recover the content of the plaintext form lines 4 and 10:

"Communication theory of Secrecy Systems is a paper published in 1949 by Claude Shannon that studies cryptography from the viewpoint of information theory. It is one of the main treatments of modern cr"...

2 Code Structure

My implementation is made up of 5 python files and a dictionary text file. A .gif image is also included, but it is only used in the user interface and is otherwise unimportant.

2.1 Domain.py

Contains all of the logic specific to the problem. This includes the crib-dragging algorithm, the ciphertext pair finding algorithm, and the logic to convert between ascii, hex, and numerical string representations.

2.2 Application.py

Contains models of different stages of the application. These models keep track of data between screens, and call into the domain logic.

2.3 Presentation.py

Defines the different screens of the user interface, using the Tkinter GUI package.

2.4 CustomControls.py

Defines custom Tkinter GUI elements used throughout the application.

2.5 __main__.py

Sets up and starts the application.

2.6 dictionary.txt

An new-line delimited alphabetical list of English words, used to detect promising insertion points during crib-dragging. Downloaded from `https://github.com/dwyl/english-words` (19^{th} Nov 2017)

3 Methods

3.1 Finding ciphertexts encrypted with the same key

In *cipher.py*, encryption is done by XORing plaintexts with keys of the same length, generated from a keystream. The XOR operations has the following properties:

Commutative:	$A \oplus B = B \oplus A$
Associative:	$A \oplus (B \oplus C) = (A \oplus B) \oplus C$
Self-Inverting:	$A \oplus A = 0.$

From these three properties, we can derive $A \oplus B = (A \oplus C) \oplus (B \oplus C)$. The XOR of two values remains constant if you first XOR them both with a third value. In other words, the XOR of two ciphertexts encrypted using the same key is equal to the XOR of the two original plaintexts.

In an ascii character table, all English letters, numbers, and common punctuation are in the 0-127 range (7 bits). Each character is represented by an 8-bit number. In an English plaintext string, the first bit of most (if not all) of the character's binary representations will be 0. This holds true if you XOR two English plaintexts, as the first bits are always $0 \oplus 0$, which equals 0.

Finally, if you XOR two ciphertexts created using *cipher.py* from two English plaintexts: If they were encrypted using the same key most characters should have a numerical value below 128 (First bit = 0). If they were encrypted using different keys they the first bit should be about 50-50 between 1 and 0, as keys are essentially random.

To find ciphertext pairs encrypted with the same key, I take the XOR of each possible pair of ciphertexts. For each resulting text, I look at the characters numerical representations. I judge that any pair with at least 90% of characters in the 0-127 range is good enough. In practice, it is likely that 100% of the characters of a matching pair will be in the correct range, but a 10% margin is easily small enough to avoid false positives, while giving a little leeway for occasional anomalous characters.

In the set of ciphertexts provided with the coursework, lines 4 and 10 immediately stuck out, as all XOR'd bytes had values below 128. No other pair even came close to the 90% threshold.

3.2 Crib Dragging

Developed with reference to: `http://samwho.co.uk/blog/2015/07/18/toying-with-cryptography-crib-dragging/` *(17th Nov 2017)*

After finding a pair of ciphertexts created with the same key, we can retrieve xor of the original plaintexts. If we then need to guess the contents of one plaintext to find the contents of the other. To do this, I implemented a method known as crib-dragging, in which the user substitutes words into a guessed plaintext at different positions, and see what the other plaintext would look like.

My application allows the user to enter a string, then for each position that string could be inserted, it calculates the resulting contents in the other plaintext. It does this by XORing the input string with a equal-length substrings of the XOR of the two ciphertexts from different positions.

3.3 Identifying promising insertion positions

After performing crib dragging with a string, my application analyses the output string for each insertion position and figures out which ones could be substrings of a valid English plaintext.

To do this I first replace any valid punctuation with spaces. This classification takes into account which symbols require leading or trailing spaces, and whether those spaces could have been cut-off at the edges of the string.

Next, I split the string into words based on groups of spaces and other word-boundary characters such as hyphens. I match each word against a dictionary (see section 2.6). If there are words up against the start or end of the string, the code checks if they are the end or start of words in the dictionary, and if the string is entirely made up of one word, the code instead checks if it is a substring of any word in the dictionary. All strings are converted to lower-case before comparison. Finally, any word that can be converted to a float is also deemed valid, to allow for numbers (even with decimal points) because they are not included in the dictionary. If all words in the string pass the matching process, then it is deemed a likely English substring.