COMS W4995 Design Using C++ Design Document: Cryptanalysis of Encrypted Text Using Concurrency in C++

Swapnil Paliwal (sp3911), Arun Ram-Mohan (amr2356)

Overview

Preliminary

We expect readers to have some prior knowledge about how the Vigenère cipher and Enigma cipher (both commercial and military versions) work. We recommend that readers review [1], [2], and [3] before proceeding if they have no prior knowledge.

Inspiration

CRYPTOGRAPHY AND CRYPTANALYSIS

1. **Vigenère cipher:** Invented by Blaise de Vigenère, was one of the most influential and secure encryption algorithms. The cipher was considered to be unbreakable for 300 years [11]. It was referred to as le Chiffre indéchiffrable (undecipherable cipher).

   The core strength of the algorithm was its invulnerability to frequency analysis. Russians most widely used a version of the Vigenere cipher in WWI [1].

   Much credit is accredited to Kasiski for breaking the Vigenere cipher. However, Charles Babbage was the first one to derive and use it in modern times. However, it is argued that there were cryptographers in the past who applied a similar strategy to break the cipher [6]. The Kasiski examination deploys a method for finding matches in the ciphertext and finding periods of those matches. Adhered to this, the technique is to deploy frequency analysis within the given period.

   The cryptanalysis algorithm designs [6, 13] inspired us. Thus, we aimed to deploy a method that requires no complex logic in cryptanalysis. The focus was to develop a push-and-play algorithm where a user just provides a ciphertext and the program returns the plaintext. The logic behind the algorithm was inspired by quadgram analysis [3].

2. **Enigma:** Arthur Scherbius, while working at Scherbius & Ritter, developed an electrical circuit of Alberti's cipher [10, 12], which we now know as the Enigma.

   The system, according to many, is considered a fearsome system of encryption in history. It was, no doubt, the most historical cipher of great importance in the world [6].

   The Enigma machine had three rotors, a reflector, a plugboard, an input QWERTY keyboard, and lamps that reflected the cipher characters [12].

   The only difference between the commercial and the military Enigma was the plugboard [2]. In his initial design, Arthur proposed that six plugs out of a total of thirteen be used.

However, the plugs were upgraded to ten during the war. For ten plugs, there were "150,738,274,937,250" possible combinations. Making it impossible even for modern computers to break.

Marian Rejewski, before Alan Turing, broke the Military version of Enigma with three rotors and six plugs, which were in use till 1939 [14]. However, the added complexity of the ten plugs and other details approached Rejewski's machine's limitation. Alan Turing and British intelligence had an advantage now because of Rejewski's work; they saved much time. Alan Turing deployed an attack that we now know as a known plaintext attack [5, 15, 8, 9]. Thus, breaking the cipher and helping British intelligence to tap into all the communications over the channel.

Stuart Milner-Barry (a cryptanalyst at Bletchley Park) wrote: “I do not imagine that any war since classical times, if ever, has been fought in which one side read consistently the main military and naval intelligence of the other." [6]

We were inspired by the approach Marian Rejewsk [14]i, and Alan Turing [15] deployed during WWII. We brainstormed and aimed to generate a robust algorithm that could have been used to approach the problem. We combined the learnings from Turing [15] and Marian Rejewski's work [14] and deployed a known-plaintext attack that works well [8, 9]. Although, no cryptanalysis attack guarantees a 100% success rate. We claim that our approach works for myriad cases to make it practical.

We were also motivated to test if multiple settings could yield the same results. We were confident that multiple settings could be found because of the Enigma machine's flaws [12], and our hypothesis was tested to be confirmed by [7]. We prove our results on the Enigma machine portrayed by [4], we extend this machine to a military setting.

PROGRAMMING

Given such an open-ended project assignment, we were eager to apply our background knowledge to create an ambitious program relevant to our interests, which could hopefully also capture the interest of others due to the historical significance of the Vigenère and Enigma ciphers to the two World Wars of the 20th century, as discussed above.

At the same time, we also wished to apply techniques from modern C++ from the 21st century, which we have been learning in class, to these problems.

From the class lectures, and from the text *A Tour of C++*,[16] we have learned about: the benefits of designing code around class hierarchies; making use of standard libraries for containers, algorithms, and concurrency; and techniques for designing applications in a modular, object-oriented fashion without adding runtime cost to algorithms or overcomplicating the storage of data.

So, because of the nature of the cryptanalysis problems we wanted to try to solve, we were motivated to use whatever techniques we could from modern C++17 to help us arrive at workable solutions.

Project Goals

1. Implement Vigenère cipher
2. Test and validate the code
3. Implement commercial Enigma
4. Implement military Enigma
5. Test and validate both commercial and military Enigma code
6. Develop a scoring function for the hill-climbing cryptanalysis algorithm - Commercial Enigma and Vigenère cipher
7. Test the cryptanalysis code with the ciphertext
8. Develop a novel logic to the cryptanalysis of the military Enigma cipher
9. Test the code with the known ciphertext
10. Develop a scoring function for the known-plaintext cryptanalysis algorithm
11. Implement the known-plaintext attack and test it against the ciphertext of the known plaintext
12. Combine cryptanalysis algorithms for Vigenère, commercial Enigma and military Enigma ciphers in some elegant way into one C++ program
13. Given an unknown ciphertext of indeterminate origin, find a way to determine which cipher was used to encrypt it, and obtain the original plaintext
14. Create the basic foundation of a cryptanalysis program that can possibly be extended to solve other ciphers beyond the three indicated above
15. Use modern C++17 to find good solutions to all of the above problems

The Vigenère Cipher

Our Approach

The `vigenere.h` file focuses on implementing the cipher. Further, the header file implements a cryptanalysis function accessed by the `cryptanalysis.cpp` file.

To encrypt a message using Vigenère cipher [1], we implement the function `encryption` that takes the character's ASCII value and the key (ASCII) and encrypts the message. Thus, we are working with the integers.

```
int encryption(int plaintext_ascii, int key) {…}
```

The key can shrink or expand based on the length of the plaintext. Thus, to accommodate this, we implement the `key_update` function that sets the key equivalent to the plaintext length.

```
string key_update (string key_for_enc_n_dec, int plaintext_length) {...}
```

The encrypting function is responsible for performing encryption that results in ASCII values. The program outputs the ASCII values in the form of characters used by "`constexpr char int_to_char`."

```
constexpr char int_to_char [26] = {...}
```

The cryptanalysis function is a robust state-of-the-art algorithm that enables a user to crack a Vigenère cipher that would require billions of years to break via brute force in just seconds.

The algorithm assumes the key as "AA." and attempts 26 possible combinations for first place, and updates the setting where the score is highest. It repeats the same for the second location.

```
      VigenereText(string t, size_t l): encrypted{key_update(t,l)},
decrypted{key_update(t,l)}, length{l}, key{"AA"},
score{min_quadgram_score} {}
```

Note: For the scoring function, we are using quadgram analysis[3]. In theory, if the correct key credentials are guessed, the score via quadgram will be the highest. The scoring basis is from the precomputed quadgram file, which is imported using `#include "quadgram_analysis.h"`.

Once the two key values are fixed, the algorithm adds the third key character and assigns it the value of "A" (`i=0`) and repeats the same process for the third character.

```
string decryption_key_attempt (VigenereText &ciphertext, const int loc)
{
      …
      std::future<double> score =
      std::async(decryption_key_attempt_score, ciphertext, loc, i);
      …
      int best_score_index =
      std::max_element(std::begin(decryption_score_list),
      std::end(decryption_score_list)) -
      std::begin(decryption_score_list);
      ...
      return_key[loc] =  int_to_char[best_score_index];
      …
}
```

Thus to guess a 10 character long key, the algorithm must only employ 260 guesses. In contrast, brute force would require checking 141,167,095,653,376 combinations.

Key requirements that influenced design decisions

1. The encryption and decryption algorithm's input had to be converted to an integer and converted back to English characters after encryption.
2. The key size had to be adjusted based on the input message. If the key is shorter than the message, the key characters must be repeated till it covers the entire message, and if the key is longer than the message, it must shrink.

3. The scoring function had to access the qgram array that comprises the scores of the `quadgram`.
4. A logic for cutting scores in the cryptanalysis function had to be constructed. We deployed a `basetext` that was responsible for adjusting the length of input text. In theory, the two English texts of the same lengths will have scores in certain boundaries. A scoring function in the milieu of this logic had to be constructed.

The Commercial Enigma Cipher

Our Approach

The enigma.h file implements the version of Enigma delineated by [4]. The [4] reflects a commercial version of the cipher; however, we add the plugboard to the same design and convert it to the military design Enigma machine, as the only difference between the two is the plugboard.

In Enigma, the encryption and decryption algorithm is the same.

```
short EnigmaText::rotor_direction_output(const short
input_output_char_loc) {...}
```

The commercial Enigma sets the rotor and ring settings and proceeds further with the encryption or decryption process.

```
void EnigmaText::initialize(const string key, const string ring_setting)
{
      …
location_on_initialization_array=character_location(rotor_one.output_cha
racter, ring_setting[0]);

rotor_shift(rotor_one.output_character,location_on_initialization_array
);
      …

}
```

Before encryption, we request the user to provide us with the rotor and ring settings. Once we have the credentials, we proceed with the initialize function. Once the machine is set to the correct rotor and ring settings, we need the user's message (`decrypted` function is responsible for that; used by the `enigmaEncryption.cpp`).

The encryption function is responsible for shifting the rotors [4, 5] and returning the ciphertext.

```
string EnigmaText::encryption_decryption(const bool decryption) {...}
```

The encryption function also employs the rotor_direction_output function, which flows through the machine and outputs the ciphertext character, character-by-character.

```
short EnigmaText::rotor_direction_output(const short
input_output_char_loc) {...}
```

The cryptanalysis function utilizes ciphertext-only attack. Before we use the cryptanalysis function, we need pre-processing. For this, we derive motivation from Marian Rejewski. We pre-store the values of all the rotor and ring settings, i.e., 17,576 (ignoring redundancy by using the same storage data structure; array), and then proceed with the cryptanalysis function.

```
bool EnigmaText::cryptanalysis() {
      …
      for(int i=0;i<26;++i) {
            for(int j=0;j<26;++j){
                  for(int k=0;k<26;++k){
                        all[l] += all[l] + alphabet[i] + alphabet[j] +
alphabet[k];
                        ++l;
                  }
            }
      }
      ...
 }
```

The cryptanalysis function like Vigenère cipher relies on quadgram sequence analysis. To approach 308,915,776 possible combinations, we deploy a novel strategy. We fix a ring setting and then proceed by altering different rotor settings. We check the score for each one of them and update the base scores. Now, we are looking for settings that get us the highest score. We look for three such events when the score is constant and then return the result to the user to ensure we have the correct settings.

```
bool EnigmaText::cryptanalysis() {
      key = all[j];
      ring_setting = all[i];
      …
      …
      if(...) {
            rotor_one.input_character[0] = key[0];
            ...
            rotor_two.output_character[0] = ring_setting[0];
      }
}
```

Key requirements that influenced design decisions

1. Machine initialization had to focus on setting the rotor and ring settings based on the input. The code had to move the virtual rotors.
2. The reflector, rotor, ring credentials had to be hardcoded, and the entire program had to be written around these static structures.
3. The encryption function had to reflect the design and feel of the actual Enigma machine. We aimed to create an actual virtual replica of the device.
4. The cryptanalysis function had to try 17,576X17,576 combinations of rotor and ring settings. However, with each setting, the quadgram score had to be tested, as done for Vigenere.

The Military Enigma Cipher

Our Approach

The problem is a bit different for military Enigma, and there is a plugboard that adds the most complexity during the cryptanalysis and implementation.

The `enigma_plugboard.h` file has two functions: implementing the cipher and constructing the cryptanalysis function.

The rotor, ring, and message input operations remain the same as the commercial Enigma. However, the plugboard code is added in the input in the form of two characters, i.e., plug input and the end input. The plug input is where the first plug enters, and the end input is where the end of the first plug is placed [2, 5].

```
void EnigmaPlugboardText::read_plugboard() {
	…
	for(...) {
		…
		cin>>loc_holder[0];
		…
		cin>>loc_holder[1];
		…
	}
}
```

The message input passes through the plugboard, and characters are substituted, and then we encrypt these characters. Once the message is encrypted via the `encryption_decryption` function (like before), the output is passed through the plugboard, and the ciphertext is rendered on the screen.

```
void EnigmaPlugboardText::encrypt(const string key, const string
ring_setting) {
	…
```

```
	for(...){
		encrypted[i] =
sub_list[character_location(alphabet,encrypted[i])];
	}
}
```

The cryptanalysis function relies on the known-plaintext attack. Thus, the message once provided by the user is stored in the `knownPlainText.txt` file.

```
void read_decrypted() {
	...
	ofstream kptfile(kptfilename);
	if (kptfile.is_open()) { kptfile << decrypted; kptfile.close(); }
	...
}
```

The cryptanalysis function relies on the scoring function, which is a known plaintext function. The cryptanalysis function after obtaining right settings proceeds with guessing the plugboard settings. The function `location_of_best_loc` is used to derive the two max scores.

The scores are stored in the `tracking_one` and `tracking_two` vector, which forms the basis of two sets of possible plugboard combinations that yield the correct results.

```
for(int i=0;i<26;++i) {
	current_sub_loc = location_of_best_loc(i,-1);
	if(current_sub_loc[0]!=0 || current_sub_loc[1]!=0)
tracking_one.push_back(current_sub_loc);
	current_sub_loc = location_of_best_loc(i,current_sub_loc[1]);
	if(current_sub_loc[0]!=0 || current_sub_loc[1]!=0)
tracking_two.push_back(current_sub_loc);
}
```

Key requirements that influenced design decisions

1. Plugboard had to influence the input and the output to and from the machine. The shuffling of characters required careful alterations. Any unwanted changes could harm the correctness of the ciphertext.
2. The cryptanalysis function needs a known plaintext to work on cryptanalysis. Since the encryption and the cryptanalysis files were separate, we had to derive a logic where the text provided as an input to the military Enigma file could be taken as an input for the cryptanalysis file.

3. The fresh logic for the cryptanalysis function had to be derived that could work for military Enigma. We visioned a similar approach to cryptanalysis as we did for the commercial Enigma. We guess the rotor and ring settings. The new approach relied on the assumption that if the rotor and ring settings are correct. It is the case that the score will be the highest. Further, the only thing the plugboard cryptanalysis function had to do is try manual combinations from q->m, w->m, e->m,r->m, and so on. Deriving the logic and converting the code to fit that logic into an efficient code required many manual iterations and further structural analysis.

Structure of the Project

The three ciphers discussed above, while significantly different from each other in terms of solution methods, represent programming problems which have several major features in common:

- Each takes as input a text string, of which each character is one of 26 capital letters.
- Each uses a scoring algorithm to progress toward solutions with best scores.
- Each returns several text strings as output, including the decrypted text and the keys/machine settings used.

Therefore, we thought it made sense to design the program in as modular a fashion as possible, using a templated function and a class hierarchy as covered in *A Tour of C++*. This way, no code would need to be duplicated, the logical structure of the code could remain easy to maintain and debug, and if we or anyone else wished to add more cipher algorithms making use of scoring-based techniques to the program's repertoire, this would be easy to do with very minimal changes to existing code.

Additionally, during our early testing and development, we discovered very quickly that, if the cipher that was used to encrypt a given ciphertext is unknown, we have essentially no way to quickly determine which one of the three it was, or rule out any, without simply trying all three available decryption methods and seeing which succeeds and which fails.
However, running each algorithm one at a time would be problematic for the user. This is because, for example, a long text that was encrypted using the commercial Enigma cipher will take a very long time to complete Vigenère cryptanalysis (unsuccessfully).

We solved this problem by making use of concurrency, using the `std::async` function covered in class. This way, especially for lengthy inputs, the cryptanalysis method using the correct cipher will most likely finish and return correct results first, and once this is done, there is no need for the user to wait for the other (incorrect) processes to finish.

Modularity

In order to achieve the modularity we wanted, we first created a class hierarchy with the base class `GenericCipherText`, defined in `generic_cipher.h`. This base class contains all the basic data we would expect any ciphertext object to have: the ciphertext string `encrypted`, its plaintext form `decrypted`, the length of the string `length`, and a value `score` that can contain the score of the decrypted text according to whichever scoring technique is used (with the default value -1e10 set in the constructor).

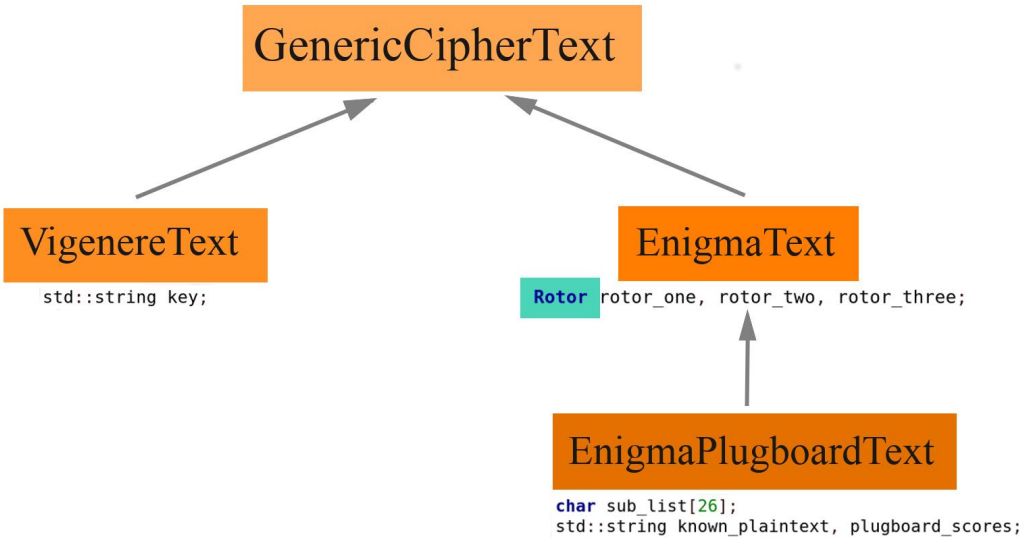
We also added the pure virtual method `cryptanalysis()`, which should ultimately execute the cryptanalysis algorithm in derived classes, the pure virtual method `settings()` which should output the settings found, and the method `read_decrypted()`, just to make reading a plaintext from the command line slightly easier:

```
class GenericCipherText {
public:
      string encrypted, decrypted;
      size_t length;
      double score;
      GenericCipherText(string t): encrypted{t}, decrypted{t},
length{t.length()}, score{min_quadgram_score} {}
      GenericCipherText(): encrypted{}, decrypted{}, length{0},
score{min_quadgram_score} {}
      void read_decrypted() {cin >> decrypted; encrypted=decrypted;
length=decrypted.length();}
      virtual bool cryptanalysis() = 0;
      virtual string settings() = 0;
};
```

Next, we created classes `VigenereText` defined in `vigenere.h`, and `EnigmaText` defined in `enigma.h`, both derived from `GenericCipherText`.
The class `VigenereText` contains one extra piece of data, which is the string `key` used for encryption, for which the default value is “AA”, set in the constructor.
The class `EnigmaText` contains three additional objects of class `Rotor`, which are initialized in the `EnigmaText()` constructor using arrays of characters that mimic the actual rotors on the Enigma machine.

Finally, we created a class `EnigmaPlugboardText` defined in `enigma_plugboard.h`. This class is derived from `EnigmaText`, but contains an additional array of characters `sub_list` which represents the military Enigma plugboard, as well as a string `known_plaintext` to store the known plaintext, and a string `plugboard_scores` to store the plugboard data returned by the algorithm.

From this point forward, we were able to fill out each of the classes in their separate header files with the appropriate `cryptanalysis()` algorithms for each type of cipher. But, structuring the classes this way ensured that the most basic level of the interface was common to all of them.
This is a diagram of the class structure that our program uses, following the example from [16]:



Concurrency

In order to be able to run multiple cryptanalysis algorithms simultaneously, we made use of the C++ function `std::async`. [17]

First, in `cryptanalysis.cpp`, we created a very simple class called `Cipher` to store all the data that we wanted to get back from each of the separate algorithms: the decrypted text as a string `decrypted`, any other key/settings data in a string `settings`, and the time taken by the algorithm as a double `time_taken`.

Next, we created a templated function `timed_analysis()`, which accepts a label string `label` to be used in the output, and an object `ciphertext` of any class derived from `GenericCipherText`. (Note: Because we are using C++17 rather than C++20, we were not able to use concepts, or other methods of restricting what types `ciphertext` could be.)

```
template<typename T>
Cipher timed_analysis(std::string label, T ciphertext) {
      using namespace std::chrono;
      auto start = high_resolution_clock::now();
      bool success = ciphertext.cryptanalysis();
      auto end = high_resolution_clock::now();
      double time_ms = duration_cast<milliseconds>(end-start).count();
      Cipher result {ciphertext.decrypted, ciphertext.settings(),
time_ms/1000.0};
      if(success) /* ...print result to screen*/
      return result;
```

```
}
```

This function executes and times the `cryptanalysis()` method particular to the `ciphertext` object used, using `std::chrono::high_resolution_clock`, and returns the information in an object of type `Cipher`.

Finally, the `main()` function of `cryptanalysis.cpp` does little more than create an object of type `std::vector<std::future<Cipher>>`, of which each entry is specified using a lambda as an argument to the `std::async` function:

```
cipher_tasks.push_back(std::async([&]{return timed_analysis("Vigenere",
VigenereText(input_text));}));
cipher_tasks.push_back(std::async([&]{return timed_analysis("Commercial
Enigma", EnigmaText(input_text));}));
cipher_tasks.push_back(std::async([&]{return timed_analysis("Military
Enigma", EnigmaPlugboardText(input_text));}));
```

This creates in memory an asynchronous process for each type of cryptanalysis algorithm, as specified by the derived class in which it is defined. The output we want from each algorithm is written to standard output by the corresponding `timed_analysis` function as it completes.

Here is an example of typical output, in which an encrypted text string is passed in to `input_text`, and the three `timed_analysis` processes indicated above are carried out simultaneously. The “Military Enigma” process ends first immediately with no output, because no `knownPlainText.txt` file was provided. Next, the “Commercial Enigma” process finishes, and returns correct plaintext and rotor/ring settings, because in fact, the original text string was encrypted using commercial Enigma. Finally, the “Vigenere” process finishes after much longer, and returns an incorrect plaintext and key, because the input was not a Vigenere coded message.

```
arun@x2360e [7:10pm] [Floor;Tails]
/u/arun/TeamCipher/combined $ ./cryptanalysis
Ciphertext: PFKDWJFJTYQHWRIGTXSYIOATBNPJTFUMNEYCPPHUSRUCWMPETNCIREKAFJDFNCOBSBNBPTUJTPPDDLQDLPLYY
CWJRRAJSHQEQYHEYKNACHUZRSLFRSBCUIMTQZMYLJEAYQLAYBFATCRJLVXWOFUWGSJKRPUJUVVRYVAWBKTXQTRNADJZCYMDUB
MFNXLCKVVNJVSAQWZRDLCSWNZFMBRXCNJDOXACVHHWQDQWSZDVHISSUNQF

Commercial Enigma:
THENAVYISALLCLEARFORTHEDAYITISTWELVEHUNDREDHOURWEAREALLSETFORTHEDAYWEAREMOVINGATASTEADYPACEWILLRE
ACHTARGETINNEXTFEWDAYSWEAREADVICINGAIRFORCEANDARMYTOHELPUSEXPEDITETHEPROCESSBYPERFORMINGDAILYCOLL
ABORATEDDRILLSATFOURTEENHUNDREDHOURSHEILHITLER
Rotor Setting: JND  Ring Setting: ACP   Score: -1507.713205
Time taken: 37.093 s

Vigenere:
TTERENTIERRIENCESTRATIVERESSIONSKITTERINGUESTRUMMEANTROUNDERIEVEREASTIMSELESTOPPORTANDREDIERRESPE
ASIMPERIENTENTERELATESSENTERENTENTHEREDECIATENTERESPIERSBURINCHERENCESSINSTANDERESSENTERNINGLANAT
CHMANDRESTATERSTANDREASTRANCIENDERENOTHERSITUT
Key: WMGMSWMBPHZZSEGCBEBYPGFPKJXRLRHUDWFJLYZHMXQKDVVSHJCVYNWGSGZOFYTXBXNJWLIRPELLKXBOXYSYLZFFOJWS
BDYPMYPWMVJJUDHGNFSBAOQCBEUBMMTUUFRHUDSTUKBXPAJJSRKDKOQERKFTZOASMITKUEWJZGBFIGZUAQGVLUULQOTBWKDPE
KVAJCQTEWMOMHKZWUVOUIRKZWFDWEJCIFZSDAMFOMPCAEBAMU
Score: -960.822232
Time taken: 117.526 s
arun@x2360e [7:14pm] [Floor;Tails]
/u/arun/TeamCipher/combined $
```

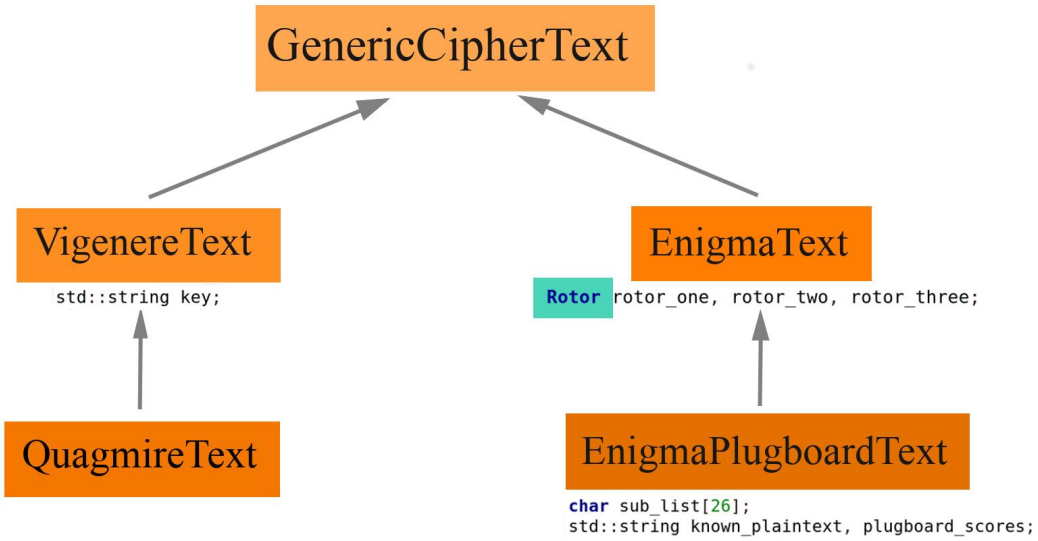
Here is another example of typical output, in which the same three `timed_analysis` processes are carried out simultaneously on a different `input_text`, and the “Military Enigma” process finishes first and correctly because in fact, the original text string was encrypted using military Enigma. The other two processes finish later, with incorrect results as expected:

```
arun@x2360e [7:59pm] [Floor;Tails]
/u/arun/TeamCipher/combined $ ./cryptanalysis
Ciphertext: BUMHWJTDGHEFKOROIWTHIQRIJCAJTMOMMPKVFNAUEJUITMNSCPXLRYKGYXSMPKBFEJVJHIUWIWOOJLFWKWEYH
XKQRZLBOWTSMHNKKLPXWAWWGSESZSJXUZUIYVMBKBDKOOIFHXQWDXLFKECWGQSNGNMKMNCWPUYRSRIOWIHCXGLNFQNGDHGBUJ
UGACKJKPNQWBETQWVFPKXEUMMUMBJZXPUYOVCXLYYLFLSWEOMJBPDCRXIQ

Military Enigma:
OYSAAVUTUBMMPFSEEQZDTISGPQTTIEDWSYQAOGPDTFDPUUHLKIPHAOLPURRGAVXKVCSOZVRSUGLFZGCISEUEREDIASGFYKNLS
JFWUFWZSITPHERCEETBAAELMHRHUCZVLRSBDRJQGFUKULTAEULYOUSLNMVSLNKDVIKXYFQLZUZSUZWFSVSLRNTJOUIELORVKE
UWORQYRIWFTKKEATHPDJIVEQUEIQBBVEBUBHYHKTYZITLG
Rotor Setting: JND  Ring Setting: ACP   Score: 31
        Plugboard Settings can be Seen Below, Use Ones with Highest Scores:
A H 29, B J 36, C X 42, D Y 30, E S 50, F Q 38, G O 35, H Y 43, I T 45, J V 32, K L 41, L Y 29, M
 N 30, N P 42, O Z 32, P Z 32, Q Y 32, S Y 30, T Y 29, V Z 34, X Z 33, Y Z 31,

A F 29, B Z 32, C P 33, D V 30, E H 32, F P 34, G Z 32, H P 34, I Z 32, J X 31, K Y 30, L X 29, M
 Z 29, N Z 32, O Q 32, P X 32, Q X 32, S Z 29, T X 29, V X 32, X Y 31,
Time taken: 35.826 s

Commercial Enigma:
ZBVKERYMEYWSNMALMCCCVJCDZTCZDYEHTMJLEKGWNUTQHAOJDMENEMBLEAQQSTCZAUFKWJAZASKIEKSGDFWSXPDYHHTSKKHEV
EKFGWJHNOLSWETPMPAIRROFADPKHUQCINKCODTTRJJSITOYIDLIEOSFSFYHHYVDZSNIGKBTZIDXQIGLRRLDJACHXITOJAERON
TISTMRHEEREVNMCQSAWXPAUSAOEZRDDCJVPWTMGEFJQJWW
Rotor Setting: MZU  Ring Setting: ACR   Score: -2102.985456
Time taken: 38.308 s

Vigenere:
EPRINESTREASTERINCESSENERALLESHOUTENTEREVOIDENTINGEROOMISTERESENTSEVERIENTERRIFICALLECONVERATIONS
ELENTENTIVISELVERESSIANTIONERSTATINGLOOKINGUESTILLAINTREMENTENERSTANCOMETICALLENTERSTERESERINEVER
RIENDERSEREFOREVERSTATINEVEREDNESSENTERINCEALL
Key: XFVZJFBKPDENRKAGVUPPQMEESCPYPUHYSWGIMJJQJVMFPZUKPJTUDKYYGEOVLSXSLRRODRMSVDKXSDAOIWTNDVWDWVUB
VOFFUDCGXSLKDSBOOOTXVBFFCRUVFNYOGKLROVASBMCITPYZQAKDYFHNYATTIBYJWQLNBZPOJGTQTNDLFFCCQOCKCQQPWKGSC
CJVSKNLMJSBYSTNQHYDIGFIFWUFGIYCHHUISHAEABTFKHQANX
Score: -960.986124
Time taken: 114.245 s
arun@x2360e [8:01pm] [Floor;Tails]
/u/arun/TeamCipher/combined $
```

Extensibility

The Vigenère and Enigma ciphers, while probably the most historically famous of all ciphers that can be solved using scoring algorithms as we have done, are far from the only ones. For example, the Beaufort, Porta, and Quagmire ciphers are similar to Vigenère in that they all rely on polyalphabetic substitution, and therefore in theory should be susceptible to strategies similar to what we have used. As such, these are good candidates for future additions to this project.

If we, or anyone else, wished to add a cryptanalysis algorithm for, say, the Quagmire cipher, to this project, the first step would be to define a new class `QuagmireText`, derived either from `GenericCipherText` or possibly from `VigenereText`, depending on how much of our existing Vigenère algorithm is reusable for the Quagmire decryption technique used.



Then, once the Quagmire cryptanalysis has been implemented in the `QuagmireText::cryptanalysis()` method, and the `QuagmireText::settings()` method is made to return the right output, all that remains is to add this line to the `main()` function in `cryptanalysis.cpp`:

```
cipher_tasks.push_back(std::async([&]{return timed_analysis("Quagmire",
QuagmireText(input_text));}));
```

Once this is done, the program will concurrently run all algorithms specified by the list of types used in processes in `cipher_tasks`, including Quagmire, and return results as each one finishes. The output will look similar to the sample output shown above, with a “Quagmire:” section added.

Testing and Measurements

We compiled and tested our code in two different hardware/operating system environments. We used a PC running Windows 10, and compiled each of our executable files using this command:
`g++ -O2 -std=c++17 filename.cpp -o filename.exe -pthread -lpthread`
It was in this environment that we performed the time measurements used for analysis.
But, to make sure our code was not too platform/compiler-specific, we also compiled and ran it on a computer running CentOS Linux 7.9.2009 Core (x86-64) with 12 Intel Xeon Gold 6136 CPUs @ 3.00 GHz, using this command:
`clang++ -O2 -std=c++17 filename.cpp -o filename.exe -pthread -lpthread`

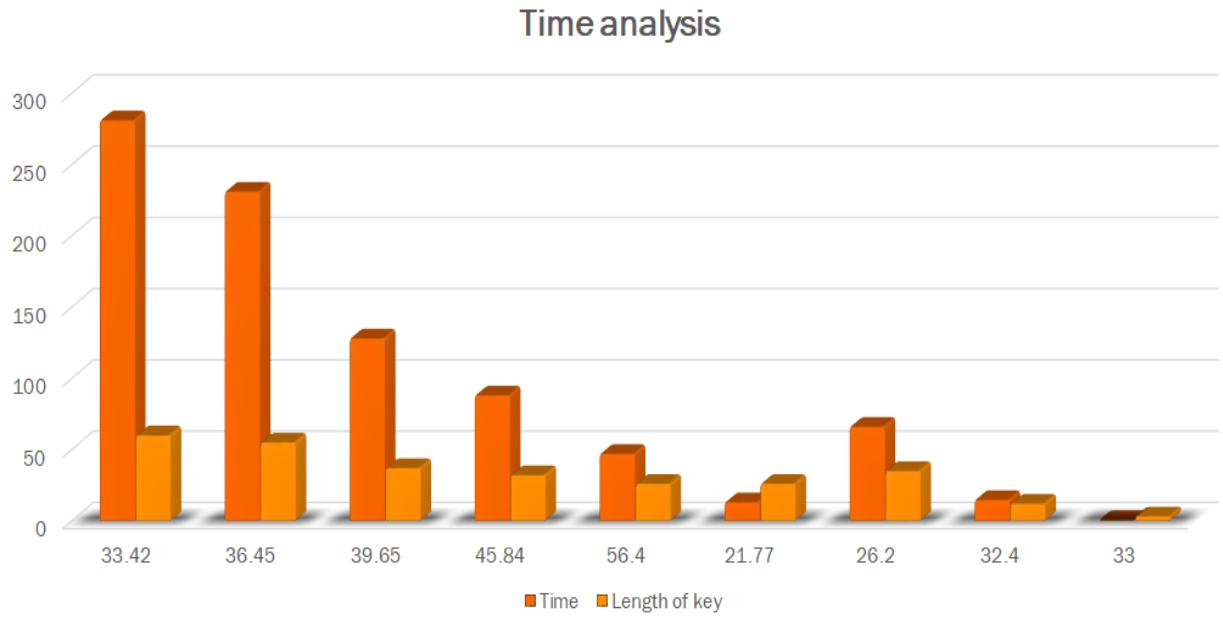
Vigenère Algorithm Results

To test our code and its competence, we analyze our algorithm via the time it takes to break the different length cipher codes.

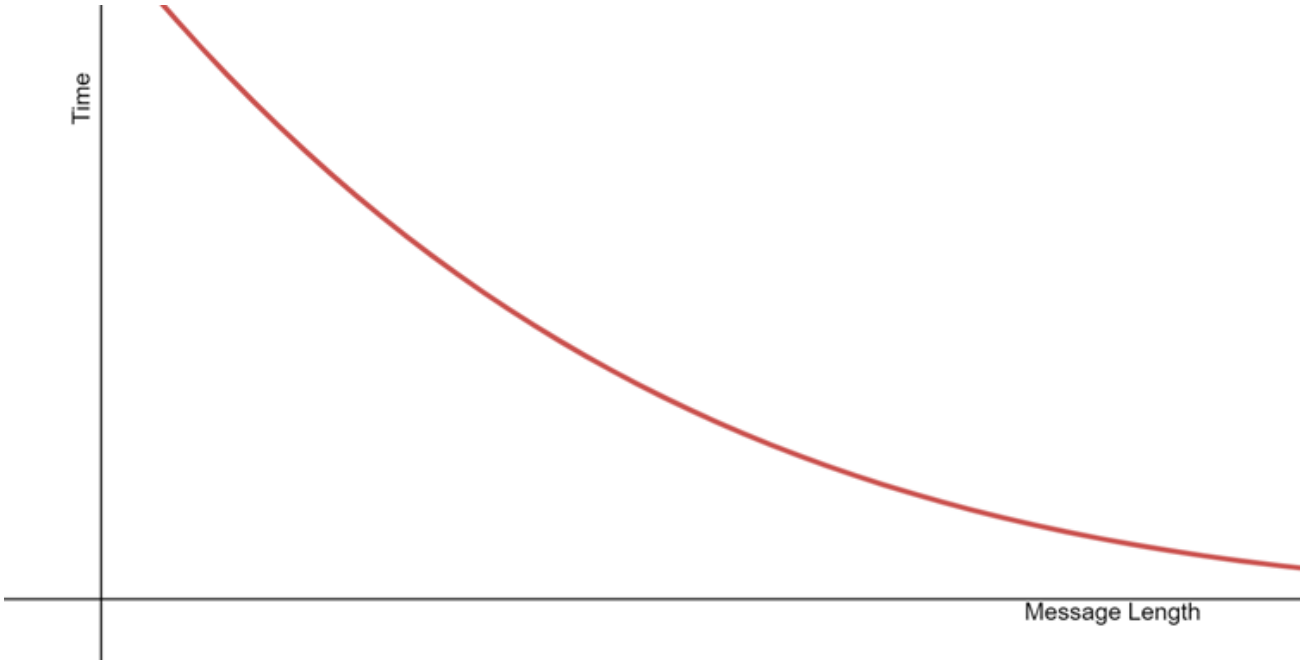
The analysis of the Vigenère cipher has two parts, and those are:

1. Analyzing the time our cryptanalysis function takes to break the code with varying ciphertext lengths and keys.
2. The time our cryptanalysis function takes when we fix the key length and vary the ciphertext length.

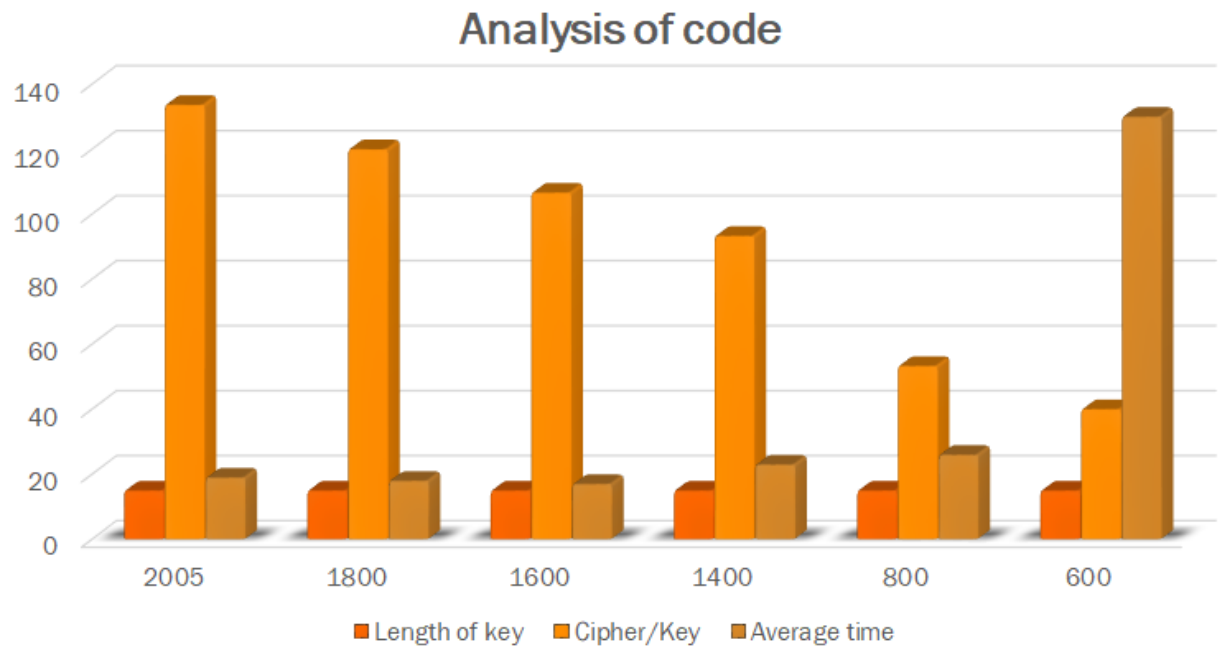
We are using 20 different ciphertexts under each length of ciphertexts (2005, 1467, 917, 566, 389, 99) and vary the keys' length to get vital data points. The varying key length gets us a cipher-to-key ratio and the execution time, which becomes the basis of the first graph plot.



For the second graph, results were derived by taking the average execution cost of 20 different ciphers from each category with the fixed-length key. We plot the curves using Newton's interpolation method.

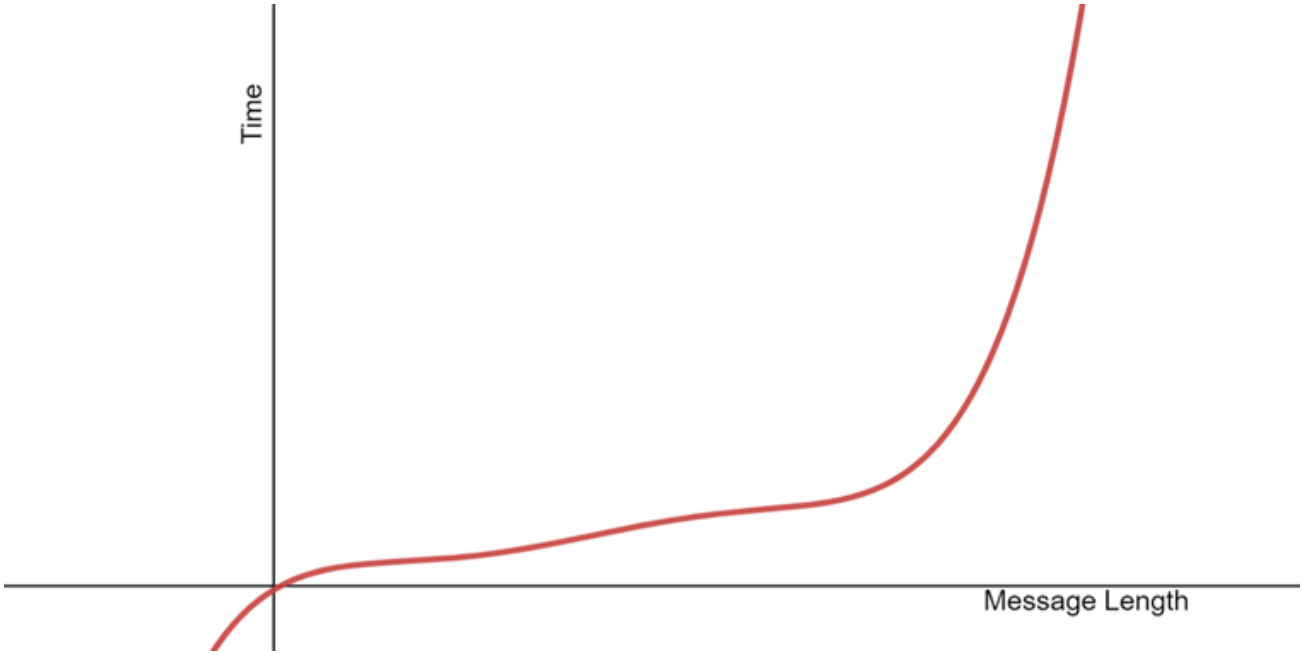


The image below delineates the graph for execution time versus length of cipher for experiment 2.



Commercial Enigma Algorithm Results

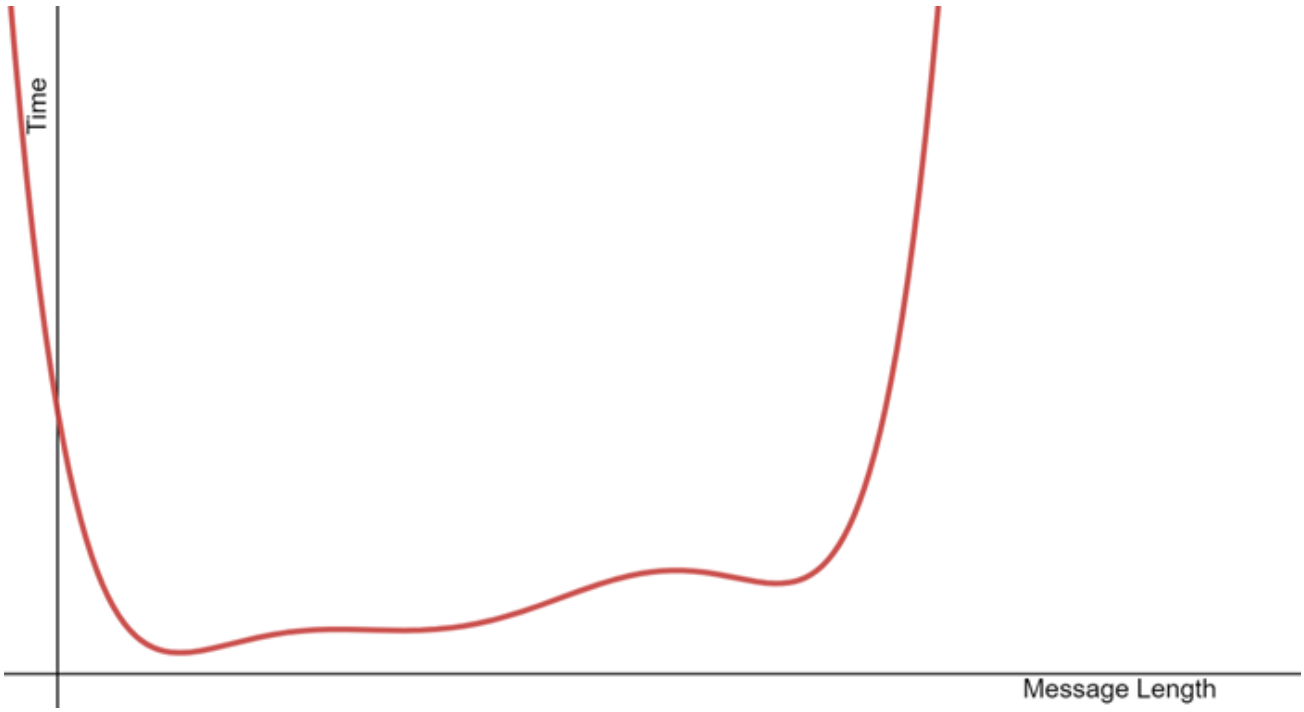
Commercial Enigma is the Enigma version where there is no presence of the plugboard. We utilized ten different ciphertexts under each category of different ciphertext lengths (65, 100, 150, 200, 250, 300) to validate our cryptanalysis function's performance. The experiment was straightforward as the only measurable was the average time. The graph below depicts the exponential growth when the length of ciphertext increases. We see exponential growth in the graph is because of the structure of the code we have written; however, in theory, the larger the ciphertext length, the more cryptanalysis function must process the decryption code to get the results, and the efficiency of code dominates the impact of the result.



Military Enigma Algorithm Results

The cryptanalysis of the military Enigma is different from the commercial Enigma because of the scoring functions we deploy while deriving results. The commercial Enigma utilizes quadgram analysis and Ciphertext-only attack. However, the military Enigma deploys a known plaintext attack.

To measure the military Enigma's cryptanalysis functions efficiency, we used a similar approach as delineated in the commercial Enigma section. The graph below reflects exponential growth as the ciphertext length increases, but it is faster than the ciphertext-only attack. Thus, proving our initial assumption that a known-plaintext attack is faster than a ciphertext-only attack, as more information is available to the scoring function to work.



Next steps

Further Military Enigma Algorithm Development

Currently, our Military Enigma algorithm is limited to the “known plaintext” approach, and while it works well on ciphertexts with lengths typical for the original Enigma machine, we would ideally prefer to be able to analyze ciphertexts with unknown plaintexts, of arbitrary length, directly. This is a goal that will require more development.

Adding Other Ciphers

As indicated in the “Extensibility” section above, there are many other ciphers that we could try to develop cryptanalysis algorithms for, and our project is structured in such a way that makes it easy to add as many as can be devised using scoring-based strategies. A possible future goal for this project could be to develop a more universal cryptanalysis application, in which case the main limitation to its power will probably be the number of threads available on the hardware used.

References

1. Vigenere cipher. (2020, April 4). In Wikipedia. https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher
2. Enigma machine. (2020, April 4). In Wikipedia. https://en.wikipedia.org/wiki/Enigma_machine
3. James Lyons. (2012). Practical Cryptography - Quadgram Statistics as a Fitness Measure. http://practicalcryptography.com/cryptanalysis/text-characterisation/quadgrams/

4. Michael C. Koss. (2003). Paper Enigma. https://www.apprendre-en-ligne.net/crypto/bibliotheque/PDF/paperEnigma.pdf
5. History and Modern Cryptanalysis of Enigma's Pluggable Reflector. Olaf Ostwald and Frode Weierud. https://cryptocellar.org/pubs/HMCE_UKWD.pdf
6. Singh, Simon (1999). The Code Book. Anchor Books, Random House. pp. 63–78. ISBN 0-385-49532-3.
7. James Lyons. (2012). Practical Cryptography - Cryptanalysis of Enigma, Part 2. http://practicalcryptography.com/cryptanalysis/breaking-machine-ciphers/cryptanalysis-enigma-part-2/
8. Known-plaintext attack. (2020, April 4). In Wikipedia. https://en.wikipedia.org/wiki/Known-plaintext_attack
9. Kerckchoff's principle. (2020, April 4). In Wikipedia. https://en.wikipedia.org/wiki/Kerckhoffs%27s_principle
10. History of Enigma. (2020, April 4). In Crypto Museum. .https://www.cryptomuseum.com/crypto/enigma/hist.htm
11. History of Vigenère Cipher. (2020, April 4). In Crypto Museum. https://www.cryptomuseum.com/crypto/vigenere/
12. Enigma Machine. (2020, April 4). In Brilliant. https://brilliant.org/wiki/enigma-machine/
13. The Vigenere Cipher -- A Polyalphabetic Cipher. (2020, April 4). http://user.it.uu.se/~elenaf/Teaching/Krypto2003/vigenere.html
14. How Polish Mathematicians Deciphered the Enigma. MARIAN REJEWSKI. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.692.9386&rep=rep1&type=pdf
15. US 6812 Bombe Report. 1944. "6812th Signal Security Detachment," APO 413, US Army. NARA, College Park, Md., Record Group 457, NSA Historical Collection, Box 970, No. 2943 Formatted by Tony Sale, Bletchley Park (2002), 39-40. https://www.codesandciphers.org.uk/documents/bmbrpt/usbmbrpt.pdf
16. Bjarne Stroustrup (2018). A Tour of C++. Addison-Wesley. p.57. ISBN-13: 978-0-13-499783-4.
17. http://en.cppreference.com/w/cpp/thread/async