University of Cape Town

EEE3097S

Design

Final Report

Rory Schram
SCHROR002

Natasha Soldin
SLDNAT001

17/10/2022

Contents

Chapter 1

Introduction

The following project design is for an ARM based digital IP using an STM32F051-microcontroller. This design aims to retrieve, compress and encrypt data from an Inertial Measurement Unit (IMU) sensor. This type of sensor includes an Accelerometer, a Gyroscope, a Magnetometer and a Barometer. This design will be implemented as a buoy installed on an ice 'pancake' in the Southern Ocean to collect data about the ice and wave dynamics.

This data will then be transmitted using the Iridium communication network, which is a global satellite communications network. However, this is extremely costly and therefore the data would need to be compressed to reduce its size. The data is also encrypted for security purposes.

The following report is a final report for the project and includes:

- A Requirement Analysis: where requirement, specifications and acceptance test protocols are derived from the project's description.
- A Paper Design: where a theoretical version of the project was proposed by exploring the background, feasibility, possible bottlenecks, available algorithms choices, subsystem design and experimental design.
- A Validation using Simulated Data (**Progress Report 01**): or simulated implementation of the project where sample data is used in a data analysis, simulated experiment, results and re-evaluated acceptance test procedures.
- A Validation using IMU Data (**Progress Report 02**): or practical implementation of the project where data is retrieved from the IMU is used in a IMU module for data validation, practical experiment, results and re-evaluated acceptance test procedures.
- A conclusion and plan for future work.

Chapter 2

Admin

2.1 Contributions

Each team member's contributions for the following project are tabulated in table 2.1 below. The table makes reference to section and page numbering.

Team Member	Contribution		Section Number	Page Number
Natasha Soldin	Paper Design	Requirement Analysis	3	5
		Encryption Algorithm Comparison	4.2.2	14
		Feasibility Analysis	4.1.2	10
		Possible Bottlenecks	4.1.3	10
		Acceptance Test Protocols	3.3	7
	Progress Report 1	Data Analysis	5.1	23
		Acceptance Test Protocols	5.4	35
	Progress Report 2	IMU Module	6.1	36
		Acceptance Test Protocols	6.4	50
Rory Schram	Paper Design	Compression Algorithm Comparison	4.2.1	11
		Subsystem Design	4.3	15
		Figures of Merit	4.3.6	20
		Experiment Design	4.4	21
		Development Timeline	2.2.1	3
	Progress Report 1	Experiment Setup	5.2	28
		Results	5.3	30
	Progress Report 2	Experiment Setup	6.2	42
		Results	6.3	45

Table 2.1: Contribution Table

2.2 Project Management Tool

This project was managed using Asana as its project management software. With which a timeline was created that helped the team members stay up to date on tasks and deliverables. The following figure 2.1 below shows a screenshot of the project management board.

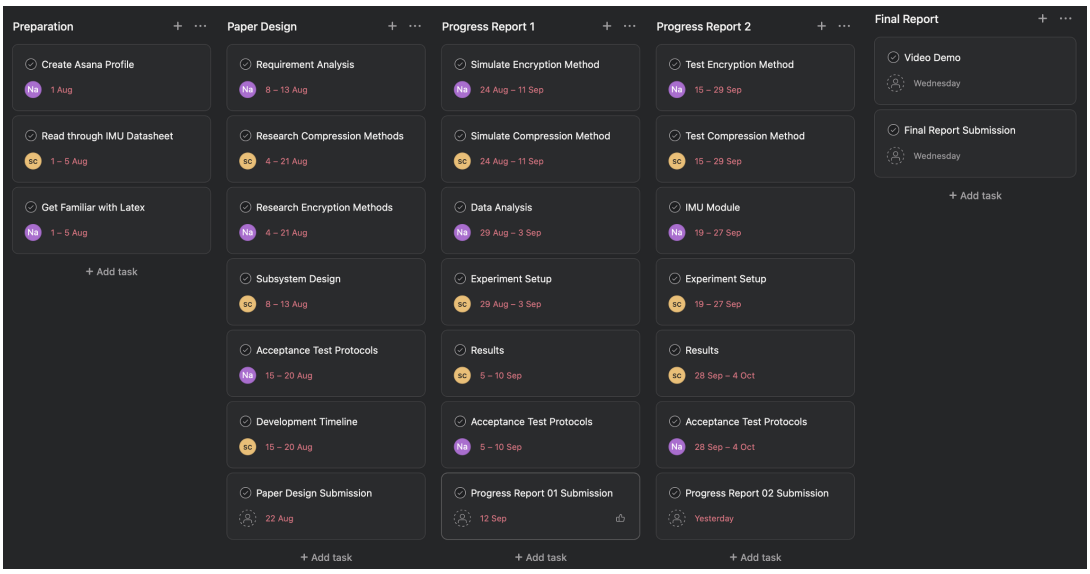
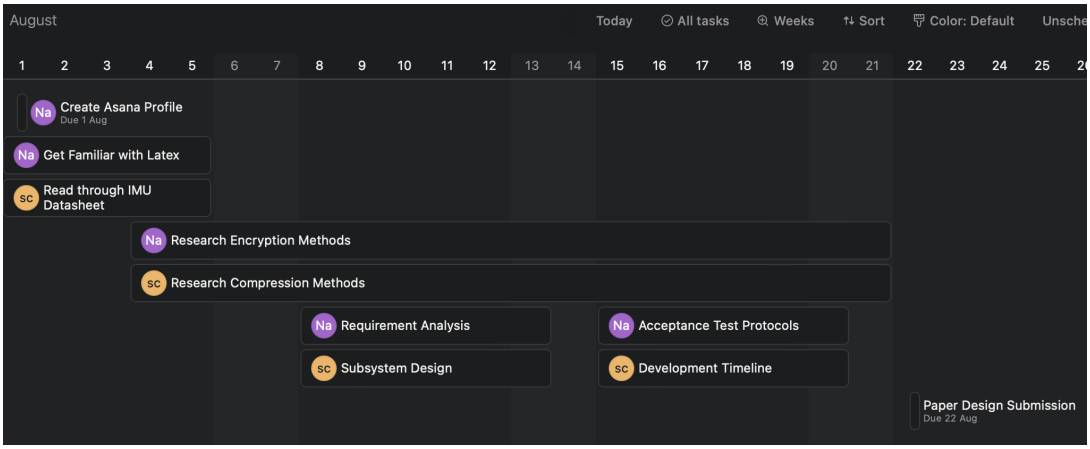


Figure 2.1: Asana Board

2.2.1 Timeline

Asana, the Project Management Tool was used to generate a timeline. Figure 2.2 below shows the timeline. The project remained on time throughout its course and experienced no delays.



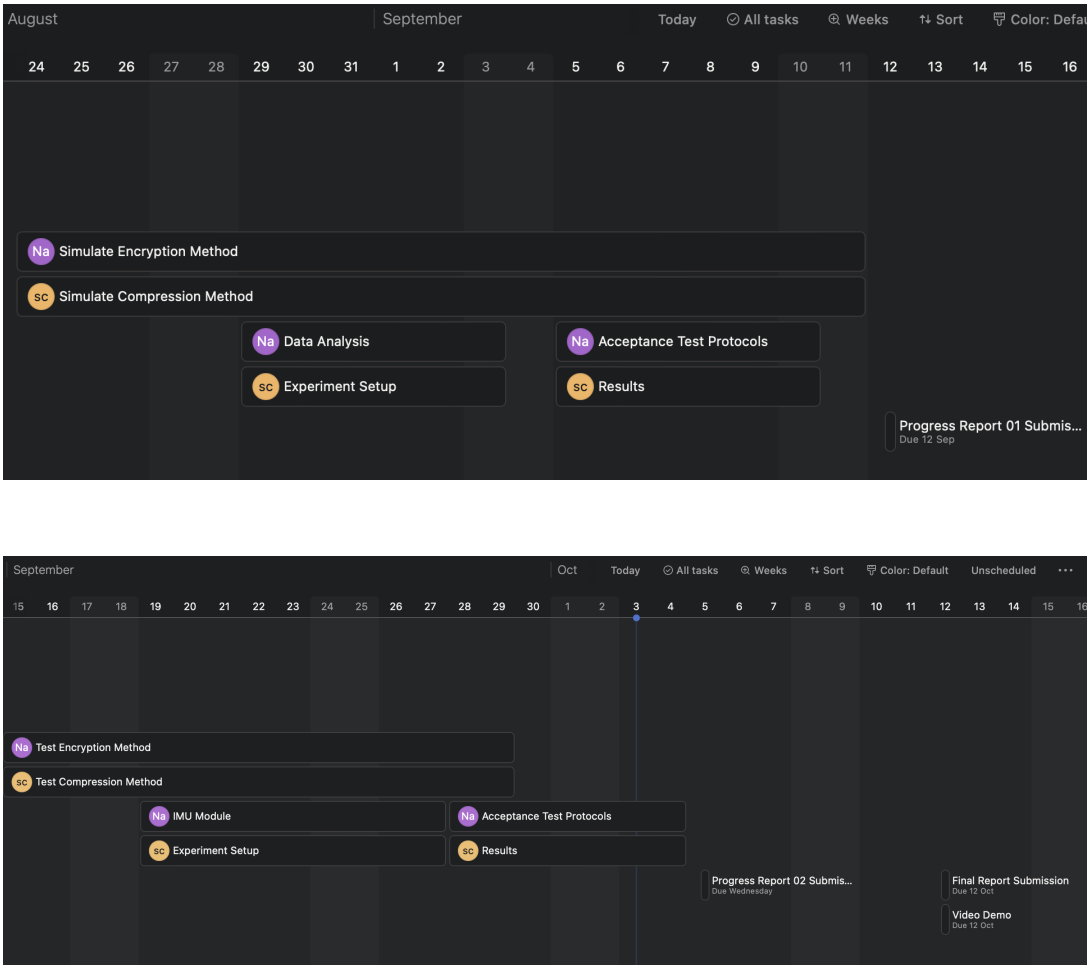


Figure 2.2: Asana Timeline

2.3 GitHub

Github was used extensively throughout the project's duration as it aided the students in version control and collaboration. The git repository can be accessed here.

Chapter 3

Requirement Analysis

3.1 Requirements

The following are the project's base requirements that are derived from the project's description:

1. The project should be designed to utilize the ICM-20649 IMU sensor even though that sensor is not available to use during the project's planning stage.
2. The project should be designed to utilize the STM32F051 microcontroller.
3. The data obtained from the IMU sensor should be compressed to reduce the cost of transmission because the transmission of data using Iridium is extremely costly.
4. The data obtained from the IMU sensor should be encrypted to increase security of the data.
5. Minimal loss/ damage should apply to the decrypted and decompressed data.
6. Limit power consumption by reducing the amount of processing done in the processor and minimizing the computation required.

3.2 Specifications

Each of the above mentioned Requirements have the following corresponding specifications which quantitatively describe them and provide then with numerical conditions:

1. The design should adhere to the electrical specifications in figure 3.1 below as obtained from the ICM-20649's datasheet.

Typical Operating Circuit of section 4.2, VDD = 1.8 V, VDDIO = 1.8 V, T_A = 25°C, unless otherwise noted.

PARAMETER	CONDITIONS	MIN	TYP	MAX	UNITS	NOTES
SUPPLY VOLTAGES						
VDD		1.71	1.8	3.6	V	1
VDDIO		1.71	1.8	3.6	V	1
SUPPLY CURRENTS						
Gyroscope Only (DMP & Accelerometer disabled)	Low-Noise Mode		2.67		mA	2
Accelerometer Only (DMP & Gyroscope disabled)	Low-Noise Mode		760		µA	2
Gyroscope + Accelerometer (DMP disabled)	Low-Noise Mode		3.21		mA	2
Gyroscope Only (DMP & Accelerometer disabled)	Low-Power Mode, 102.3 Hz update rate, 1x averaging filter		1.23		mA	2, 3
Accelerometer Only (DMP & Gyroscope disabled)	Low-Power Mode, 102.3 Hz update rate, 1x averaging filter		68.9		µA	2, 3
Gyroscope + Accelerometer (DMP disabled)	Low-Power Mode, 102.3 Hz update rate, 1x averaging filter		1.27		mA	2, 3
Full-Chip Sleep Mode			8		µA	2
TEMPERATURE RANGE						
Specified Temperature Range	Performance parameters are not applicable beyond Specified Temperature Range	-40		+85	°C	1

Figure 3.1: ICM-20649 Electrical Specifications

2. Both algorithms needs to be coded in C/C++ in STM32CUBEIDE to be compatible with the STM32F051 microcontroller.

3. The compression specifications are as follows:

   (a) The compression and decompression of the IMU data will be done using a dictionary compression method as described in section 4.2.1 below.

   (b) The compression ratio should be between 40% and 60%.

4. The encryption specifications are as follows:

   (a) The encryption and decryption of the IMU data will be done using an AES encryption method as described in section 4.2.2 below. The similarity percentage between the pre-encrypted data and the post-encrypted data should be less than 20%.

   (b) The encryption key should be either 128, 192 or 256 bits to ensure adequate security.

5. The decrypted and decompressed data should reflect 25% of the Fourier coefficients of the original IMU data

6. The C/C++ programs should be of time complexity order n [O(n)] to reduce the amount of processing done and thus the power consumption.

3.3 Acceptance Test Procedure

An acceptance test procedure (ATP) was derived for each Requirements and Specifications pair as listed in sections 3.1 and 3.2 above to test whether or not the project meets the specifications.

1. Ensure that the data obtained from the sensor hat closely follows the structure of the sample data provided as this comes from the design's intended sensor (ICM-20649 IMU).
2. Run the algorithms in a C/C++ IDE as well as on the STM32F051 to test that it works.
3. The compression ATPs are as follows:
   (a) Check that the file size of the compressed data is considerably less than the file size of the original data.
   (b) Calculate the compression ratio between the uncompressed and compressed data.
   $$ratio = \frac{uncompressed}{compressed}$$
4. The encryption ATPs are as follows:
   (a) Compare the encrypted data to the un-encrypted data to ensure minimal similarities which indicates reasonable level of encryption. Calculate the similarity percentage.
   (b) Ensure decrypted data is exactly the same as the data before encryption to make sure that no data loss has occured during the encryption phase.
5. The fourier transform of the decrypted and decompressed data should be compared to the fourier transform of the original data to test if 25% of the lower fourier co-efficients have been retained.
6. Test the time complexity of the compression and encryption programs by running multiple tests with different dataset sizes and plot the correlation.

Each ATP correlates to a requirement and specification pairing. To demonstrate the correlation, each has been given a corresponding number and tabulated in table 3.1 below.

Requirements	**Specifications**	**Acceptance Test Protocols**
R1	S1	A1
The project should be designed to utilize the ICM-20649 IMU sensor even though that sensor is not available to use during the project's planning stage.	The design should adhere to the electrical specifications of the ICM-20649 IMU.	Ensure that the data obtained from the sensor hat closely follows the structure of the sample data provided as this comes from the design's intended sensor (ICM-20649 IMU).
R2	S2	A2
The project should be designed to utilize the STM32F051 microcontroller.	Both algorithms needs to be coded in C/C++ in STM32CUBEIDE to be compatible with the STM32F051 microcontroller.	Run the algorithms in a C/C++ IDE as well as on the STM32F051 to test that it works.
R3	S3.1	A3.1
The data obtained from the IMU sensor should be compressed to reduce the cost of transmission because the transmission of data using Iridium is extremely costly.	The compression and decompression of the IMU data will be done using a dictionary compression method.	Check that the file size of the compressed data is considerably less than the file size of the original data.
	S3.2	A3.2
	The compression ratio should be between 40% and 60%	Calculate the compression ratio between the uncompressed and compressed data.
R4	S4.1	A4.1
The data obtained from the IMU sensor should be encrypted to increase security of the data.	The encryption and decryption of the IMU data will be done using an AES encryption method.	Compare the encrypted data to the un-encrypted data to ensure minimal similarities which indicates reasonable level of encryption and calculate percentage similarity.
	S4.2	A4.2
	The encryption key should be either 128, 192 or 256 bits to ensure adequate security.	Ensure decrypted data is exactly the same as the data before encryption to make sure that no data loss has occured during the encryption phase.
R5	S5	A5
Minimal loss/ damage should apply to the decrypted and decompressed data.	The decrypted and decompressed data should reflect 25% of the Fourier coefficients of the original IMU data.	The fourier transform of the decrypted and decompressed data should be compared to the fourier transform of the original data to test if 25% of the lower fourier co-efficients have been retained.
R6	S6	A6
Limit power consumption by reducing the amount of processing done in the processor and minimizing the computation required.	The C/C++ programs should be of time complexity order n [O(n)] to reduce the amount of processing done and thus the power consumption.	Test the time complexity of the compression and encryption programs by running multiple tests with different dataset sizes and plot the correlation.

Table 3.1: Requirements, Specifications and ATPs

Chapter 4

Paper Design

4.1 Background

4.1.1 User Cases

There are multiple user cases for a system containing an IMU sensor [1] and reading off its motion data. A few of them are:

- A GPS system requires movement data to determine the location and speed of a car to plot routes to a destination. The IMU's accelerometer would provide information regarding the speed of the car and the magnetometer would provide information regarding the direction of the car. This data would require compression as it involves satellite transmission and would require encryption as it is private information.
- Virtual reality (VR) or augmented reality (AR) systems require movement data to describe the head movement and orientation of the 'player'. The IMU's accelerometer would provide information regarding the speed of the head movements and the gyroscope would provide information regarding the orientation of the device. This data requires compression as it is large in size.
- Drones require movement data to aid in flight control, stability and guidance. The IMU's accelerometer would provide information regarding the speed of the drone, the magnetometer would provide information regarding the direction of the drone and the gyroscope would provide information regarding the orientation of the drone. This data would require compression as it often involves satellite or wireless transmission and would require encryption as drone application often involves the use of sensitive data (i.e. military applications).

4.1.2 Feasibility Analysis

Upon consideration of the project description and its Requirements, the project is decided to be feasible. For the following reasons:

- There are many readily available predefined compression and encryption libraries that can be implemented in this project. Which greatly reduces the complexity of the project and time needed to complete the project.
- The project description is relatively simple and straightforward.
- The timeline given for the project is reasonable.
- Similar projects have been completed and successfully implemented in the past.

The possible bottlenecks specified in section 4.1.2 below greatly impact the feasibility of the project. Majority are due to the fact that previous projects utilised raspberry pi and python which in combination offer more memory space, easier data retrieval and better algorithm variety.

4.1.3 Possible Bottlenecks

The project is vulnerable to the following possible bottlenecks:

- Student's lack of knowledge surrounding C/C++ programming.
- Limitations of the STM320F0 microcontroller's memory space and computing power.
- Data procurement from sensor resulting in faulty/ incorrect data.
- Data transmission to different sub-systems of the project could result in losses or faults.
- Incompatibility of data format with compression and encryption libraries.

4.2 Algorithm Analysis

The following analysis compares available compression and encryption algorithms to determine which will be most appropriate in this projects context - keeping in mind the memory space constrained nature of the STM32F051 microcontroller.

4.2.1 Comparison of Compression Algorithms

Time series data is becoming more and more prevalent in today's data-centric world. The single largest issue with this is how to deal with the massive amounts of data produced, and consequently, how to handle storing this data. Systems with high write rates, such as IoT sensors, can cause massive storage concerns. Any device that is writing constantly and producing time series data, over time may need a large amount of storage space. For example, a Boeing 787 may generate half a terabyte of sensor data per flight. [2] This fact reiterates the importance compression when it comes to utilizing time series data in your projects. Compression of the IMU data is especially important for this project, because the cost of data transmission is so incredibly high due to the Iridium Satellite Communication network being utilized for data transmission. While this network has the impressive capability of being able to communicate with every remote locations all around the globe, it comes at a high monetary cost. This cost can be reduced dramatically by proper implementation of compression algorithms on the time series data that has been produced by the IMU module. The following comparison will delve into what exactly compression is, how it it is implemented and which algorithms where considered for this project.

Generally, compression algorithms take advantage of particular properties of time series data, that can then be exploited to reduce the overall size of the data set. These properties can be broken into 3 categories, as seen below. [3]

- **Redundancy**: Sometimes certain portions of data appear more often than other portions of data in time series data, this can be exploited for our purposes by leaving out these repeated portions of data and instead representing them with a piece of data that takes up much less space.
- **Approximabilty**: Often sensor can end up producing time series data that can be approximated by function that have been designed/calculated beforehand.
- **Predictabilty**: In today's ever more advanced digital world, sometimes it is possible to predict future time series data using deep learning neural networks.

There are a large number of compression algorithms available for use, with every one differing drastically in complexity, functionality and efficiency. Every compression algorithm can be described as one of the following. [3]

- **Non-adaptive or adaptive**: A non-adaptive algorithm does not need to train on a

data set beforehand to work efficiently and properly, whereas an adaptive algorithm learns from past data to compress the newer data being produced.

- **Lossy or Lossless**: To explain this concept, lets assume that you have a data set D_1, which has a size determined to be the number of bits required to store the data set, defined as s_1, and a compression algorithm defined as the function F, which takes in a data set and returns that same data set after the compression algorithm has been applied. Therefore, $F(D_1) = D_1^{'}$ and the size of $D_1^{'}$ is $s_1^{'}$. For a compression algorithm to be of any use, $s_1^{'} < s_1$. A lossless compression algorithm is one where the inverse compression algorithm produces the same data set as the one that was originally input into the algorithm. Therefore, $F^{-1}(D_1^{'}) = D_2$, where $D_2 = D_1$ the size of the output $s_2 = s_1$. If, after compression and decompression, $D_2 = D_1$ and $s_2 = s_1$, the compression algorithm is said to be lossless. If $D_2 \neq D_1$ and $s_2 < s_1$ then the algorithm is said to be lossy.

- **Symmetric or Non-symmetric**: If a compression system uses the same algorithm to compress and decompress data, only changing the direction of data flow, then it is said to be symmetric. If a compression system uses completely different algorithms for compression and decompression, then it is said to be non-symmetric.

As mentioned above, there are multiple types of compression algorithms. Below is an overview of different types of compression algorithms, each one incorporating various features mentioned above. [3]

1. **Dictionary Based**: This compression principle is based off of the fact that over a long time, time series data sets will have repeated common segments. After an initial training phase on a similar data set, a dictionary of segments will be created. The compression algorithm will then search through this dictionary and if a segment is found in the data that matches what is in the dictionary, then that length of text will be be extracted, and instead it will be replaced by an index that can be traced back to the dictionary.

2. **Functional Approximation**: The idea behind functional approximation is that time series data can be represented by a mathematical function. Finding a mathematical function for the entire time series would not be possible however, so what happens is the data set is split up into little segments, and an approximation for these segments is made. Since devising a function that would perfectly match the data set is a very process heavy procedure, the best that can be done is a close approximation. This method greatly reduces the data set size, because the only data that needs to be stored are the functions of approximation themselves, however this results in lossy compression, because of the multiple functions only roughly approximating the time series data. The strength behind this method however, is that no training needs to take place, the algorithm will work out of the box.

3. **Autoencoders**: Autoencoders are unsupervised machine learning networks that apply backpropagation. [4] Autoencoders aim to take input data and compress it into a smaller deconstructed form, called the code, which can then be rebuilt at a later stage with the same neural network, just operating in the reverse direction.

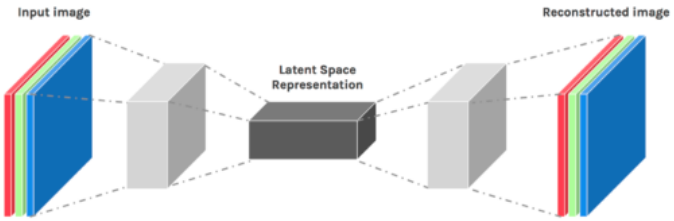


Figure 4.1: Overview of autoencoded compression [4]

4. **Sequential Algorithms**: Sequential algorithms, at their base level, incorporate several of the best compression techniques into one large algorithm. Some of the most popular techniques include, but are not limited to, the following:

   - Huffman coding
   - Delta encoding
   - Run-length encoding
   - Fibonacci binary encoding

   Combining these algorithms, whether using many of them, or simply two of them, results in a very robust and efficient compression algorithm.

After much consideration, it was determined that the most important compression aspects were the following:

1. The compression algorithm needs to be able to run on the STM32F051 in a C/C++ implementation.
2. The compression needs to be lightweight, in other words, it cannot take up the limited resources available for use on the STM32F051.
3. The compression must be simple. This is to reduce the time of development, and to save on resources on the STM32F051 micro-controller.

Bearing all of the prior information in mind, the compression technique that fits all of the requirements best would be the dictionary type compression. This is because dictionary compression is lightweight, fast, and simple to implement. The main types of dictionary compression are Tristan, Conrad, A-LZSS, and D-LZW. Due to these techniques being the most suitable for the compression needs of this projects, one of them will be used.

4.2.2 Comparison of Encryption Algorithms

Encryption methods can be subdivided into two categories [5]:

- Symmetric Encryption: encryption that involves a single key to encrypt and decrypt data. This a slighlty older technique that offers less security of data.
- Asymmetric Encryption: encryption that involves two keys, a public key and a private key that are mathematically linked together. Data is encrypted using the public key but can only be decrypted by the private key. This more modern method offers a greater level of security to the data but is more complicated to implement.

Although Asymmetric encryption is safer and offers a greater level of encryption to the data, it is more complicated to implement and for this context unnecessary. It is not necessary as the system will be deployed in a remote area of the Southern Ocean and the data collected by the sensor is not sensitive or at risk of tampering/ theft. Thus, Symmetric encryption will suffice.

There are several Symmetric encryption methods to consider, of which three will be examined in this comparison: Advanced Encryption Standard (AES), Data Encryption Standard (DES) and Triple Data Encryption Standard (3DES). Some of the key factors of the above encryption methods are tabulated in table 4.1 below: [6] [7]

Key	**AES**	**DES**	**3DES**
Definition	Advanced Encryption Standard	Data Encryption Standard	Triple Data Encryption Standard
Designed By	Vincent Rijmen and Joan Daemen	IBM	IBM/ Walter Tuchman
Year	2001	1978	1981
Key Length	128, 192, 256 bits	56 bits	112, 168 bits
Size	128 bits	64 bits	64 bits
Security Level	Most secure	Less secure	More secure than DES
Derived from	Square Cipher	Lucifer Cipher	DES
Memory Used	14,7 KB	18,2 KB	20,7 KB

Table 4.1: Comparison between Symmetric Encryption Methods

DES is no longer in use as it is considered to be insecure. This insecurity stems from its short 56-bit key. 3DES [8], although more secure than DES, is less commonly used today. The national Institute of Standards and Technology (NIST) recently released a proposal stating that all forms of 3DES will be deprecated through 2023 and disallowed from 2024. Therefore for longevity purposes this encryption method will not be used. Thus, AES will be used because it is commonly used today, it offers an adequate level of security and is lightweight in nature which reduces the resource usage on the STM32F051.

4.3 Subsystem Design

For this project, the following major subsystems exist:

1. The STM32F051 microcontroller
2. The IMU (Inertial Measurement Unit) sensor
3. The external computing device (personal laptop)

These subsystems and their relationships can be graphically shown in figure 4.2 below:

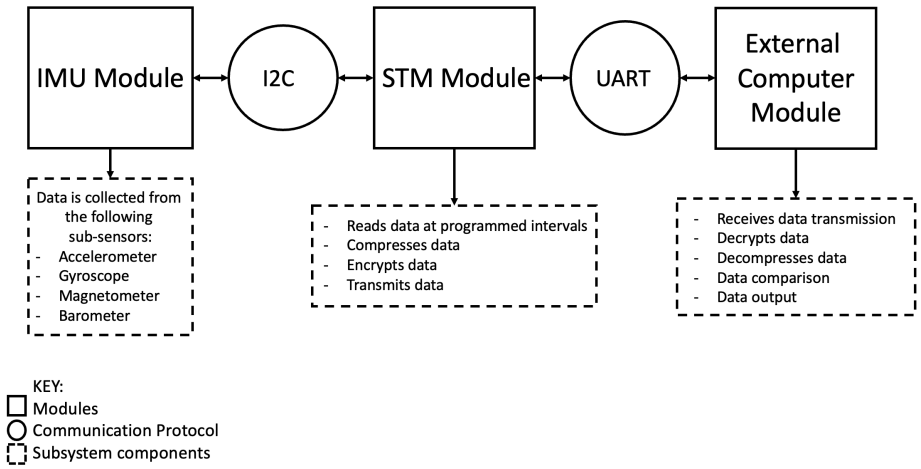


Figure 4.2: Block Diagram of Subsystems

4.3.1 Sub-subsystems

STM32F051 Microcontroller Subsystem

Within the STM32F051 microcontroller subsystem there are three main sub-subsystems. Two of which are the compression and encryption blocks that will be coded onto the microcontroller and form the basis for the project and the third sub-subsystem is the communication protocols that exist between the microcontroller and the external computing devices and between the microcontroller and the IMU sensor.

IMU Subsystem

The IMU subsystem only has one sub-subsystem, the communication protocol between IMU sensor and the STM32F051 microcontroller. This allows the data to be read of and transmitted to the STM32F051 microcontroller.

External Computing Device Subsystem

Within the external computing device subsystems there are three main sub-subsystems. The first is a communication protocol between the computer and the STM32F051 microcontroller and the second and third are the decryption and decompression blocks that will handle the transmitted data.

4.3.2 Subsystem and Sub-subsystems Requirements

STM32F051 Microcontroller Subsystem Requirements

1. **Compression**
   (a) The STM32F051 microcontroller must compress the sensor data read from the IMU as transmission of the data is costly.
   (b) The compression algorithm must adequately reduce the size of the IMU sensor data because time series data can can be dense in nature.
   (c) The compression algorithm must be lightweight and not require too much computing power or resources as the STM32F051 microcontroller is limited is those respects.
   (d) The compression algorithm needs to preserve the first 25% of the Fourier coefficients that are contained within the sensor data read from the IMU.
2. **Encryption**
   (a) The STM32F051 must encrypt the compressed IMU sensor data for security purposes.
   (b) This encryption should, similarly to the compression, be lightweight and not too resource heavy, due to the STM32F051 microcontroller's limited available resources.
   (c) The encryption algorithm needs to preserve the first 25% of the Fourier coefficients that are contained within the sensor data read from the IMU.
3. **Communication Protocols**
   (a) The STM32F051 microcontroller must be able to communicate with the IMU module. This module is a digital sensor and so a digital communication protocol will have to be implemented to allow for this communication to happen.
   (b) The communication between the STM32F051 microcontroller and the IMU module must happen at regular, controlled intervals.
   (c) The STM32F051 microcontroller must be able to communicate with the external computing device. This communication needs to cater for two-way communication as to allow for the compressed and encrypted data to be sent to the external computing device for processing, and on the other hand, to allow for data to be sent to the STM32F051 microcontroller itself for acceptance test procedures.

IMU Subsystem Requirements

1. **Communication Protocols**

   (a) The IMU module needs to communicate with the STM32F051 module at regular intervals for data transmission purposes. As stated in the previous requirements, this communication must happen at regular, controllable intervals.

External Computing Device Subsystem Requirements

1. **Communication Protocol**

   (a) The external computing device needs to communicate with the STM32F051 module. As stated before, this communication protocol must allow for two-way communication for data retrieval and testing purposes.

2. **De-encryption**

   (a) The external computer needs to be able to properly de-encrypt the transmitted data from the STM32F051. This decryption should return the compressed data.

3. **De-compression**

   (a) The external computer needs to be able to properly de-compress the data after decryption. This decryption should be fast and efficient.

   (b) The de-compression should preserve 25% of the Fourier coefficients contained within the data.

4.3.3 Subsystem and Sub-subsystems Specifications

STM32F051 Microcontroller Subsystem Specifications

1. **Compression**

   (a) The compression algorithm chosen is a dictionary compression algorithm.

   (b) The compression algorithm needs to achieve a compression ratio of between 40% and 60%.

   (c) Dictionary compression is a very lightweight compression algorithm because it doesn't deal with mathematical operations as much as the other algorithms. The time complexity of the algorithm needs to be of order n [O(n)].

   (d) The compression algorithm will be a lossless implementation, so as to perfectly preserve the first 25% of the Fourier coefficients contained within the raw sensor data.

2. **Encryption**

   (a) The encryption algorithm chosen is a AES encryption algorithm.

(b) The AES encryption method is very lightweight as it only takes up 14,7KB of memory and offers multiple key length choices. The time complexity of the algorithm needs to be of order n [O(n)]

(c) The encryption algorithm will be a lossless implementation, so as to perfectly preserve the first 25% of the Fourier coefficients contained within the raw sensor data.

3. **Communication Protocols**

(a) The I2C communication protocol will be used in this project for communication between the STM32F051 and the digital IMU module.

(b) I2C communication has many different speed modes. These are as follows, standard mode: 100 kbit/s, full speed: 400 kbit/s, fast mode: 1 mbit/s, high speed: 3,2 Mbit/s. [9]

(c) The communication between the STM32F051 and the IMU module will be controlled by a timing structure governed by the internal clock of the STM32F051. This allows for precise timing of the data reading.

(d) Communication between the STM32F051 and an external computing device will utilize the UART (Universal Asynchronous Receive Transmit) communication protocol. An FTDI device will be utilized to convert the UART signal to USB which is what our personal laptops utilize. As the name suggests, UART allows for two-way communication. UART allows for a maximum data transmission speed of around 5 Mbps.[10]

IMU Subsystem Specifications

1. **Communication Protocol**

(a) As mentioned beforehand, the IMU module will be connected into the system via the I2C communication protocol with measurements being taken at predefined intervals controlled by the onboard oscillator of the STM32F051.

External Computing Device Subsystem Specifications

1. **Communication Protocol**

(a) As mentioned beforehand, the communication between the STM32F051 and the external computing device will take place over UART communication protocol.

2. **De-encryption**

(a) Decryption of the AES level encrypted data will take place on the external computing device.

3. **De-compression**:

(a) De-compression will take pace on the external computer. This process will be as fast as the compression process and will produce the exact same data as was initially compressed, due to the fact that dictionary, lossless compression is being used. A fully lossless compression algorithm choice means that all of the first 25% of the Fourier components contained within the data can be extracted.

4.3.4 Inter-Subsystem and Inter-Sub-subsystems Interactions

As mentioned in section 4.3 above the main subsystems in the project are the STM32F051 microcontroller, the IMU sensor and the external computing device. Section 4.3.1 goes on to detail that the main sub-subsystems are the compression/decompression blocks, the encryption/decryption blocks and the communication protocols.

The main interactions of the entire system are as follows. The IMU module performs measurements that are then stored in the memory addresses of the module itself. The measurements are read into the STM32F051 microcontroller via the I2C communication protocol. The data is then processed on the STM32F051 microcontroller, this processing involves formatting the raw data, compressing this formatted data, and then encrypting this data using the aforementioned compression and encryption algorithms. The compressed and encrypted data is then to transmitted to the external computing device. This communication takes place via the UART communication protocol. Once the transmitted data is on the external computing device, decryption and decompression takes place. The data is then stored for viewing and for further processing to take place. Additional processing could be, for example, performing Fourier transforms to analyse the data in the frequency domain.

The last interactions that may take place are for acceptance test procedures. This involves taking the IMU module out of the system itself and feeding data to the STM32F051 microcontroller directly. This would involve the following. Raw data on the external computing device is sent over the UART communication protocol to the STM32F051 microcontroller. The STM32F051 microcontroller compresses and encrypts the data sent from the external computer and then sends it back to the external computer whereupon decryption and de-compression can take place. The original data can then be compared to the recently received data and a comparison can take place.

4.3.5 UML Diagram

The UML diagram in figure 4.3 below is similar to the block diagram in figure 4.2 above. It differs in that it is only concerned with the STM32F051 micro controller and external computer modules and incorporates more technical coding-related terms and structures.

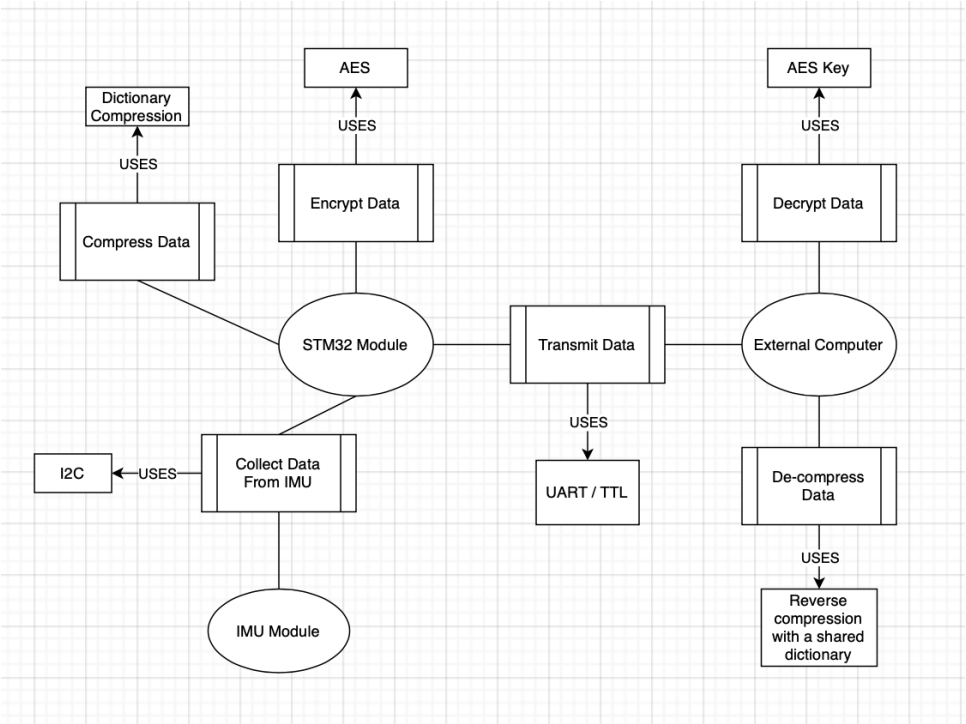


Figure 4.3: UML diagram depicting system interactions

4.3.6 Figures of Merit

Figures of merit are the quantities used to describe the efficiency of a device. It is challenging to set numerical values to the experiment during the planning phase and therefore these figures of merit will act as an estimate but are subject to change when project implementation begins. The figures of merit are:

- The decrypted and decompressed data should reflect 25% of the Fourier coefficients of the original IMU data.
- The encryption algorithm should use a 128, 192 or 256 bit key size.
- The compression algorithm should have a compression ratio of between 40% and 60%.
- The STM32F051 microcontroller baud rate is 38400 bps.

4.4 Experiment Design

The following details a preliminary experiment set-up that can be used to test the specification and corresponding ATPs in table 3.1. Although there are further specifications detailed in the Subsystem and Sub-subsystems Specifications section above, given the limited time frame of the project, this initial implementation will focus on the base Requirements and not the Subsystem and Sub-subsystems Requirements. It is recommended in Future Work that the sub-system requirements also be taken into consideration.

4.4.1 Overall System Experiment Design

The following experiment aims to test the communication between the subsystems and overall system functionality. This experiment aims to test acceptance test procedures A1, A2 and A5 from table 3.1 above. The method is as follows:

1. Connect the system fully including: IMU, STM32F051 micrcontroller and PC
2. Configure communication protocols: USB to UART between the STM32F051 micrcontroller and the computer and I2C between the STM32F051 micrcontroller and the IMU.
3. Take in data for a specified amount of time and send to the PC. This specified amount of time will depend on the STM32F051 micrcontroller's memory space limitations and are specified in section 6.2 below.
4. Analyse this data to ensure that correct or valid IMU sensor data has been produced and analyse it further in the frequency domain by taking the Fourier Transform.
5. Pass the data through the compression and encryption blocks and obtain the data ready for transmission.
6. Transmit this data to the PC
7. Decrypt and decompress the data on the PC and ensure no loss has occurred.
8. Analyse this data in the frequency domain by taking the Fourier Transform to ensure that the first 25% of Fourier co-efficients have been retained.

4.4.2 Compression Experiment Design

The compression experiment aims to test the compression algorithm to determine its efficiency and accuracy. This experiment aims to test acceptance test procedures A3.1, A3.2 and A6 from table 3.1 above. The method is as follows:

- The compression algorithm will run on a dataset of known size and a time measurement will be taken for the program's runtime.

- This will be repeated on 5 datasets of varying size. The dataset sizes will vary in the simulated and practical experiments (i.e. larger datasets sizes can be used in the theoretical simulated Experiment Setup and smaller dataset sizes will be used in the practical Experiment Setup due to memory constraints of the STM32F051 microcontroller). The dataset sizes are specified in the respective experiment setups in sections 5.2 and 6.2 below.
- A graph will be plotted to determine the time complexity of the compression algorithm. This graph will plot dataset size vs. runtime.
- The compression ratio will be calculated and compared to the ideal case.
- The compressed file will then be uncompressed and compared to the original data to test for signs of data loss.
- The Fourier transform of the pre-compressed and uncompressed data are compared to ensure that 25% of the Fourier coefficients are retained.

4.4.3 Encryption Experiment Design

The encryption experiment aims to test the encryption algorithm to determine its efficiency and accuracy. This experiment aims to test acceptance test procedures A4.1, A4.2 and A6 from table 3.1 above. The method is as follows:

- The encryption algorithm will run on a dataset of known size and a time measurement will be taken for the program's runtime.
- This will be repeated on 5 datasets of varying size. The dataset sizes will vary in the simulated and practical experiments (i.e. larger datasets sizes can be used in the theoretical simulated Experiment Setup and smaller dataset sizes will be used in the practical Experiment Setup due to memory constraints of the STM32F051 microcontroller). The dataset sizes are specified in the respective experiment setups in sections 5.2 and 6.2 below.
- A graph will be plotted to determine the time complexity of the encryption algorithm. This graph will plot dataset size vs. runtime.
- The encrypted file will then be decrypted and compared to the original data to test for signs of data loss or tampering.
- The Fourier transform of the encrypted and decrypted data are compared to ensure that 25% of the Fourier coefficients are retained.

Chapter 5

Validation using Simulated Data

The following section includes the validation of the project using Simulated or Old Data (provided IMU data sample as analysed in section 5.1 below) or simulated implementation of the project. This is detailed by the first progress report submission. The simulated implementation is necessary to ensure the proposed algorithms are compatible with IMU data structure and that they meet the relevant specifications (S3.1, S3.2, S4.1, S4.2 S6 in table 3.1 above). The simulated project version provides a 'golden measure' for the experiment as it acts as a benchmark or fully functioning, if not slighty un-optimised, version of the project with which future versions can be compared. Therefore, Python programming language is used for reasons detailed in section 5.2.1 below.

5.1 Data Analysis

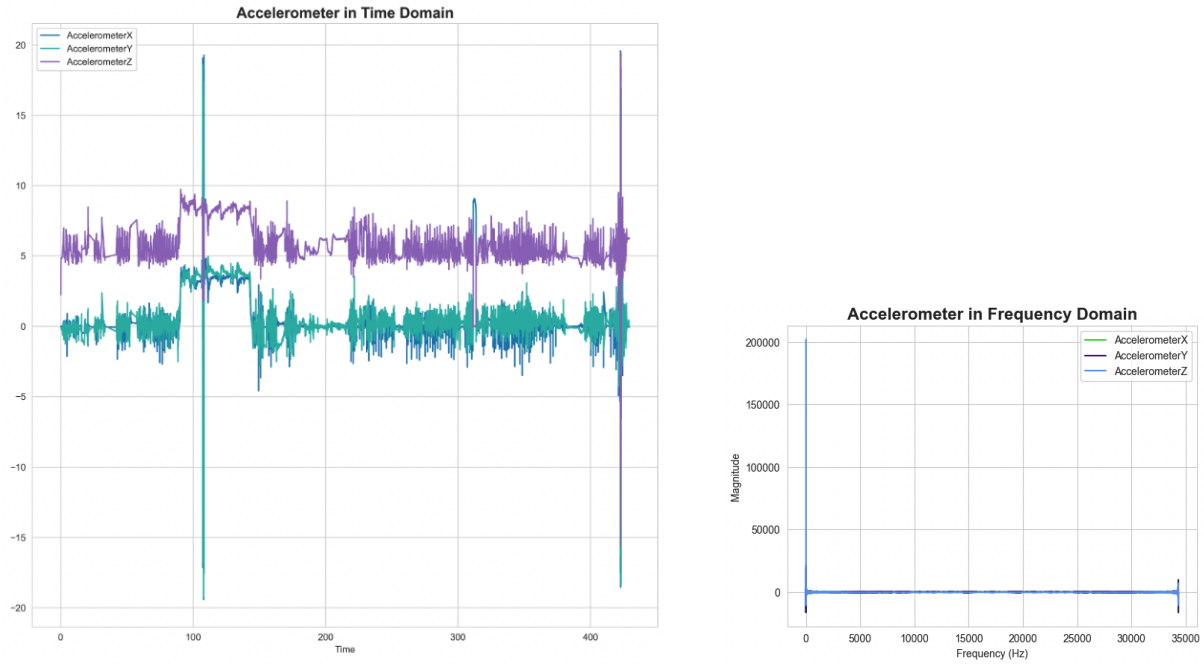
The data chosen for initial analysis is the provided 'Walking Around Example Data' file. This data was chosen as it is large in size which allows for in depth and comprehensive graphical examination of the data.

This data was chosen in default as the students have not yet received the IMU sensors at this time of the project and therefore cannot use the data retrieved directly from the sensor. It is expected in the future that the data come directly from the IMU connected to the STM32F051 microcontroller. This retrieved data will be verified by visual analysis to confirm that the communication link between the computer and the STM32F051 microcontroller yielded correct data and through IMU Validation Tests. The limitation of using this 'example' data is that the first two ATPs (A1 and A2) as detailed in table 3.1 above cannot be tested however the remaining ATPs (A3 - A6) can be.

For this analysis, the data that will be focused on is that which comes from the Accelerometer, Magnetometer and Gyroscope sensors as these are on the ICM-20649 IMU sensor that the project is designed to utilize. This data is specified in the x, y and z directions.

5.1.1 Accelerometer Data Analysis

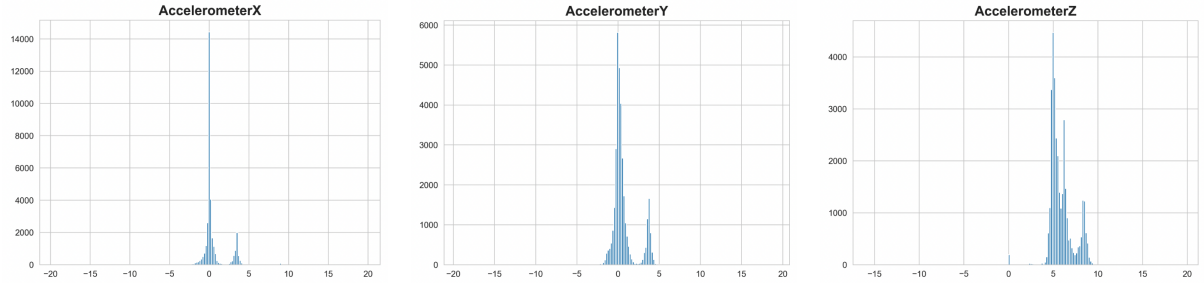
The Accelerometer data is plotted in the time and frequency domains in figure 5.1 below. The frequency domain plot was obtained by taking the Fourier Transform of the data.



(a) Accelerometer Time Domain (b) Accelerometer Frequency Domain

Figure 5.1: Accelerometer in time and frequency domains

A histogram was plotted for the x, y and z direction time domain Accelerometer data in figure 5.2 below.



(a) x Accelerometer histogram (b) y Accelerometer histogram (c) z Accelerometer histogram

Figure 5.2: Accelerometer histograms in time domain

The properties of the Accelerometer data are tabulated in table 5.1 below:

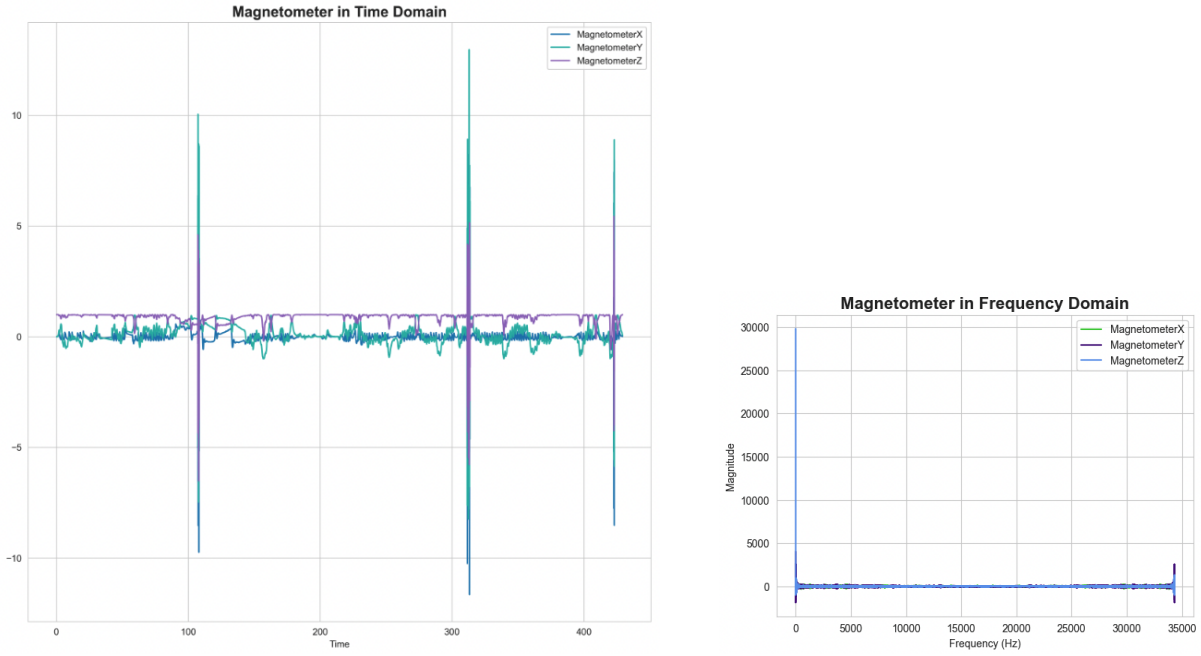
Catgeory	AccelerometerX	AccelerometerY	AccelerometerZ
Count	34328	34328	34328
Mean	0,532169	0,627148	5,886018
Standard Deviation	1,614881	1,580636	1,370541
Minimum	-19,410999	-19,2278	-15,6586
Maximum	19,5781	18,9123	19,457701

Table 5.1: Properties of Accelerometer Data

The Accelerometer data has two Gaussian distributions as shown by figure 5.2 above. This suggests that there are two major 'events' occurring during data retrieval. The Gaussian distributions suggests that the data is symmetric about the two means (or events) and that the data is adequately 'random'.

5.1.2 Magnetometer Data Analysis

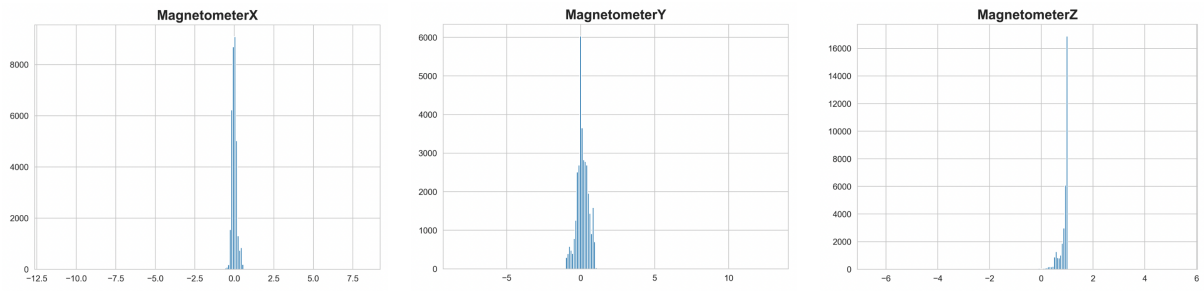
The Magnetometer data is plotted in the time and frequency domains in figure 5.3 below. The frequency domain plot was obtained by taking the Fourier Transform of the data.



(a) Magnetometer Time Domain (b) Magnetometer Frequency Domain

Figure 5.3: Magnetometer in time and frequency domains

A histogram was plotted for the x, y and z direction time domain Magnetometer data in figure 5.4 below.



(a) x Magnetometer histogram (b) y Magnetometer histogram (c) z Magnetometer histogram

Figure 5.4: Magnetometer histograms in time domain

The properties of the Magnetometer data are tabulated in table 5.2 below:

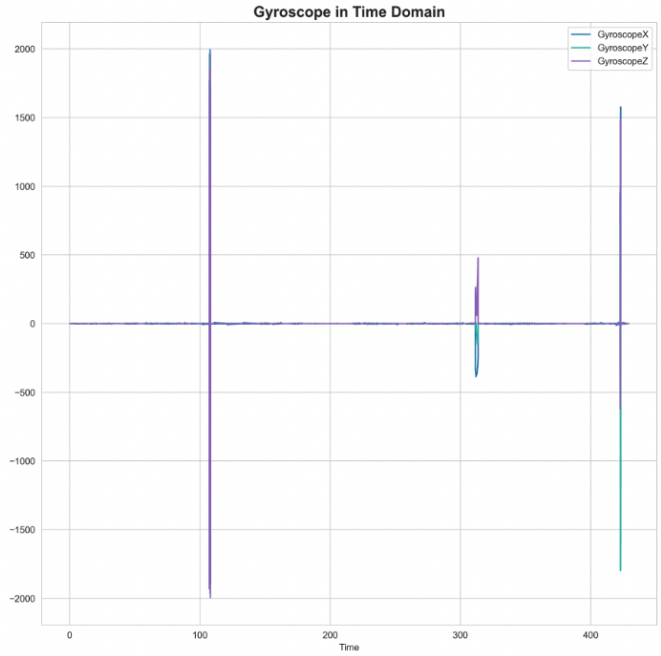
Catgeory	**MagnetometerX**	**MagnetometerY**	**MagnetometerZ**
Count	34328	34328	34328
Mean	-0,006781	0,117376	0,868291
Standard Deviation	0,427628	0,57007	0,317025
Minimum	-11,645	-8,2363	-6,5104
Maximum	8,2309	12,9682	5,4401

Table 5.2: Properties of Magnetometer Data

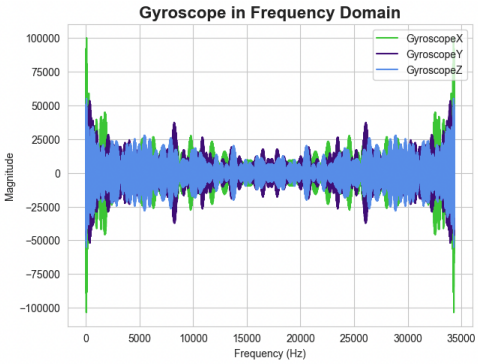
The Magnetometer data follows a Gaussian distribution as shown by figure 5.4 above. This suggests that the data is relatively symmetric about the mean and that the data is adequately 'random'. The low standard deviations of data suggest that the data does not deviate greatly from the mean.

5.1.3 Gyroscope Data Analysis

The Gyroscope data is plotted in the time and frequency domains in figure 5.5 below. The frequency domain plot was obtained by taking the Fourier Transform of the data.



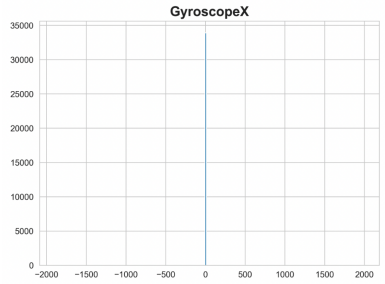
(a) Gyroscope Time Domain



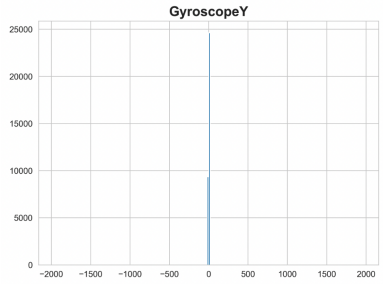
(b) Gyroscope Frequency Domain

Figure 5.5: Gyroscope in time and frequency domains

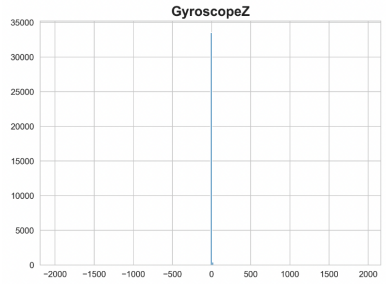
A histogram was plotted for the x, y and z direction time domain Gyroscope data in figure 5.6 below.



(a) x Gyroscope histogram



(b) y Gyroscope histogram



(c) z Gyroscope histogram

Figure 5.6: Gyroscope histograms in time domain

The properties of the Gyroscope data are tabulated in table 5.3 below:

Catgeory	**GyroscopeX**	**GyroscopeY**	**GyroscopeZ**
Count	34328	34328	34328
Mean	-0,533702	-0,52153	1,172048
Standard Deviation	79,11129	76,232833	73,234824
Minimum	-1893,536621	-1964,573242	-1993,78064
Maximum	1993,292847	1963,902344	1963,841431

Table 5.3: Properties of Gyroscope Data

The Gyroscope data follows a Gaussian distribution as shown by figure 5.6 above. This

suggests that the data is relatively symmetric about the mean and that the data is adequately 'random'.

5.2 Experiment Setup

Before experiments were performed, the decision of order of encryption and compression of the data needed to be made. Compression relies on patterns and predictability of data and encryption randomises the data resulting in a reduction in patterns and predictability which would reduce the efficiency of compression. Therefore, data should first be compressed and then encrypted. [11]. The following experiment set-ups loosely follow the preliminary Experiment Design outlined in section 4.4 above. It only includes the Compression Experiment Design and the Encryption Experiment Design and not the Overall System Experiment Design as this portion is a simulated experiment and does not test the overall system which includes hardware implementation.

5.2.1 Simulations

For the simulations, python programs were used. Although the project requires code in C/C++ for STM32F051 microcontroller compatibility, python allows for easier and more flexible testing to occur. Python is appropriate at this phase of the project as the main focus is simulation based and will not involve the use of the STM32F051 yet. The main algorithms are the compression algorithm as stated in Appendix A.1 [12] below and the encryption algorithm as stated in Appendix A.2 [13] below. A third comparison algorithm is also used as stated in Appendix A.3 [14] below to compare data to ensure minimal loss, this algorithm outputs a percentage difference between two inputted files.

The main pythons script is set up such that the compression algorithm accepts a varying number of lines of data from a csv file containing sample data. This data set of variable size is then compressed and passed to the encryption method to be encrypted. Decryption and subsequent decompression take place before the data is compared to the original data to test for data loss. The time taken for compression, encryption, decompression and decryption was measured using a timing algorithm as stated in Appendix A.4 [15].

5.2.2 Compression Block

The aim of the compression experiment will be to determine in the compression algorithm meets the specifications and corresponding ATPs. The following method will be followed to test the compression algorithm:

- The compression algorithm A.1 will run on a dataset of known size and a time measurement will be taken for the compression part of the program's runtime.

- This will be repeated on 23 datasets of varying sizes. The varying sizes of data input are determined by the number of lines inputted into the program. This ranges from 10 lines to 33 010 lines.
- A graph will be plotted to determine the time complexity of the compression algorithm. This graph will plot dataset size vs. runtime.
- The compression ratio will be calculated for each of the 23 compression cases and this will also be plotted.
- The compressed file will then be decompressed for the 23 compression cases and a graph will be plotted to determine the time complexity of the decompression algorithm. This graph will plot dataset size vs. runtime.
- The decompressed data will then be compared to the original data using the comparison algorithm A.3 to test for signs of data loss.
- The Fourier transform of the original data and the decompressed data will be verified to have the same first 25% Fourier coefficients.

The compression block expects to receive raw data and produce compressed data of reduced size.

5.2.3 Encryption Block

The aim of the encryption experiment will be to determine in the encryption algorithm meets the specifications and corresponding ATPs. The following method will be followed to test the encryption algorithm:

- The encryption algorithm A.2 will run on a compressed dataset of known size and a time measurement will be taken for the program's runtime.
- This will be repeated on 23 datasets of varying sizes. The varying sizes of data input are determined by the number of lines inputted into the compression program. This ranges from 10 lines all the way up to 34 010 lines.
- A graph will be plotted to determine the time complexity of the encryption algorithm. This graph will plot dataset size vs. runtime.
- The encrypted file will be compared to the pre-encrypted file using the comparison algorithm A.3 to test for minimal similarities and thus adequate encryption.
- The encrypted file will then be decrypted for the 23 compression cases and a graph will be plotted to determine the time complexity of the decryption algorithm. This graph will plot dataset size vs. runtime.
- The encrypted file will then be decrypted and compared to the original data using the comparison algorithm A.3 to test for signs of data loss or tampering.

The encryption block expects to receive compressed data and produce encrypted data of increased randomization.

5.3 Results

The results of the above mentioned experimental setups for the compression and encryption blocks are shown below.

The program was setup such that the dataset size (i.e. number of lines of csv file) was varied and the compression, encryption, decryption and decompression algroithms were automated. The program automatically records (in csv format): the number of lines in the csv file (i.e. the dataset size), the timing of these four methods, the size of the original data, the size of the compressed data, the compression ratio and a boolean variable stating whether or not the decompressed data matches the original data. Table 5.4 below shows the results of the experiment.

Lines of Data	Compress Time (s)	Encrypt Time (s)	Decrypt Time (s)	Decompress Time (s)	Original Size (B)	Compressed Size (B)	Compression Ratio	Data Comparison
10	5.698204040527344e-05	0.008265018463134766	0.008541345596313477	5.412101745605469e-05	4004	1407	2.845771144278607	TRUE
1510	0.0014219284057617188	1.2346031665802002	1.1993489265441895	0.0005030632019042969	636341	199383	3.191550934633344	TRUE
3010	0.0030820369720458984	2.483834981918335	2.5102078914642334	0.0009360313415527344	1260976	407195	3.096737435381083	TRUE
4510	0.008454084396362305	3.7066709995269775	3.6556930541992188	0.001850128173828125	1882197	619049	3.0404652943466512	TRUE
6010	0.011243820190429688	5.106395721435547	5.0341410636901855	0.0037460327148437 5	2502724	827160	3.0256830601092894	TRUE
7510	0.015943050384521484	6.29496693611145	6.266692876815796	0.004539966583251953	3116791	1037100	3.005294571401022	TRUE
9010	0.012837886810302734	7.821256875991821	7.764508962631226	0.005251884460449219	3732423	1246056	2.9953894528014793	TRUE
10510	0.010243654251098633	8.91141414642334	8.88274598121643	0.0071642398834228516	4348083	1451452	2.995678120943717	TRUE
12010	0.01190495491027832	9.457523107528687	9.51429033279419	0.0070052146911621 09	4975697	1662961	2.9920707701503524	TRUE
13510	0.014448165893554688	11.50040602684021	11.365385293960571	0.007726192474365234	5605845	1869210	2.999045051117852	TRUE
15010	0.01681685447692871	11.94100308418274	12.004706144332886	0.009037017822265625	6242389	2049173	3.0462967255570907	TRUE
16510	0.017476797103881836	13.139831066131592	13.046177864074707	0.011100292205810547	6879158	2219491	3.099430455000719	TRUE
18010	0.019752979278564453	15.192198991775513	15.078176975250244	0.01158905029296875	7517537	2393168	3.1412491726447955	TRUE
19510	0.024883031845092773	15.257004737854004	15.120185852050781	0.01331782341003418	8145650	2603725	3.1284601868476893	TRUE
21010	0.02616596221923828	17.117594718933105	17.084326028823853	0.013182878494262695	8775760	2807400	3.1259385908669945	TRUE
22510	0.027812957763671875	18.642353057861328	18.54386305809021	0.013921022415161133	9401424	3018094	3.115020274385092	TRUE
24010	0.02629399299621582	21.60689091682434	23.784772872924805	0.014039993286132812	10028875	3228572	3.106288167028643	TRUE
25510	0.02746891975402832	21.42377495765686	21.103504180908203	0.009458780288696289	10646412	3438397	3.096330063107896	TRUE
27010	0.02765512466430664	20.83461093902588	20.83729362487793	0.015753984451293945	11274051	3645814	3.0932375295996997	TRUE
28510	0.0314488410949707	23.77382493019104	23.65444588661194	0.016793251037597656	11897055	3856139	3.0852246249422026	TRUE
30010	0.030972957611083984	24.934611320495605	25.514904260635376	0.011677980422973633	12526243	4061942	3.083806464986452 5	TRUE
31510	0.0359091758728027 34	26.269142150878906	25.221153020858765	0.019836902618408203	13161134	4223094	3.1164672157427704	TRUE
33010	0.041114091873168945	26.270262002944946	25.94036602973938	0.0164029598236084	13786535	4429737	3.1122694191551328	TRUE

Table 5.4: Simulated Experimental Data

5.3.1 Compression Block

Data pertaining to the compression block of the experiment from table 5.4 above is discussed and where relevant plotted, below.

The dataset size is considerably reduced as a result of the compression algorithm, for instance a dataset size of of 13786535B compressed to a file size of 4429737B which means A3.1 in table 3.1 above is met. Figure 5.7 below shows how the runtime of the compression algorithm changes over varying dataset sizes (number of lines of data).

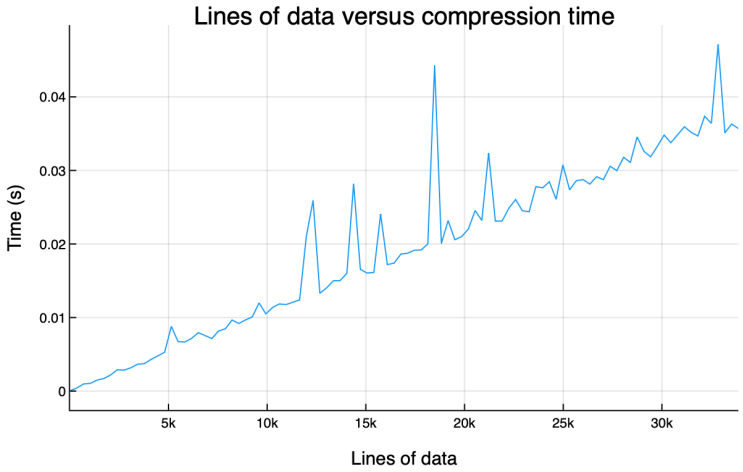


Figure 5.7: Graph showing the time required to compress varying sizes of input data.

It can concluded from figure 5.7 above that the lz4 compression algorithm implemented in this particular setup has a time complexity of order n [O(n)] which meets A6 in table 3.1 above. This is to be expected from the lz4 compression library.

Figure 5.8 below shows how the runtime of the decompression algorithm changes over varying dataset sizes (number of lines of data).

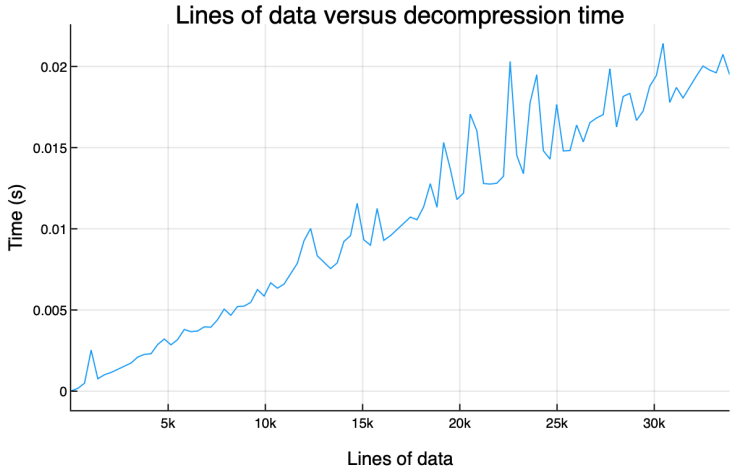


Figure 5.8: Graph showing the time required to decompress varying sizes of input data.

As with the results from the compression times, it can concluded from figure 5.8 above that the time complexity of the decompression algorithm is also order n [O(n)] which meets A6 in table 3.1 above.

Figure 5.8 below shows the change in compression ratio over varying dataset sizes (number of lines of data).

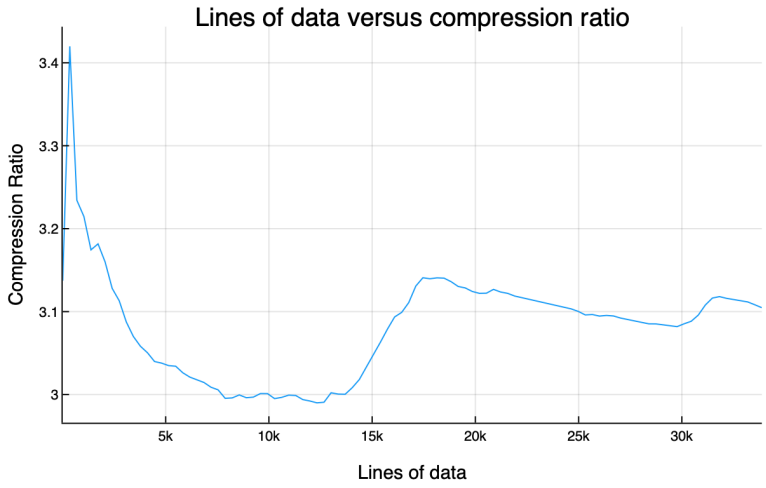


Figure 5.9: Graph showing the compression ratio over varying sizes of input data.

It can be seen that the compression ratio for the system starts off fluctuating and as the dataset increases in size, the compression ratio stabilizes to around 3,1 which meets A3.2 in table 3.1 above.

The comparison algorithm as stated in appendix A.3 outputted a percentage of 100% as shown by the output in figure 5.10 below. This suggest that there is no difference between the original data and the decompressed data.

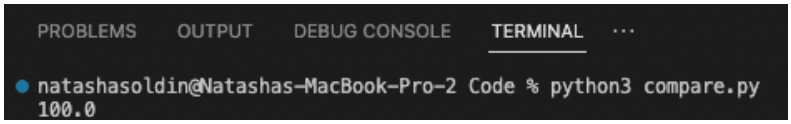
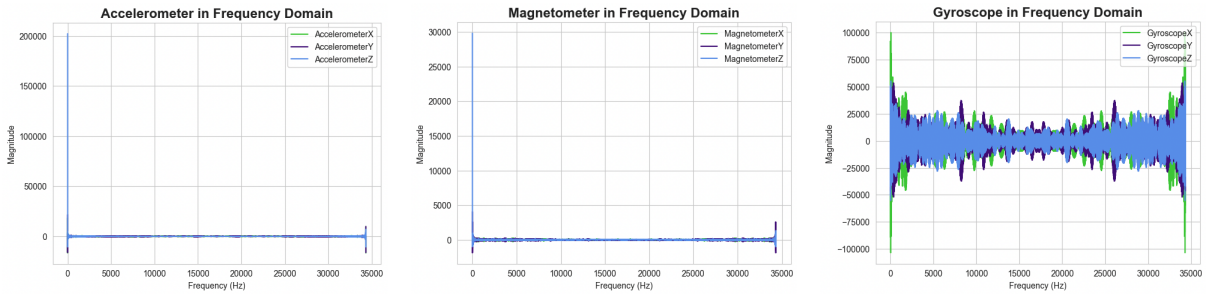


Figure 5.10: Output of the Comparison Algorithm for Decompressed Data

This further infers that the Fourier transforms of the two datasets will be the same and therefore the first 25% of the Fourier coefficients were retained successfully. Further proof was obtained by taking the Fourier transform of the Accelerometer, Gyroscope and Magnetometer post decompression as shown by figure 5.11 below.



(a) Accelerometer (b) Magnetometer (c) Gyroscope

Figure 5.11: Frequency Domain

These graphs can be compared to the frequency domain representations of the original data in figures 5.1b, 5.3b and 5.5b in the Data Analysis above which confirms a retention of all Fourier co-efficients and thus meeting A5 in table 3.1 above.

5.3.2 Encryption Block

Data pertaining to the encryption block of the experiment from table 5.4 above is discussed and where relevant plotted, below..

Figure 5.12 below shows how the runtime of the encryption algorithm changes over varying dataset sizes (number of lines of data).

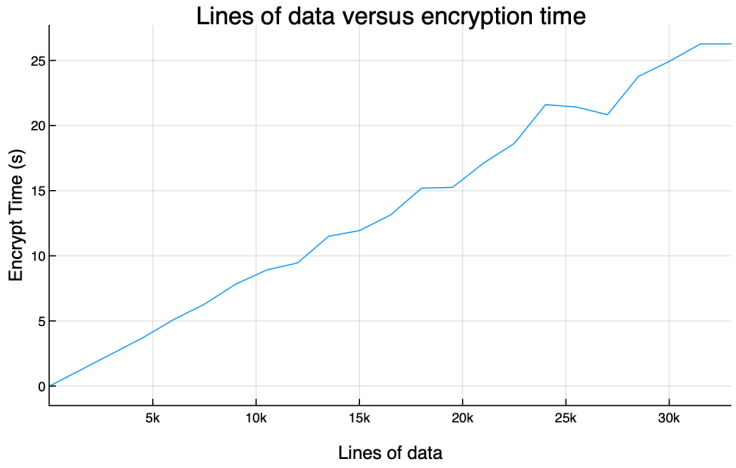


Figure 5.12: Graph showing the time required to encrypt varying sizes of input data.

It can concluded from figure 5.12 above that the AES encryption algorithm implemented in this particular setup has a time complexity of order n [O(n)] which meets A6 in table 3.1.

Figure 5.13 below shows how the runtime of the decryption algorithm changes over varying dataset sizes (number of lines of data).

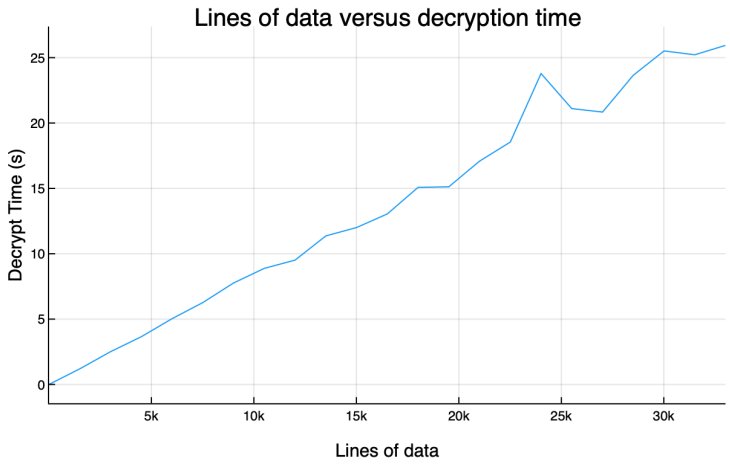


Figure 5.13: Graph showing the time required to decrypt varying sizes of input data.

As with the results from the encryption times, it can concluded from figure 5.8 above that the time complexity of the decryption algorithm is also order n [O(n)] which meets A6 in table 3.1 above.

The comparison algorithm as stated in appendix A.3 outputted a percentage of 13.79% as shown by the output in figure 5.14 below. This suggest that there is adequate difference between the compressed (pre-encrypted) data and the encrypted data which meets A4.1 in table 3.1 above.

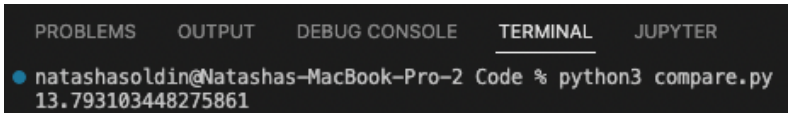


Figure 5.14: Output of the Comparison Algorithm for Encrypted Data

The comparison algorithm as stated in appendix A.3 outputted a percentage of 100% as shown by the output in figure 5.15 below. This suggest that there is no difference between the compressed (pre-encrypted) data and the decrypted data which meets A4.2 in table 3.1 above.

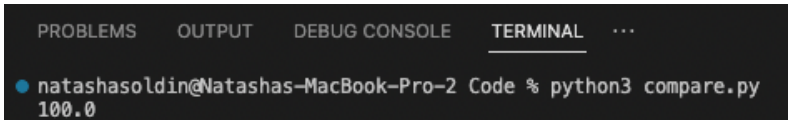


Figure 5.15: Output of the Comparison Algorithm for Decrypted Data

Note: to perform the above comparisons of encrypted and pre-encrypted or decrypted data (i.e. compressed data) requires manual manipulation (i.e. copying and pasting). This is because the compressed data is in binary array format and cannot be accessed directly, but a 'crude' text representation can be accessed which was submitted to the

encryption and comparison algorithms.

It is relevant to note that the time taken for the encryption algorithm is larger than initially anticipated. This can be attributed to the fact that the input into the encryption algorithm is in the form of a string of substantial size which renders the algorithms inefficient. This is not expected to be an issue in the future because C/ C++ is faster than python as it is a compiled language while Python is an interpreted language [16]. In addition, the sample sizes of data that will be sent into the encryption algorithm will be considerably smaller which is more efficient to encrypt. Therefore, in the next phase of the project when code is implemented in C/C++, the encryption algorithm is expected be faster.

5.4 Acceptance Test Protocols

From the results in section 5.3 above, the ATPs can be tested to determine whether they have been met or not. This is tabulated in table 6.4 below:

ATP Number	**Has the ATP been met?**	**Reason**
A1	N/A	The student had not yet received the IMU sensor and therefore cannot obtain data from it. Within this progress report the provided 'Walking Around' IMU data was used as it is large in size and allows for multiple tests to be performed on it.
A2	N/A	This progress report does not yet require code in C/C++ as it is simulation focused. Therefore python was used for easier and more flexible testing to be performed.
A3.1	✓	A file size of 13786535B compressed to a file size of 4429737B. This is considerably less and therefore the compression had the desired affect in terms of size reduction.
A3.2	✓	The compression ratio was calculated to be around 3,1 as seen in figure 5.9 above. This is adequate as the project required a compression ratio between 40% and 60%
A4.1	✓	The similarity between the pre-encrypted data and post-encypted data is minimal or less that 20% as seen in figure 5.14 above. This enough to consider the encryption to be of adequate randomisation.
A4.2	✓	The pre-encrypted data is compared to the post-encrypted data and shows no loss of data as shown in figure 5.15 above.
A5	✓	The lower 25% of the fourier co-efficients have been retained when comparing the fourier transform of the original data to the decrypted and decompressed data as seen in figure 5.11 above.
A6	✓	The time complexity of the algorithms is O(n) as seen in figures 5.7, 5.8, 5.12 and 5.13 above.

Table 5.5: ATP Meeting and Reasoning

The ATPs and corresponding specification were not changed. Although the first two ATPs (A1 and A2) were not met, it is because they were not applicable to this portion of the project as the IMU sensors are not available to the student for data retrieval and verification. Use of the sample data allows for testing of the remaining ATPs (A3 - A6)

Chapter 6

Validation Using IMU Data

The following section includes the validation of the project using the IMU Data (as validated in section 6.1 below) or practical implementation of the project. This is detailed by the second progress report submission. The practical implementation is necessary to ensure data can be read directly from the IMU connected to the STM32F051 microcontroller (and therefore to test S1 and S2 in table 3.1 above) and to implement the algorithms in C to be compatible with the STM32F051 microcontroller (and therefore to test S3.1, S3.2, S4.1, S4.2, S5 and S6 in table 3.1 above). This hardware implementation is the next logical step towards the final project's product.

6.1 IMU Module

6.1.1 IMU Comparison

The project is intended to use the ICM-20649 IMU sensor but the project utilizes the ICM-20948 IMU sensor due to availability constraints. The following table 6.1 summarizes the key differences between to the sensors:

	ICM-20649 IMU	ICM-20948 IMU
Device	6-Axis MEMS Motion Tracking Device	9-Axis MEMS Motion Tracking Device
Supply Voltage	1.71V - 3.6V	1.95V - 3.6V
Temperature Range	- 40 C to +85 C	- 40 C to +85 C
Gyroscope	Digital-output X-, Y-, and Z-axis angular rate sensors (gyroscopes) with a user-programmable full-scale range of ± 500 dps, ± 1000 dps, ± 2000 dps, and ± 4000 dps and integrated 16-bit ADCs	Digital-output X-, Y-, and Z-axis angular rate sensors (gyroscopes) with a user-programmable full-scale range of ± 250 dps, ± 500 dps, ± 1000 dps, and ± 2000 dps, and integrated 16-bit ADCs
Accelerometer	Digital-output X-, Y-, and Z-axis accelerometer with a programmable full scale range of $4g$, $8g$, $16g$, and $30g$ and integrated 16-bit ADCs	Digital-output X-, Y-, and Z-axis accelerometer with a programmable full scale range of ± 2g, ± 4g, ± 8g, and ± 16 g, and integrated 16-bit ADCs
Magnetometer	N/A	3-axis silicon monolithic Hall-effect magnetic sensor with magnetic concentrator. Output data resolution of 16-bits and full scale measurement range is ± 4900 μT
Low Pass Filter	Digitally-programmable	User-selectable
Communications	I2C at up to 100 kHz (standard-mode) or up to 400 kHz (fast- mode), or SPI at up to 7 MHz.	I2C at up to 100 kHz (standard-mode) or up to 400 kHz (fast- mode), or SPI at up to 7 MHz.
Shock Tolerance	10,000g shock tolerant	20,000g shock tolerant
Application	High Impact Application	N/A

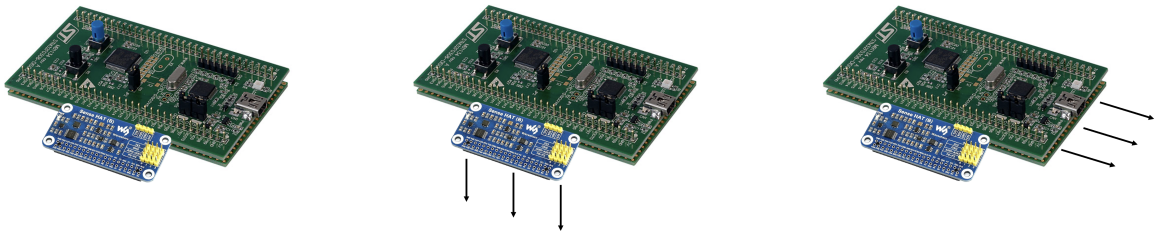
Table 6.1: IMU Sensor Comparison

Specification 1 (S1) in table 3.1 above ensures IMU sensor compatibility between the project and the intended ICM-20649 IMU sensor. This specification ensures that the electrical specifications of the project align with the electrical specifications of the intended ICM-20649 IMU sensor as shown by figure 3.1 in the Paper Design above. This ensures that extrapolation between the projects current implementation and future intended implementation is as seamless as possible.

6.1.2 IMU Validation Tests

The sensor data validated and analyzed in the following IMU module was the accelerometer data as this is one of the sensors on the ICM-20649 IMU sensor that this project is concerned with. Therefore the additional sensors on the ICM-20948 IMU can be ignored at this stage of the design.

- **V1 - Hardware Validation:** when correctly connected the LED on the IMU should be illuminated signaling correct configuration. After the powering up of the STM32F051, the I2C interface writes to certain memory addresses to take the IMU module out of sleep mode. This follows the 'start-up' sequence and then data should be able to be extracted from the IMU.
- **V2 - Sensor Validation - Accelerometer Validation:**
  - **V2.1 - Motion Validation:** the acceleration readings should be read for three cases: the system should be left stationary on a flat surface, dropped and allowed to free fall and moved from a stationary position with effective, unidirectional acceleration. As shown by figure 6.1 below:



(a) Stationary Position (b) Free Fall (c) Accelerated in one direction

Figure 6.1: Accelerometer Validation

– **V2.2 - Rotation Validation:** the acceleration readings should be read for two cases: the system should be rotated along the x-axis and then rotated along y-axis. As shown by figure 6.2 below:

(a) x-Axis Rotation (b) y-Axis Rotation

Figure 6.2: Gyroscope Validation

– **V2.3 - Pendulum Movement:** a common movement for IMU validation is to attach the system to a pendulum and allow it to swing. As shown by figure 6.3 below:

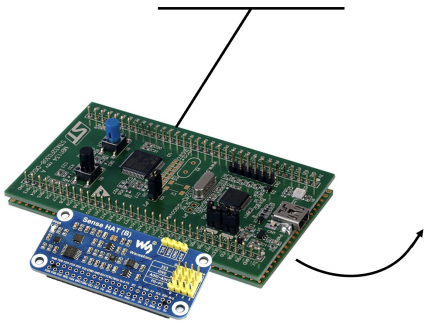


Figure 6.3: Pendulum Movement

6.1.3 IMU Validation Results

- **V1 - Hardware Validation:** The results of the hardware validation can be seen in figure 6.4 below. The IMU's light is illuminated when connected signalling correct configuration:

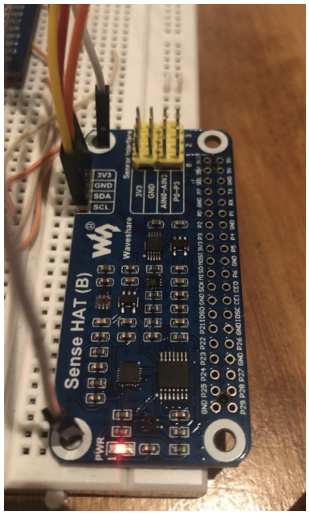


Figure 6.4: IMU Connection

- **V2 - Sensor Validation - Accelerometer Validation:**
  - **V2.1 - Motion Validation:** the results from experiment a) the stationary position experiment above can be graphically plotted as shown by figure 6.5 below:

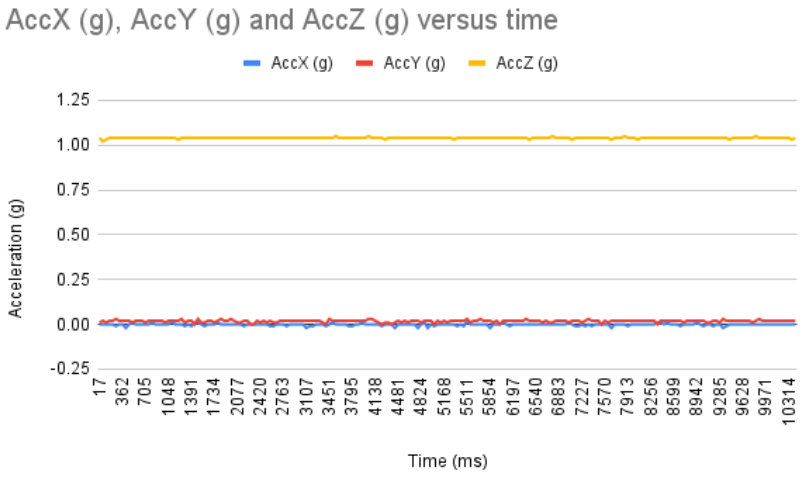


Figure 6.5: Accelerometer Validation Results 2.1a - Stationary Position

Figure 6.5 above demonstrates that when the IMU module is in a stationary, stable environment with no external forces except for gravitational force, the x and y acceleration is 0 and the z acceleration is constant due to gravity.

The results from experiment b) the free fall validation experiment above can be graphically plotted as shown by figure 6.6 below:

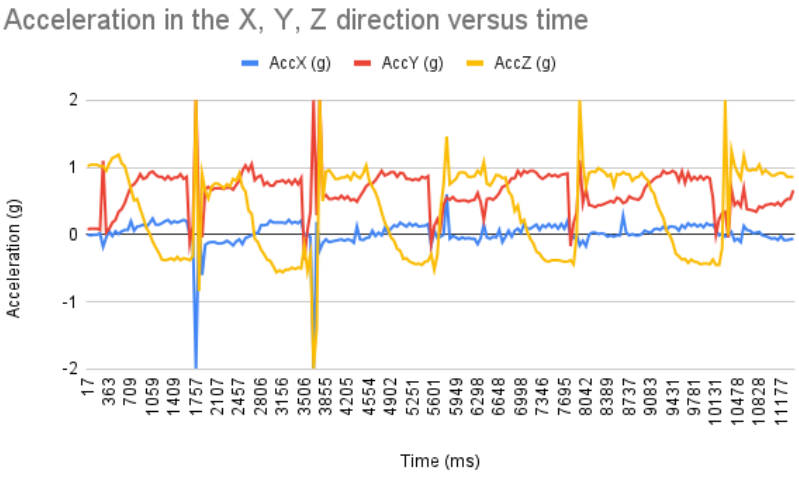


Figure 6.6: Accelerometer Validation Results 2.1b - Free Fall

The acceleration values can be separated into their x, y and z components and plotted separately shown in figure 6.7 below:

(a) x component of acceleration (b) y component of acceleration (c) z component of acceleration

Figure 6.7: Accelerometer Validation Results 2.1b - Free Fall

Above are the x, y, and z components of acceleration after a simple drop test was applied to the IMU module. The module was lifted and dropped onto a hard surface repeatedly over a time period of about 10 seconds. This resulted in hard spikes in the acceleration in the 3 different axis directions as a result of free fall's sudden halt.

The results from experiment c) the uni-directional acceleration validation experiment above can be graphically plotted as shown by figure 6.8 below:

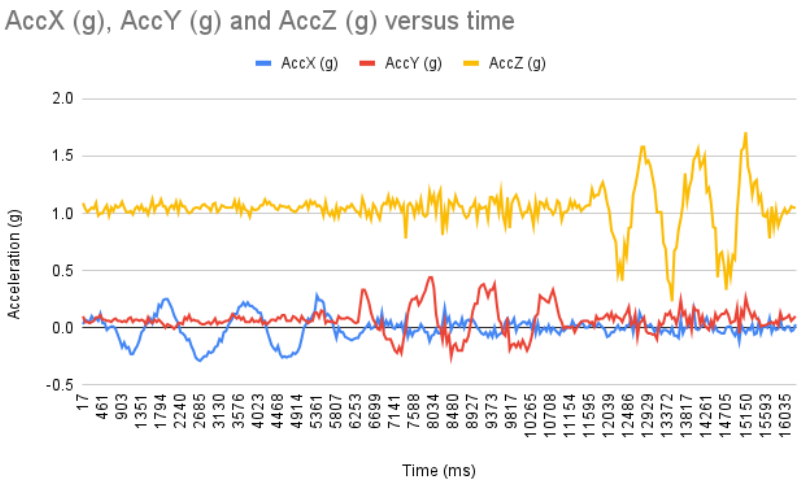


Figure 6.8: Accelerometer Validation Results 2.1c - Straight Line Movement

As can be seen, the acceleration in the different x, y, and z directions is increasing and decreasing as linear movement is experienced by the IMU at varying accelerations.

– **V2.2 - Rotation Validation:** the results from the rotation validation experiment above can be graphically plotted as shown by figure 6.9 below:

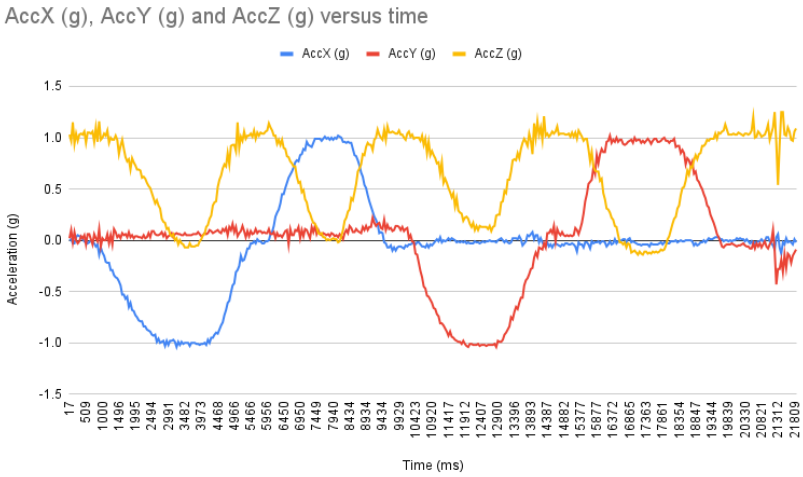


Figure 6.9: Rotation Validation Results

Both rotations (around x and y axis) are combined in a single experiment and plotted in figure 6.9 above. It can be seen that angular acceleration results in oscillatory motion in the x, y and z components of acceleration.

– **V2.3 - Pendulum Movement:** the results from the pendulum movement validation experiment can be graphically plotted as shown by figure 6.10 below:

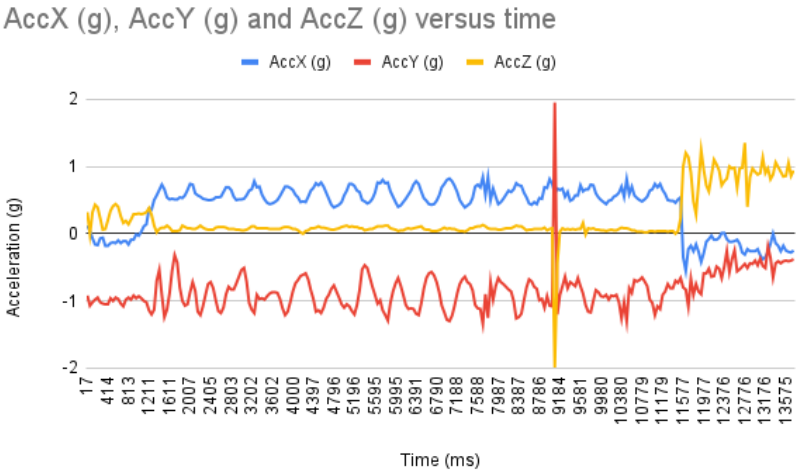


Figure 6.10: Pendulum Movement Results

The IMU validation test results shown in figures 6.5 to 6.10 above align with the expected output of the system. This signifies correct operation of the IMU sensor and correct output data formatting. This meets A1 in table 3.1 above.

6.2 Experiment Setup

In the practical implementation, the code needed to be written in C/C++. C was chosen as the preferred language for the project as it is more compatible with the STM32F051 microcontroller and STMCubeIDE.

Due to the limitations placed on the project as a result of the use of the STM32F051 microcontroller as detailed in section 4.1.3 above. Both compression and encryption algorithms were changed to simpler versions. This change was due to the inability of the lz4 compression and AES encryption algorithms chosen in sections 4.2 above to be adapted to operate on the STM3250F1 microcontroller. These algorithms require inclusion and header files and as well as additional resources which results in their being too 'heavy' and requiring too much memory space to be run of the STM3250F1 microcontroller. The compression algorithm was changed to an adapted version of LZSS compression called Heatshrink [17] and the encryption algorithm was changed to a basic ASCII shift encryption algorithm.

Although the compression algorithm was changed from the lz4 algorithm to the LZSS algorithm, it is still a type of dictionary compression algorithm which was chosen in section 4.2.1 above to be best suited to the project's context. However, the encryption algorithm chosen in section 4.2.2 above to be best suited to the project is a symmetric encryption method and the new ASCII shift encryption method is not of that type. The reason for this is that due to the limited space, a priority needed to be placed on one of the algorithms to allow it to be more advanced than the other and this priority was placed on compression.

Compression takes precedence over encryption in this project's context because the data transmission using iridium is extremely costly, compression of the data is vital to ensure cost reduction. The data is not sensitive in nature and therefore extensive encryption of the data is not necessary - the group has instead chosen to implement a simpler encryption algorithm and rely somewhat on the Iridium's network security. In addition, the compressed data is already reasonably different enough from the original data due to the manipulations performed by a dictionary compression algorithm. Therefore a simple encryption method suffices to ensure minimal similarities to the original data.

The specifications and ATPs were changed accordingly and are reflected in the new table in table 6.3 below.

The compression and encryption experimental blocks below follow a similar structure to those in the simulated based experiment in Validation using Simulated Data chapter. The biggest difference however is due to the space constraints on the STM32F051 microcontroller, fewer and smaller dataset sizes were chosen and the arising results are less exhaustive than before.

The compression algorithm's effectiveness is hypothesised to be hindered as the initial dataset size may be too small to be effectively compressed. Thus, it is hypothesises that the compression ratio will be significantly lower than previously but the hope is that the dataset can still be compressed enough to meet the project's specifications.

Another difference from the previous chapter is that the decryption and decompression algorithms will not be tested or timed as they do not directly relate to the core aspects of the project and will occur post-data transmission on the receiving PC. Decryption and decompression will still occur in this experiment to ensure no data is lost and thus that the lower Fourier co-efficients are retained.

A different string comparison algorithm will be used in this experiment as stated in Appendix B.3 below. This differs from the previous comparison algorithm A.3 as it compares two strings instead of two files and outputs the similarity as a percentage. This is more appropriate given the small dataset size inputs in this experiment.

The following experiment set-ups loosely follow the preliminary Experiment Design outlined in section 4.4 above. It includes the Compression Experiment Design, the Encryption Experiment Design and the Overall System Experiment Design as this portion is a practical experiment and includes hardware implementation.

6.2.1 Overall System

The aim of the overall system experiment block is to assess the overall functionality of the system.

1. Connect the system fully including: IMU, STM32F051 micrcontroller and PC

2. Configure communication protocols: USB to UART between the STM32F051 micrcontroller computer and I2C between the STM32F051 micrcontroller and the IMU.

3. Take in data for a specified amount of time and send to the PC.

4. Analyse this data in the frequency domain by taking the Fourier Transform.

5. Pass the data through the compression and encryption blocks and obtain the data ready for transmission.

6. Transmit this data to the PC

7. Decrypt and Decompress the data on the PC and ensure no loss has occurred.

8. Analyse this data in the frequency domain by taking the Fourier Transform to ensure that the first 25% of Fourier co-efficients have been retained.

6.2.2 Compression Block

The aim of the compression experiment will be to determine in the compression algorithm meets the specifications and corresponding ATPs. The following method will be followed to test the compression algorithm:

1. The compression algorithm B.1 will run on a dataset of known size and a time measurement will be taken for the compression part of the program's runtime.

2. This will be repeated on 5 datasets of varying sizes. The varying sizes of data input are determined by the time the data is being read off the sensor. The time will range from 0,5 to 5 seconds.

3. A graph will be plotted to determine the time complexity of the compression algorithm. This graph will plot dataset size vs. runtime.

4. The compression ratio will be calculated for each of the 5 compression cases and this will also be plotted.

5. The compressed file will then be decompressed.

6. The decompressed data will then be compared to the original data using a comparison algorithm B.3 to test for signs of data loss.

7. The Fourier transform of the original data and the decompressed data will be verified to have the same first 25% Fourier coefficients.

The compression block expects to receive raw data and produce compressed data of reduced size.

6.2.3 Encryption Block

The aim of the encryption experiment will be to determine in the encryption algorithm meets the specifications and corresponding ATPs. The following method will be followed to test the encryption algorithm:

1. The encryption algorithm B.2 will run on a compressed dataset of known size and a time measurement will be taken for the program's runtime.
2. This will be repeated on 5 datasets of varying sizes. The varying sizes of data input are determined by the time the data is being read off the sensor. The time will range from 0,5 to 5 seconds.
3. A graph will be plotted to determine the time complexity of the encryption algorithm. This graph will plot dataset size vs. runtime.
4. The encrypted data will be compared to the pre-encrypted data using the comparison algorithm B.3 to test for minimal similarities and thus adequate encryption.
5. The encrypted file will then be decrypted and compared to the original data using a comparison algorithm B.3 to test for signs of data loss or tampering.

The encryption block expects to receive compressed data and produce encrypted data of increased randomization.

6.3 Results

The results of the above mentioned experimental setups for compression and encryption blocks are tabulated in table 6.2 below. This table includes: the time that the data is read for, the size of the original dataset (number of character), the compression time, the size of the compressed dataset (number of character), the compression ratio and the encryption time.

Data Intake Time (s)	Original Dataset Size (no. characters)	Compression Time (s)	Compressed Dataset Size (no. characters)	Compression Ratio	Encryption Time (s)
0,5	176	312	1062	0,165725047	286
1	269	316	1060	0,253773585	284
2	471	321	1058	0,445179584	288
3	670	328	1056	0,634469697	291
4	871	334	1059	0,822474032	295
5	990	337	1058	0,935727788	300

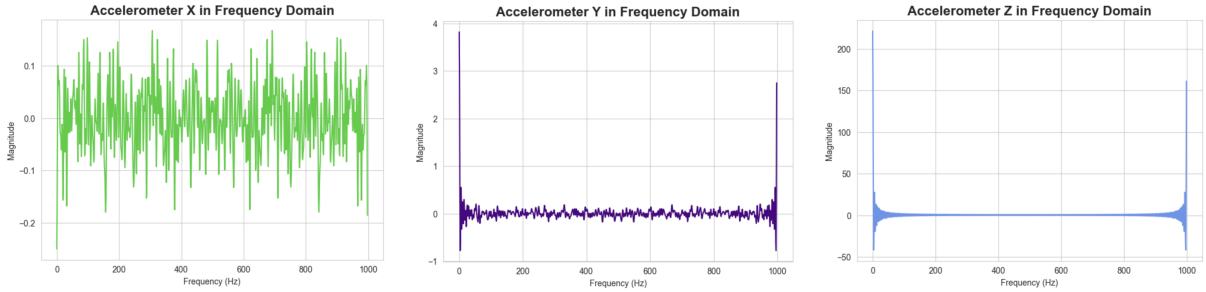
Table 6.2: Practical Experimental Data

Note: the maximum time that the data was able to be read in for was 5 seconds. This is because of the memory space limitation of the STM32F051 microcontroller.

6.3.1 Overall System

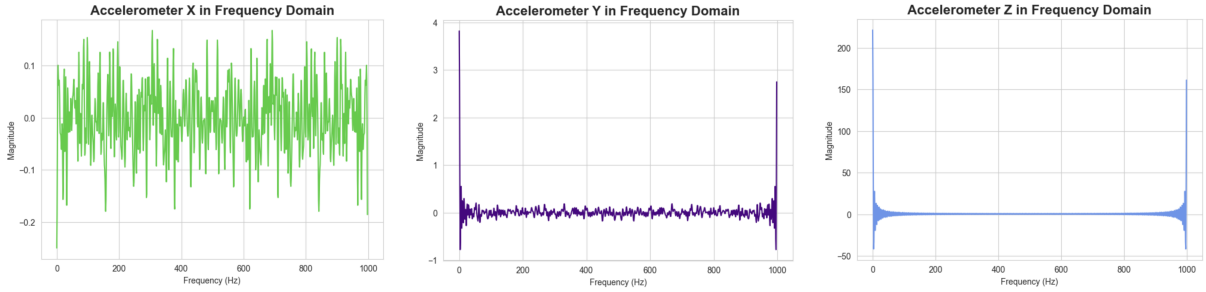
The results that can be drawn from the overall system functionality would be the retention of Fourier co-efficients which corresponds to R5, S5 and A5 in table 6.3 below. This is tested by analysing the pre-compressed/encrypted data against the post-decompressed/decrypted data in the frequency domain. Because the algorithms used are lossless, the data is exactly the same and therefore its frequency domain data will be the same thus the Fourier co-efficients are retained.

To illustrate this, the accelerometer data can be examined (but this would be applicable to all data obtained from the sensor). The frequency domain plots of the pre-encrypted and pre-compressed (i.e. original) accelerometer data is shown in figure 6.11 below and the frequency domain plots of the decrypted and decompressed (i.e. transmitted) accelerometer data is shown in figure 6.12 below:



(a) x component of acceleration (b) y component of acceleration (c) z component of acceleration

Figure 6.11: Accelerometer Frequency Domain of Original Data



(a) x component of acceleration (b) y component of acceleration (c) z component of acceleration

Figure 6.12: Accelerometer Frequency Domain of Transmitted Data

It can be seen from figures 6.11 and 6.12 above that the frequency domain representations are unchanged and therefore all Fourier Co-efficients are retained. Thus, the lower 25% of the Fourier coefficients are retained which meets A5 in table 6.3 below.

6.3.2 Compression Block

Data pertaining to the compression block of the experiment from table 6.2 above is discussed and where relevant plotted below.

A dataset size of 176 characters results in a compressed dataset size of 1062 characters. This is not a reduced size and thus A3.1 in table 6.3 below is not met. This can be attributed to the small dataset sizes and reduced efficiency of the compression algorithm. Figure 6.13 below shows how the runtime of the compression algorithm changes over varying dataset sizes.

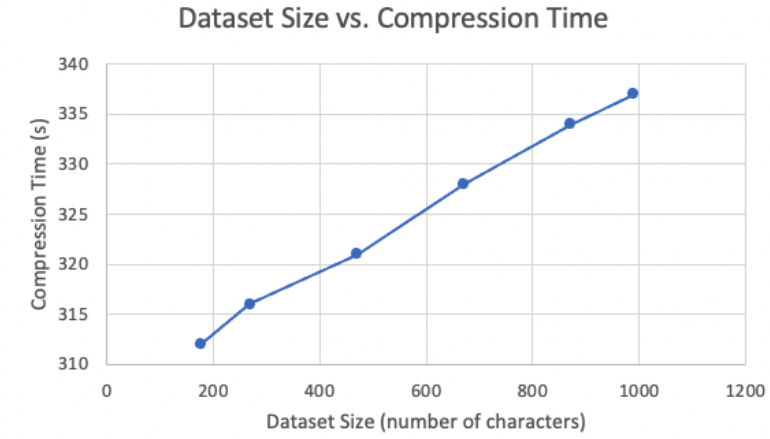


Figure 6.13: Graph showing the time required to compress varying sizes of input data.

It can concluded from figure 6.13 above that the LZSS compression algorithm implemented in this particular setup has a time complexity of order n [O(n)] which meets A6 in table 6.3 below. This is to be expected from the LZSS compression library.

Figure 6.14 below shows the change in compression ratio over varying dataset sizes.

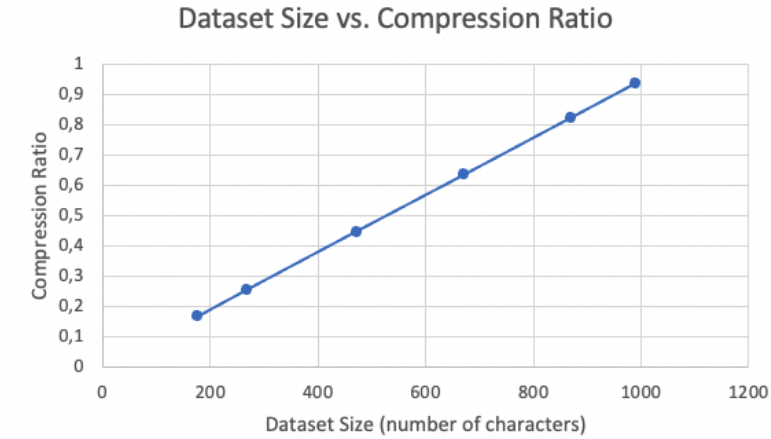


Figure 6.14: Graph showing the compression ratio over varying sizes of input data.

It can be seen that the compression ratio increases as the dataset size increases. At smaller dataset sizes the compression ratio is insufficient and does not meet the project's specifications. However, as the dataset size exceeds about 471 characters the compression ratio achieves a desirable value. This can again be attributed to the constraints of the STM32F051 microcontroller. In use, the system should be required to intake data for a minimum of 2 seconds as this will ensure an adequate compression ration (i.e. about 40%) - therefore A3.2 in table 6.3 below is considered to be met.

The comparison algorithm as stated in Appendix B.3 outputted a percentage of 100% as shown by the output in figure 6.15 below. This suggests that there is no difference between the original data and the decompressed data.

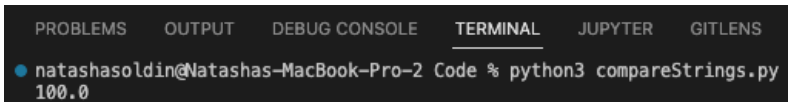


Figure 6.15: Output of the Comparison Algorithm for Decompressed Data

This further infers that the Fourier transforms of the two datasets will be the same and therefore the first 25% of the Fourier coefficients were retained successfully. This is illustrated in figures 6.11 and 6.12 above.

6.3.3 Encryption Block

Data pertaining to the encryption block of the experiment from table 6.2 above is discussed and where relevant plotted below.

The ASCII shift encryption algorithm in C operates as expected which meets A2 in table 6.3 below. Figure 6.16 below shows how the runtime of the encryption algorithm changes over varying dataset sizes.

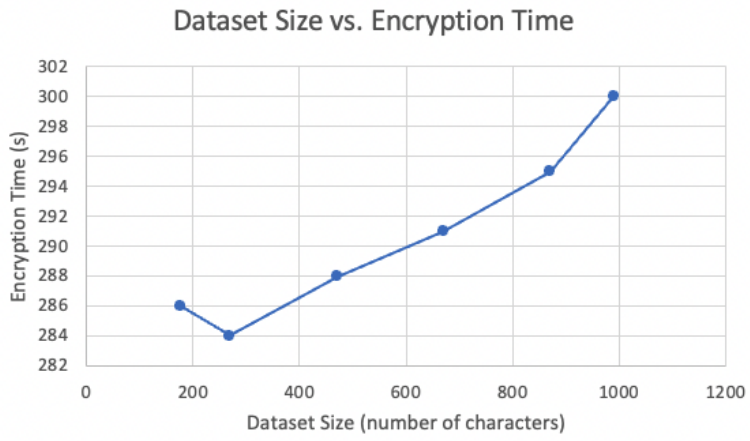


Figure 6.16: Graph showing the time required to encrypt varying sizes of input data.

It can concluded from figure 6.16 above that the ASCII shift encryption algorithm implemented in this particular setup has a time complexity of order n [O(n)] which meets A6 in table 6.3 below.

The comparison algorithm as stated in Appendix B.3 outputted a percentage of 15.37% as shown by the output in figure 6.17 below. This suggest that there is adequate difference between the compressed (pre-encrypted) data and the encrypted data which meets A4.1 in table 6.3 below. This similarity percentage is less than in the simulated based Experiment Setup which is to be expected as a result from downgrade of the encryption algorithm from symmetric encryption to an ASCII shift encryption method.

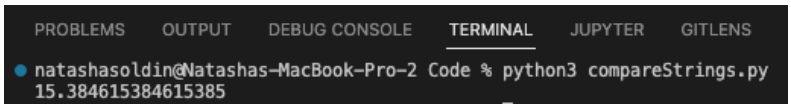


Figure 6.17: Output of the Comparison Algorithm for Encrypted Data

The comparison algorithm as stated in Appendix B.3 outputted a percentage of 100% as shown by the output in figure 6.18 below. This suggest that there is no difference between the compressed (pre-encrypted) data and the decrypted data which meets A4.2 in table 6.3 below.

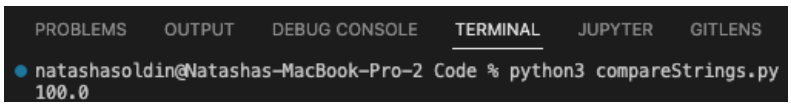


Figure 6.18: Output of the Comparison Algorithm for Decrypted Data

6.4 Acceptance Test Protocols

Due to the use of different compression and encryption algorithms, changes needed to be made to the requirement, specification and corresponding ATPs. Those pertaining to the compression algorithm were left unchanged as the algorithm is still of dictionary compression type but those pertaining to the encryption algorithm were changed. The updates made to the ATP table 3.1 above can be seen in new ATP table 6.3 below:

Requirements	**Specifications**	**Acceptance Test Protocols**
R1	S1	A1
The project should be designed to utilize the ICM-20649 IMU sensor even though that sensor is not available to use during the project's planning stage.	The design should adhere to the electrical specifications of the ICM-20649 IMU.	Ensure that the data obtained from the sensor hat closely follows the structure of the sample data provided as this comes from the design's intended sensor (ICM-20649 IMU).
R2	S2	A2
The project should be designed to utilize the STM32F051 microcontroller.	Both algorithms needs to be coded in C/C++ in STM32CUBEIDE to be compatible with the STM32F051 microcontroller.	Run the algorithms in a C/C++ IDE as well as on the STM32F051 to test that it works.
R3	S3.1	A3.1
The data obtained from the IMU sensor should be compressed to reduce the cost of transmission because the transmission of data using Iridium is extremely costly.	The compression and decompression of the IMU data will be done using a dictionary compression method.	Check that the file size of the compressed data is considerably less than the file size of the original data.
	S3.2	A3.2
	The compression ratio should be between 40% and 60%	Calculate the compression ratio between the uncompressed and compressed data.
R4	S4	A4
The data obtained from the IMU sensor should be encrypted to increase security of the data.	The encryption and decryption of the IMU data will be done using an ASCII shifting encryption method.	Compare the encrypted data to the un-encrypted data to ensure minimal similarities which indicates reasonable level of encryption.
R5	S5	A5
Minimal loss/ damage should apply to the decrypted and decompressed data.	The decrypted and decompressed data should reflect 25% of the Fourier coefficients of the original IMU data.	The fourier transform of the decrypted and decompressed data should be compared to the fourier transform of the original data to test if 25% of the lower fourier co-efficients have been retained.
R6	S6	A6
Limit power consumption by reducing the amount of processing done in the processor and minimizing the computation required.	The C/C++ programs should be of time complexity order n [O(n)] to reduce the amount of processing done and thus the power consumption.	Test the time complexity of the compression and encryption programs by running multiple tests with different dataset sizes and plot the correlation.

Table 6.3: Requirements, Specifications and ATPs Updated

From the results in section 6.3 above, the ATPs can be tested to determine whether they have been met or not. The first ATP (A1) is not tested in the Experiment Setup like the other ATPs (A2 - A6). A1 is tested in the IMU Module section within the IMU Validation Tests subsection. This is tabulated in table 6.4 below:

ATP Number	**Has the ATP been met?**	**Reason**
A1	✓	The data obtained from the IMU closely follows the sample data provided and is validated as 'correct' IMU data in the validation tests of the IMU Module in section 6.1 above.
A2	✓	The algorithms were coded in C/C++ to be compatible with the STM32F051 micrcontroller.
A3.1	×	The compression algorithm's efficiency was hindered by the smaller dataset sizes resulting in inadequate dataset size reduction.
A3.2	✓	The compression ratio was above 40% granted that the data intake time was above 2 as seen in figure 6.14. This is adequate as the project required a compression ratio between 40% and 60%.
A4	✓	The similarity between the pre-encrypted data and post-encypted data is 15,38% as seen in figure 6.17 above which is less than 20%. This is enough to consider the encryption to be of adequate randomisation.
A5	✓	The lower 25% of the fourier co-efficients have been retained when comparing the fourier transform of the original data to the decrypted and decompressed data as seen in figures 6.11 and 6.12 above.
A6	✓	The time complexity of the algorithms is O(n) as seen in figures 6.13 and 6.16 above.

Table 6.4: ATP Meeting and Reasoning

The ATPs and corresponding specification were not changed - however one was omitted due to the encryption algorithm change. All ATPs were tested and most were met in this implementation of the project. However A3.1 was not met due to the reduced efficiency of the compression algorithm as a result of the small dataset sizes taken.

This is not the fault of the compression algorithm implementation, as if larger dataset were taken, the compressed data would be reduced in size. This is solely due to the constraints placed on the students as a result of the use of an STM32F051 microcontroller and will be addressed in Future Work.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The project laid out in this report included:

- A Requirement Analysis: where requirement, specifications and acceptance test protocols are derived from the project's description.
- A Paper Design: where a theoretical version of the project was proposed by exploring the background, feasibility, possible bottlenecks, available algorithms choices, subsystem design and experimental design.
- A Validation using Simulated Data: or simulated implementation of the project where sample data is used in a data analysis, simulated experiment, results and re-evaluated acceptance test procedures.
- A Validation using IMU Data: or practical implementation of the project where data is retrieved from the IMU is used in a IMU module for data validation, practical experiment, results and re-evaluated acceptance test procedures.

In conclusion, the project was completed in the specified time and meets most of the user requirements specified in the project's description. Although A3.1 was not met, this was not a fault of the code but instead the restrained implementation of the project. The compression ratio is still adequate if a reasonable dataset sizes is taken. Therefore, the project was considered successful and will be further developed in future implementations.

7.2 Future Work

The project was implemented such that the base Requirements are met. However, due to the subsystem nature of this design there are further Subsystem and Sub-subsystems Requirements pertaining to the project that were not considered in this initial project implementation due to it's time constrained duration. In future work, these subsystem requirements and specification as specified in section 4.3.2 and 4.3.3 above should be considered and the project should be altered to meet.

The main issues facing the project were due to the use of the STM32F051 microcontroller and thus C programming language. Both placed strict constraints on what was possible to achieve in the project and greatly limited the students. As a result the compressed data was larger in size than the original data.

In future work, the students will attempt to optimise the existing program (which was not possible in this project's duration due to time constraints) but ideally will choose to use a different microcontroller for example a raspberry pi. This would allow the students to implement the python programs as explained in the Validation using Simulated Data section above and removes the memory space constraints. This would allow for: both the compression and encryption methods to be adequately advanced and for reasonable dataset sizes to be taken from the sensor. With this it is hypothesised that acceptance test procedure A3.1 would also be met.

Bibliography

[1] "The importance of imu motion sensors," Sep 2019.

[2] P. Asghari, A. M. Rahmani, and H. H. S. Javadi, "Internet of things applications: A systematic review," *Computer Networks*, vol. 148, pp. 241–261, 2019.

[3] G. Chiarot and C. Silvestri, "Time series compression: a survey," *ArXiv*, vol. abs/2101.08784, 2021.

[4] S. Deb, "How to perform data compression using autoencoders?," Sep 2020.

[5] SSL2BUY Wiki, "Symmetric vs. asymmetric encryption - what are differences?." `https://www.ssl2buy.com/wiki/symmetric-vs-asymmetric-encryption-what-are-differences`, 2021.

[6] Tutorials Point, "Difference between AES and DES ciphers." `https://www.tutorialspoint.com/difference-between-aes-and-des-ciphers`, 2021.

[7] M. N. A. Wahid, A. Ali, B. Esparham, and M. Marwan, "A comparison of cryptographic algorithms: Des, 3des, aes, rsa and blowfish for guessing attacks prevention," *Journal of Computer Science Applications and Information Technology*, 2018.

[8] J. Lake, "What is 3des encryption and how does des work?," Feb 2022.

[9] I2C-Bus, "Speed."

[10] S. Mishra, N. K. Singh, and V. Rousseau, "Chapter 10 - sensor interfaces," in *System on Chip Interfaces for Low Power Design* (S. Mishra, N. K. Singh, and V. Rousseau, eds.), pp. 331–344, Morgan Kaufmann, 2016.

[11] A. Ahmad, "Encryption and compression of data," Feb 1960. `https://security.stackexchange.com/questions/19969/encryption-and-compression-of-data`.

[12] "Lz4 compression library bindings for python."

[13] "Simple aes-ctr example." `https://cryptobook.nakov.com/symmetric-key-ciphers/aes-encrypt-decrypt-examples`.

[14] user9323924user9323924, "How to compare and count the differences between two files line by line?," Aug 1965. `https://stackoverflow.com/questions/49659668/how-to-compare-and-count-the-differences-between-two-files-line-by-line`.

[15] "How do i get time of a python program's execution?." https://stackoverflow.com/questions/1557571/how-do-i-get-time-of-a-python-programs-execution/12344609#12344609, journal=Stack Overflow, author=PaulMc, year=1957, month=Feb.

[16] "Difference between c and python," Jun 2022. https://www.interviewbit.com/blog/difference-between-c-and-python/.

[17] Atomicobject, "Atomicobject/heatshrink: Data compression library for embedded/real-time systems." https://github.com/atomicobject/heatshrink.

[18] "C program to encrypt and decrypt the string (source code)." http://www.trytoprogram.com/c-examples/c-program-to-encrypt-and-decrypt-string/.

[19] H. Jain, "What is sequencematcher() in python?," Oct 2022. https://www.educative.io/answers/what-is-sequencematcher-in-python.

Appendix A

Simulated Algorithms

A.1 Compression Python Algorithm

The following code was used to implement dictionary compression in python [12]:

A.1.1 Compress Method

```
def compress(data):
    # Convert imported csv data to bytes format.
    data_as_bytes = str.encode(data)

    # Compress the data.
    Timing.startlog()
    compressed = lz4.frame.compress(data_as_bytes)
    global iTimeToCompress
    iTimeToCompress = Timing.endlog()

    # Get the sizes of the data, compressed and not compressed.
    global iSizeOriginal
    iSizeOriginal = getsizeof(data_as_bytes)

    global iSizeCompressed
    iSizeCompressed = getsizeof(compressed)

    global iCompressionRatio
    iCompressionRatio = iSizeOriginal/iSizeCompressed

    # Write the binary compressed data to a file.
    with open('compressed_data.txt', 'wb') as f:
        f.write(compressed)
```

A.1.2 Decompress Method

```
def decompress(data):
    # Decompress the data.
    Timing.startlog()
    global decompressed
    decompressed = lz4.frame.decompress(data)
    global iTimeToDecompress
    iTimeToDecompress = Timing.endlog()

```

```
    #Convert back from byte array.
    decompressed = decompressed.decode()

    # Write decompressed data to output file.
    with open('decompressed_decrypted_data.csv', 'w') as f:
        f.write(decompressed)
```

A.2 Encryption Python Algorithm

The following code was used to implement AES encryption in python [13]:

A.2.1 Encrypt Method

```
def encrypt(data):
    # Derive a 256-bit AES encryption key from the password
    password = "hello"
    passwordSalt = os.urandom(16)
    global key
    key = pbkdf2.PBKDF2(password, passwordSalt).read(32)

    # Encrypt the plaintext with the given key:
    #   ciphertext = AES-256-CTR-Encrypt(plaintext, key, iv)
    global iv
    iv = secrets.randbits(256)
    aes = pyaes.AESModeOfOperationCTR(key, pyaes.Counter(iv))

    # Encrypt the data.
    Timing.startlog()
    ciphertext = aes.encrypt(data)
    global iTimeToEncrypt
    iTimeToEncrypt = Timing.endlog()

    # Write the binary compressed encrypted data to a file.
    with open('compressed_encrypted_data.txt', 'wb') as f:
        f.write(ciphertext)
```

A.2.2 Decrypt Method

```
def decrypt(data):


    # Decrypt the ciphertext with the given key:
    #   plaintext = AES-256-CTR-Decrypt(ciphertext, key, iv)
    aes = pyaes.AESModeOfOperationCTR(key, pyaes.Counter(iv))

    # Decrypt the data.
    Timing.startlog()
    decrypted = aes.decrypt(data)
    global iTimeToDecrypt
    iTimeToDecrypt = Timing.endlog()

    # Write the binary compressed decrypted data to a file.
    with open('compressed_decrypted_data.txt', 'wb') as f:
        f.write(decrypted)
```

A.3 File Comparison Algorithm

The following code was used to implement data comparison of files in python [14]

```
count = 0
total = 0
file1name = "data.csv"
file2name = "decompressed_decrypted_data.csv"

with open(file1name) as file1, open(file2name) as file2:
    for line_file_1, line_file_2 in zip(file1, file2):
        total += 1
        if line_file_1 != line_file_2:
            count += 1

percentage = ((total-count)/total)*100
print(percentage)
```

A.4 Timing Python Algorithm

The following code was used to implement relative timing measurement in python [15]:

```
"""
Python Practical 2 Code for Timing
Keegan Crankshaw
EEE3096S Prac 2 2019
Date: 7 June 2019
Adapted from Paul McGuire's answer on Stack Overflow
https://stackoverflow.com/questions/1557571/how-do-i-get-time-of-a-python-programs-execution
    /12344609#12344609
"""

from time import time, strftime, localtime
from datetime import timedelta

start = ''

def secondsToStr(elapsed=None):
    if elapsed is None:
        return strftime("%Y-%m-%d %H:%M:%S", localtime())
    else:
        return str(timedelta(seconds=elapsed))

def startlog():
    global start
    start = time()
    #log("Starting log")


def log(s, elapsed=None):
    #line = "="*40
    #print(line)
    #print(secondsToStr(), '-', s)
    if elapsed:
        print("Elapsed time:", elapsed)
    #print(line)

def endlog():
    global start
    end = time()
    elapsed = end-start
    #log("End Timing", secondsToStr(elapsed))
    return str(elapsed)
```

Appendix B

Practical Algorithms

B.1 Compression C Algorithm

The following code was used to implement dictionary compression in C [17]:

B.1.1 compression method in main.c

```
1  void compress(uint8_t *input, uint32_t input_size, uint8_t *compressed) {
2      static heatshrink_encoder hse;
3
4      heatshrink_encoder_reset(&hse);
5
6      size_t comp_sz = input_size + (input_size/2) + 4;
7      uint8_t *comp = calloc(input_size, sizeof(uint8_t));
8
9      size_t count = 0;
10     uint32_t sunk = 0;
11     uint32_t polled = 0;
12
13
14     while (sunk < input_size) {
15         heatshrink_encoder_sink(&hse, &input[sunk], input_size - sunk, &count) >= 0;
16         sunk += count;
17
18         if (sunk == input_size) {
19             heatshrink_encoder_finish(&hse);
20         }
21
22         HSE_poll_res pres;
23         do {                    /* "turn the crank" */
24             pres = heatshrink_encoder_poll(&hse, &comp[polled], comp_sz - polled, &count);
25             polled += count;
26
27         } while (pres == HSER_POLL_MORE);
28         if (sunk == input_size) {
29             heatshrink_encoder_finish(&hse);
30         }
31     }
32
33     //memmove(compressed, &polled, sizeof(uint32_t));
34     memcpy(compressed, comp, polled);
35
36
37     free(comp);
38 }
```

B.1.2 heatshrink_encoder.c

```
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include "heatshrink_encoder.h"

typedef enum {
    HSES_NOT_FULL,              /* input buffer not full enough */
    HSES_FILLED,                /* buffer is full */
    HSES_SEARCH,                /* searching for patterns */
    HSES_YIELD_TAG_BIT,         /* yield tag bit */
    HSES_YIELD_LITERAL,         /* emit literal byte */
    HSES_YIELD_BR_INDEX,        /* yielding backref index */
    HSES_YIELD_BR_LENGTH,       /* yielding backref length */
    HSES_SAVE_BACKLOG,          /* copying buffer to backlog */
    HSES_FLUSH_BITS,            /* flush bit buffer */
    HSES_DONE,                  /* done */
} HSE_state;

#if HEATSHRINK_DEBUGGING_LOGS
#include <stdio.h>
#include <ctype.h>
#include <assert.h>
#define LOG(...) fprintf(stderr, __VA_ARGS__)
#define ASSERT(X) assert(X)
static const char *state_names[] = {
    "not_full",
    "filled",
    "search",
    "yield_tag_bit",
    "yield_literal",
    "yield_br_index",
    "yield_br_length",
    "save_backlog",
    "flush_bits",
    "done",
};
#else
#define LOG(...) /* no-op */
#define ASSERT(X) /* no-op */
#endif

// Encoder flags
enum {
    FLAG_IS_FINISHING = 0x01,
};

typedef struct {
    uint8_t *buf;               /* output buffer */
    size_t buf_size;            /* buffer size */
    size_t *output_size;        /* bytes pushed to buffer, so far */
} output_info;

#define MATCH_NOT_FOUND ((uint16_t)-1)

static uint16_t get_input_offset(heatshrink_encoder *hse);
static uint16_t get_input_buffer_size(heatshrink_encoder *hse);
static uint16_t get_lookahead_size(heatshrink_encoder *hse);
static void add_tag_bit(heatshrink_encoder *hse, output_info *oi, uint8_t tag);
static int can_take_byte(output_info *oi);
static int is_finishing(heatshrink_encoder *hse);
static void save_backlog(heatshrink_encoder *hse);

/* Push COUNT (max 8) bits to the output buffer, which has room. */
static void push_bits(heatshrink_encoder *hse, uint8_t count, uint8_t bits,
    output_info *oi);
static uint8_t push_outgoing_bits(heatshrink_encoder *hse, output_info *oi);
static void push_literal_byte(heatshrink_encoder *hse, output_info *oi);

#if HEATSHRINK_DYNAMIC_ALLOC
heatshrink_encoder *heatshrink_encoder_alloc(uint8_t window_sz2,
        uint8_t lookahead_sz2) {
```

```
72      if ((window_sz2 < HEATSHRINK_MIN_WINDOW_BITS) ||
73          (window_sz2 > HEATSHRINK_MAX_WINDOW_BITS) ||
74          (lookahead_sz2 < HEATSHRINK_MIN_LOOKAHEAD_BITS) ||
75          (lookahead_sz2 >= window_sz2)) {
76          return NULL;
77      }
78
79      /* Note: 2 * the window size is used because the buffer needs to fit
80       * (1 << window_sz2) bytes for the current input, and an additional
81       * (1 << window_sz2) bytes for the previous buffer of input, which
82       * will be scanned for useful backreferences. */
83      size_t buf_sz = (2 << window_sz2);
84
85      heatshrink_encoder *hse = HEATSHRINK_MALLOC(sizeof(*hse) + buf_sz);
86      if (hse == NULL) { return NULL; }
87      hse->window_sz2 = window_sz2;
88      hse->lookahead_sz2 = lookahead_sz2;
89      heatshrink_encoder_reset(hse);
90
91  #if HEATSHRINK_USE_INDEX
92      size_t index_sz = buf_sz*sizeof(uint16_t);
93      hse->search_index = HEATSHRINK_MALLOC(index_sz + sizeof(struct hs_index));
94      if (hse->search_index == NULL) {
95          HEATSHRINK_FREE(hse, sizeof(*hse) + buf_sz);
96          return NULL;
97      }
98      hse->search_index->size = index_sz;
99  #endif
100
101     LOG("-- allocated encoder with buffer size of %zu (%u byte input size)\n",
102         buf_sz, get_input_buffer_size(hse));
103     return hse;
104 }
105
106 void heatshrink_encoder_free(heatshrink_encoder *hse) {
107     size_t buf_sz = (2 << HEATSHRINK_ENCODER_WINDOW_BITS(hse));
108 #if HEATSHRINK_USE_INDEX
109     size_t index_sz = sizeof(struct hs_index) + hse->search_index->size;
110     HEATSHRINK_FREE(hse->search_index, index_sz);
111     (void)index_sz;
112 #endif
113     HEATSHRINK_FREE(hse, sizeof(heatshrink_encoder) + buf_sz);
114     (void)buf_sz;
115 }
116 #endif
117
118 void heatshrink_encoder_reset(heatshrink_encoder *hse) {
119     size_t buf_sz = (2 << HEATSHRINK_ENCODER_WINDOW_BITS(hse));
120     memset(hse->buffer, 0, buf_sz);
121     hse->input_size = 0;
122     hse->state = HSES_NOT_FULL;
123     hse->match_scan_index = 0;
124     hse->flags = 0;
125     hse->bit_index = 0x80;
126     hse->current_byte = 0x00;
127     hse->match_length = 0;
128
129     hse->outgoing_bits = 0x0000;
130     hse->outgoing_bits_count = 0;
131
132     #ifdef LOOP_DETECT
133     hse->loop_detect = (uint32_t)-1;
134     #endif
135 }
136
137 HSE_sink_res heatshrink_encoder_sink(heatshrink_encoder *hse,
138         uint8_t *in_buf, size_t size, size_t *input_size) {
139     if ((hse == NULL) || (in_buf == NULL) || (input_size == NULL)) {
140         return HSER_SINK_ERROR_NULL;
141     }
142
143     /* Sinking more content after saying the content is done, tsk tsk */
144     if (is_finishing(hse)) { return HSER_SINK_ERROR_MISUSE; }
```

```
145
146     /* Sinking more content before processing is done */
147     if (hse->state != HSES_NOT_FULL) { return HSER_SINK_ERROR_MISUSE; }
148
149     uint16_t write_offset = get_input_offset(hse) + hse->input_size;
150     uint16_t ibs = get_input_buffer_size(hse);
151     uint16_t rem = ibs - hse->input_size;
152     uint16_t cp_sz = rem < size ? rem : size;
153
154     memcpy(&hse->buffer[write_offset], in_buf, cp_sz);
155     *input_size = cp_sz;
156     hse->input_size += cp_sz;
157
158     LOG("-- sunk %u bytes (of %zu) into encoder at %d, input buffer now has %u\n",
159         cp_sz, size, write_offset, hse->input_size);
160     if (cp_sz == rem) {
161         LOG("-- internal buffer is now full\n");
162         hse->state = HSES_FILLED;
163     }
164
165     return HSER_SINK_OK;
166 }
167
168
169 /***************
170  * Compression *
171  ***************/
172
173 static uint16_t find_longest_match(heatshrink_encoder *hse, uint16_t start,
174     uint16_t end, const uint16_t maxlen, uint16_t *match_length);
175 static void do_indexing(heatshrink_encoder *hse);
176
177 static HSE_state st_step_search(heatshrink_encoder *hse);
178 static HSE_state st_yield_tag_bit(heatshrink_encoder *hse,
179     output_info *oi);
180 static HSE_state st_yield_literal(heatshrink_encoder *hse,
181     output_info *oi);
182 static HSE_state st_yield_br_index(heatshrink_encoder *hse,
183     output_info *oi);
184 static HSE_state st_yield_br_length(heatshrink_encoder *hse,
185     output_info *oi);
186 static HSE_state st_save_backlog(heatshrink_encoder *hse);
187 static HSE_state st_flush_bit_buffer(heatshrink_encoder *hse,
188     output_info *oi);
189
190 HSE_poll_res heatshrink_encoder_poll(heatshrink_encoder *hse,
191         uint8_t *out_buf, size_t out_buf_size, size_t *output_size) {
192     if ((hse == NULL) || (out_buf == NULL) || (output_size == NULL)) {
193         return HSER_POLL_ERROR_NULL;
194     }
195     if (out_buf_size == 0) {
196         LOG("-- MISUSE: output buffer size is 0\n");
197         return HSER_POLL_ERROR_MISUSE;
198     }
199     *output_size = 0;
200
201     output_info oi;
202     oi.buf = out_buf;
203     oi.buf_size = out_buf_size;
204     oi.output_size = output_size;
205
206     while (1) {
207         LOG("-- polling, state %u (%s), flags 0x%02x\n",
208             hse->state, state_names[hse->state], hse->flags);
209
210         uint8_t in_state = hse->state;
211         switch (in_state) {
212         case HSES_NOT_FULL:
213             return HSER_POLL_EMPTY;
214         case HSES_FILLED:
215             do_indexing(hse);
216             hse->state = HSES_SEARCH;
217             break;
```

```
218             case HSES_SEARCH:
219                 hse->state = st_step_search(hse);
220                 break;
221             case HSES_YIELD_TAG_BIT:
222                 hse->state = st_yield_tag_bit(hse, &oi);
223                 break;
224             case HSES_YIELD_LITERAL:
225                 hse->state = st_yield_literal(hse, &oi);
226                 break;
227             case HSES_YIELD_BR_INDEX:
228                 hse->state = st_yield_br_index(hse, &oi);
229                 break;
230             case HSES_YIELD_BR_LENGTH:
231                 hse->state = st_yield_br_length(hse, &oi);
232                 break;
233             case HSES_SAVE_BACKLOG:
234                 hse->state = st_save_backlog(hse);
235                 break;
236             case HSES_FLUSH_BITS:
237                 hse->state = st_flush_bit_buffer(hse, &oi);
238             case HSES_DONE:
239                 return HSER_POLL_EMPTY;
240             default:
241                 LOG("-- bad state %s\n", state_names[hse->state]);
242                 return HSER_POLL_ERROR_MISUSE;
243             }
244 
245             if (hse->state == in_state) {
246                 /* Check if output buffer is exhausted. */
247                 if (*output_size == out_buf_size) return HSER_POLL_MORE;
248             }
249         }
250     }
251 
252     HSE_finish_res heatshrink_encoder_finish(heatshrink_encoder *hse) {
253         if (hse == NULL) { return HSER_FINISH_ERROR_NULL; }
254         LOG("-- setting is_finishing flag\n");
255         hse->flags |= FLAG_IS_FINISHING;
256         if (hse->state == HSES_NOT_FULL) { hse->state = HSES_FILLED; }
257         return hse->state == HSES_DONE ? HSER_FINISH_DONE : HSER_FINISH_MORE;
258     }
259 
260     static HSE_state st_step_search(heatshrink_encoder *hse) {
261         uint16_t window_length = get_input_buffer_size(hse);
262         uint16_t lookahead_sz = get_lookahead_size(hse);
263         uint16_t msi = hse->match_scan_index;
264         LOG("## step_search, scan @ +%d (%d/%d), input size %d\n",
265             msi, hse->input_size + msi, 2*window_length, hse->input_size);
266 
267         bool fin = is_finishing(hse);
268         if (msi > hse->input_size - (fin ? 1 : lookahead_sz)) {
269             /* Current search buffer is exhausted, copy it into the
270              * backlog and await more input. */
271             LOG("-- end of search @ %d\n", msi);
272             return fin ? HSES_FLUSH_BITS : HSES_SAVE_BACKLOG;
273         }
274 
275         uint16_t input_offset = get_input_offset(hse);
276         uint16_t end = input_offset + msi;
277         uint16_t start = end - window_length;
278 
279         uint16_t max_possible = lookahead_sz;
280         if (hse->input_size - msi < lookahead_sz) {
281             max_possible = hse->input_size - msi;
282         }
283 
284         uint16_t match_length = 0;
285         uint16_t match_pos = find_longest_match(hse,
286             start, end, max_possible, &match_length);
287 
288         if (match_pos == MATCH_NOT_FOUND) {
289             LOG("ss Match not found\n");
290             hse->match_scan_index++;
```

```
291             hse->match_length = 0;
292             return HSES_YIELD_TAG_BIT;
293         } else {
294             LOG("ss Found match of %d bytes at %d\n", match_length, match_pos);
295             hse->match_pos = match_pos;
296             hse->match_length = match_length;
297             ASSERT(match_pos <= 1 << HEATSHRINK_ENCODER_WINDOW_BITS(hse) /*window_length*/);
298 
299             return HSES_YIELD_TAG_BIT;
300         }
301     }
302 
303     static HSE_state st_yield_tag_bit(heatshrink_encoder *hse,
304             output_info *oi) {
305         if (can_take_byte(oi)) {
306             if (hse->match_length == 0) {
307                 add_tag_bit(hse, oi, HEATSHRINK_LITERAL_MARKER);
308                 return HSES_YIELD_LITERAL;
309             } else {
310                 add_tag_bit(hse, oi, HEATSHRINK_BACKREF_MARKER);
311                 hse->outgoing_bits = hse->match_pos - 1;
312                 hse->outgoing_bits_count = HEATSHRINK_ENCODER_WINDOW_BITS(hse);
313                 return HSES_YIELD_BR_INDEX;
314             }
315         } else {
316             return HSES_YIELD_TAG_BIT; /* output is full, continue */
317         }
318     }
319 
320     static HSE_state st_yield_literal(heatshrink_encoder *hse,
321             output_info *oi) {
322         if (can_take_byte(oi)) {
323             push_literal_byte(hse, oi);
324             return HSES_SEARCH;
325         } else {
326             return HSES_YIELD_LITERAL;
327         }
328     }
329 
330     static HSE_state st_yield_br_index(heatshrink_encoder *hse,
331             output_info *oi) {
332         if (can_take_byte(oi)) {
333             LOG("-- yielding backref index %u\n", hse->match_pos);
334             if (push_outgoing_bits(hse, oi) > 0) {
335                 return HSES_YIELD_BR_INDEX; /* continue */
336             } else {
337                 hse->outgoing_bits = hse->match_length - 1;
338                 hse->outgoing_bits_count = HEATSHRINK_ENCODER_LOOKAHEAD_BITS(hse);
339                 return HSES_YIELD_BR_LENGTH; /* done */
340             }
341         } else {
342             return HSES_YIELD_BR_INDEX; /* continue */
343         }
344     }
345 
346     static HSE_state st_yield_br_length(heatshrink_encoder *hse,
347             output_info *oi) {
348         if (can_take_byte(oi)) {
349             LOG("-- yielding backref length %u\n", hse->match_length);
350             if (push_outgoing_bits(hse, oi) > 0) {
351                 return HSES_YIELD_BR_LENGTH;
352             } else {
353                 hse->match_scan_index += hse->match_length;
354                 hse->match_length = 0;
355                 return HSES_SEARCH;
356             }
357         } else {
358             return HSES_YIELD_BR_LENGTH;
359         }
360     }
361 
362     static HSE_state st_save_backlog(heatshrink_encoder *hse) {
363         LOG("-- saving backlog\n");
```

```
364     save_backlog(hse);
365     return HSES_NOT_FULL;
366 }
367
368 static HSE_state st_flush_bit_buffer(heatshrink_encoder *hse,
369         output_info *oi) {
370     if (hse->bit_index == 0x80) {
371         LOG("-- done!\n");
372         return HSES_DONE;
373     } else if (can_take_byte(oi)) {
374         LOG("-- flushing remaining byte (bit_index == 0x%02x)\n", hse->bit_index);
375         oi->buf[(*oi->output_size)++] = hse->current_byte;
376         LOG("-- done!\n");
377         return HSES_DONE;
378     } else {
379         return HSES_FLUSH_BITS;
380     }
381 }
382
383 static void add_tag_bit(heatshrink_encoder *hse, output_info *oi, uint8_t tag) {
384     LOG("-- adding tag bit: %d\n", tag);
385     push_bits(hse, 1, tag, oi);
386 }
387
388 static uint16_t get_input_offset(heatshrink_encoder *hse) {
389     return get_input_buffer_size(hse);
390 }
391
392 static uint16_t get_input_buffer_size(heatshrink_encoder *hse) {
393     return (1 << HEATSHRINK_ENCODER_WINDOW_BITS(hse));
394     (void)hse;
395 }
396
397 static uint16_t get_lookahead_size(heatshrink_encoder *hse) {
398     return (1 << HEATSHRINK_ENCODER_LOOKAHEAD_BITS(hse));
399     (void)hse;
400 }
401
402 static void do_indexing(heatshrink_encoder *hse) {
403 #if HEATSHRINK_USE_INDEX
404     /* Build an index array I that contains flattened linked lists
405      * for the previous instances of every byte in the buffer.
406      *
407      * For example, if buf[200] == 'x', then index[200] will either
408      * be an offset i such that buf[i] == 'x', or a negative offset
409      * to indicate end-of-list. This significantly speeds up matching,
410      * while only using sizeof(uint16_t)*sizeof(buffer) bytes of RAM.
411      *
412      * Future optimization options:
413      * 1. Since any negative value represents end-of-list, the other
414      *    15 bits could be used to improve the index dynamically.
415      *
416      * 2. Likewise, the last lookahead_sz bytes of the index will
417      *    not be usable, so temporary data could be stored there to
418      *    dynamically improve the index.
419      * */
420     struct hs_index *hsi = HEATSHRINK_ENCODER_INDEX(hse);
421     int16_t last[256];
422     memset(last, 0xFF, sizeof(last));
423
424     uint8_t * const data = hse->buffer;
425     int16_t * const index = hsi->index;
426
427     const uint16_t input_offset = get_input_offset(hse);
428     const uint16_t end = input_offset + hse->input_size;
429
430     for (uint16_t i=0; i<end; i++) {
431         uint8_t v = data[i];
432         int16_t lv = last[v];
433         index[i] = lv;
434         last[v] = i;
435     }
436 #else
```

```
437     (void)hse;
438 #endif
439 }
440
441 static int is_finishing(heatshrink_encoder *hse) {
442     return hse->flags & FLAG_IS_FINISHING;
443 }
444
445 static int can_take_byte(output_info *oi) {
446     return *oi->output_size < oi->buf_size;
447 }
448
449 /* Return the longest match for the bytes at buf[end:end+maxlen] between
450  * buf[start] and buf[end-1]. If no match is found, return -1. */
451 static uint16_t find_longest_match(heatshrink_encoder *hse, uint16_t start,
452         uint16_t end, const uint16_t maxlen, uint16_t *match_length) {
453     LOG("-- scanning for match of buf[%u:%u] between buf[%u:%u] (max %u bytes)\n",
454         end, end + maxlen, start, end + maxlen - 1, maxlen);
455     uint8_t *buf = hse->buffer;
456
457     uint16_t match_maxlen = 0;
458     uint16_t match_index = MATCH_NOT_FOUND;
459
460     uint16_t len = 0;
461     uint8_t * const needlepoint = &buf[end];
462 #if HEATSHRINK_USE_INDEX
463     struct hs_index *hsi = HEATSHRINK_ENCODER_INDEX(hse);
464     int16_t pos = hsi->index[end];
465
466     while (pos - (int16_t)start >= 0) {
467         uint8_t * const pospoint = &buf[pos];
468         len = 0;
469
470         /* Only check matches that will potentially beat the current maxlen.
471          * This is redundant with the index if match_maxlen is 0, but the
472          * added branch overhead to check if it == 0 seems to be worse. */
473         if (pospoint[match_maxlen] != needlepoint[match_maxlen]) {
474             pos = hsi->index[pos];
475             continue;
476         }
477
478         for (len = 1; len < maxlen; len++) {
479             if (pospoint[len] != needlepoint[len]) break;
480         }
481
482         if (len > match_maxlen) {
483             match_maxlen = len;
484             match_index = pos;
485             if (len == maxlen) { break; } /* won't find better */
486         }
487         pos = hsi->index[pos];
488     }
489 #else
490     for (int16_t pos=end - 1; pos - (int16_t)start >= 0; pos--) {
491         uint8_t * const pospoint = &buf[pos];
492         if ((pospoint[match_maxlen] == needlepoint[match_maxlen])
493             && (*pospoint == *needlepoint)) {
494             for (len=1; len<maxlen; len++) {
495                 if (0) {
496                     LOG("  --> cmp buf[%d] == 0x%02x against %02x (start %u)\n",
497                         pos + len, pospoint[len], needlepoint[len], start);
498                 }
499                 if (pospoint[len] != needlepoint[len]) { break; }
500             }
501             if (len > match_maxlen) {
502                 match_maxlen = len;
503                 match_index = pos;
504                 if (len == maxlen) { break; } /* don't keep searching */
505             }
506         }
507     }
508 #endif
509
```

```
    const size_t break_even_point =
      (1 + HEATSHRINK_ENCODER_WINDOW_BITS(hse) +
          HEATSHRINK_ENCODER_LOOKAHEAD_BITS(hse));

    /* Instead of comparing break_even_point against 8*match_maxlen,
     * compare match_maxlen against break_even_point/8 to avoid
     * overflow. Since MIN_WINDOW_BITS and MIN_LOOKAHEAD_BITS are 4 and
     * 3, respectively, break_even_point/8 will always be at least 1. */
    if (match_maxlen > (break_even_point / 8)) {
        LOG("-- best match: %u bytes at -%u\n",
            match_maxlen, end - match_index);
        *match_length = match_maxlen;
        return end - match_index;
    }
    LOG("-- none found\n");
    return MATCH_NOT_FOUND;
}

static uint8_t push_outgoing_bits(heatshrink_encoder *hse, output_info *oi) {
    uint8_t count = 0;
    uint8_t bits = 0;
    if (hse->outgoing_bits_count > 8) {
        count = 8;
        bits = hse->outgoing_bits >> (hse->outgoing_bits_count - 8);
    } else {
        count = hse->outgoing_bits_count;
        bits = hse->outgoing_bits;
    }

    if (count > 0) {
        LOG("-- pushing %d outgoing bits: 0x%02x\n", count, bits);
        push_bits(hse, count, bits, oi);
        hse->outgoing_bits_count -= count;
    }
    return count;
}

/* Push COUNT (max 8) bits to the output buffer, which has room.
 * Bytes are set from the lowest bits, up. */
static void push_bits(heatshrink_encoder *hse, uint8_t count, uint8_t bits,
        output_info *oi) {
    ASSERT(count <= 8);
    LOG("++ push_bits: %d bits, input of 0x%02x\n", count, bits);

    /* If adding a whole byte and at the start of a new output byte,
     * just push it through whole and skip the bit IO loop. */
    if (count == 8 && hse->bit_index == 0x80) {
        oi->buf[(*oi->output_size)++] = bits;
    } else {
        for (int i=count - 1; i>=0; i--) {
            bool bit = bits & (1 << i);
            if (bit) { hse->current_byte |= hse->bit_index; }
            if (0) {
                LOG("  -- setting bit %d at bit index 0x%02x, byte => 0x%02x\n",
                    bit ? 1 : 0, hse->bit_index, hse->current_byte);
            }
            hse->bit_index >>= 1;
            if (hse->bit_index == 0x00) {
                hse->bit_index = 0x80;
                LOG(" > pushing byte 0x%02x\n", hse->current_byte);
                oi->buf[(*oi->output_size)++] = hse->current_byte;
                hse->current_byte = 0x00;
            }
        }
    }
}

static void push_literal_byte(heatshrink_encoder *hse, output_info *oi) {
    uint16_t processed_offset = hse->match_scan_index - 1;
    uint16_t input_offset = get_input_offset(hse) + processed_offset;
    uint8_t c = hse->buffer[input_offset];
    LOG("-- yielded literal byte 0x%02x ('%c') from +%d\n",
        c, isprint(c) ? c : '.', input_offset);
```

```
583     push_bits(hse, 8, c, oi);
584 }
585
586 static void save_backlog(heatshrink_encoder *hse) {
587     size_t input_buf_sz = get_input_buffer_size(hse);
588
589     uint16_t msi = hse->match_scan_index;
590
591     /* Copy processed data to beginning of buffer, so it can be
592      * used for future matches. Don't bother checking whether the
593      * input is less than the maximum size, because if it isn't,
594      * we're done anyway. */
595     uint16_t rem = input_buf_sz - msi; // unprocessed bytes
596     uint16_t shift_sz = input_buf_sz + rem;
597
598     memmove(&hse->buffer[0],
599         &hse->buffer[input_buf_sz - rem],
600         shift_sz);
601
602     hse->match_scan_index = 0;
603     hse->input_size -= input_buf_sz - rem;
604 }
```

B.1.3 heatshrink_decoder.c

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include "heatshrink_decoder.h"
4
5  /* States for the polling state machine. */
6  typedef enum {
7      HSDS_TAG_BIT,               /* tag bit */
8      HSDS_YIELD_LITERAL,         /* ready to yield literal byte */
9      HSDS_BACKREF_INDEX_MSB,     /* most significant byte of index */
10     HSDS_BACKREF_INDEX_LSB,     /* least significant byte of index */
11     HSDS_BACKREF_COUNT_MSB,     /* most significant byte of count */
12     HSDS_BACKREF_COUNT_LSB,     /* least significant byte of count */
13     HSDS_YIELD_BACKREF,         /* ready to yield back-reference */
14 } HSD_state;
15
16 #if HEATSHRINK_DEBUGGING_LOGS
17 #include <stdio.h>
18 #include <ctype.h>
19 #include <assert.h>
20 #define LOG(...) fprintf(stderr, __VA_ARGS__)
21 #define ASSERT(X) assert(X)
22 static const char *state_names[] = {
23     "tag_bit",
24     "yield_literal",
25     "backref_index_msb",
26     "backref_index_lsb",
27     "backref_count_msb",
28     "backref_count_lsb",
29     "yield_backref",
30 };
31 #else
32 #define LOG(...) /* no-op */
33 #define ASSERT(X) /* no-op */
34 #endif
35
36 typedef struct {
37     uint8_t *buf;               /* output buffer */
38     size_t buf_size;            /* buffer size */
39     size_t *output_size;        /* bytes pushed to buffer, so far */
40 } output_info;
41
42 #define NO_BITS ((uint16_t)-1)
43
44 /* Forward references. */
```

```c
static uint16_t get_bits(heatshrink_decoder *hsd, uint8_t count);
static void push_byte(heatshrink_decoder *hsd, output_info *oi, uint8_t byte);

#if HEATSHRINK_DYNAMIC_ALLOC
heatshrink_decoder *heatshrink_decoder_alloc(uint16_t input_buffer_size,
                                             uint8_t window_sz2,
                                             uint8_t lookahead_sz2) {
    if ((window_sz2 < HEATSHRINK_MIN_WINDOW_BITS) ||
        (window_sz2 > HEATSHRINK_MAX_WINDOW_BITS) ||
        (input_buffer_size == 0) ||
        (lookahead_sz2 < HEATSHRINK_MIN_LOOKAHEAD_BITS) ||
        (lookahead_sz2 >= window_sz2)) {
        return NULL;
    }
    size_t buffers_sz = (1 << window_sz2) + input_buffer_size;
    size_t sz = sizeof(heatshrink_decoder) + buffers_sz;
    heatshrink_decoder *hsd = HEATSHRINK_MALLOC(sz);
    if (hsd == NULL) { return NULL; }
    hsd->input_buffer_size = input_buffer_size;
    hsd->window_sz2 = window_sz2;
    hsd->lookahead_sz2 = lookahead_sz2;
    heatshrink_decoder_reset(hsd);
    LOG("-- allocated decoder with buffer size of %zu (%zu + %u + %u)\n",
        sz, sizeof(heatshrink_decoder), (1 << window_sz2), input_buffer_size);
    return hsd;
}

void heatshrink_decoder_free(heatshrink_decoder *hsd) {
    size_t buffers_sz = (1 << hsd->window_sz2) + hsd->input_buffer_size;
    size_t sz = sizeof(heatshrink_decoder) + buffers_sz;
    HEATSHRINK_FREE(hsd, sz);
    (void)sz;   /* may not be used by free */
}
#endif

void heatshrink_decoder_reset(heatshrink_decoder *hsd) {
    size_t buf_sz = 1 << HEATSHRINK_DECODER_WINDOW_BITS(hsd);
    size_t input_sz = HEATSHRINK_DECODER_INPUT_BUFFER_SIZE(hsd);
    memset(hsd->buffers, 0, buf_sz + input_sz);
    hsd->state = HSDS_TAG_BIT;
    hsd->input_size = 0;
    hsd->input_index = 0;
    hsd->bit_index = 0x00;
    hsd->current_byte = 0x00;
    hsd->output_count = 0;
    hsd->output_index = 0;
    hsd->head_index = 0;
}

/* Copy SIZE bytes into the decoder's input buffer, if it will fit. */
HSD_sink_res heatshrink_decoder_sink(heatshrink_decoder *hsd,
        uint8_t *in_buf, size_t size, size_t *input_size) {
    if ((hsd == NULL) || (in_buf == NULL) || (input_size == NULL)) {
        return HSDR_SINK_ERROR_NULL;
    }

    size_t rem = HEATSHRINK_DECODER_INPUT_BUFFER_SIZE(hsd) - hsd->input_size;
    if (rem == 0) {
        *input_size = 0;
        return HSDR_SINK_FULL;
    }

    size = rem < size ? rem : size;
    LOG("-- sinking %zd bytes\n", size);
    /* copy into input buffer (at head of buffers) */
    memcpy(&hsd->buffers[hsd->input_size], in_buf, size);
    hsd->input_size += size;
    *input_size = size;
    return HSDR_SINK_OK;
}


/*****************
```

```
 * Decompression *
 *****************/

#define BACKREF_COUNT_BITS(HSD) (HEATSHRINK_DECODER_LOOKAHEAD_BITS(HSD))
#define BACKREF_INDEX_BITS(HSD) (HEATSHRINK_DECODER_WINDOW_BITS(HSD))

// States
static HSD_state st_tag_bit(heatshrink_decoder *hsd);
static HSD_state st_yield_literal(heatshrink_decoder *hsd,
    output_info *oi);
static HSD_state st_backref_index_msb(heatshrink_decoder *hsd);
static HSD_state st_backref_index_lsb(heatshrink_decoder *hsd);
static HSD_state st_backref_count_msb(heatshrink_decoder *hsd);
static HSD_state st_backref_count_lsb(heatshrink_decoder *hsd);
static HSD_state st_yield_backref(heatshrink_decoder *hsd,
    output_info *oi);

HSD_poll_res heatshrink_decoder_poll(heatshrink_decoder *hsd,
        uint8_t *out_buf, size_t out_buf_size, size_t *output_size) {
    if ((hsd == NULL) || (out_buf == NULL) || (output_size == NULL)) {
        return HSDR_POLL_ERROR_NULL;
    }
    *output_size = 0;

    output_info oi;
    oi.buf = out_buf;
    oi.buf_size = out_buf_size;
    oi.output_size = output_size;

    while (1) {
        LOG("-- poll, state is %d (%s), input_size %d\n",
            hsd->state, state_names[hsd->state], hsd->input_size);
        uint8_t in_state = hsd->state;
        switch (in_state) {
        case HSDS_TAG_BIT:
            hsd->state = st_tag_bit(hsd);
            break;
        case HSDS_YIELD_LITERAL:
            hsd->state = st_yield_literal(hsd, &oi);
            break;
        case HSDS_BACKREF_INDEX_MSB:
            hsd->state = st_backref_index_msb(hsd);
            break;
        case HSDS_BACKREF_INDEX_LSB:
            hsd->state = st_backref_index_lsb(hsd);
            break;
        case HSDS_BACKREF_COUNT_MSB:
            hsd->state = st_backref_count_msb(hsd);
            break;
        case HSDS_BACKREF_COUNT_LSB:
            hsd->state = st_backref_count_lsb(hsd);
            break;
        case HSDS_YIELD_BACKREF:
            hsd->state = st_yield_backref(hsd, &oi);
            break;
        default:
            return HSDR_POLL_ERROR_UNKNOWN;
        }

        /* If the current state cannot advance, check if input or output
         * buffer are exhausted. */
        if (hsd->state == in_state) {
            if (*output_size == out_buf_size) { return HSDR_POLL_MORE; }
            return HSDR_POLL_EMPTY;
        }
    }
}

static HSD_state st_tag_bit(heatshrink_decoder *hsd) {
    uint32_t bits = get_bits(hsd, 1);  // get tag bit
    if (bits == NO_BITS) {
        return HSDS_TAG_BIT;
    } else if (bits) {
```

```
191             return HSDS_YIELD_LITERAL;
192         } else if (HEATSHRINK_DECODER_WINDOW_BITS(hsd) > 8) {
193             return HSDS_BACKREF_INDEX_MSB;
194         } else {
195             hsd->output_index = 0;
196             return HSDS_BACKREF_INDEX_LSB;
197         }
198     }
199
200     static HSD_state st_yield_literal(heatshrink_decoder *hsd,
201             output_info *oi) {
202         /* Emit a repeated section from the window buffer, and add it (again)
203          * to the window buffer. (Note that the repetition can include
204          * itself.)*/
205         if (*oi->output_size < oi->buf_size) {
206             uint16_t byte = get_bits(hsd, 8);
207             if (byte == NO_BITS) { return HSDS_YIELD_LITERAL; } /* out of input */
208             uint8_t *buf = &hsd->buffers[HEATSHRINK_DECODER_INPUT_BUFFER_SIZE(hsd)];
209             uint16_t mask = (1 << HEATSHRINK_DECODER_WINDOW_BITS(hsd)) - 1;
210             uint8_t c = byte & 0xFF;
211             LOG("-- emitting literal byte 0x%02x ('%c')\n", c, isprint(c) ? c : '.');
212             buf[hsd->head_index++ & mask] = c;
213             push_byte(hsd, oi, c);
214             return HSDS_TAG_BIT;
215         } else {
216             return HSDS_YIELD_LITERAL;
217         }
218     }
219
220     static HSD_state st_backref_index_msb(heatshrink_decoder *hsd) {
221         uint8_t bit_ct = BACKREF_INDEX_BITS(hsd);
222         ASSERT(bit_ct > 8);
223         uint16_t bits = get_bits(hsd, bit_ct - 8);
224         LOG("-- backref index (msb), got 0x%04x (+1)\n", bits);
225         if (bits == NO_BITS) { return HSDS_BACKREF_INDEX_MSB; }
226         hsd->output_index = bits << 8;
227         return HSDS_BACKREF_INDEX_LSB;
228     }
229
230     static HSD_state st_backref_index_lsb(heatshrink_decoder *hsd) {
231         uint8_t bit_ct = BACKREF_INDEX_BITS(hsd);
232         uint16_t bits = get_bits(hsd, bit_ct < 8 ? bit_ct : 8);
233         LOG("-- backref index (lsb), got 0x%04x (+1)\n", bits);
234         if (bits == NO_BITS) { return HSDS_BACKREF_INDEX_LSB; }
235         hsd->output_index |= bits;
236         hsd->output_index++;
237         uint8_t br_bit_ct = BACKREF_COUNT_BITS(hsd);
238         hsd->output_count = 0;
239         return (br_bit_ct > 8) ? HSDS_BACKREF_COUNT_MSB : HSDS_BACKREF_COUNT_LSB;
240     }
241
242     static HSD_state st_backref_count_msb(heatshrink_decoder *hsd) {
243         uint8_t br_bit_ct = BACKREF_COUNT_BITS(hsd);
244         ASSERT(br_bit_ct > 8);
245         uint16_t bits = get_bits(hsd, br_bit_ct - 8);
246         LOG("-- backref count (msb), got 0x%04x (+1)\n", bits);
247         if (bits == NO_BITS) { return HSDS_BACKREF_COUNT_MSB; }
248         hsd->output_count = bits << 8;
249         return HSDS_BACKREF_COUNT_LSB;
250     }
251
252     static HSD_state st_backref_count_lsb(heatshrink_decoder *hsd) {
253         uint8_t br_bit_ct = BACKREF_COUNT_BITS(hsd);
254         uint16_t bits = get_bits(hsd, br_bit_ct < 8 ? br_bit_ct : 8);
255         LOG("-- backref count (lsb), got 0x%04x (+1)\n", bits);
256         if (bits == NO_BITS) { return HSDS_BACKREF_COUNT_LSB; }
257         hsd->output_count |= bits;
258         hsd->output_count++;
259         return HSDS_YIELD_BACKREF;
260     }
261
262     static HSD_state st_yield_backref(heatshrink_decoder *hsd,
263             output_info *oi) {
```

```
    size_t count = oi->buf_size - *oi->output_size;
    if (count > 0) {
        size_t i = 0;
        if (hsd->output_count < count) count = hsd->output_count;
        uint8_t *buf = &hsd->buffers[HEATSHRINK_DECODER_INPUT_BUFFER_SIZE(hsd)];
        uint16_t mask = (1 << HEATSHRINK_DECODER_WINDOW_BITS(hsd)) - 1;
        uint16_t neg_offset = hsd->output_index;
        LOG("-- emitting %zu bytes from -%u bytes back\n", count, neg_offset);
        ASSERT(neg_offset <= mask + 1);
        ASSERT(count <= (size_t)(1 << BACKREF_COUNT_BITS(hsd)));

        for (i=0; i<count; i++) {
            uint8_t c = buf[(hsd->head_index - neg_offset) & mask];
            push_byte(hsd, oi, c);
            buf[hsd->head_index & mask] = c;
            hsd->head_index++;
            LOG("  -- ++ 0x%02x\n", c);
        }
        hsd->output_count -= count;
        if (hsd->output_count == 0) { return HSDS_TAG_BIT; }
    }
    return HSDS_YIELD_BACKREF;
}

/* Get the next COUNT bits from the input buffer, saving incremental progress.
 * Returns NO_BITS on end of input, or if more than 15 bits are requested. */
static uint16_t get_bits(heatshrink_decoder *hsd, uint8_t count) {
    uint16_t accumulator = 0;
    int i = 0;
    if (count > 15) { return NO_BITS; }
    LOG("-- popping %u bit(s)\n", count);

    /* If we aren't able to get COUNT bits, suspend immediately, because we
     * don't track how many bits of COUNT we've accumulated before suspend. */
    if (hsd->input_size == 0) {
        if (hsd->bit_index < (1 << (count - 1))) { return NO_BITS; }
    }

    for (i = 0; i < count; i++) {
        if (hsd->bit_index == 0x00) {
            if (hsd->input_size == 0) {
                LOG("  -- out of bits, suspending w/ accumulator of %u (0x%02x)\n",
                    accumulator, accumulator);
                return NO_BITS;
            }
            hsd->current_byte = hsd->buffers[hsd->input_index++];
            LOG("  -- pulled byte 0x%02x\n", hsd->current_byte);
            if (hsd->input_index == hsd->input_size) {
                hsd->input_index = 0; /* input is exhausted */
                hsd->input_size = 0;
            }
            hsd->bit_index = 0x80;
        }
        accumulator <<= 1;
        if (hsd->current_byte & hsd->bit_index) {
            accumulator |= 0x01;
            if (0) {
                LOG("  -- got 1, accumulator 0x%04x, bit_index 0x%02x\n",
                accumulator, hsd->bit_index);
            }
        } else {
            if (0) {
                LOG("  -- got 0, accumulator 0x%04x, bit_index 0x%02x\n",
                accumulator, hsd->bit_index);
            }
        }
        hsd->bit_index >>= 1;
    }

    if (count > 1) { LOG("  -- accumulated %08x\n", accumulator); }
    return accumulator;
}

```

```
337 HSD_finish_res heatshrink_decoder_finish(heatshrink_decoder *hsd) {
338     if (hsd == NULL) { return HSDR_FINISH_ERROR_NULL; }
339     switch (hsd->state) {
340     case HSDS_TAG_BIT:
341         return hsd->input_size == 0 ? HSDR_FINISH_DONE : HSDR_FINISH_MORE;
342
343     /* If we want to finish with no input, but are in these states, it's
344      * because the 0-bit padding to the last byte looks like a backref
345      * marker bit followed by all 0s for index and count bits. */
346     case HSDS_BACKREF_INDEX_LSB:
347     case HSDS_BACKREF_INDEX_MSB:
348     case HSDS_BACKREF_COUNT_LSB:
349     case HSDS_BACKREF_COUNT_MSB:
350         return hsd->input_size == 0 ? HSDR_FINISH_DONE : HSDR_FINISH_MORE;
351
352     /* If the output stream is padded with 0xFFs (possibly due to being in
353      * flash memory), also explicitly check the input size rather than
354      * uselessly returning MORE but yielding 0 bytes when polling. */
355     case HSDS_YIELD_LITERAL:
356         return hsd->input_size == 0 ? HSDR_FINISH_DONE : HSDR_FINISH_MORE;
357
358     default:
359         return HSDR_FINISH_MORE;
360     }
361 }
362
363 static void push_byte(heatshrink_decoder *hsd, output_info *oi, uint8_t byte) {
364     LOG(" -- pushing byte: 0x%02x ('%c')\n", byte, isprint(byte) ? byte : '.');
365     oi->buf[(*oi->output_size)++] = byte;
366     (void)hsd;
367 }
```

B.2 Encryption C Algorithm

The following code was used to implement ASCII shift encryption in C [18]:

```
1 void encrypt(uint8_t *input, uint32_t input_size, uint8_t *encrypted) {
2   for(int i = 0; i < input_size; i++)
3         encrypted[i] = input[i] + 5; //the key for encryption is 3 that is added to ASCII value
4 }
```

B.3 String Comparison Algorithm

The following code was used to implement data comparison of strings in python [19]

```
1 import difflib
2
3 string1 = "Time (ms),AccX (g),AccY (g),AccZ (g) 17,-0.01,0.01,1.04 127,-0.01,0.02,1.03"
4 string2 = "3A313"
5
6 temp = difflib.SequenceMatcher(None,string1 ,string2)
7
8 percentage = temp.ratio()*100
9 print(percentage)
```