**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 the aes\_decode.cpp file 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.

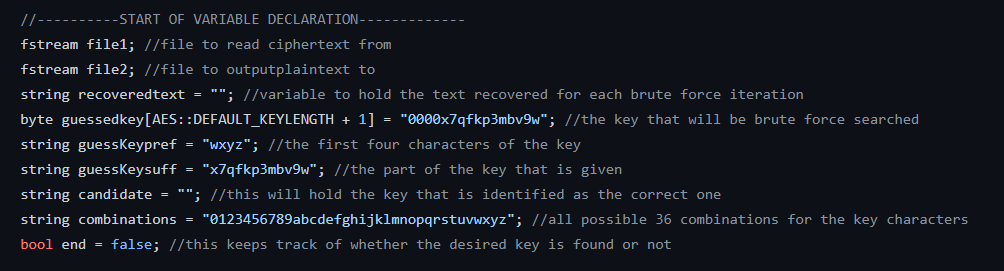


Figure 1: Declaration of variables

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

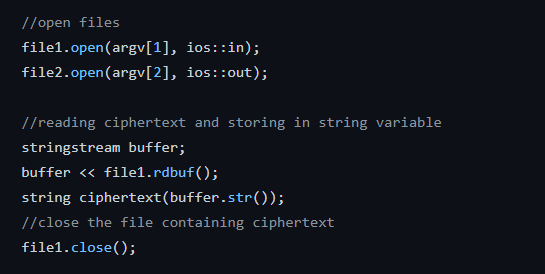


Figure 2: File input/output

Figure 3 shows our implementation of exhaustive key search. The first line of code was for the bonus assignment to record the running time of our program. Our nested for loops iterate through characters to find the key pattern. Each for loop goes through the combinations string 36 times because this is the number of valid characters for the key. Once we have a key, we will pass it and the ciphertext to the aes\_decode function. After that, the recovered text will be passed to the percentageCalc function to check whether at least 90% of the recovered text file represents standard alphanumeric English characters or basic punctuation. If the recovered text does not reach 90%, the program decides it is not the original plaintext and we have not successfully retrieved the key. If it does reach 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.

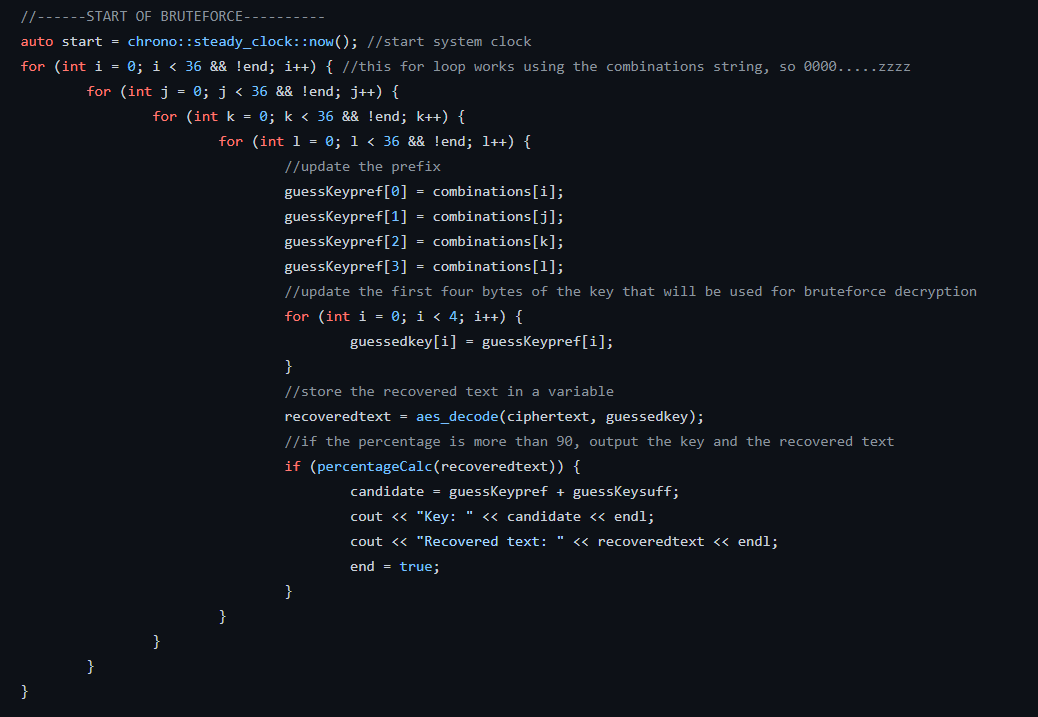


Figure 3: Implementation of exhaustive key search via brute force

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

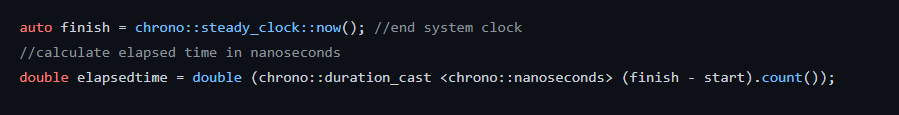


Figure 4: Stopping the system clock to facilitate counting the elapsed time

Figure 5 contains the code that calculates the percentage of recovered text representing standard alphanumeric English characters or basic punctuation symbols. We implemented a for loop to iterate through the recovered text per character and check if it satisfies certain ASCII codes corresponding with the valid characters and symbols. The for loop adds 1 percentage point every time a symbol falls under the specified ASCII codes. When finished, it calculates the percentage and return trues if it reached 90%.

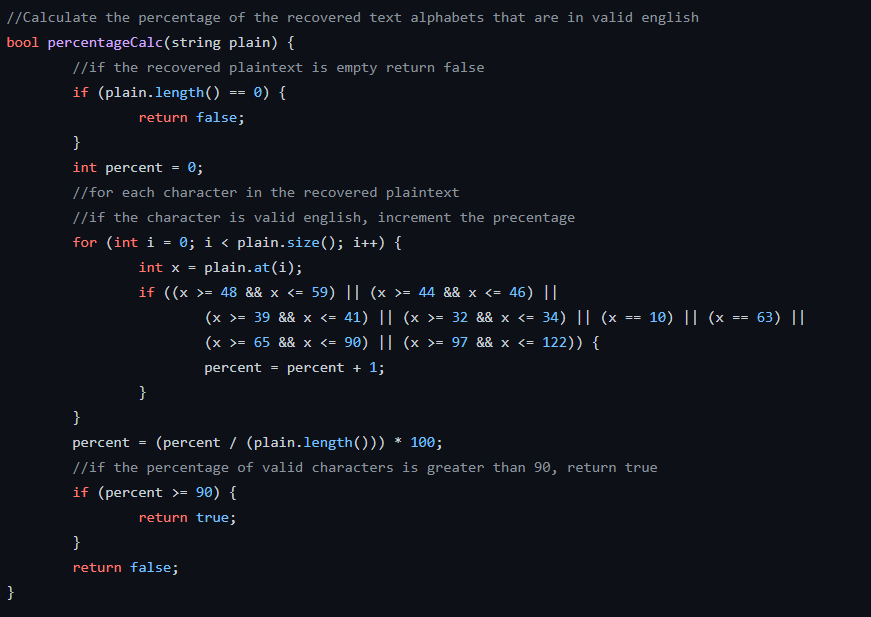


Figure 5: Analysis of the percentage of recovered text that represents standard English characters

The code presented in Figure 6 displays and converts the elapsed time to seconds to satisfy the bonus point requirement given for this project.

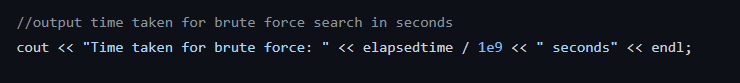


Figure 6: Display and conversion of elapsed time

# The Result

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 Figure 7 and Figure 8.

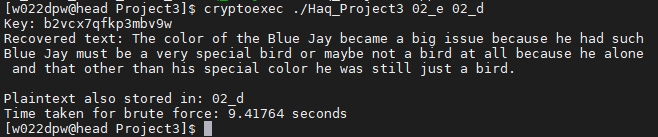


Figure 7: Decryption of the test file 02.e

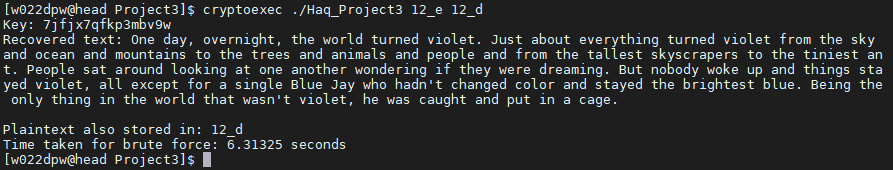


Figure 8: Decryption of the test file 12.e

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 1: Complete statistics for test file decoding

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