**CEG 4750-01: Information Security**

**Amina Haq**

**David P. Wilson**

**Jean Pauline Arcita**

**Dr. Meilin Liu**

**29th March 2021**

**Project 3 Report**

# Introduction

Our goal for this project was to develop a program that could perform a brute force attack to break the encryption on a set of provided files encoded with AES. We were provided the last 96 bits of the 128-bit key, so our program only had to find the first 16. We selected our test files based on a modulus operation on one of our member’s University ID (UID) numbers. The rest of this report will describe the input to this task, the work itself, and the results.

# The Input

We chose to use the UID number belonging group member David Wilson. To determine which test files we should use for the program, we took the last four digits of David’s UID as an integer: 3925. We then performed a modulus operation on this integer and its reverse-ordered sibling, 5293. The results are shown in Equations 1 and 2.

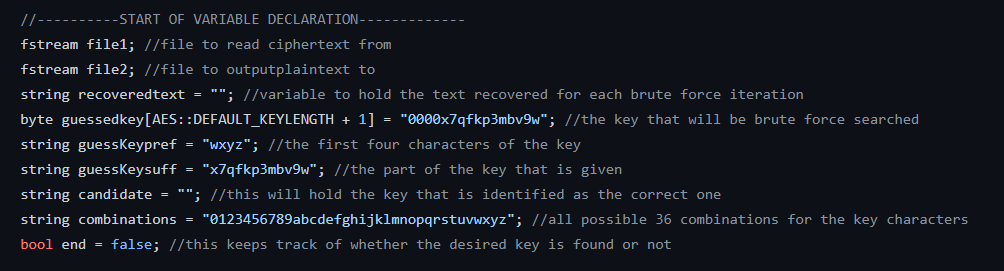
|  |  |
| --- | --- |
|  | (1) |
|  | (2) |

As a result, we chose to use test files 02.e and 12.e in our program.

# The Code

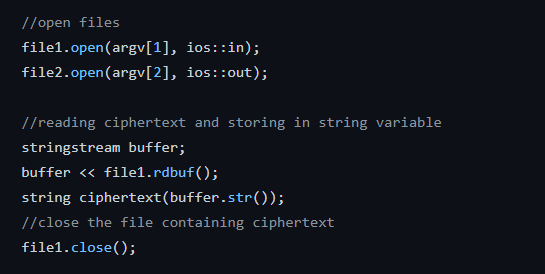
We used aes\_decode.cpp that was given with this project to implement brute force functionality.

Figure 1 contains variable declarations that we needed to implement functionalities to satisfy the requirements for this project.



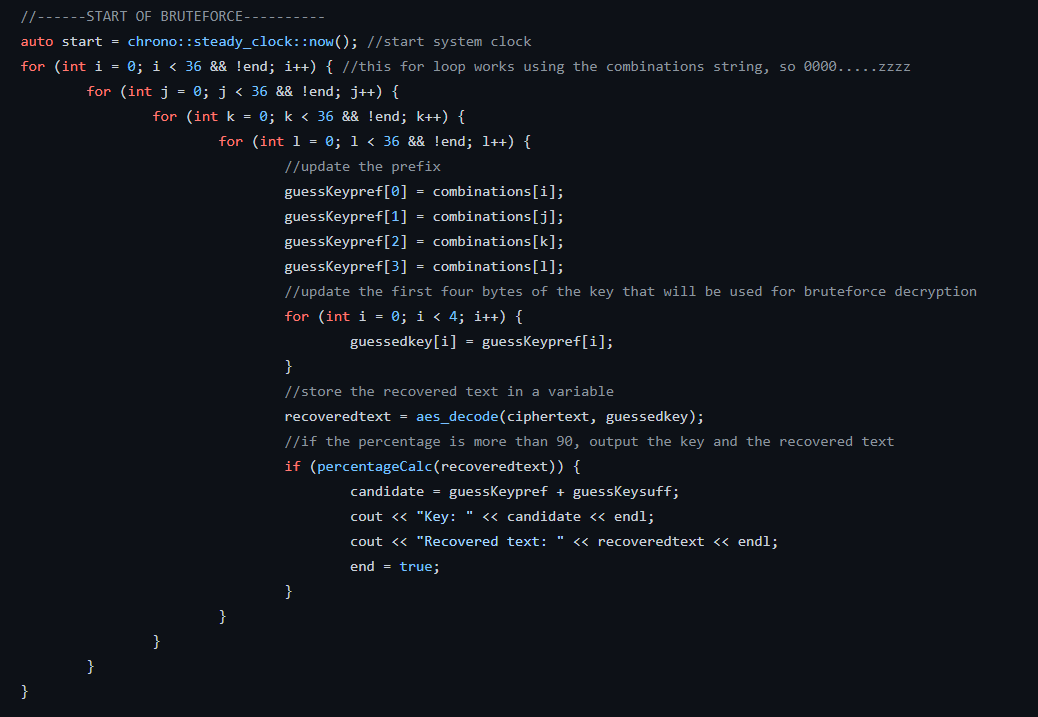
**Figure 1: Declaration of variables**

Figure 2 contains opening encrypted file and the file that will contain recovered text. The last line of code contains “file1.close” was done to minimize the number of disk I/O.



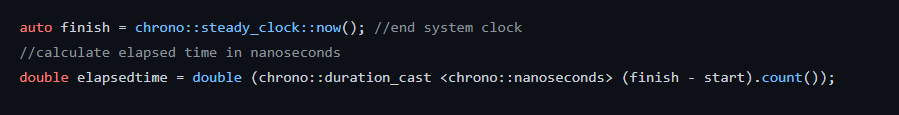
**Figure 2: File Input/Output**

Figure 3 shows how exhaustive key search was implemented. First line of code is for the bonus points to record the running time of our program to find the correct key. Our nested for loops were used to check every character that we will use to brute force key. Each for loop goes through 36 times because this is the number of characters that will be used to implement exhaustive key search. Once we have a key that we want to try to decrypt the file, we will pass it into aes\_decode function to decode it. After that, the recovered text will be passed into percentageCalc function shown in Figure 5 to check whether the recovered text file reached at least 90% of standard alphanumeric English characters or basic punctuation. If the recovered text does not reach 90%, then recovered text is not the original plaintext and we have not successfully retrieved the right key. If it reaches 90%, then it means that we have recovered the key. Given this, we will save the candidate key, display the recovered text, and end the loop.



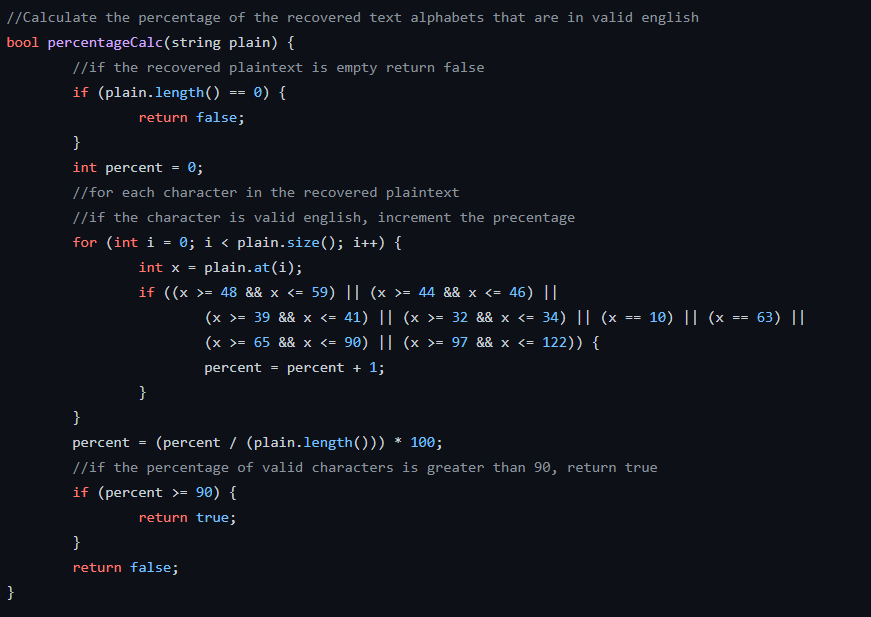
**Figure 3: Implementation of Exhaustive Key Search (Brute Force)**

After the nested for loop, we stop the system clock and compute the elapsed time to recover correct key as shown in Figure 4.



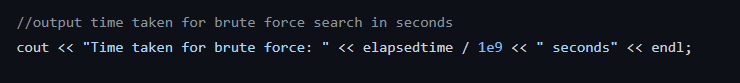
**Figure 4: Stopping system clock**

Figure 5 contains the code to calculate the percentage of recovered text based on standard alphanumeric English characters or basic punctuations. We have implemented a for loop to iterate through the recovered text per character and checks if it satisfies the following ASCII codes. These ASCII codes represents alphanumeric English characters or basic punctuations. The for loop adds 1% every time it falls under specified ASCII codes. We calculate the percentage and return true if it reached 90% and false if it does not.



**Figure 5: Percentage calculation of recovered text**

Figure 6 displays and converts the elapsed time to seconds to satisfy the bonus points given for this project.



**Figure 6: Display and conversion of elapsed time**

# The Work and the Results

Our program was designed to iterate through all possible combinations of the 16-bit subkey from “0000” to “zzzz.” Each time it did this, it would scan through the output to determine the percentage of the results that could be interpreted as standard alphanumeric English characters or basic punctuation. When this percentage reached at least 90%, we would stop and print the output. The results of these operations are shown in Figures 7 and 8.

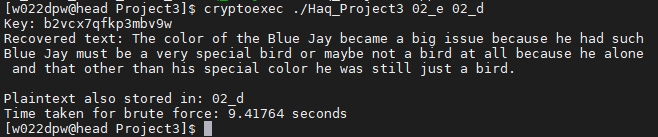


Figure 7: Decryption of the test file 02.e

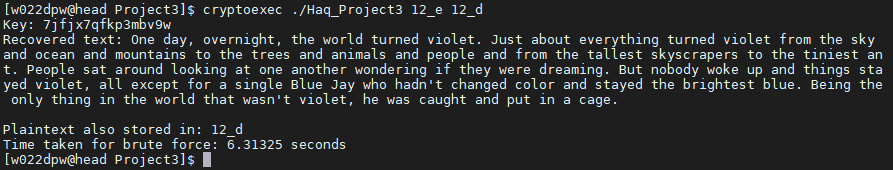


Figure 8: Decryption of the test file 12.e

As the figures show, both files were fully decoded into proper English. The sub-key for 02.e turned out to be **b2vc**, while the sub-key for 12.e turned out to be **7jfj**. We also completed the bonus object by calculating the elapsed time it took to find these keys via the brute force method. For convenience, these values will be recounted in Table 1.

Table : Complete statistics for test file decoding

|  |  |  |
| --- | --- | --- |
| **Test File** | **Key** | **Elapsed Time** |
| 02.e | b2vcx7qfkp3mbv9w | 9.41764 seconds |
| 12.e | 7jfjx7qfkp3mbv9w | 6.31325 seconds |