

Table of Contents

Data	2
Time Domain	2
Frequency Domain	4
Experimentation	5
System Functionality	5
Compression Subsystem	6
Encryption Subsystem	6
Results	7
System Functionality	7
Compression Subsystem	7
Encryption Subsystem	8
ATPs	9
Admin Documentation	11
Distribution of Work	11
Project Management Tools	11
Development timeline	12
References	12

Data

Given two sets of data: the first, a data collection of an IMU placed in the pocket of the user, whilst walking around, sitting and standing for five minutes and the second , a data collection of an IMU placed on a fixed rotation program, rotating about one of its axes for over ten minutes. We have decided on using the first data set with the IMU in a user's pocket (reading accelerometer and gyroscope data) as it contains more complexities (i.e walking,sitting, standing) compared to a fixed rotation device. Though we do not have a dataset that resembles that of an IMU in the arctic (which is where our project will be adapted), the arctic conditions and complexities are unknown and thus we have decided it would be wise to deal with a data set with more gyroscope and accelerometers complexities such that of an IMU in a pocket. Since the user will be walking and stopping to sit down, the movements may be irregular and may induce rotations about all axes which can be comparable to a degree to the unknown movement of an IMU in the arctic.

This data set contains a large number of entries which is suitable to run our compression and encryption speed ATPs. We will be able to vary the size of the data and use this varying data to perform compression and encryption speed, compression ratio tests and allow us to improve the efficiency of our subsystems accordingly. The data set also contains noise which is expected from our given IMU, allowing us to demonstrate our compression test and determine if at least 25% of non-noisy data may be extracted.

Time Domain

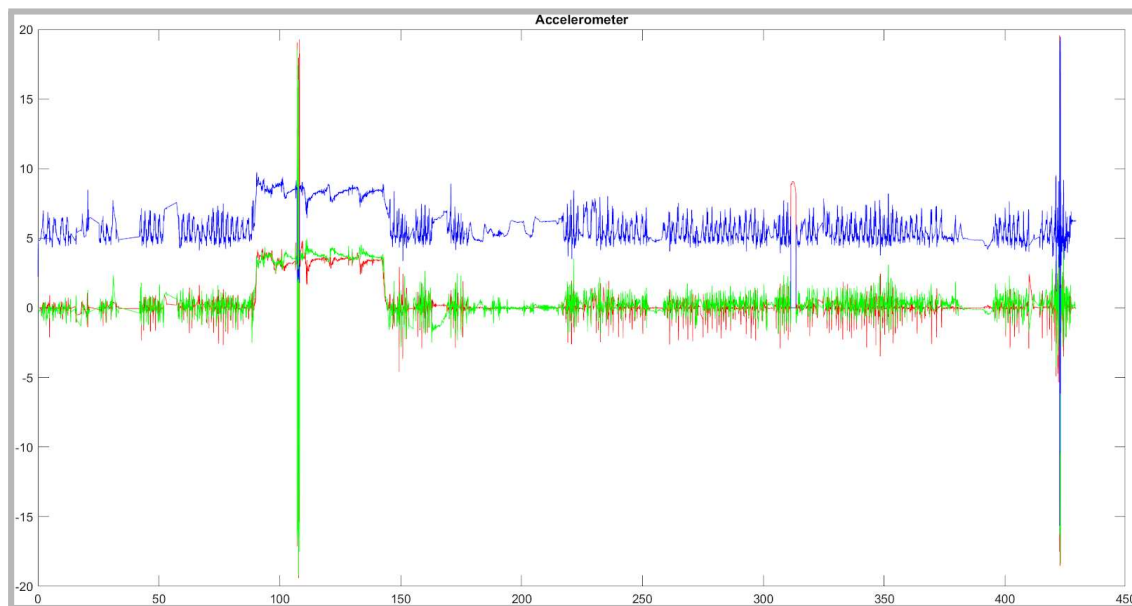


Figure 1: Accelerometer in the time domain with x-axis in red, y-axis in green, z-axis in blue

Below are the graphs of the data set, showcasing the accelerometer and gyroscope data in the time domain. .

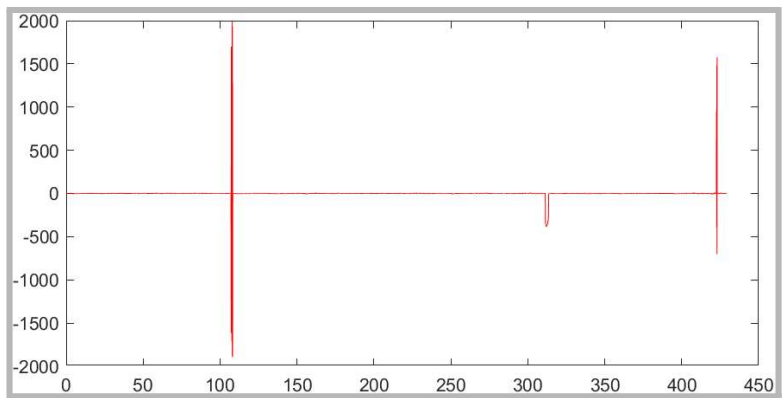


Figure 2: Gyroscope x-axis data in the time domain

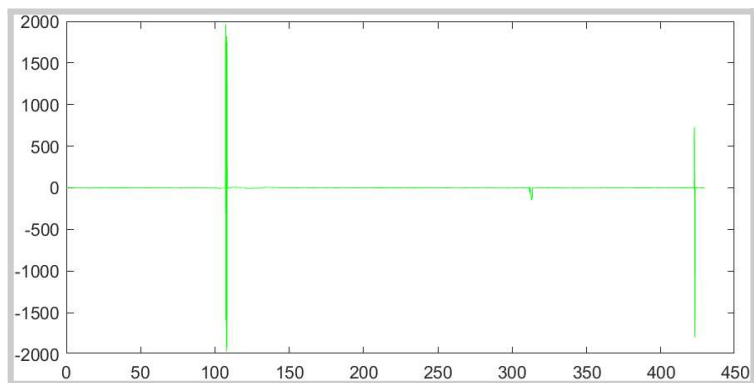


Figure 3: Gyroscope y-axis data in the time domain

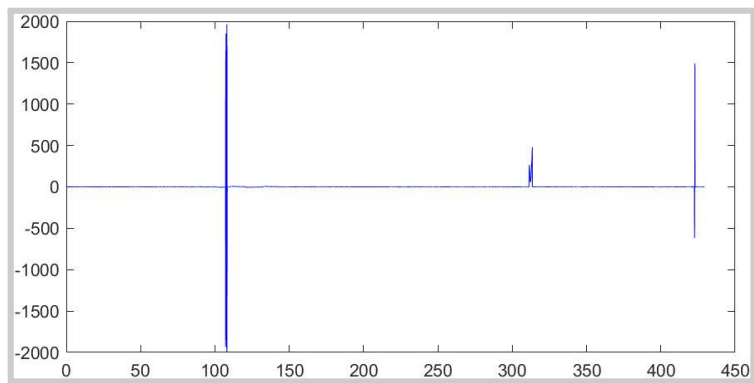


Figure 4: Gyroscope z-axis data in the time domain

Frequency Domain

Below are the graphs of the data set, showcasing the accelerometer and gyroscope data in the frequency domain.

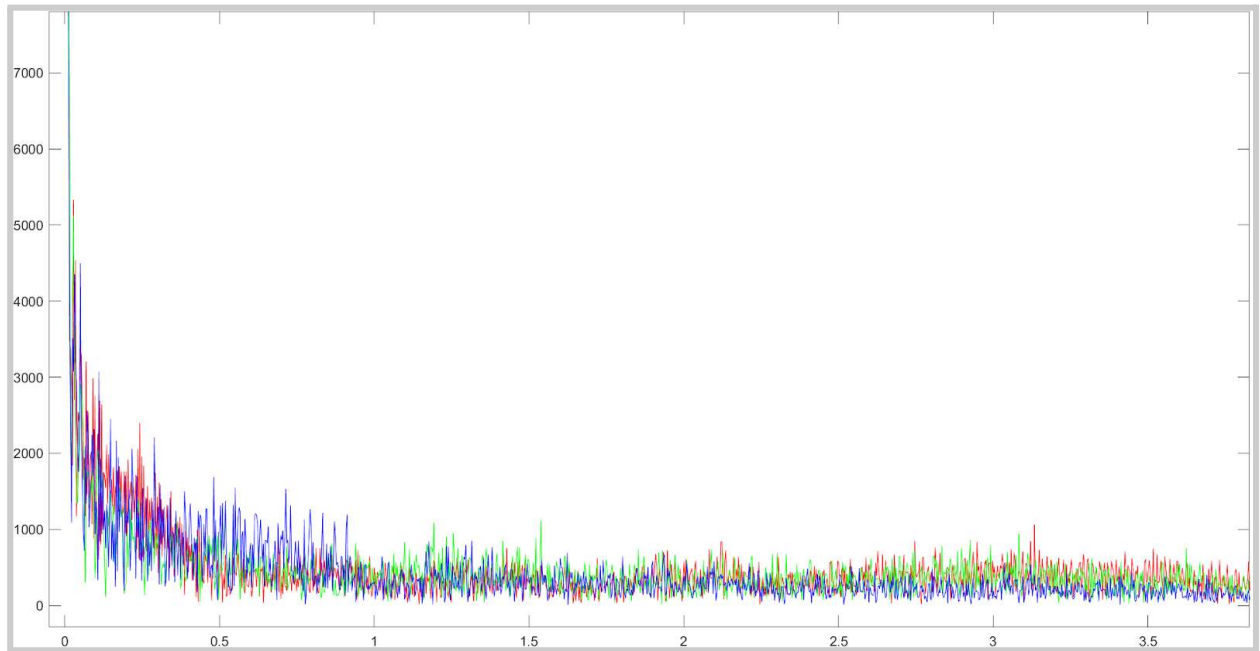


Figure 5: Accelerometer in the frequency domain with x-axis in red, y-axis in green, z-axis in blue

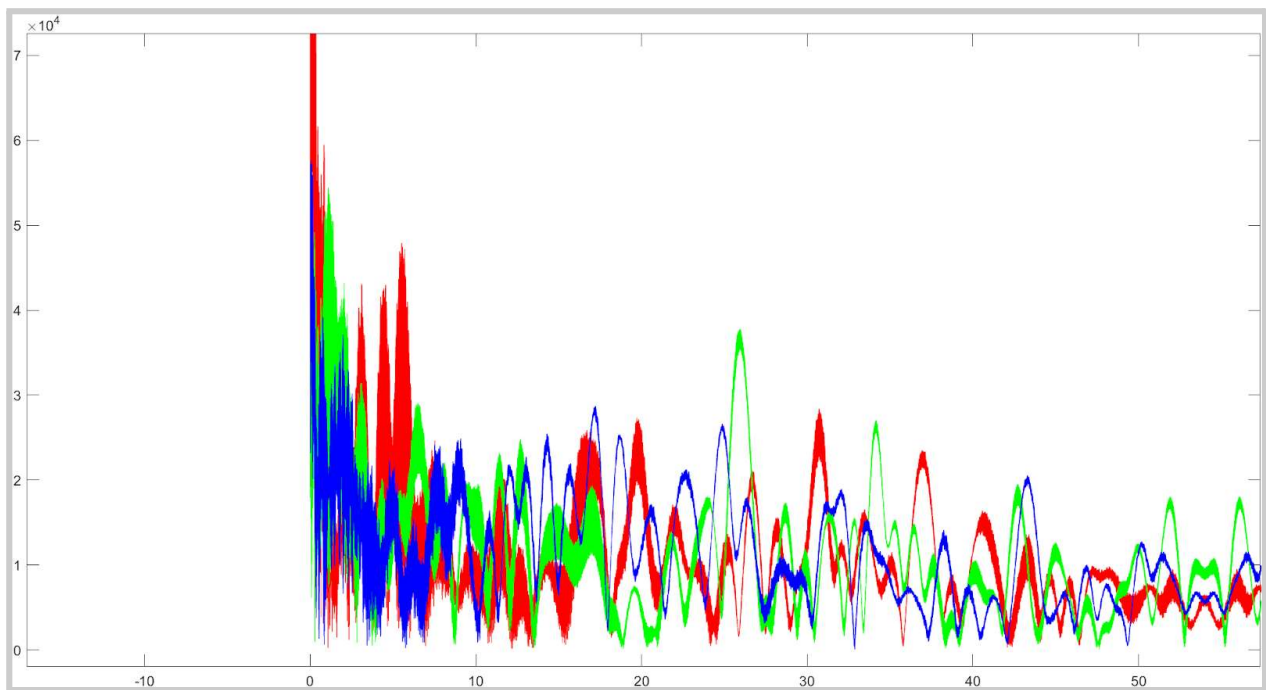


Figure 6: Gyroscope in the frequency domain with x-axis in red, y-axis in green, z-axis in blue

Experimentation

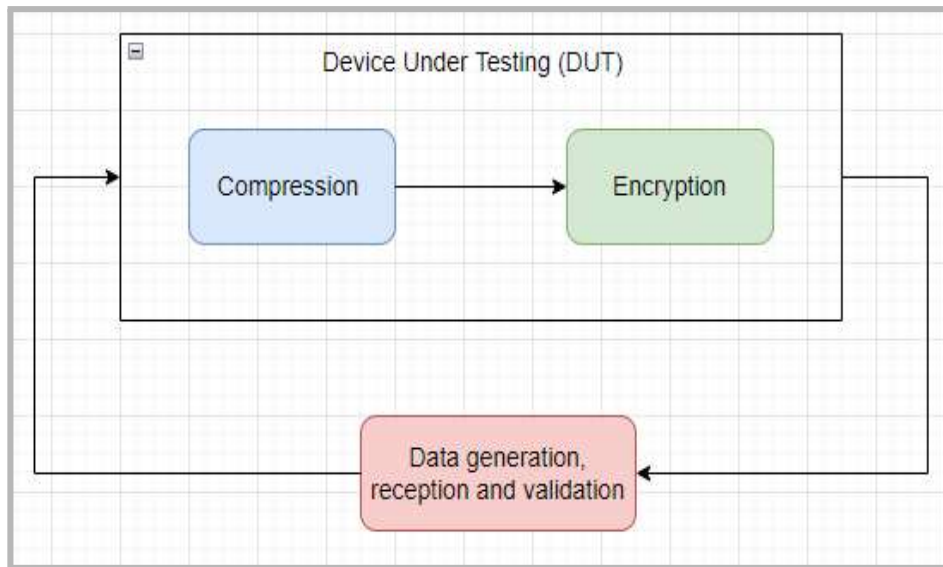


Figure 7: A loop-in test was used in this experiment

Given an stm32f0 microcontroller and usb-to-uart module, we were able to form a serial connection between our PC and the microcontroller. The serial connection allows for our IMU data to be sent to the microcontroller for compression and encryption processing, as well as to be retrieved. Our IMU data is stored within a .csv file located on a PC however, unlike the raspberry pi's ability to store user files, the micro-controller does not have a file I/O system in place for us to store files, and as a result we have opted to send each line from the .csv file as a string for processing. This method of input will not be a problem as this method will most likely be similar to when we sample sensor data from an attached IMU. Below we will describe how we aim to run our experiments to test our overall system functionality, as well as discuss the experiments set for the compression and encryption submodules.

System Functionality

In order to test the system functionality of the system, we will create a subset of data consisting of a few rows of records and test the compression and encryption subblocks separately and finally collectively. We will utilize a program called CoolTerm which will allow us to send string lines of data to our microcontroller as well as capture the data output from our microcontroller to which we can then save as a text file. Using CoolTerm we will send each line of our IMU subset data into our microcontroller buffer via the serial port connection. The buffer will allow for the storage of our input lines of data whilst allowing us to process each string of data simultaneously[3]. The first string line of data from the buffer will be sent to the compression block where each line will be encoded and sent to a second buffer. As this new buffer of encoded data fills, data from the buffer will be transmitted to the serial port where CoolTerm will capture and store the encoded data in a text file for compression analysis. If the compressed text file is satisfactory, we similarly follow the same procedure for encryption where now we send a string of data from the compressed text file for encryption and retrieve the encrypted

data from the buffer, storing the data in a text file for encryption analysis. We will then reverse the process, inputting encrypted text file data retrieving both the decrypted data and decompressed data. Once satisfactory we will combine the compression and encryption subblocks where we will input strings of IMU data and only retrieve the final encrypted and compressed data, and lastly reversing the process to retrieve the decrypted and decompressed data.

Compression Subsystem

This subsystem will have 3 tests i.e compression, decompression and data Integrity. During compression we will read the .csv file data, Huffman encodes it, store the compressed data in an array/buffer and then output the data via the serial port which will then be converted into a .txt file on the PC for analysis.

For decompression it's the reverse we will read the .txt file data, Huffman decodes it, store the decompressed data in an array/buffer and then output the data via the serial port which will then be converted into a .csv on the PC for analysis.

During analysis, we simply compare the file sizes of the input and output files to prove that has occurred the data Integrity test also takes place during this analysis where we will check if the same data is within the test .csv file and the compressed and decompressed .csv file.

Encryption Subsystem

The subsystem will consist of three experiments, namely the encryption test, decryption test and efficiency test. Within the encryption test, I'll test if input data has indeed been transformed into ciphertext. From CoolTerm I'll input string lines of data from the compression text file, into the microcontroller where each will be stored in the buffer created for the subblock. The subblock will then process each data stored in the buffer at a time, performing AES-128 encryption to which will then be placed in another buffer for serial data retrieval. Once the data has been captured with CoolTerm and saved in a text file, a comparison will be made with the results and that of the original input data text file. On our PC, an online tool or python script will be used to compare each line from the input and output text file and determine if there are any similarities. For successful encryption, the online tool and/or python script should result in 0% similarities for each line compared.

Within the decryption test, I'll reverse the process, using our output encrypted data as input, expecting the compressed data to be retrieved. We may use the same online tool or python script to compare the newly decrypted text file and compression text file, and for a successful decryption, the online tool and/or python script should result in 100% similarities for each line compared.

The decryption test will also have a security test to determine if the encrypted data is 100% secured. In this test, the security key used to decrypt the data will be changed simulating a

malicious user trying to decrypt the data. Once the data is decrypted using this incorrect key, the decrypted data will be compared with the compression text file. For 100% security, the online tool and/or python script should result in 0% similarities for each line compared.

Lastly, the efficiency test will include measuring the time taken to encrypt and decrypt the compressed data and finding ways to increase its efficiency. Eric Tobias[5], has determined that AES-128 is faster than AES-256, however, this theory may be tested, as well AES-192. The AES library used[4] allows us to utilize different modes of AES, consisting of the CBC, CTR and EBC modes. Each mode may be tested, to determine the optimal performance of the encryption algorithm

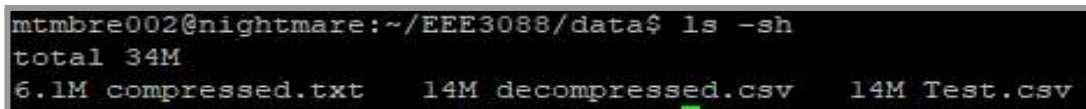
Results

System Functionality

The serial connection and communication between our PC and microcontroller was successful. The microcontroller was able to receive the csv and text file subset data string lines as expected and was able to store these contents in the created buffer without any loss of contents. Both the encryption and compression subblocks were able to function individually, and linking the subblocks to perform the compression, encryption, decryption and decompression steps collectively was successful. After the encryption process and the separate decompression process, data from the buffer was printed from both processes to the serial port and CoolTerm was able to capture the data and save these datasets as textfiles for compression and encryption result analysis.

Compression Subsystem

For the compression test, the output .txt file size is smaller than the input .csv file proof that compression has occurred.



```
mtmbre002@nightmare:~/EEE3088/data$ ls -sh
total 34M
6.1M compressed.txt    14M decompressed.csv    14M Test.csv
```

Figure 8: Input and Output file

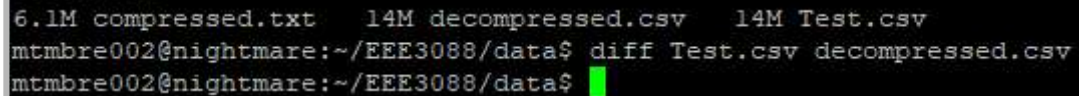
Input file = Test.csv, Output file = compressed.txt

$$\text{Compression ratio} = \frac{\text{Original file}}{\text{compressed file}} = \frac{14M}{6.1M} = 2.3$$

Note: File was compressed to 44% of its original size and still retained 100% of its original data.

The same applies to the decompression test as well. The input file (compressed.txt) is smaller than the output file (decompressed.csv).

As for the results of the last test the data integrity test, the subsystem passed as there is no difference between the original data and the data that went through compression and decompression

A terminal window with a black background and white text. The first line shows file sizes: '6.1M compressed.txt 14M decompressed.csv 14M Test.csv'. The second line shows the command 'mtmbre002@nightmare:~/EEE3088/data\$ diff Test.csv decompressed.csv'. The third line shows the result 'mtmbre002@nightmare:~/EEE3088/data\$' followed by a green cursor block.

```
6.1M compressed.txt 14M decompressed.csv 14M Test.csv
mtmbre002@nightmare:~/EEE3088/data$ diff Test.csv decompressed.csv
mtmbre002@nightmare:~/EEE3088/data$
```

Figure 9: differences the Original file and processed data file

This result shows that the compression algorithm is lossless thereby retaining data integrity

Encryption Subsystem

The encryption surpassed 2 of the 3 experiments. With the encryption test, I was able to extract the ciphertext from CoolTerm, and the comparison test between the ciphertext and the original compression text file resulted in 0% comparison in each line. Within the decryption test, I was also able to extract the decrypted compression data, and according to our simulations, the decrypted text file had a 100% match to the original compression text file. I can conclude that AES-128 is secure, as changing decryption keys did not result in a perfect decryption. The results of the encryption performance showcasing each of the experiments above will be recorded and may be viewed within our github repository.

As stated, only 2 of the 3 experiments succeeded. At this moment I have not been able to perform an efficiency test and perform any necessary speedups, as the functionality of the encryption algorithm has been the primary focus. It is expected that AES-128 will produce the fastest rate of encryption[5] however testing this theory will come at a later stage.

ATPs

Table 1: ATPS from Paper Design

Specification(s) ID	Specification(s)	Figure of Merits
DS003	We shall use Zstd algorithm to compress our data from the IMU	The algorithm is supposed to compress the data from the simulations so that we will be able to retrieve at least 25% of the data
DS004	The adaptive mode of Zstandard is supposed to be set to the minimum required	Fastest time of execution
DS005	The microcontroller will communicate with the IMU in order to receive the data	Print out data from the IMU to prove retrieval
DS006	The use of the algorithm BlowFish to encrypt the compressed IMU data	Sending an unencrypted file to the microcontroller and retrieving it, seeing if the input data matches the yield data. The data should not match
DS007	Choosing an encryption algorithm that limits the number of processors done on the microcontroller.	Time how long it takes for the file to be encrypted and returned from the microcontroller, generating an average speed.

Table 2: ATP Paper Design Review

Specification(s) ID	Specification(s)	Review
DS003	We shall use the Zstd algorithm to compress our data from the IMU	The figure of merit was met but using a different algorithm i.e Huffman encoding algorithm
DS004	The adaptive mode of Zstandard is supposed to be set to the minimum required	Was not achieved due to the change in the Huffman encoding algorithm but is still relatively fast as it is ideal for compressing text or program files
DS005	The microcontroller will communicate with the IMU in order to receive the data	A serial connection was successfully established between the microcontroller and PC to simulate that of an IMU, allowing for data packets to be stored on and received from the microcontroller

DS006	The use of the algorithm BlowFish to encrypt the compressed IMU data	A working BlowFish encryption and decryption could not be created as online support for debugging was not sufficient and as a result AES-128 library which is just as efficient was used.
DS007	Choosing an encryption algorithm that limits the number of processors done on the microcontroller.	AES library provided by kokke[4] is a simplified version of the general AES library, intended in areas where memory and processing power is constrained. Efficiency testing is yet to commence

Table 3: Updated ATP table

Specification(s) ID	Specification(s)	Review
DS003	We shall use the Huffman encoding algorithm to compress our data from the IMU	It is lossless so that we are able to retrieve at least 25% of the Fourier transform coefficients
DS004	The Huffman encoding scheme takes advantage of the disparity between frequencies and uses less storage for the frequently occurring characters at the expense of having to use more storage for each of the more rare characters.	Fast as it is efficient in compressing text or program files
DS005	The microcontroller will communicate with the IMU in order to receive the data & a serial connection between the microcontroller and PC will be established	A serial connection was successfully established between the microcontroller and PC, allowing for data packets to be stored on and received from the microcontroller
DS006	The use AES library to encrypt and decrypt the compressed IMU data and ensure security.	AES-128 was used to successfully encrypt and decrypt the compressed .csv data ensuring 100% security
DS007	Choosing an encryption algorithm that limits the number of processors done on the microcontroller.	The AES library provided by kokke[5] is a simplified version of the general AES library, intended for areas where memory and processing power are constrained. Efficiency testing is yet to commence

Admin Documentation

Distribution of Work

Table 4: contribution of each of the team members

Work	Member
Admin documents	MTMBRE002 & STLMOU001
Data	STLMOU001
Experimentation & System Functionality	STLMOU001
Compression Subsystem	MTMBRE002
Encryption Subsystem	STLMOU001
Results & System Functionality	MTMBRE002
Compression Subsystem	MTMBRE002
Encryption Subsystem	STLMOU001
ATPs	MTMBRE002 & STLMOU001

Project Management Tools

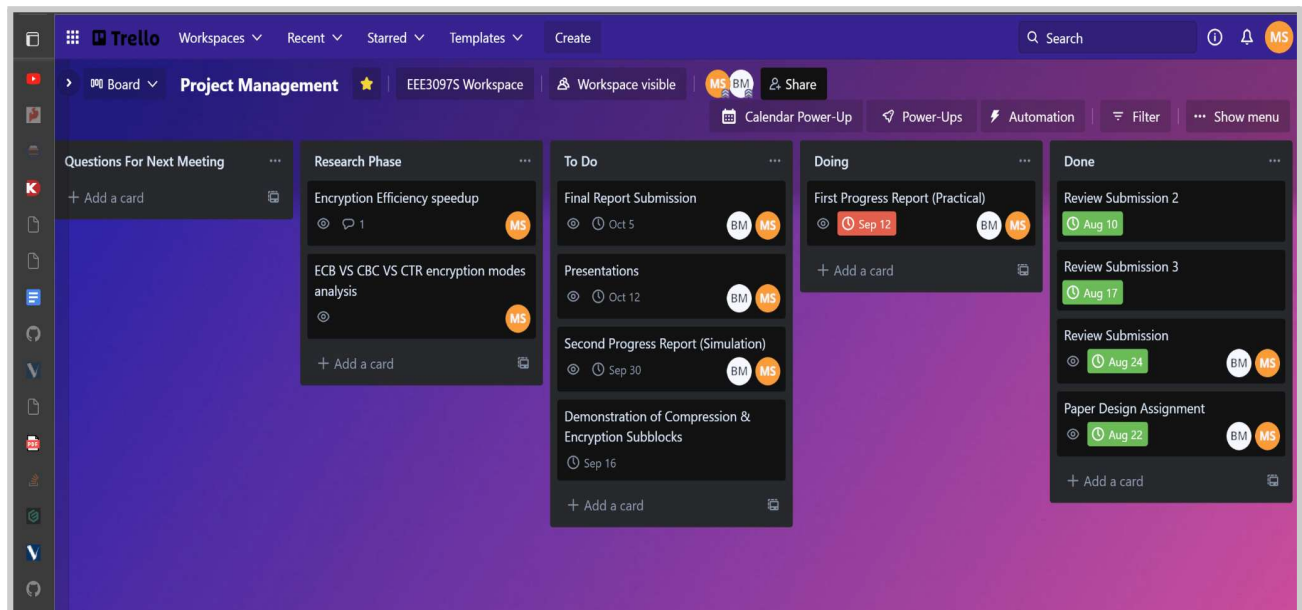


Figure 10: Our Project Management Tool

- Here is the link to our github repository [here](#)

Development timeline

The project is still planned to be completed within the course timeline. There are no changes to our timeline.

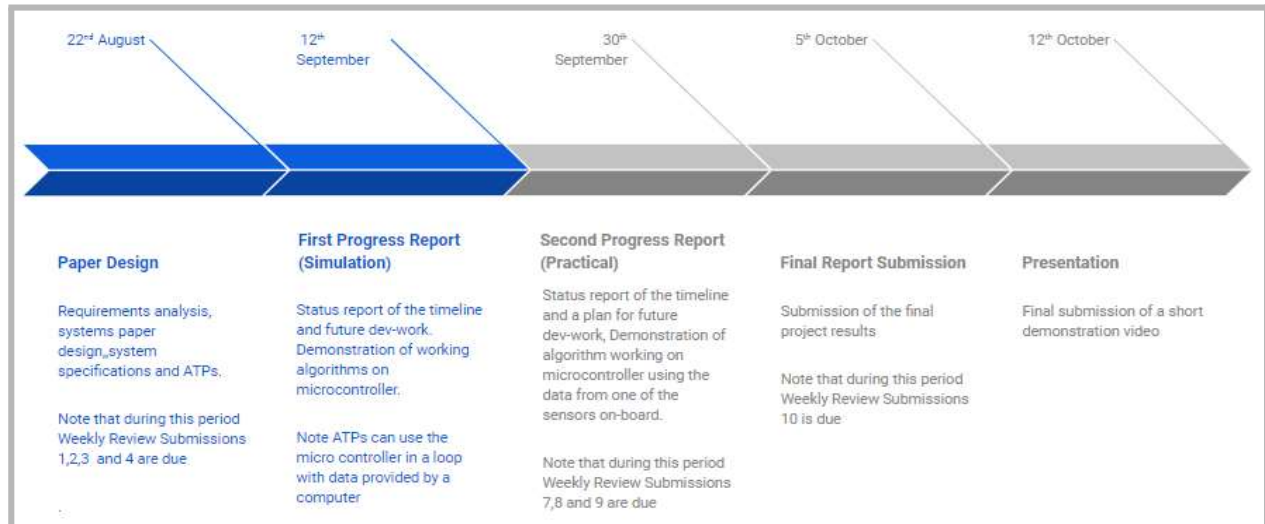


Figure 11: Our Project Development Timeline

References

- [1]"GitHub - Bestoa/huffman-codec: Very simple 8 bits huffman encoder/decoder with pure C.", *GitHub*, 2022. [Online]. Available: <https://github.com/Bestoa/huffman-codec>. [Accessed: 11- Sep- 2022].
- [2]"GitHub - jgesc/huffman-coding: An implementation of Huffman Coding Compression written in C", *GitHub*, 2022. [Online]. Available: <https://github.com/jgesc/huffman-coding>. [Accessed: 11- Sep- 2022].
- [3] Huffman Coding - Greedy Algorithm
- [3]S. Schmit, "Software FIFO Buffer for UART Communication", *Engineering and Component Solution Forum - TechForum | Digi-Key*, 2022. [Online]. Available: <https://forum.digikey.com/t/software-fifo-buffer-for-uart-communication/13476>. [Accessed: 12- Sep- 2022].
- [4]Kokke(2020) Tiny AES in C[Source Code]. GitHub - kokke/tiny-AES-c: Small portable AES128/192/256 in C
- [5]E. Tobias, "128 or 256 bit Encryption: Which Should I Use? - Ubiq", *Ubiq*, 2021. [Online]. Available: <https://www.ubiqsecurity.com/128bit-or-256bit-encryption-which-to-use/#:~:text=Picking%20Between%20AES%2D128%20and%20AES%2D256&text=AES%2D128%20is%20faster%20and,which%20should%20never%20happen%20anyway>). [Accessed: 12- Sep- 2022].
- [7]*Web.stanford.edu*, 2022. [Online]. Available: <https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1176/assnFiles/assign6/huffman-encoding-supplement.pdf>. [Accessed: 12- Sep- 2022].