Name: Andrew Hayes
Student ID: 21321503
E-mail: a.hayes18@universityofgalway.ie

CT437

2025–03–22

Assignment 2: Using & Benchmarking Block Ciphers with OpenSSL

# 1 Block Cipher Benchmarking

| Cipher | Key Size (bits) | Mode | Data Size (MB) | Encryption Time (s) | Decryption Time (s) |
|---|---|---|---|---|---|
| AES | 128 | ECB | 100 | 0.011645 | 0.016149 |
| AES | 128 | ECB | 1000 | 0.115766 | 0.138276 |
| AES | 128 | CBC | 100 | 0.012991 | 0.015155 |
| AES | 128 | CBC | 1000 | 0.120283 | 0.144772 |
| AES | 128 | CTR | 100 | 0.013145 | 0.014195 |
| AES | 128 | CTR | 1000 | 0.123406 | 0.148595 |
| AES | 256 | ECB | 100 | 0.011633 | 0.016037 |
| AES | 256 | ECB | 1000 | 0.117581 | 0.146535 |
| AES | 256 | CBC | 100 | 0.012724 | 0.014844 |
| AES | 256 | CBC | 1000 | 0.123072 | 0.152202 |
| AES | 256 | CTR | 100 | 0.010960 | 0.014497 |
| AES | 256 | CTR | 1000 | 0.125039 | 0.146826 |
| ARIA | 128 | ECB | 100 | 0.513804 | 0.514874 |
| ARIA | 128 | ECB | 1000 | 5.165819 | 5.148774 |
| ARIA | 128 | CBC | 100 | 0.520173 | 0.513266 |
| ARIA | 128 | CBC | 1000 | 5.163050 | 5.168916 |
| ARIA | 128 | CTR | 100 | 0.523422 | 0.516402 |
| ARIA | 128 | CTR | 1000 | 5.185706 | 5.171717 |
| ARIA | 256 | ECB | 100 | 0.511218 | 0.523186 |
| ARIA | 256 | ECB | 1000 | 5.175878 | 5.191240 |
| ARIA | 256 | CBC | 100 | 0.531489 | 0.520085 |
| ARIA | 256 | CBC | 1000 | 5.211425 | 5.235387 |
| ARIA | 256 | CTR | 100 | 0.520419 | 0.526104 |
| ARIA | 256 | CTR | 1000 | 5.242562 | 5.266137 |
| Camellia | 128 | ECB | 100 | 0.430678 | 0.422496 |
| Camellia | 128 | ECB | 1000 | 4.305158 | 4.320622 |
| Camellia | 128 | CBC | 100 | 0.435216 | 0.440686 |
| Camellia | 128 | CBC | 1000 | 4.362768 | 4.397620 |
| Camellia | 128 | CTR | 100 | 0.446206 | 0.442398 |
| Camellia | 128 | CTR | 1000 | 4.446463 | 4.461865 |
| Camellia | 256 | ECB | 100 | 0.441961 | 0.441616 |
| Camellia | 256 | ECB | 1000 | 4.473938 | 4.449186 |
| Camellia | 256 | CBC | 100 | 0.447059 | 0.440549 |
| Camellia | 256 | CBC | 1000 | 4.448629 | 4.426780 |
| Camellia | 256 | CTR | 100 | 0.436507 | 0.451490 |
| Camellia | 256 | CTR | 1000 | 4.416296 | 4.432459 |

Table 1: Benchmarking results from TSV file

To benchmark the CPU time of my program, I used the standard POSIX `getrusage()` function declared in the `<sys/resource.h>` header, and wrote the collected experimental data to a tab-separated value (TSV) file defined in my `benchmark.h` header file, as I generally prefer TSV to CSV due to its increased human-readability in plaintext form. I made sure to only open and write to the results file *after* each benchmark had been measured to ensure that the data was accurate. The above table of results is generated automatically from the TSV file using the LaTeX `pgfplots` package, and the below bar charts were generated using a simple Python script. Since I already run a Linux-based operating system on my laptop, I was able to run these benchmarks natively on my machine, both making it slightly easier for me to run the tests but also likely greatly improving the performance of the block ciphers as the program had full access to my system resources instead of limited virtualised hardware.

```
[andrew-hayes@arch] ~/currsem/CT437/assignments/assignment2/code  (master) +
 % ls                                                                    ✓
Permissions Size User   Date Modified    Git Name
.rw-r--r--  3.6k andrew 2025-03-20 22:14  -- benchmark.c
.rw-r--r--   589 andrew 2025-03-20 22:14  -- benchmark.h
.rw-r--r--  1.5k andrew 2025-03-21 02:29  -- plot.py
[andrew-hayes@arch] ~/currsem/CT437/assignments/assignment2/code  (master) +
 % gcc -o benchmark benchmark.c -lcrypto -lrt                            ✓

[andrew-hayes@arch] ~/currsem/CT437/assignments/assignment2/code  (master) +
 % ./benchmark                                                        ✓ 1s
Benchmarking completed. Results saved in results.tsv
[andrew-hayes@arch] ~/currsem/CT437/assignments/assignment2/code  (master) +
 % python3 plot.py results.tsv                                    ✓ 2m 13s
[andrew-hayes@arch] ~/currsem/CT437/assignments/assignment2/code  (master) +
```

Figure 1: Compiling and running the benchmarking program

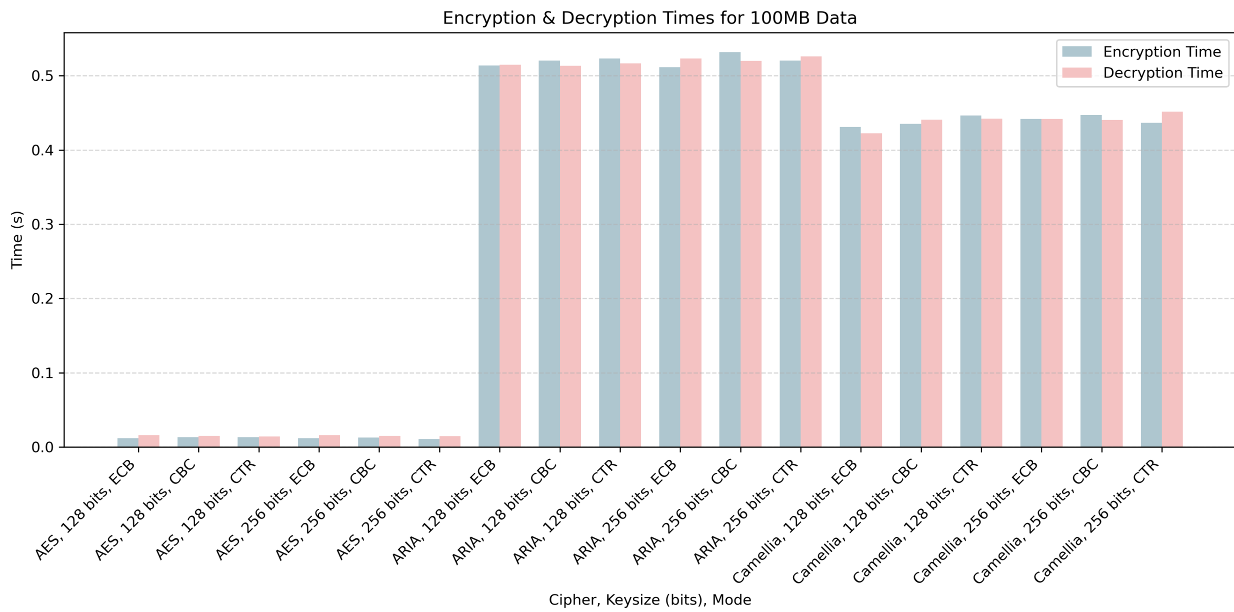


Figure 2: Encryption & decryption times for 100MB data

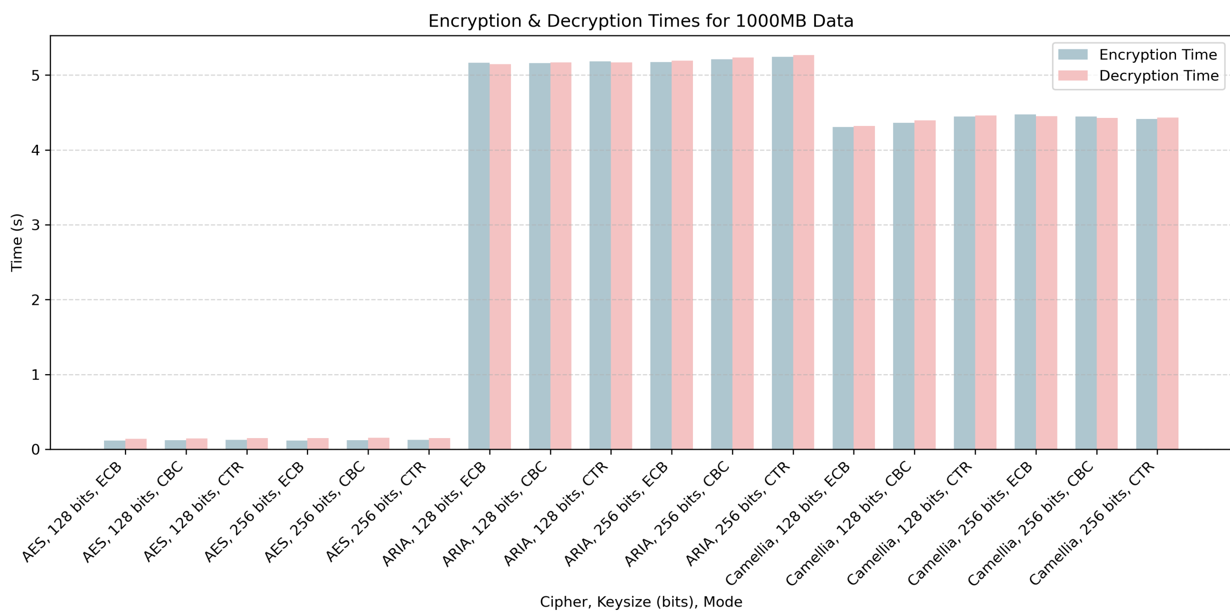


Figure 3: Encryption & decryption times for 1000MB data

The above bar charts contain the time taken for encryption & decryption for each cipher, key size, & mode combination tested; the blue columns represent the time taken for encryption and the red columns represent the time taken for decryption. The graphs are separated by the size of the data being encrypted / decrypted. What is most immediately obvious to me when looking at these two graphs is how similar the two are: the results and relative scaling look almost identical until you look at the $y$-axis and see that each time result is about 10 times higher in the 1000MB experiment than in the 100MB experiment, which makes

perfect sense, as the one experiment involves 10 times the data of the other. Since block ciphers process only fixed-size blocks, processing each block (theoretically) takes a constant amount of time, and so the CPU time of the algorithm will scale linearly with the amount of data to be encrypted or decrypted.

The next feature of the bar charts that is immediately obvious is that the blue columns are approximately the same height as the red columns: encryption time $\approx$ decryption time. This is what one would expect, as the block ciphers in question are **symmetric**: the operations used for encryption & decryption are inverses of each other. The slight variations in some results can be attributed to fluctuations in background processes consuming CPU on my laptop when I ran the tests. The one exception here are the AES results: relative to the amount of time taken by encryption for the AES results, the decryption results are quite a bit higher. Furthermore, the decryption results are consistently higher than the encryption results, unlike the other algorithms where it appears to be more or less random noise. This is because of **AES-NI** (Advanced Encryption Standard New Instructions): a set of hardware instructions on Intel/AMD CPUs that allow hardware acceleration of AES operations. OpenSSL automatically detects if AES-NI is supported at runtime, and will automatically take advantage of these hardware accelerations if available. On Linux-based operating systems, you can check if your CPU has AES flags in its CPU information to check if AES-NI is supported by running `grep -c aes /proc/cpuinfo`.

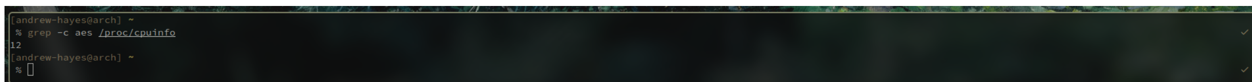


Figure 4: Output of `grep -c aes /proc/cpuinfo` showing that my CPU supports AES-NI

As can be seen from the terminal screenshot above, my laptop's CPU does indeed support AES-NI, which explains not only the slightly slower (relatively speaking) AES decryption times, but also why the AES times are so much faster than both the ARIA and Camellia times. Ordinarily, for similar amounts of data, one would expect that AES would take roughly the same amount of time as ARIA or Camellia, and possibly be outperformed by something like Camellia as AES has more lookup tables & transformations than Camellia, which can be costly in software implementations. However, since my CPU has hardware-acceleration for AES operations with AES-NI, these operations occur in the hardware rather than the software, and are therefore much, much faster.

We can also see from the bar charts that ARIA is consistently slower than Camellia; as there is no hardware acceleration for either, this is down purely to their software implementations / algorithmic design. This can be explained by the fact that ARIA has a computationally intensive round function that uses a substitution-permutation network structure (like AES) with multiple layers of S-box substitutions, diffusion matrices, & key-dependent transformations. On the other hand, Camellia uses a Feistel network which is more lightweight and better-suited for software execution.

If you look carefully at the graphs, you can see that encryption & decryption with a 256-bit key is just slightly slower than encryption & decryption with a 128-bit key. A larger key size means more encryption rounds and more computation per block, thus making using 256-bit key slower than a 128-bit key. However, this speed decrease is usually well worth it, as it makes the encryption far stronger and much more difficult to brute force.

The last aspect of the bar charts to discuss is the impact of the various modes on encryption & decryption time. The differences here are the most difficult to see in the bar charts but are there if you look closely; they may be easier to see in the tabulated results.

1. ECB mode is the fastest of the three modes because each block is encrypted independently with no chaining or initialisation vector required, making it simple & fast, and easy to parallelise in both encryption & decryption. However, it is not particularly secure: identical plaintext blocks will result in identical ciphertext blocks, and it can leak data patterns in the encrypted data.

2. CTR mode is the second fastest, as it works by converting the block cipher into a stream cipher and uses a counter & a nonce to generate a keystream. It encrypts the counter, then `XOR`s it with the plaintext. This is highly parallelisable for both encryption & decryption, and uses the same logic for both. It is also much more secure than ECB mode, as the counter never repeats.

3. The slowest of the three modes is CBC mode: in CBC mode, each plaintext block is `XOR`ed with the previous ciphertext block before encryption, requiring an IV for the first block. This makes it more secure than ECB as it removes repeating patterns, but it is not parallelisable as each block depends on the previous, making encryption much slower. However, decryption can be parallelised somewhat, as ciphertext blocks can be decrypted independently before they are `XOR`ed with the previous ciphertext block, but it is not clear from the results whether or not such parallelisation was utilised, and if it was, it had little impact on the performance.

## 2 Implementing & Benchmarking Triple-DES

| Cipher | Key Size (bits) | Mode | Data Size (MB) | Encryption Time (s) | Decryption Time (s) |
|---|---|---|---|---|---|
| TripleDES | 192 | ECB | 100 | 2.998096 | 2.828663 |
| TripleDES | 192 | CBC | 100 | 3.034894 | 2.988420 |
| TripleDES | 192 | ECB | 1000 | 29.170444 | 28.621023 |
| TripleDES | 192 | CBC | 1000 | 30.095597 | 29.524169 |

Table 2: Benchmarking results from TSV file

```
[andrew-hayes@arch] ~/currsem/CT437/assignments/assignment2/code (master) +
 % ls
Permissions Size User   Date Modified    Git Name
.rw-r--r--  1.4k andrew 2025-03-22 11:03  -- 3des_plot.py
.rw-r--r--   242 andrew 2025-03-22 10:42  -- 3des_results.tsv
.rwxr-xr-x   17k andrew 2025-03-22 11:05  -- benchmark
.rw-r--r--  3.6k andrew 2025-03-20 22:14  -- benchmark.c
.rw-r--r--   589 andrew 2025-03-20 22:14  -- benchmark.h
.rwxr-xr-x   17k andrew 2025-03-22 10:39  -- benchmark_3des
.rw-r--r--  3.1k andrew 2025-03-22 11:00  -- benchmark_3des.c
.rw-r--r--  1.5k andrew 2025-03-21 02:29  -- plot.py
.rw-r--r--  1.4k andrew 2025-03-21 16:19  -- results.tsv
[andrew-hayes@arch] ~/currsem/CT437/assignments/assignment2/code (master) +
 % gcc -o benchmark_3des benchmark_3des.c -lcrypto -lrt

[andrew-hayes@arch] ~/currsem/CT437/assignments/assignment2/code (master) +
 %
[andrew-hayes@arch] ~/currsem/CT437/assignments/assignment2/code (master) +
 % ./benchmark_3des
3DES Benchmarking completed. Results saved in 3des_results.tsv
[andrew-hayes@arch] ~/currsem/CT437/assignments/assignment2/code (master) +
 % bat 3des_results.tsv

     File: 3des_results.tsv

 1   Cipher  Key Size   Mode   Data Size (MB)  Encryption Time (s) Decryption Time (s)
 2 ~ TripleDES   192 ECB 100 2.998096    2.828663
 3 ~ TripleDES   192 CBC 100 3.034894    2.988420
 4 ~ TripleDES   192 ECB 1000    29.170444   28.621023
 5 ~ TripleDES   192 CBC 1000    30.095597   29.524169

[andrew-hayes@arch] ~/currsem/CT437/assignments/assignment2/code (master) +
 % python3 3des_plot.py
[andrew-hayes@arch] ~/currsem/CT437/assignments/assignment2/code (master) +
 %
```

Figure 5: Compiling & running the benchmarking program

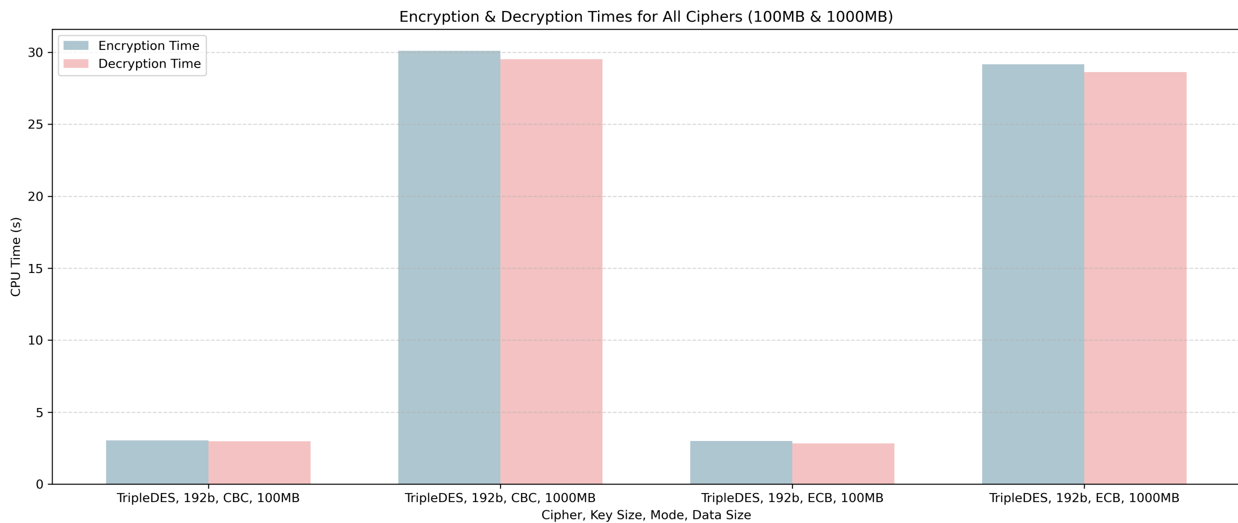


Figure 6: Encryption & decryption times for Triple-DES

As before, what is immediate obvious about this graph is that the times for 1000MB of data are pretty much exactly 10 times that of the times for 100MB of data, as Triple-DES is a block cipher and so the computational time scales linearly with input data size. We can also see that decryption is generally a little faster than encryption but is more or less the same speed. This is likely due in part due to background noise, but is also possibly due to internal caching behaviour or other optimisations for decryption in the OpenSSL implementation. In general, however, one would expect encryption & decryption to take more or less the same amount of time, as Triple-DES is a symmetric algorithm and therefore decryption requires the same steps as encryption, just inverted. As before, since ECB encrypts each block independently, it is faster than CBC due to being parallelisable, whereas CBC has dependency between blocks and is therefore not parallelisable and thus slower.

The main comparison to be drawn from comparing the Triple-DES results to the results from the first experiment is that Triple-DES is much slower, taking 5–10 times more time to execute than any other algorithm from the first experiment. This

is down to three primary reasons: DES operates on 64-bit blocks which is less efficient, is very outdated so has no hardware acceleration and not much software optimisation, and because Triple-DES runs DES three times, so it's automatically going to be 3 times slower. For example Triple-DES took about 6 times longer to run than Camellia in ECB mode on 1000MB of data: $\sim$30 seconds versus $\sim$5 seconds. This makes sense, as we would expect DES to be about twice as slow as Camellia and, Triple-DES will naturally be three times slower than DES, adding up to a $6\times$ speed decrease.