Parallel Computing

Log book for CW1

Harrison Marcks

# Introduction

This logbook will be detailing the identification, design, development and benchmarking process of the Parallel Computing AES128 Cipher Cracking coursework assignment.

First, I will detail the identification of the problem as it is stated in the assignment brief. Next will be detailed the research made regarding the problem identified, including any libraries required, programs used, algorithms, methodologies and, benchmarking strategies used. Following this will be the implementation log, tests and finally the benchmarking and results.

Contents

[Introduction 1](#_Toc529314861)

[Contents 1](#_Toc529314862)

[Background and Task Identification 3](#_Toc529314863)

[Research 3](#_Toc529314864)

[Libraries 3](#_Toc529314865)

[Programs 3](#_Toc529314866)

[Algorithms 3](#_Toc529314867)

[Generate and test (Wikipedia, 2008) 3](#_Toc529314868)

[Depth-first (Pelletier-Thibert, 1876) (Pelletier-Thibert, 1876) 3](#_Toc529314869)

[Breadth-First (Zuse, 2003) (Wikipedia, 2012) 4](#_Toc529314870)

[Dictionary Attack (Atwood, 2009) (Wikipedia, 2018) 4](#_Toc529314871)

[Rainbow Table (Oechslin, 2003) (Wikipedia, 2018) 4](#_Toc529314872)

[Benchmarking strategies and Result Presentations 5](#_Toc529314873)

[Design 5](#_Toc529314874)

[Methodology 5](#_Toc529314875)

[Steps 5](#_Toc529314876)

[Tasks 6](#_Toc529314877)

[Implementation 6](#_Toc529314878)

[03/11/18 6](#_Toc529314879)

[04/11/2018 6](#_Toc529314880)

[06/11/18 7](#_Toc529314881)

[Tests 7](#_Toc529314882)

[Timings, Results, Benchmarking 7](#_Toc529314883)

[References 7](#_Toc529314884)

# Background and Task Identification

Advanced Encryption Standard (AES) is a symmetric key encryption algorithm first developed when weaknesses in the original DES key were found. It was developed by two Belgian cryptographers and originally named the “Rijndael Cipher”. It has a 128bit block size with configurable key lengths of 128, 192 and 256 bits.

The task is to create an AES128 cipher cracker, both parallel (openMP and MPI) and serially, then comparing the results and performing some form of benchmarking for each method implemented.

The cipher is to be cracked using an exhaustive brute-force algorithm to try and gain the key and decrypt the file.

# Research

## Libraries

The three main libraries that will be used for this work will be openSSL, openMP and MPI C++ libraries, as detailed in the brief. For the sake of simplicity and few dependencies no other “non-standard” libraries will be used, instead simple classes will be written as required.

## Programs

The only program used will be openSSL. This will be to generate the original AES encrypted text for brute force decryption, as there is little point in spending time implementing something that has already been done to a higher quality.

## Algorithms

There are three suggested brute force algorithms to try and implement, more details will be given below

### Generate and test (Wikipedia, 2008)

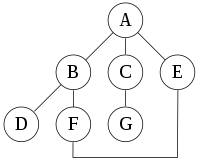
A *generate and test* system of brute-forcing involves each potential solution being generated in turn and tried; regardless of its fit, suitability or likelihood of success. The general idea involves setting a first candidate, validating it and then either, if it’s correct outputting the result and finishing, or generating the next candidate and repeating.

The major issue with this method is how quickly the number of possible answers increases in magnitude, which can be shown using factorials:

A 10-character key has 10! (3,628,800) possible combinations. If there are 11 characters, this means there are now 11! (39,916,800) combinations. An 11x or 1100% increase in possible solutions for a 10% increase in characters.

### Depth-first (Pelletier-Thibert, 1876) (Pelletier-Thibert, 1876)

A depth first algorithm entails going to the very bottom of a tree or graph from the top node first and exhausting each node at the bottom of said tree first before returning back up the tree to move onto the next lowest node.



1

2

3

4

5

6

7

(Image credit: Wikipedia, <https://en.wikipedia.org/wiki/Depth-first_search>, extra marks made by me)

The image above assumes that the search has a left bias, but it shows that the algorithm will go the lowest node that that node points to. The main issue with this algorithm in respect to the assignment is that it requires the depth of the highest node to be known before-hand.

### Breadth-First (Zuse, 2003) (Wikipedia, 2012)

A breadth-first strategy involves traversing the entire level of a tree before moving onto the next level. Using the same graph as above (in depth-first) the algorithm will instead move through:

A -> B -> C -> E -> D-> F -> G

It is in effect the opposite of a depth search node, only instead of drilling down, visits its direct neighbours instead.

### Dictionary Attack (Atwood, 2009) (Wikipedia, 2018)

A dictionary attack is a search algorithm that instead of generating random results or traversing tress, instead will access a list of common passphrases used. These lists can be thousands, even millions, of lines long.

Once it’s accessed this list, it’ll pull a single passphrase, test it, and then either succeed or try with the next on the list. These attacks are often faster than standard Generate-and-Search algorithms because users’ passwords are rarely uniformly spread out.

### Rainbow Table (Oechslin, 2003) (Wikipedia, 2018)

A rainbow table is a precomputed table for reversing cryptographic hash functions. Passwords for users are stored as hashes in databases such that they can only be viewed when the correct password is entered and then re-hashed and compared. Obtaining this hash and then re-entering it would not work however because that would re-hash the password and fail, still requiring a brute force or dictionary attack to compute the hash used. Rainbow tables solve this by needing only the hashed value.

However, describing rainbow tables is beyond the scope (and word limit) of this report, however deserves a mention purely for completeness.

## Benchmarking strategies and Result Presentations

For the purposes of this report the benchmarking will be done only between each attempt at running the de-cipher tool, i.e. comparing serial, openMP and openMPI. Each run will be performed several times with the same key and encrypted text, then the average for each component will be taken and that will be given as the final score for each example.

Each run will be recorded and will hopefully work to build a picture of algorithmic improvement as development goes on. Both as the tasks themselves become more succinct and as the parallelisation is more fine-tuned and improved upon.

# Design

## Methodology

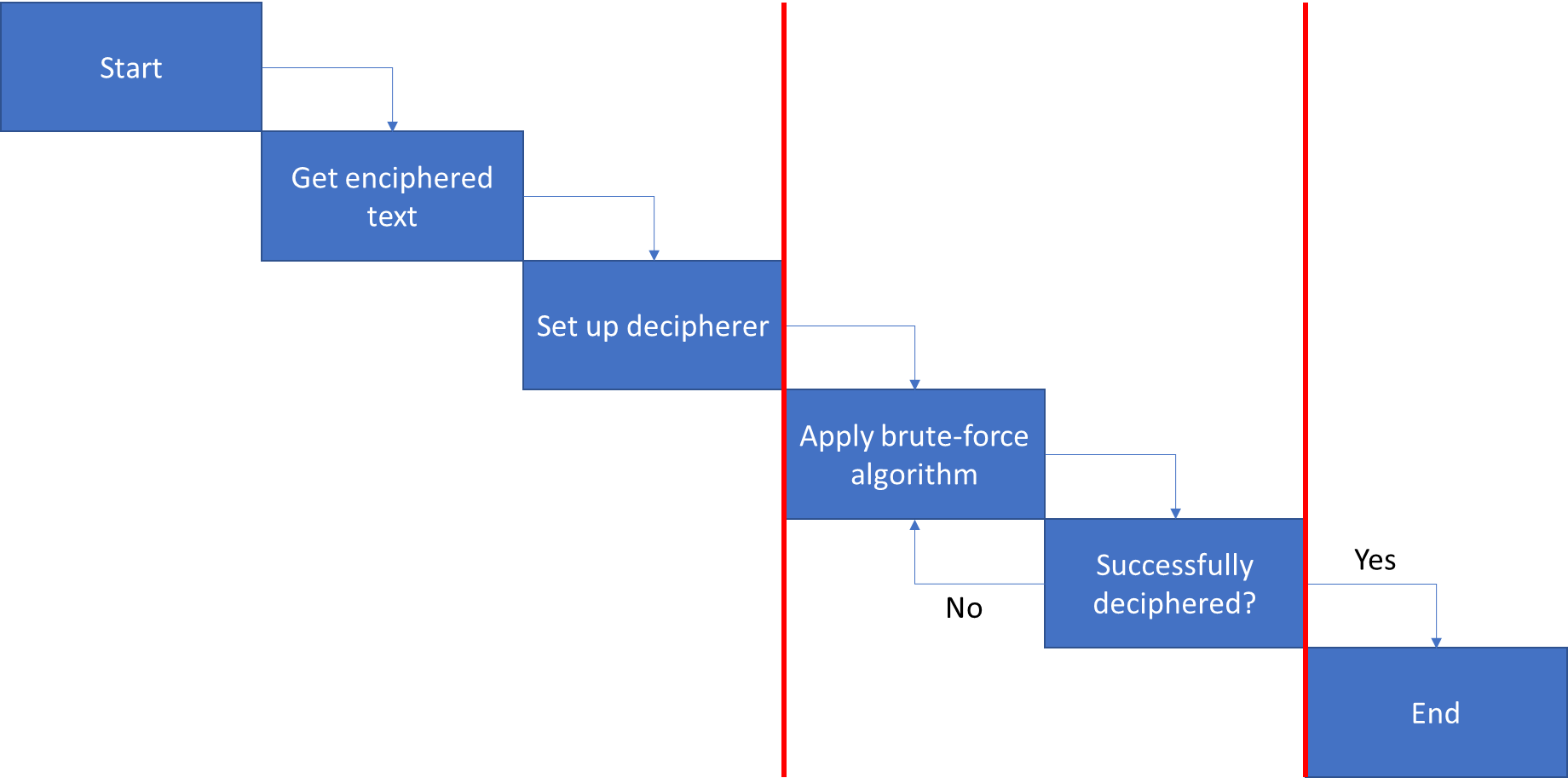
For the purposes and intent of this report, a Generate-and-Test brute-force search will be used to perform the key finding. After defining the generic steps, the best area to parallelise needs to be identified. Once that has been identified, the tasks needed to complete the homogenous and heterogenous tasks need to be singled out and decomposed in an appropriate way for each part.

For versioning, I’ll be using each step as a version. Once the Serial version of the code is done, that will mark version 0.3, once the homogenous part is working, version 0.6, and once the heterogenous part is working, version 0.9.

Version 1 will represent each component working separately to solve the problem. Subsequent versions will take place as improvements are made both to general program running and parallelised algorithms.

## Steps

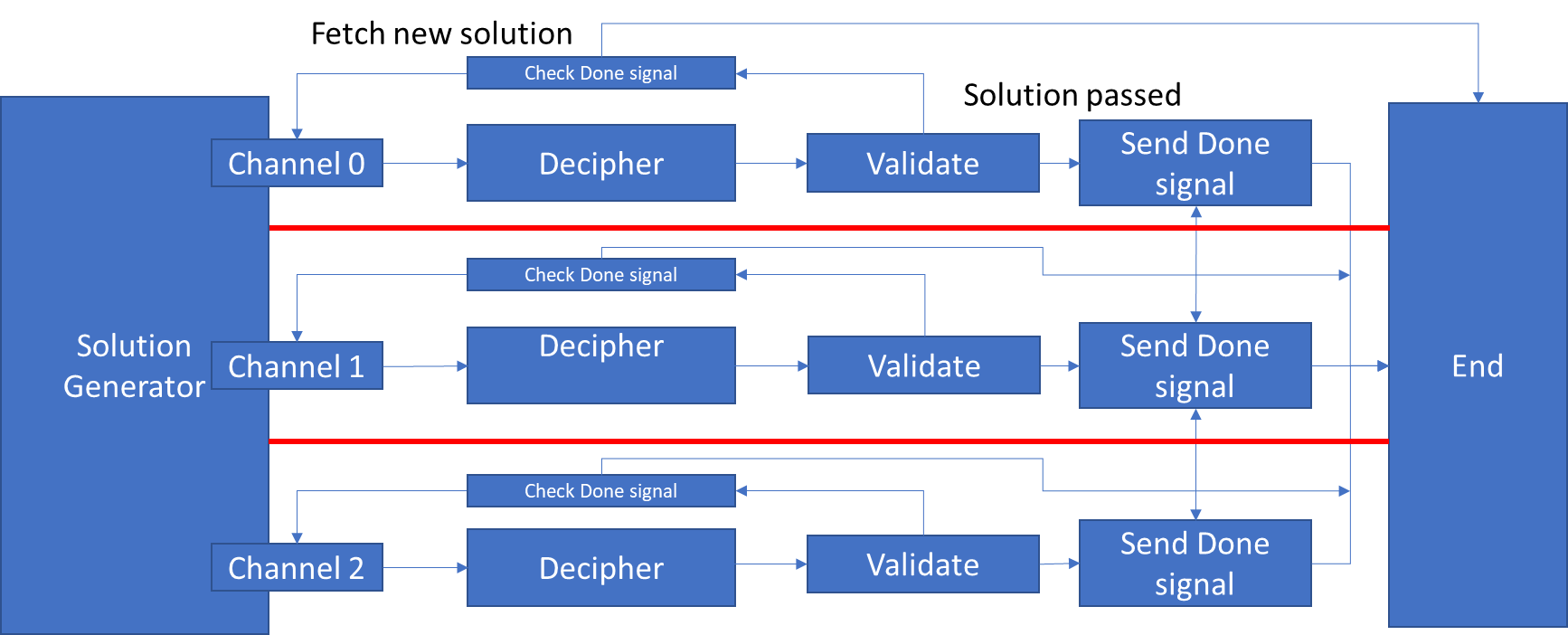
The basic steps the program needs to take consist of:



And this will be largely true for all methods, however the parallelisation will occur in during the “Apply Brute-Force Algorithm” and during the “Successfully Deciphered?” parts of the running.

Regarding the threads, each one will need someway of communicating to the other threads that a result has been found.

As for handing out generated data, the best way to do that, I believe, would be by having a system that auto-generates n amount of possible solutions at once and simply dishes them out as they’re needed, in its own thread. As soon as a solution is “taken”, it generates a new one. It can do this on several different “channels”. Each channel being used specifically by a single thread. The issue here is if the solution generator cannot generate solutions as quickly as each thread conducts its tests.



It’ll also be important that no two threads perform the same test, as that could waste time quite considerably. So, the generator needs to make sure every solution is unique to every other solution. It could be beneficial to store up solutions in a queue to make sure that there is at least some backlog for the decipherers to draw from.

## Tasks

As far as serial running is concerned, the steps outlined previously will be performed as a single thread. It just won’t need to perform an error check or send a done signal, so those functions will be stripped out. Lessons from making the serial part will also help to improve efficiency in the threads. Like deciding which objects need to be deleted and re-created, and which ones can just be hot-swapped and changed that way, without having to repeatedly create and destroy objects.

As for the actual decryption part, according to the top answer (random65537, 2011), all that’s needed to decrypt is a *pair* of deciphered blocks that can be *xor*’d together; which will boil down to a reduction algorithm. To decrypt the enciphered text the EVP\_DecryptUpdate() function must be called (potentially) multiple times. This means that this area of the deciphering can also be performed in parallel, thus improving the speed.

# Implementation

## 03/11/18

solutionGenerator, KeyStore and threadHandler classes are created. Immediately It is apparent that the algorithm for incrementing the string will be tricky. Plan involving string manipulation, trying to avoid too many objects floating around.

## 04/11/2018

keySegment class was created as an answer to the previous sessions problems. Once that was tested and proven to be working I moved on to creating the decipher handler and the decipher agents. The Handler will track each agent. Each agent will be dedicated to deciphering the text using each solution provided to it. How to perform the decryption in an OO way that works still eludes me.

## 06/11/18

After talking out how the decryption functions work with a course mate, reviewing my own knowledge and doing some extra research, I concluded that each decryption block of the enciphered text can be deciphered in parallel. I also updated the unsigned chars to instead use uint8\_t\*, after also realising that the key and initialisation vector will always be a set size of 16bytes (or 128bits, as per the name of the encryption method. This has massively improved performance compared to using std::strings and objects per key segment. Couple with my original issues of converting between char and unsigned char. It also helped me to simplify how I had to perform the updates across different key blocks.

After a great deal of digging I found an example of looping over the EVP\_CipherUpdate() function from (Kulkarni, 2017). This gave me a good example (and basis) on how to use the EVP\_DencryptUpdate() function in a loop and, by proxy put that into a parallelised algorithm. The EVP\_CipherUpdate() function is a more generic interface for swapping between encryption and decryption.

The next step now is to create an enciphering class that makes it easier to handle the IV and key (due to the unit8\_t\*’s being used as opposed to char\*’s), then to test everything works in serial. Following that I will create the OpenMPI and OpenMP methods in the deciphering agent. The final part will be creating a timing class that can time how long each method takes to run.

## 09/11/2018



(Graph From: <https://security.stackexchange.com/a/86956> )

The graph above shows that each block is decrypted using the previous block’s enciphered text xor’d with the current blocks key deciphered text. This will form the basis of the parallelised decryption.

I’m still working on the file reader, instead using the C file handling instead of the C++ file streams as it gives me more control over types for easier handling of uint8\_t’s required by the openSSL library.

I am currently facing issues with segmentation faults within the openSSL functions.

## 20/11/18

After many failed attempts I have now decided to dial back the level parallelisation in favour of getting a working prototype out. I have simplified the cipher class to perform only decryption and encryption.

Instead of decrypting the entire enciphered text in parallel, I am now simply performing each decrpytion “as it comes” per working agent.

Each agent will have its own *copy* of the ssl class. Although SSL still refuses to work and no amount of googling is helping in any shape way or form.

## 21/11/2018

After adding the final lengths together the decryption worked! Next step is to plug it into each method (Serial, OpenMP, MPI).

I need agents that will be performing the decription. They will receive a pointer/reference to a queue that contains the key, each time they will simply pop\_front() the queue and test. A seperate thread will handle constant generation of the solutions for all the queues (each agent will have its own queue).

## 23/11/2018

Due to my own misunderstanding of OpenMPs purpose (and my own understanding of modern threading techniques) I have now realised that my old method would not work as OpenMP works exclusively as a branching method. Nothing else. My new strategy involves making copies for anything that might be shared, manually, using locks to ensure no conflict, and locks are now also employed on the key generator to ensure no conflicts ensure (and to make sure that the sharing is made explicit instead of relying on OpenMPs implicity).

## 28/11/2018

Having completed the OpenMP part, I moved onto MPI. After establishing how to use it, my plan was to emulate what openmp does as close as possible (as I have already proven that it is quicker than serial).

However a manjor issue in MPI is managing sends and receives effectively. Having a kill switch for other kernels is also vital in making sure the program actually *ends.* MPI::Comm::AllReduce was used with the MPI\_LOR flag to or together all threads done flags as this seemed a much quicker and explicit way of performing this check (compared to MPI::Comm::AllGather/AllGatherV).

The Master is in charge of dishing out keys to other kernels as they need it. This is achieved by each thread sending a request tag to the master thread and then waiting to receive the key (with a key tag). The Master performs Irecv’s on each possible kernel polling for another kernel to send it a request. When a request is received, ti sends that kernel a new key. The solution builds, however as I am not on campus I cannot test it. It does build however, so instead I’ll work on refactoring my current solution to make it more readable.

## 03/12/2018

I found out today that I can ssh into the cluster from off-campus. I cloned my repo and wrote a proper yet simple makefile for the project, using the mpic++ compiler.

The first issue I came across was that the c++ compiler was older than the one I use at home. After fixing issues caused by compiler differences, I discovered that, for some reason, the openMP pragma line causes a segfault in the program (even though it works perfectly fine with the mpic++ compiler on my home machine).

Seperating the openMP and MPI tasks into seperate executables (using someithng akin to abstract factories). I’m no closer to finding the solution for openMPs segfaulting. Small scale openMP tests have shown that openMP *can* run, however, I’ve yet to figure out the issue with mine.

As a work-around I’ve decided to build two seperate executables using ifdef’s.

## 05/12/2018

Having discovered a segfault in my implemntation of the MPI segment, I’ve been working on figuring out where that’s going wrong.

I’ve also updated all the README.md files to match the current new (and settled) implementation, i.e. having two seperate executables so that mpi can actully get a point where it can run.

# Tests

# Timings, Results, Benchmarking

# References

Atwood, J. (2009) *Dictionary Attacks 101* [Online]. Available from: https://blog.codinghorror.com/dictionary-attacks-101/ [Accessed 31 October 2018].

Bull, I. (2018) *Tutorial: AES Encryption and Decryption with OpenSSL* [Online]. Available from: https://eclipsesource.com/blogs/2017/01/17/tutorial-aes-encryption-and-decryption-with-openssl/ [Accessed 31 October 2018].

Gerrie Veerman, D.P. (2012) *Distributed Password Cracking Platform*. PhD Thesis. Amsterdam: Universiteit van Amsterdam Universiteit van Amsterdam, System & Network Engineering.

Kulkarni, A. (2017) *Encrypting/Decrypting a file using OpenSSL EVP* [Online]. Available from: https://medium.com/@amit.kulkarni/encrypting-decrypting-a-file-using-openssl-evp-b26e0e4d28d4 [Accessed 06 November 2018].

Manavski, S.A. (2007) CUDA COMPATIBLE GPU AS AN EFFICIENT HARDWARE. *IEEE International Conference on Signal Processing and Communications*, pp.24-27.

Mason, J. (2017) *Advanced Encryption Standard (AES)* [Online]. Available from: https://thebestvpn.com/advanced-encryption-standard-aes/ [Accessed 31 October 2018].

Oechslin, P. (2003) Making a Faster Cryptanalytic Time-Memory. *Advances in Cryptology - CRYPTO 2003*, 2729, pp.617-32.

openSSL. (2018) *Enc* [Online]. Available from: https://wiki.openssl.org/index.php/Enc [Accessed 31 October 2018].

OpenSSL. (n.d.) *https://www.openssl.org/docs/man1.0.2/crypto/EVP\_EncryptInit.html* [Online]. Available from: https://www.openssl.org/docs/man1.0.2/crypto/EVP\_EncryptInit.html [Accessed 31 October 2018].

Pelletier-Thibert, J. (1876) École polytechnique of Paris. *French engineer of the telegraph*, X.

random65537. (2011) *Do any crypto libraries take advantage of Windows GPU API Direct Compute* [Online]. Available from: https://security.stackexchange.com/questions/8678/do-any-crypto-libraries-take-advantage-of-windows-gpu-api-direct-compute [Accessed 31 October 2018].

Roraz. (2015) *Why does the first call to DecryptUpdate in AES-CBC return 16 fewer bytes?* [Online]. Available from: https://security.stackexchange.com/a/86956 [Accessed 09 November 2018].

Shneir, B. (2007) *Secure Passwords Keep You Safer* [Online]. Available from: https://www.schneier.com/essays/archives/2007/01/secure\_passwords\_kee.html [Accessed 31 October 2018].

Wikipedia. (2008) *Brute Force Search* [Online]. Available from: https://en.wikipedia.org/wiki/Brute-force\_search [Accessed 31 October 2018].

Wikipedia. (2010) *Depth First Search* [Online]. Available from: https://en.wikipedia.org/wiki/Depth-first\_search [Accessed 31 October 2018].

Wikipedia. (2012) *Breadth First Search* [Online]. Available from: https://en.wikipedia.org/wiki/Breadth-first\_search [Accessed 31 October 2018].

Wikipedia. (2018) *Dictionary Attack* [Online]. Available from: https://en.wikipedia.org/wiki/Dictionary\_attack [Accessed 31 October 2018].

Wikipedia. (2018) *Rainbow table* [Online]. Available from: https://en.wikipedia.org/wiki/Rainbow\_table [Accessed 31 October 2018].

Zaro, J. (16) *How to use OpenSSL in GCC?* [Online]. Available from: https://stackoverflow.com/questions/1894013/how-to-use-openssl-in-gcc [Accessed 31 October 2018].

Zuse, K. (2003) Konrad Zuse in Hopferau im Allgäu—Z4 und Plankalkül. *Informatik-Spektrum*, 26(5), pp.