The Vigenère Cipher

Avi Stein

17 December 2016

COMP 112 – 02

**Abstract**:

This project presents both a problem and two solutions to it. In 1553, the Vigenère Cipher was invented, and was considered an unbreakable code for over three hundred years [1]. This code was simple enough to implement, but incredibly difficult to break by hand. Because of this, the Vigenère code was used for bank passwords and even as an encryption for military secrets during the Civil War. For this project, I built and subsequently broke the Vigenère code in two different ways. I used a classic version of the cipher, rather than a variation, and created a brute force attack and a dictionary attack against it. This project allowed me to gain deeper insight into the complexity of the foundations of cryptography and information theory.

**Introduction and Background**:

Probably the most well-known cipher ever created was the Caesar cipher, where each letter would be mapped to one other letter. The assignment between the two letters depends on a number, which shifts the letter by that number in the alphabet. This cipher is weak in a couple ways, but primarily because a brute force attack is an incredibly viable attack against it, and the code would be broken in a second at most. (Another possibility would be a frequency analysis, which will be discussed later).

However, despite being only a step up from a Caesar cipher, the Vigenère cipher can withstand a brute force attack from modern computers. The cipher takes two inputs: a sentence or word message, and a key word. The first letter in the key word is converted to a number (“a” = 0, “b” = 1, etc.) and a Caesar cipher is applied to the first letter in the message to be encrypted. The second letter in the key word is then converted and applied to the second letter, and so on. If the key word is shorter than the message to encrypt, then the key is repeated (e.g. LEMONLEM if LEMON was the key word).

The simplicity of this pattern fascinated me, and really epitomizes the ideal nature of security. It’s very easy to lock a door from the inside, but very hard to unlock the door from the outside without a key. In this case, the lock mechanism would be the design of the encryption protocol.

I first heard of the Vigenère cipher when I was in middle school, and thought “I bet this isn’t too hard to crack, but I probably need to know how to program to do it.” I was somehow very right and very wrong at the same time. Knowing how to program has given me a better appreciation for the design of such a simple code and how complex it can rapidly become. The two methods I used to break the code were probably the two most straight forward attacks, and they were used for even the most simplified cases I could come up with.

The three goals I wished to achieve in this project were fairly simple. First, build the classic version of the Vigenère cipher, where each letter in the key maps to each letter in the message. Second, build a brute force attack that will break the encryption. Third, build a dictionary attack that will break the encryption. I also made sure there were very specific inputs. First, everything must be in lowercase. There can’t be any spaces or capital letters or punctuation or special characters. Second, the message can only be one word from the dictionary (the dictionary I used was the one stored deep inside all Unix machines). Third, the key word must also be one word from that same dictionary, of equal or shorter length. I defined these parameters and restrictions before I set out to break this code because it would become very difficult to impossible when variations start showing up (to be discussed later in the information theory part of this paper).

**Methods**:

The methods for these three varied considerably in terms of their difficulty. The easiest by far was writing the encryption code, which I had expected. Next difficult was the dictionary attack (which I would have expected to be the most difficult because it requires a little more thought). Most difficult was the brute force attack, for unforeseen reasons.

The first goal of encryption was a combination of two functions. The first function we wrote in lab, where it takes a word, figures out the ASCII number, and shifts it by one. This program used flow control, lists, bools, and a for loop.

The second function used the first function to operate over a list (pulled from a string). This could have been done with a map function, but this current function is much easier for me to read and debug, and I also realized that I could have done it with a higher order function just now. This function also features a for loop, flow control, and a dictionary (the data structure) to convert letters and numbers.

The second easiest goal, the dictionary attack, was more involved. It used lists to contain the potential hits, before opening up the for loops. The first for loop opens a dictionary file of the appropriate size length (that it is able to detect) and begins working its way through it. Within this for loop, lies another for loop that opens up a second dictionary. This dictionary, which only contains words of a certain length, begins working its way through as well. This second dictionary is of variable length (using string formatting), so for the first iteration, it opens a dictionary containing one-letter words, for the second iteration, it opens a dictionary containing two-letter words, etc. Inside the deepest layer of these nested for loops, there lies the test. It takes the shorter word (the key) and encrypts the longer word (the message) with it. If this matches what the encrypted message reads as, it considers this situation a hit, and writes the message and key to a list. If not, it continues to the next word in the key dictionary. In this way, the function will test every word that is shorter or equal length to the message against every word that is the length of the message.

This strategy may have come to me more quickly, because it is almost the same as a technique I use in my research. Numerical (or in this case, relative brute force) solutions to problems are certainly a way (and sometimes the only way) of solving problems, but are rarely the most efficient way.

The hardest goal was by and away the least efficient way of solving this task. It is sometimes a good benchmark to compare other algorithms to and marvel at the power of modern computing. A full brute force attack like this grows exponentially, and becomes unwieldy very quickly.

This attack hinged on a difficult conceptual problem: being able to cycle through possibilities of letters like an odometer reads out on a car. The linchpin for this problem is that the length of the word changes, or in the odometer comparison, the number of digits, so nested for loops cannot be laid out ahead of time. This problem could probably be written using a combination of recursion and iteration that would force the function to retreat back to its base case after every step of a higher digit.

I was not able to solve this problem. My solution instead (which I would like to credit Zach Berkowitz of the SCIC with having the original idea and helping me flesh it out) was to have the function cycle through numbers instead of letters, because it natively knew how to do this. To be able to connect the words with numbers, I essentially created a big number in base-26. This number can quickly be calculated using the dictionary structure that converted letters to numbers and back. Each letter would be raised to a separate order of magnitude (in base-26) based on its position in the word. This way, each number would generate a discrete word (e.g. 0 = “a”; 26^5 = “zzzzz”) and I could use a regular for loop between 0 and the highest possible number for the length of the encrypted word. Factoring the word uses the exact same principles and is able to describe the discrete word. There is also a function to fill in the space where leading zeroes would be in the numbers. This allows the message to cycle starting at “aaaaa” instead of “a”, and the keys to start at “a” instead.

The rest of the function is very similar to the dictionary attack, since they are both of the same brute-force class.

**Results**:

The results of the attacks were measured by time, because they came up with identical answers, and because time is actually more relevant than accuracy here. To measure the time, I found a method that will start a timer at the beginning of the code and stop it at the end of the code. The time of the encryption portion was not included in this table because for any message/key with any length, its run time was negligible.

The speed at which each of these attacks is able to process through possibilities is also telling of both my ability to code things well, and the complexity of the two tasks. The brute force attack processed at a rate of 7,551 guesses per second. The dictionary attack processed at a rate of 261,468 guesses per second. Here are the results for various challenges:

|  |  |  |
| --- | --- | --- |
| **Length of Word** | **Brute Force Attack** | **Dictionary Attack** |
| 1 | 0.09 seconds | 0.01 seconds |
| 3 | 11.8 hours | 8.863 seconds |
| 7 | 150 x age of the universe | 1.48 hours |
| 10 | 6.3e6 x age of the universe | 4.99 hours |
| 24 | 6.3e67 x age of the universe | 4.5 seconds |

Table 1: Run times for various word lengths.

These results are quite stunning. This quite conclusively proves the inefficacy and irrelevance of the pure brute force method in modern cryptography. Even on a supercomputer, this method would only work on particularly short messages, or weak encryptions. Considering this encryption was built in 1553, not too many encryptions will crack under a brute force attack.

I clearly did not run all of these results; I only ran the ones that are reasonable to run. The rest of these calculations are extrapolations. For example, the run time for a dictionary attack on a 24-letter word is not actually 4.5 seconds, but closer to 39 seconds because of the time it takes to maneuver through the function structure. Still, these results are very representative, in that 4.5 or 39 seconds are only one order of magnitude different, and the two attacks for the same length differ by *85 orders of magnitude*.

The reason why the brute force attack increases so quickly is because it is running the number of possibilities for the longest word times all the possibilities that come before. This exponentially increases. The reason why the dictionary attack drops off after 10-letter words is because of the same reason. There are just fewer really long words, so the sum of all the previous words are being multiplied by an increasingly small number.

**Discussion**:

This project built and then cracked an encryption code with two different methods. The topic of this project has nothing to do with anything we learned in class, but the implementation was (mostly) based on skills we learned in class.

Some of the more important skills that I learned in class that I applied to this project were lists, for loops, flow control, and file management. It is hard to name all of the skills that were used in this project because some (like splicing) are very small components, but we did spend 15+ minutes of class talking about them. These skills were the bases of the structure for the program, as I described in the methods section.

This project brought about no advancements in any field of cryptography, because the weakness of the brute force technique has been known at least since the first computer when Charles Dances’ character tries to shut down Alan Turing’s computer in Bletchley Park for being too slow (in *The Imitation Game*) [2]. It has also been noted in the cinema that the Vigenère cipher is a very strong cipher and quite resistant to attacks (as Hugh Jackman’s character found out in *The Prestige*) [3]. In this particular case, Hugh Jackman’s character performed a different type of brute force attack to break the encryption, namely a hostage negotiation in exchange for the key.

I described one method of writing the encryption function differently and more concisely, but there are generally better and more interesting ways of breaking the encryption. The first general attack that succeeded in 1863 was the Kasiski examination, which was based on the idea that over a period of a sentence or so, the key word may repeat and patterns would arise [4].

Another potential method of breaking an encryption (more difficult for this encryption) is through the use of frequency analysis. Frequency analysis acknowledges that some letters are typically used in the English language more often than others (“e” being used around 13% of the time). With a large enough sample size such that a uniform pattern emerges (as mentioned in the Kasiski examination), a frequency analysis could be a viable option and is considerably faster. However, a frequency analysis would most likely partially crack a code (decrypting only a few letters in the message) and require a human or intelligent computer to piece together that last of the puzzle pieces. I will touch on this again in the conclusions.

**Conclusions**:

Earlier, I pointed out that sometimes the run time is more important as a metric than the accuracy of the function. This project is one of those times. For a give two letter word, the dictionary attack will return 26 possible messages and their corresponding key words. This is certainly not perfect (although since both attacks are brute force class attacks, they return the same results), but runs on the order of one second, so thoroughness is not everything. This tradeoff happens with more sophisticated functions, that use patterns in the encrypted message to disentangle the message. If the encrypted message does not have the patterns, they are useless, but at least they did not take the age of the universe to run. Also, it must be noted that as the length of the word increases, the ratio of actual words to combinations of letters drops off precipitously, and the likelihood of a false positive quickly drops to near zero.

Frequency analysis and Kasiski examinations fall under the category of more efficient, but less fool-proof methods of solving this problem. If the length of the message is too short, or the key is just very long, it would likely not yield any legitimate results. In order for it to work, it would need sentences or paragraphs to crack or partially crack (which would be outside the scope of this project). With a partially cracked message (e.g. the message “TxE” where “T” and “E” are decrypted and “x” is not yet), it can be easy to guess the word as a human (“THE”). Computers with more linguistics and information theory knowledge could also work on this task.

Another problem that I completely avoided arises from using more than one word. If the message involved more than one word, parsing becomes a critical problem. For example, if a portion of a message translates to “xxxxTHATxxxx”, the message could contain the words “THAT”, “HAT”, “AT”, “HA”, and/or “A”, not to mention combinations of words from the unbroken portion of the code. The results are very context dependent, which is something humans are decent at, and computers are not very good at without being taught.

To top off everything, another level of complexity can come from opening up the available characters to work with. I chose to use only the lowercase English dictionary. With upper case and special characters (this is just in English), the run time of the brute force attack for a two letter word jumps to over a month. The more advanced techniques suffer even more complex parsing problems. This is a very good reason to include a number or punctuation in your password.

These are just a few possible variations to considerably strengthen the Vigenère cipher that was invented in 1553. Other possibilities involve using symbolic representation, like the Navajo code talkers during WWII [5], or encrypting with visual obscuration like CAPTCHA aims to do [6]. However, more modern cryptography relies heavily on RSA encryption and prime factorization algorithms which are incredibly computationally taxing, and are of an entirely different class of encryptions methods.

This project allowed me to pursue and solve (mostly) a problem I had been puzzling over for a long time, and to peak into the world of cryptography. This project has also stretched my coding abilities and forced me to be more creative in the face of difficult problems.

**References**:

[1] Bruen, Aiden A. & Forcinito, Mario A. (2011). Cryptography, Information Theory, and Error-Correction: A Handbook for the 21st Century. John Wiley & Sons. p. 21. ISBN 978-1-118-03138-4.

[2] "THE IMITATION GAME (12A)". British Board of Film Classification. 15 September 2014. Retrieved 6 November 2014.

[3] "The Prestige (2006)". British Film Institute. Retrieved June 13, 2014.

[4] Kasiski, F. W. 1863. Die Geheimschriften und die Dechiffrir-Kunst. Berlin: E. S. Mittler und Sohn.

[5] Vogel, Clayton; Johnston, Philip. "Letter to Commandant, USMC". Demonstration in California, 1942. Northern Arizona University, Cline Library. Retrieved July 20, 2011.

[6] von Ahn, Luis; Blum, Manuel; Hopper, Nicholas J.; Langford, John (May 2003). CAPTCHA: Using Hard AI Problems for Security. EUROCRYPT 2003: International Conference on the Theory and Applications of Cryptographic Techniques.