



PARALLEL PROGRAMMING COURSEWORK LOGBOOK



Matt Gordon
UWE 10026602

Introduction

This logbook describes the work undertaken to design and develop an encryption brute force search application capable of operating in three modes:

- Sequential search
- OpenMP parallelisation
- Distributed MPI parallelisation

Week 5 (24/10/18-30/10/18)

Summary

Time during this week was mostly spent researching existing design methods for AES encryption/decryption to determine how to approach the problem.

Problem Definition

The task requires that an aes-128-cbc cipher key is discovered when both the plaintext and cipher text is known. Given that the key is 128 bits wide, the number of combinations are 2^{128} assuming that all values are available.

Initial research has identified several brute force search methods including:

1. Generate and test
2. Depth-first search
3. Breadth first search

For this problem depth-first and breadth-first search do not provide any benefit over generate and test and will require a more complex approach, potentially decreasing performance. Therefore a simple generate and test approach seems to be the most sensible in this instance.

Generate and test

Generate-and-test is a “very simple algorithm that guarantees to find a solution if done systematically and there exists a solution” (Robin, 2009). When applied to the problem given, it could be implemented in the following way:

1. Generate a valid aes-128-cbc key.
2. Encrypt the plaintext using the key.
3. Test if the known cipher text matches the new encrypted text.
4. If it does, the key has been found. If not return to step 1.

AES Encryption

The advanced encryption standard has been in use since the late 1990s and been an approved method of encryption within the US government since 2002 (McCallion, 2018). It replaced the Data Encryption Standard (DES) due to concerns over the ease at which DES could be cracked by brute-force, differential cryptanalysis and linear cryptanalysis (McCallion, 2018).

AES is defined as a substitution permutation network block cipher algorithm. The algorithm takes a block of plain text and applies alternating rounds of substitution and permutation boxes to it (McCallion, 2018). These can be 128, 192 or 256 bits in size however 128 is typical as it is considered to be both collision resistant and resistant against known forms of attacks.

This task requires encryption using AES-128-CBC. AES being the encryption standard. 128 being the size of key. CBC meaning cipher block chaining where each block of plaintext is XORed with the

previous cipher text block before being encrypted. To ensure each message is unique, an initialisation vector is specified in the first block. (Rouse, 2007).

Week 6 (31/10/18-6/11/18)

Summary

Tests using the OpenSSL C++ library were attempted to determine how much of the application can make use of library functions and how much will need to be implemented from scratch.

OpenSSL Testing

The OpenSSL wiki provides an example of AES-128-CBC encryption:

https://wiki.openssl.org/index.php/EVP_Symmetric_Encryption_and_Decryption

From this I was able to modify the code and substitute values provided as part of the task:

Key: 0x23232323232373616d706c652323232323

Plaintext: "This is a top secret."

IV: 0x010203040506070809000a0b0c0d0e0f

Cipher text: 0xd6546e4c8ba8751e586c3743f521ef390253aaff4c3150b2f7ab8d6fa34559f8

I could then confirm that the test program produced the same cipher text provided in the initial task specification.

```
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <string.h>
#include <iostream>
using namespace std;

void handleOpenSSLErrors(void) {
    ERR_print_errors_fp(stderr);
    abort();
}

int main (void) {
    unsigned char ciphertext[128] = {0};
    unsigned char key[16] = {0x23, 0x23, 0x23, 0x23, 0x23, 0x73, 0x61, 0x6d, 0x70, 0x6c, 0x65, 0x23, 0x23, 0x23, 0x23, 0x23};
    unsigned char iv[16] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};
    unsigned char plaintext[21] = {0x54, 0x68, 0x69, 0x73, 0x20, 0x69, 0x73, 0x20, 0x61, 0x20, 0x74, 0x6f, 0x70, 0x73, 0x65, 0x63, 0x72, 0x65, 0x74, 0x2e};
    int plaintext_len = 21;

    EVP_CIPHER_CTX *ctx;

    int len;

    int ciphertext_len;

    if(!(ctx = EVP_CIPHER_CTX_new())) handleOpenSSLErrors();

    if(1 != EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv))
        handleOpenSSLErrors();

    if(1 != EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len))
        handleOpenSSLErrors();
    ciphertext_len = len;

    if(1 != EVP_EncryptFinal_ex(ctx, ciphertext + len, &len)) handleOpenSSLErrors();
    ciphertext_len += len;

    EVP_CIPHER_CTX_free(ctx);

    for (int i = 0; i < ciphertext_len; i++) {
        cout << hex << (int)ciphertext[i];
    }

    cout << endl;

    return ciphertext_len;
}
```

Figure 1 - Simple AES Test Application

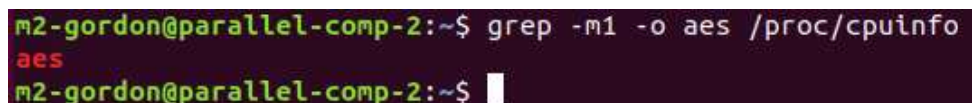
Encryption Performance

Whilst researching the OpenSSL library, it was found that many newer processors support hardware encryption/decryption using AES through an “extension to the x86 instruction set architecture for microprocessors from Intel and AMD” (Gite, 2018).

It is possible to determine if the host processor supports the AES-NI instruction set through the following command:

```
# grep -m1 -o aes /proc/cpuinfo
```

When run on one of the hosts designated for testing, it showed that they supported this. It was again repeated on my personal development system which also showed support.



```
m2-gordon@parallel-comp-2:~$ grep -m1 -o aes /proc/cpuinfo
aes
m2-gordon@parallel-comp-2:~$
```

Figure 2 - Support for AES-NI

This means that if implemented correctly, OpenSSL encryption and decryption functionality should be accelerated through the use of special processor instructions leading to a significant performance boost.

Week 7 (07/11/18-13/11/18)

Summary

Problems whilst developing the sequential search method meant that an initial prototype was developed, but no parallelisation was attempted at this stage.

Sequential Search

Building on the OpenSSL test application written last week, an iterative search loop was developed to test a set of keys that increased in value by encrypting the provided plaintext and comparing against the provided cypher text.



One of the design decisions made at this stage was how to store the key and initial vector, both of which are 128 bits. C or C++ both have no built-in data types to support a variable of this size - an integer is only guaranteed to be at least 32 bits in size (Naman-Bhalla, n.d.) Whilst it is possible to simply use an unsigned char array or four unsigned integers, it is significantly more complex to do arithmetic on them when treated as a group. This led to the design decision of developing the application in C++ as the “unistd.h” library provides access to fixed size data types of which uint128_t (unsigned integer 128 bits in size) is available. Using this data type, performing arithmetic such as incrementing all the way from 0 to 2^{128} can be achieved using standard C++ operators such as ‘++’ or ‘+=’ without needing to handle integer overflows/underflows.

With this in mind, a simple application that started at a base value and iterated up to the known key was developed. Unfortunately when this was tested, the known key and initial vector did not produce the known cipher text.



Endianness

Through extensive debugging and comparison with the OpenSSL test application written last week, it was determined that the cause of the bug was in a conversion between the uint128_t datatype and the 128 bit char array that was expected by the OpenSSL library.

My initial understanding of this problem was that I was able to take advantage of using C++ as a programming language for this task by using a reinterpret cast on the uint128_t to a char[16] as they were both stored the same way in memory as shown below:

	uint128_t – Value = 0x000102030405060708090a0b0c0d0e0f															
Memory Location																
Value	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	char array[16] = {0x00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 0A, 0B, 0C, 0D, 0E, 0F}															
Memory Location																
Value	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Upon investigation, it was found that the due to endianness of the processor, the data was in fact being stored as shown below:

	uint128_t – Value = 0x000102030405060708090a0b0c0d0e0f															
Memory Location																
Value	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
	char array[16] = {0x00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 0A, 0B, 0C, 0D, 0E, 0F}															
Memory Location																
Value	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0

However, the OpenSSL library was expecting a char array with the MSB being in position 0, like is shown in the first table. This meant the array was reversed in byte order, giving the wrong key. It is also important to note that this is not guaranteed to happen, and may not happen on all processors (Walls, 2015) meaning that a function cannot be written to correct the ordering (this would also cause significant performance degradation).

To get around this, a char array was used to store the key and initial vector, with a special “next_key” function written to handle byte overflow upon key generation.

Week 8 (14/11/18-20/11/18)

Summary

Optimisations to the sequential search algorithm were made and an OpenMP search method was successfully created.

Key Optimisations

A significant optimisation was made to reduce the search time of the application. Through task feedback it was determined that the key would never contain unprintable ASCII characters, meaning that tested values of each 16 bytes of the key would reduce from 256 to 94 (values 32-126).

To deal with the complexities this brings, the “next_key” function was modified to skip certain values whilst still not creating too much of a performance bottleneck during key generation.

Another optimisation was made where a “key mask” was created to ensure that only certain bits of the key were searched. This would be useful if specific portions of the key were already known to

the user so they were not searched unnecessarily. The key mask is modifiable in the code but not during run time.

OpenMP Parallelisation

An OpenMP search function was created to utilise the OpenMP library. It is used for multi-threaded parallel processing and suitable for shared-memory multi-processor (or core) computers (Boettcher, 2009). It has an advantage that code does not need to be modified as much as a full MPI implementation and loops can easily be parallelised through compiler instructions.

Unfortunately, design decisions for the sequential search function meant that converting the function to be compatible with OpenMP was not trivial. This was due to the basic generate and test search method in use.

The current method of searching was:

1. Encrypt the plaintext using the current key.
2. Test if the known cipher text matches the new encrypted text.
3. If it does, the key has been found. If not call "next_key" to find the next valid key and store inside a class variable, go to step 1.

This means that there is a single class variable holding the current key that is constantly being updated. If the main loop were converted to use OpenMP, this would introduce many race conditions and would not perform an exhaustive search (if the program was able to run at all).

To get around this, the method was divided in to two parts:

1. Key generation

A batch of 1 million keys is generated and stored inside dynamic memory.

2. A loop iterates over and tests these keys. If all keys have been tested and none are successful then another 1 million keys are generated and the process is repeated.

As the program is now working from a list of static pre-stored keys for most of the execution time, this loop can easily be parallelised through threading by using the "#pragma omp for" instruction.

Parallelisation can easily be confirmed as system performance monitors for the process were now showing above 100% indicating that more than one core was in use.

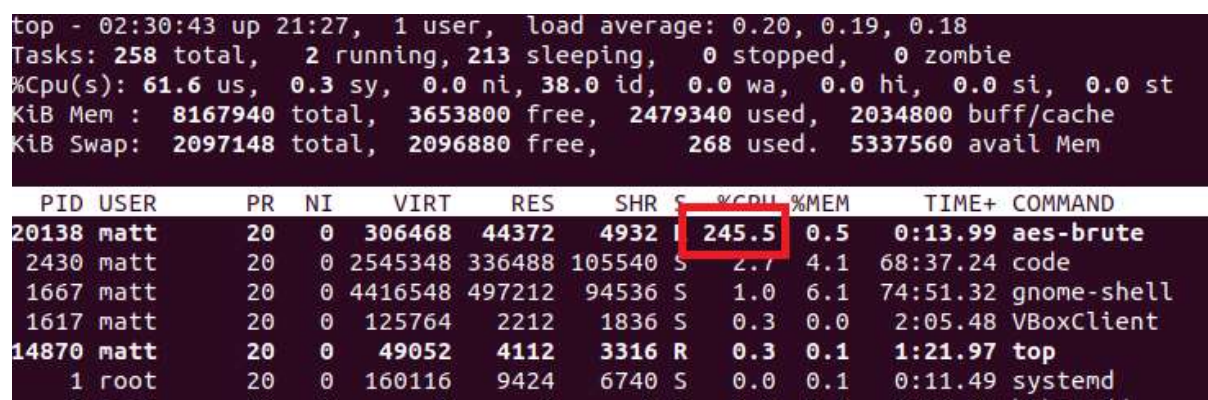


Figure 3- Process CPU Usage

Week 9 (21/11/18-27/11/18)

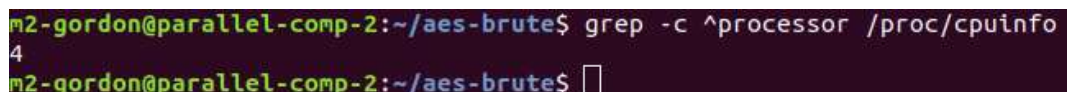
Summary

An MPI search function was created successfully providing significant performance gains.

MPI

The message passing interface allows for true parallelism by running multiple instances of an application across several nodes. These nodes will each have access to their own processor or core and will not share memory with other nodes (Kendall, n.d.).

To estimate how much benefit MPI would give, the command 'grep -c ^processor /proc/cpuinfo' can be run on the host to determine how many processor cores are available:



```
m2-gordon@parallel-comp-2:~/aes-brute$ grep -c ^processor /proc/cpuinfo
4
m2-gordon@parallel-comp-2:~/aes-brute$
```

Figure 4 - Processor Core Count

In this instance the host contains 4 cores.

MPI was installed on the test system using the command 'sudo apt-get install libopenmpi-dev' which gave access to the mpic++ command which would allow compilation of the application using mpi include files.

The implementation of an MPI function created a new significant design challenge regarding key generation. Initially it was thought that the simplest way was to give each node a set of keys to test encrypt with - a similar approach of the OpenMP function, where a million keys would be generated and given to a node to process, another million would then be generated and handed to another node etc. When the first node was finished, it would be handed another batch and the cycle would continue. When prototyping this, it was determined that the passing of a million keys between nodes (potentially over a network) was too slow. The alternative was to send a starting key to each node and let each node generate the next key locally which would be much better with regards to performance.

This method was implemented successfully with each node being passed a starting key, generating and testing the next 1 million keys, then requesting a new starting key. The master process is only responsible for handing out starting keys and does not test any encryption keys. This results in very low processor utilisation for the master node, and is possibly an area that could be looked at in the future for additional performance gains.

It was not possible to test parallelisation across multiple host computers due to a misconfiguration of the hosts. When the option '-nolocal' was given to the 'runmpi' command, the following error was received:

```
-----
No nodes are available for this job, either due to a failure to
allocate nodes to the job, or allocated nodes being marked
as unavailable (e.g., down, rebooting, or a process attempting
to be relocated to another node when none are available).
-----
```

Week 10 (28/11/18-4/11/18)

Summary

The final week of development was spent making improvements to the code readability and performance testing. See the performance testing documentation for further details.

Bibliography

- Boettcher, M. (2009, March 23). *What is OpenMP*. Retrieved from Dartmouth University:
https://www.dartmouth.edu/~rc/classes/intro_openmp/print_pages.shtml
- Gite, V. (2018, August 29). *How to find out AES-NI (Advanced Encryption) Enabled on Linux System*. Retrieved from Cyber Citi: <https://www.cyberciti.biz/faq/how-to-find-out-aes-ni-advanced-encryption-enabled-on-linux-system/>
- Kendall, W. (n.d.). *MPI Tutorial Introduction*. Retrieved from MPI Tutorial:
<http://mpitutorial.com/tutorials/mpi-introduction/>
- McCallion, J. (2018, 05 10). *What is AES encryption?* Retrieved from ITPro:
<https://www.itpro.co.uk/security/29671/what-is-aes-encryption>
- Naman-Bhalla. (n.d.). *C++ Data Types*. Retrieved from Geeks for Geeks:
<https://www.geeksforgeeks.org/c-data-types/>
- Robin. (2009, 12 14). *GENERATE-AND-TEST SEARCH*. Retrieved from World of Computing:
<http://intelligence.worldofcomputing.net/ai-search/generate-and-test-search.html>
- Rouse, M. (2007, June). *Cipher Block Chaining*. Retrieved from Tech Target:
<https://searchsecurity.techtarget.com/definition/cipher-block-chaining>
- Walls, C. (2015, October 15). *Endianness*. Retrieved from Embedded:
<https://www.embedded.com/design/mcus-processors-and-socs/4440613/Endianness>