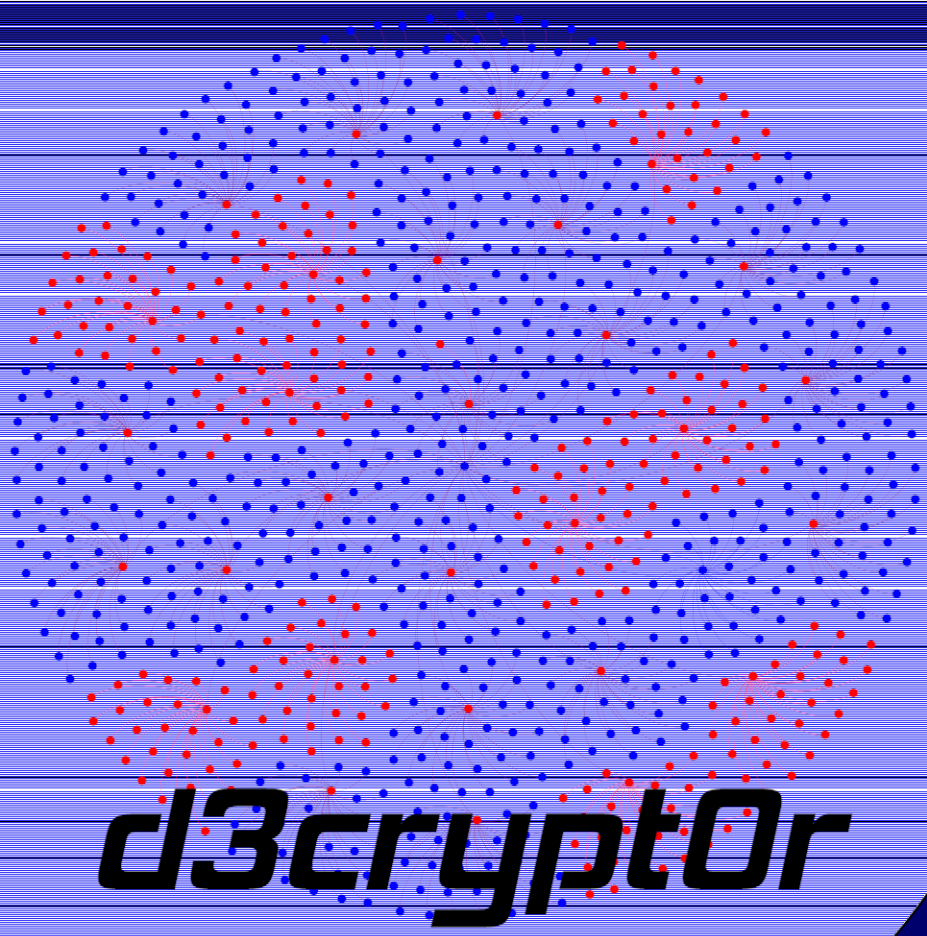

Cryptanalysis of a class of ciphers based on Search Over Bethe Lattice Technical Report

Iman Hosseini

Contents

1 INTRODUCTION

This project is the work of a one man team, Iman Hosseini. Highlights:

- Includes implementation in Python and multiplatform C++ built for linux and windows
- The python version decrypts ciphers in 60 millisecond and the C++ version in 5 milliseconds[1]
- Extensive performance comparison and analysis and quantitive exploration of the problem
- Discussion of permutation ciphers, their cryptanalysis, and their relation to this project

The compiled binaries for the C++ version are available in *Release* directory for Linux and Windows, in case you want to compile yourself, for Windows, Visual Studio Solution files are included which you can open using Visual Studio, and for Linux you can run *$clang Decry.cpp* in the directory including the source files. Note that after running the program, the input should be numbers separated by comma, exactly as in the project description: no extra spaces, wrong length, etc.

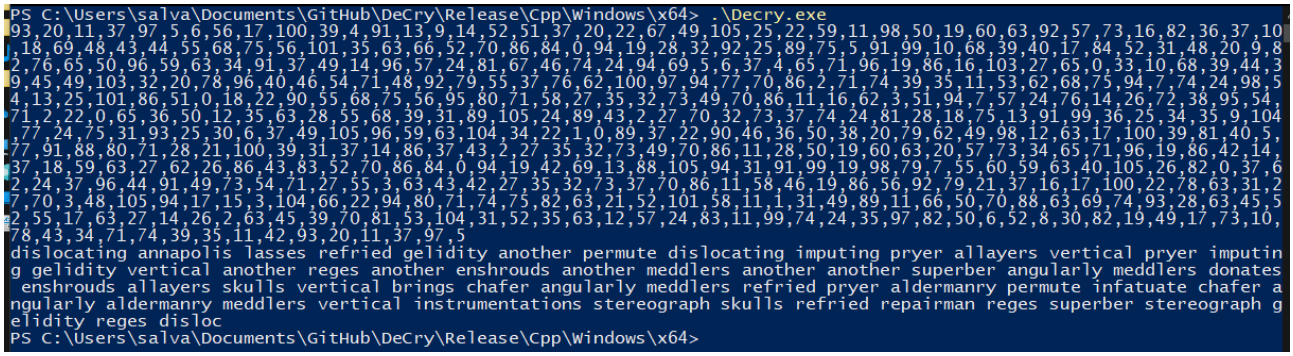

FIGURE 1: EXAMPLE RUN

[1] measured on 10th gen intel Core i7

2 INTUITION

2.1 THE CIPHER ALGORITHM

Intuitively this encryption scheme is a strengthenting of a substitution cipher, whereas in a substitution cipher the key was an automorphism (isomorphism from a set to itself) over the plaintext alphabet, here there is distinction between the alphabet of plaintexts (i.e. the english alphabet) and the ciphertext alphabet which is a set, larger than the plaintext alphabet and a symbol in plaintext alphabet may be represented by more than 1 ciphertext symbol.

This change means that the cipher is more resilient to frequency analysis attacks, by representing more frequent symbols (for example, 'a' shows up more frequently than 'q' in a text) with more cipher symbols, we can load ballance those such that the frequencies of ciphertext symbols become uniform and thus render frequency attacks useless. Frequency-based methods are in fact varied, for example assume that the dictionary contains the word 'beet', in a simple substition cipher, we would still see the duplication 'ee' as 'xx' where x is substituted for 'e', analysis of bigrams in plaintext space and the ciphertexts like this would also not work in this scheme, because now 'ee' can become 'xy'.

2.2 PART1: CHECKING PLAIN-CIPHER COMPATIBILITY

The first part is relatively easy: given a plaintext and a ciphertext we can check whether it is possible that the ciphertext is an encryption of the plaintext using this scheme, we simply build up the key until we find an inconsistency: either A) a cipher symbol is mapped to 2 different plaintext symbol or B) A plaintext is associated with too many cipher symbols (more than the number specified in the algorithm). For looking up a ciphertext given a dictionary of plaintexts, we loop through the dictionary ruling out the incompatible plaintexts.

2.3 PART2: FOR A GIVEN WORD DICTIONARY

The key in achieving the aforementioned resiliency towards frequency attack, is in the number of cipher symbols we attribute to each letter, if these numbers are not proportional to frequency of each letter, the frequency over the cipher symbols would be non-uniform and this can be used to deduce more information based on the frequencies. Intuitively, a uniform distribution is symmetric with respect to all the symbols, and it does not tell us anything, whereas imagine we had chosen the scheme such that instead of attributing 7 symbols to letter 'a' we attributed 1, then that 1 cipher symbol would appear much more than other cipher symbols in the cipher and we could single it out and deduce 'a'.

This intuition can be quantified via Shannon Entropy[2]. Before knowing that the word dictionary holds relatively small number of words (30) I was considering an approach based on frequency, with the idea that each key configuration would induce a probability distribution on the letters, based on the cipher we are analyzing. Assume 'a' is mapped to symbols 'c_1', 'c_2',.. 'c_n and denote with $P_c(x)$ the probability distribution for cipher symbol, and $P_p(x)$ the distribution for letter 'x', then we have:

$$P_p(a) = \sum_{i=1}^{i=n} P_c(c_i)$$

It is expected that this distribution will be in some sense *close* to the average probability distribution of the letters. Among different key configuration, the one which induces a letter probability distribution closer to the average frequencies is more probable to be the key. This is based on the fact that if you only have 1 sentence, the letter frequencies are stochatic, but if you incrementaly consider larger texts, the distribution of letters converges towards the average distribution. As for the measure of closeness, the best candidate would be the Kullback-Leibler divergence[3] which measures the relative entropy between 2 distribution: notice that KL-divergence is not symmetric, 1 distribution is considered to be the reference and the other, the observed empirical distribution.

Finally, regarding the scheduling algorithm, the default scheduling algorithm looks a good choice. As it is prefered for security that the distributions don't carry information: they be uniform. Imagine for letter 'a' we had 3 cipher symbols and we scheduled such that those ciphers would be used with a ratio of 1:2:3, then this could be used as a signature to find them: an attacker would look for trio's of symbols with similar ratio.

3 QUANTITIVE ANALYSIS

3.1 SOME MAGNITUDE ESTIMATION

Can we just try out all possible keys until we get it right? Why not? To get the answer first let us consider that to break the encryption, the order of elements attributed to each letter is not important, that is: even if the scheduling algorithm discriminates between translating a letter to one of the possible cipher symbols, for decryption, all cipher symbols being mapped to a letter are the same to us, so to count the number of keys, we must count the number of ways we can assort 106 token into 27 batch of sizes 19,7,1,2,4,10,2,2,5,6,1,1,3,2,6,6,2,1,5,5,7,2,1,2,1,2,1. Generally, denote the alphabet with A and the number of ciphers mapped to a letter x with

[2]https://en.wikipedia.org/wiki/Entropy_(information_theory)
[3]https://en.wikipedia.org/wiki/Kullback%E%80%93Leibler_divergence

N_x then key size would be:

$$|K| = \frac{N!}{\prod_{x \in A} N_x} \approx 10^{119}$$

Obviously too big to think about trying some random key. This number despite looking scary, gets a steep decrease as we read in the cipher and a plaintext, assuming they are the same, and build up the key as we move forward. For example, when only 12 cipher symbol are left with set sizes of 10, 2 it becomes only 66 different configurations. To have a better understanding of how it works, we ran a large number of tests for different plaintexts and ciphers where the cipher was not made from that plaintext, running our code which consumed characters and updated the key until hitting an inconsistency and measured how many characters it took, until hitting an inconsistency and realizing these two cannot be the same, the result can be seen in figure 2, and it shows that within a reasonable amount of characters being read in, it will be known whether they are not the same.

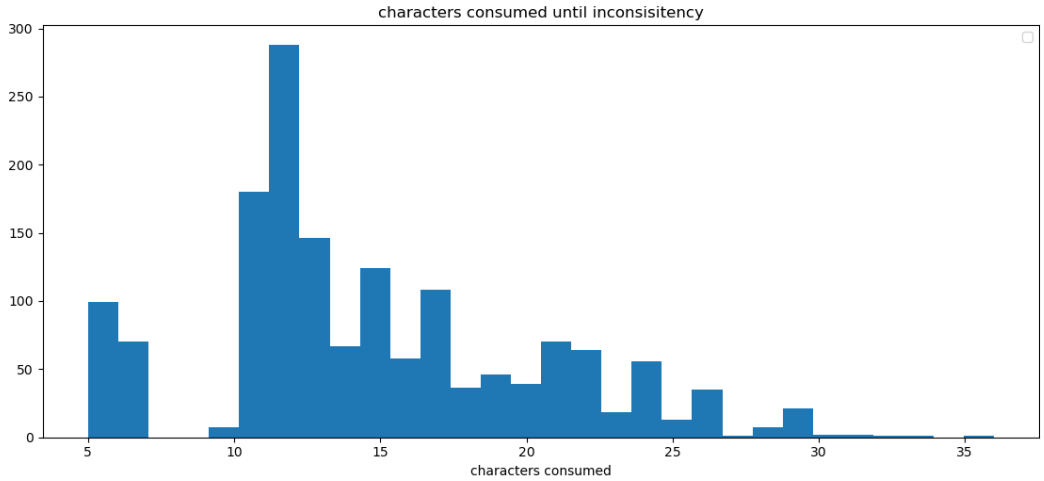

FIGURE 2: HOW MANY CHARACTERS DOES IT TAKE?

3.2 BETHE LATTICE

To attack the encryption for the second test, after realizing the size of the word dictionary, we decided to try sequences of words, and see which sequence can be and can not be our plaintext. Based on previous part, we knew that whithin a mere 30 character we would have ruled out many wrong plaintexts so this could work. The space of possible plaintexts, without a word dictionary would have been:

$$plaintext \in \{< space >, a, b, c, .., z\}^{500}$$

But given the word dictionary (W) and this format it is now a subset of the above space, namely it is:

$$plaintext \in \{\{w_0, w_1, .., w_n\} | w_i \in W and \sum_i |w_i| + n \geq 500\}$$

Where the addition with n is to account for spaces between words, and the representation of a plaintext would be the first 500 character in the w_i enjoined by spaces. This means, unlike the first space, the space of plaintexts is restricted and there is some inherent structure in this space. For example, if we impose total order on the set of words, then we can also impose partial order on the set of plaintexts [4], by defining a $>$ relation like this:

$$\{w_0, w_1, .., w_n\} > \{w'_0, w'_1, ..w'_m\} \;\; if \; n > m \; or \; w_j > w'_j \;\; and \; \forall k < j \; w_k = w'_k$$

Which is known as *dictionary order*. The significance of this order is that it can be trivially generalized to partial messages: messages that are not of length 500, which are what we encounter when we are building up our plaintext. With this order, the space is endowed with a natural traversal to visit different states until we find the plaintext which works. We begin with the empty state $[]$ then we add a single word $[0]$, then we add another $[0, 0]$ and continue until the generated plaintext would be inconsistent with the ciphertext that we want to decrypt, then we try the next word. When all the words are exhausted, we go back and change the previous word, and so on. The state of the algorithm is lists of numbers, where each number is between 0 to 29, this structure can be vizualized like figure 3 which is assuming only 3 words:

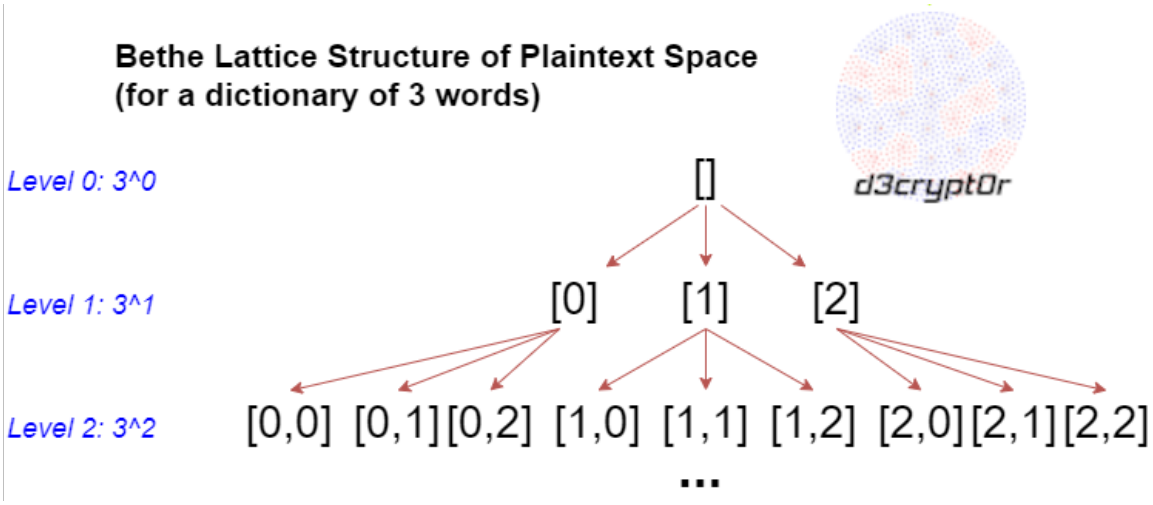

FIGURE 3: STRUCTURE OF STATES

So our approach is that based on this natural order, we traverse the possible states until reaching an acceptable state (an state of length 500) and we prune the inconsistent states, so that we do not go into those sub-trees. This structure is called a Bethe lattice [5] which is also the inspiration for our logo. We plotted how more spaces are explored, the red nodes are states

[4] https://en.wikipedia.org/wiki/Partially_ordered_set
[5] https://en.wikipedia.org/wiki/Bethe_lattice

which have not yet been explored and blue are visited nodes, the graphs are for bethe lattices with z=30 as in our case, and have been plotted for the first 3 levels.

3

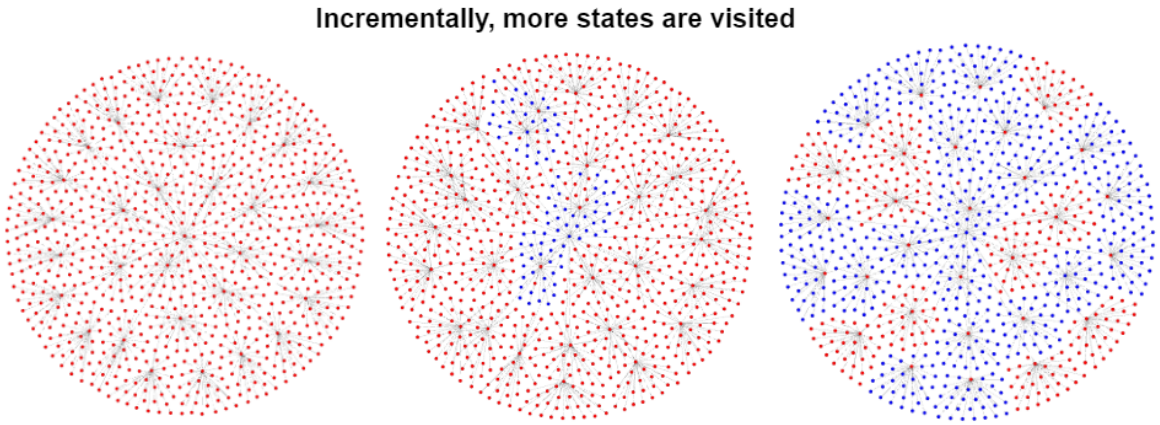

FIGURE 4: EXPLORING THE BETHE LATTICE

4 IMPLEMENTATION: D3CRYPT0R

First prototypes were made in python, including the code to generate the plots. For histograms matplotlib was used and for larger graphs, Gephi [6]. A set of random keys, and plaintexts/ciphertexts were generated for testing and for benchmarking performance. The final decryptor code was developed both in python and in C++, where the C++ version can decrypt ciphers in a matter of 5 milliseconds, the python version takes roughly x12 more time, but is still fast at 60 ms. There C++ version also has support for multi-threaded version, where the lattice is divided up between threads and each thread searches it's subspace, but this does not lead to much improvement, as the overhead of having threads is roughly of the same order as the benefit, it is expected that for larger dictionary sizes where the search becomes longer, multi-threaded version would become feasible.

The C++ version is developed with no dependency on external libraries and is thus multi-platform, and included in the release version you can find linux and windows versions for both x64 and x86. Also on Windows, we compared the performance of the code generated by different architectures and compilers (Clang, Microsoft Visual C++) which can be seen in figure 5.

Performance analaysis report by Intel VTune™ [7] shows how multi-threaded version doesn't achieve an advantage, also the report shows that running multiple threads affects processor

[6] gephi.org

[7] https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html

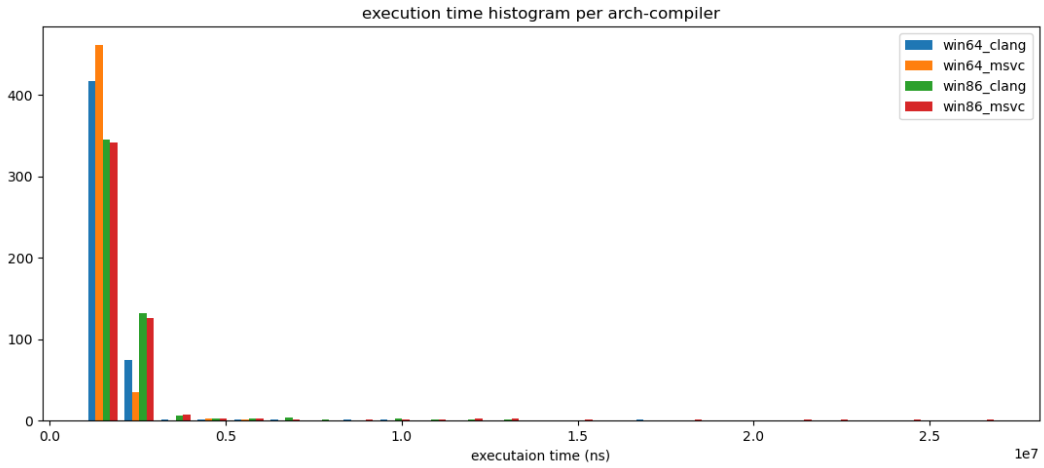

FIGURE 5: COMPILERS, HEAD-TO-HEAD

frequency, a phenomenon discussed here [8]. The memory footprint is less than 1MB, and the threading overhead is also in-line with other reports [9].

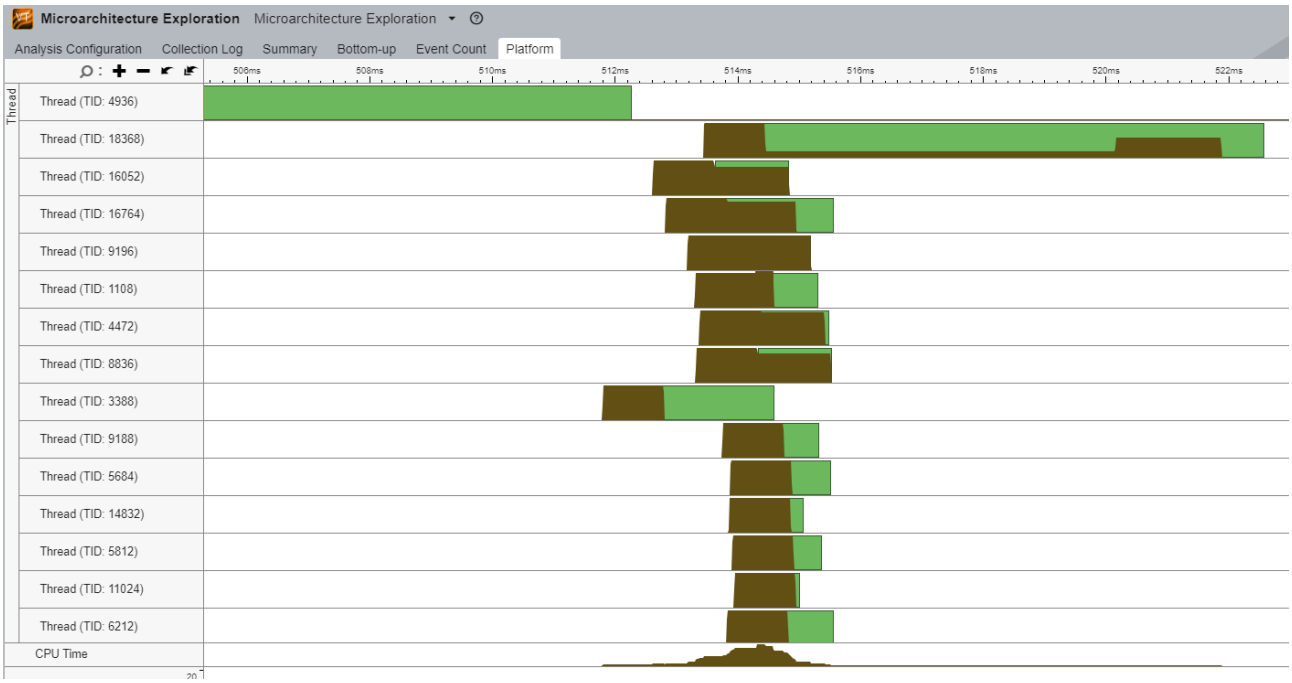

FIGURE 6: VTUNE REPORT

Achieving the 5 ms record was not an easy task, various considerations were made. The data that should statically be inside the program: the length of each row in the key, the alphabet, and the dictionaries were preprocessed by a python script, which then generated C datastructures usable in our project, the generated data went into the header file *Config.h* where the plaintext dictionary for example, is stored as a 2d array of shorts, where each short represents the number of that given letter. Because these numbers would all fit short, short was used in-

[8] https://pzemtsov.github.io/2016/12/08/turbo-boost-and-performance.html
[9] https://lemire.me/blog/2020/01/30/cost-of-a-thread-in-c-under-linux/

stead of int, and this is also easily changeable with a typedef inside that same header. Also, notice that all the data is defined to be *constexpr* so that these will be baked into the binary at compile-time, again saving time. Utility functions are also defined in the headers, for example, to print the plaintext in string form.

The main code, includes a struct holding the state as discussed in the previous part, which determines where in that Bethe Lattice we currently are, and function *consume(plc)* takes a plaintext character and based on that, updates the key table, if consuming that character leads to inconsistency it returns false, signalling that this node in the Bethe Lattice is a dead end, and thus other nodes are explored. To be able to track back, there is a rollback function which undoes the previously consumed characters, by reverting the key table, and auxillary data to previous states.

5 PLANNING

We started the project early, aiming to produce a project of good quality. At the github repository of the project the exact details of how the project proceeded can be seen. As of 17th September, we had an implementation of the encryption and decryption algorithm in Python, plus the main idea for checking whether a ciphertext can be generated from a given plaintext, and the basics of our c++ code. The initial version of this report was also ready (with logo design etc.). The report was done with LaTeXusing overleaf.

6 ON PERMUTATION CIPHERS & HOW THEY BREAK

Permutation cipher is one of the two common methods to construct ciphers (the other being substitution), whereas substitution replaces plaintext letters or n-grams with other numbers of symbols, permutation ciphers use the same letters in other arrangements. To detect whether permutation or substitution cipher is used, one solution is to look into the frequencies of the symbols, in permutation ciphers the freuquencies would be similar but the letters are changes, and this also means they can be broken similarly, there is a permutation of the letters which makes the frequencies resemble the original frequencies of the plaintext.
A class of permutation ciphers are transposition ciphers, which achieve the permutation by shifting each letter according to a regular system, in the easiest case a constant shift for every letter. Examples of such ciphers are Rail Fence which is a re-invention of much older Scytale used by greeks. In Route Cipher, the plaintext is written on a 2d grid, and the key is then a specific sequence on that grid, by which sequence you can decrypt the message. In the ZigZag cipher the letters are laid out in a zigzag pattern:

D				N				E				T				L		
	E		E		D		H		E		S		W		L		X	
		F				T				A				A				X

FIGURE 7: ZIGZAG CIPHER

The ciphertext becomes "DNETLEEDHESWLXFTAAX", for an actor with knowledge of ZigZag width it is easy to decrypt the message, and by iteratively trying different widths, it can be broken.

Transposition ciphers are prone to frequency analysis, a way to mitigate this is to use *fractionation*, which is to use multiple cipher symbols for a single letter, this is similar to what we faced in this project, and in permutation ciphers this would mean the symbols corresponding to a letter, say 'a' end up being far apart, notable examples of using fractionation is the *VIC Cipher*, used in pen & paper by a soviet spy [10].
For an alphabet of size n the number of possible permutations is $n!$ so even for alphabet as small as 20 letters this number approaches 10^{18} which means an exhaustive search of possible permutations is not a good idea.

Even though permutation and substitution ciphers are not useful anymore, and can be broken, they lived on in another form. In modern block cipher algorithms, a component of the algorithms are S-Boxes which are usef to achieve *confusion* property and hide the relation between key and the cipher, these components are implemented as 2d tables (mxn) which map m-bits of input to n-bits of output. Similarly for permutations, P-Box are used which transpose the bits across the inputs of the S-Boxes, the reason is that this would allow diffusion to be kept while transposition is done.

It has to be noted that permutation ciphers are not only useful for text, but they can be extended to multimedia as well, in that case, the permutation moves around the pixel data in an image to make the image unrecognizable. In a multimedia permutation cipher, a pixel at position (i, j) will be permuted to another position (i', j') therefore by comparing a number of known plaintexts and the corresponding ciphertexts it will be possible for an attacker to reconstruct the secret underlying permutation. Imagine, giving an image with only 1 red pixel as input and getting back the cipher: tracing the red pixel in the cipher, you would be able to deduce $W(i, j)$ where W is a tensor holding the pixel permutation. We can do this more smartly, and in the image we input, track all 256^3 distinct cases of pixels, expect that we should pick 1 to

[10] https://en.wikipedia.org/wiki/VIC_cipher

be the default null pixel to avoid collision, so with each challenge we can deduct $256^3 - 1$ of the elements in W and so for an image of size $w \times h$ it takes $\frac{wh}{256^3-1}$ attempts to deduct the whole key.