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](#_Toc528772828)

[Contents 1](#_Toc528772829)

[Background and Task Identification 2](#_Toc528772830)

[Research 2](#_Toc528772831)

[Libraries 2](#_Toc528772832)

[Programs 2](#_Toc528772833)

[Algorithms 2](#_Toc528772834)

[Generate and test (Wikipedia, 2008) 2](#_Toc528772835)

[Depth-first (Pelletier-Thibert, 1876) (Pelletier-Thibert, 1876) 2](#_Toc528772836)

[Breadth-First (Zuse, 2003) (Wikipedia, 2012) 3](#_Toc528772837)

[Dictionary Attack (Atwood, 2009) (Wikipedia, 2018) 3](#_Toc528772838)

[Rainbow Table (Oechslin, 2003) (Wikipedia, 2018) 3](#_Toc528772839)

[Benchmarking strategies and Result Presentations 4](#_Toc528772840)

[Design 4](#_Toc528772841)

[Methodology 4](#_Toc528772842)

[Steps 4](#_Toc528772843)

[Tasks 4](#_Toc528772844)

[Implementation 4](#_Toc528772845)

[Tests 4](#_Toc528772846)

[Timings, Results, Benchmarking 5](#_Toc528772847)

[References 5](#_Toc528772848)

# 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)

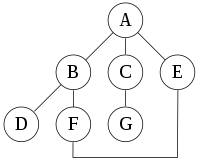
A *generate and test* system of brute-forcing involves each potential solution being generated in turn and tried; regardless of its fit, suitability or likelihood of success. The general idea involves setting a first candidate, validating it and then either, if it’s correct outputting the result and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

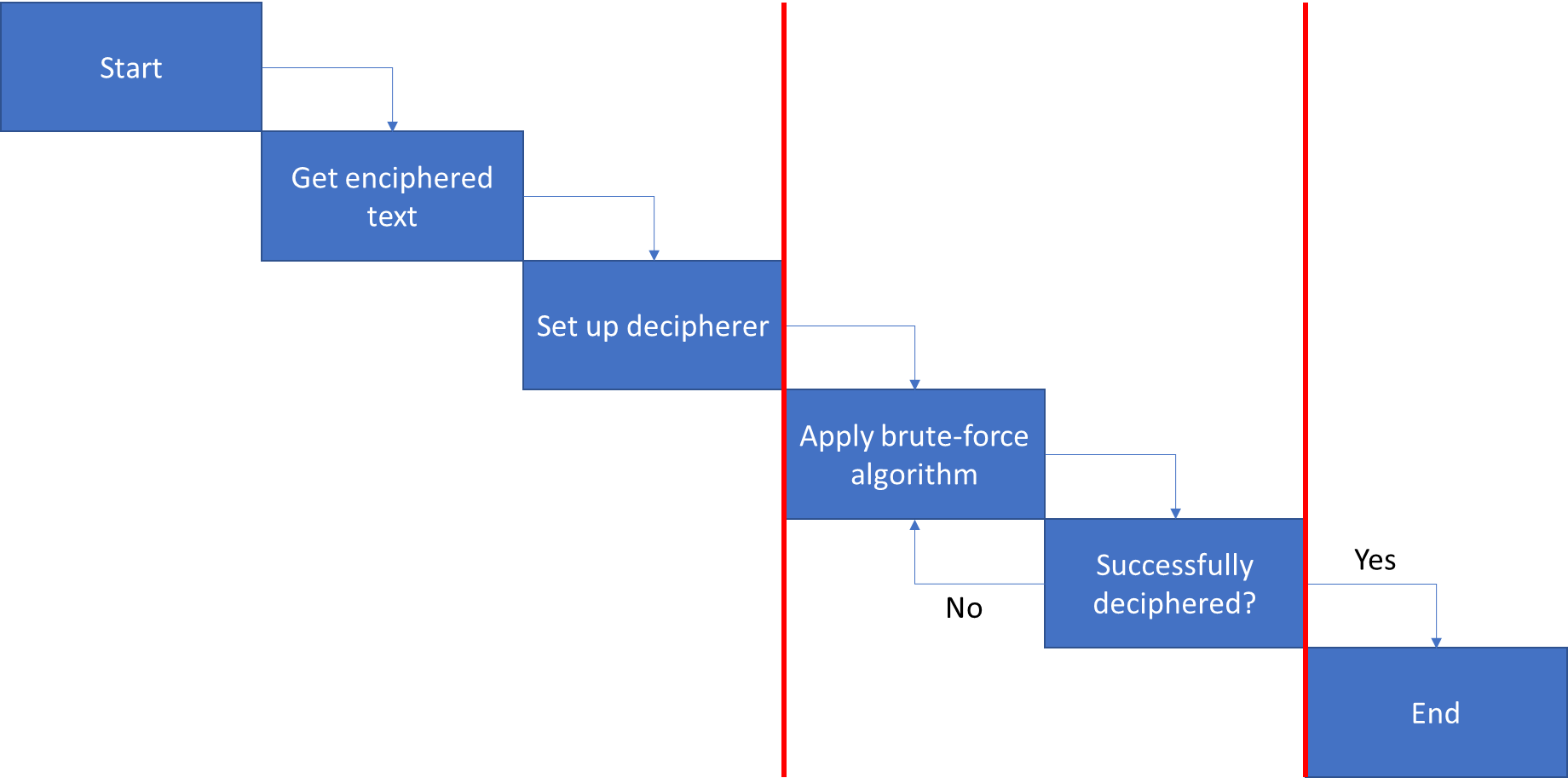
For the purposes and intent of this report, a Generate-and-Test brute-force search will be used to perform the key finding. After defining the generic steps, the best area to parallelise needs to be identified. Once that has been identified, the tasks needed to complete the homogenous and heterogenous 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 paralleised algorithms.

## Steps

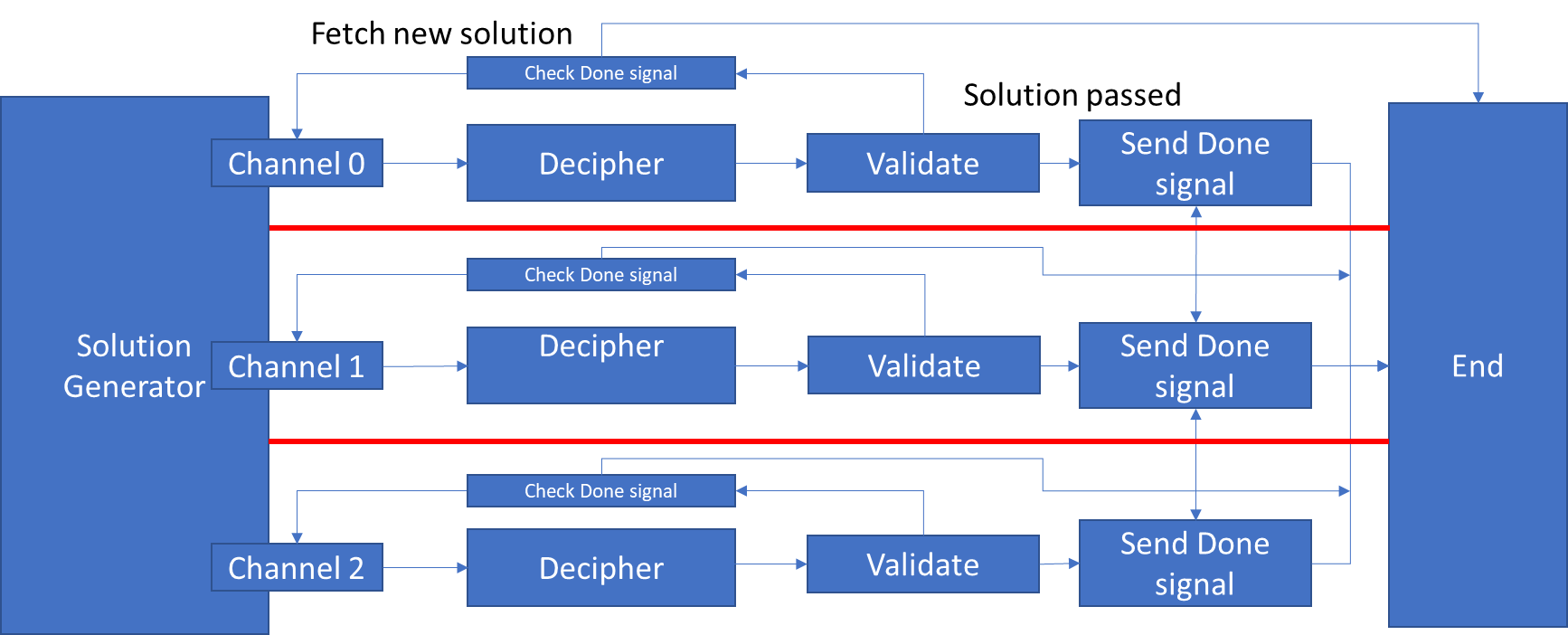
The basic steps the program needs to take consist of:



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. However, figuring out the lower-level gubbins of the AES decryption is beyond the scope of the work. Instead I will be settling for many different instances of the entire decryption unit, with each unit running in parallel but not the decryption itself (per solution).

# 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 a classified way that actually works still eludes me.

# 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.

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].

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.354-58.