EEE3097S Milestone 4 Submission:
Final Report

Taine de Buys (DBYTAI001) and Mike Altshuler (ALTMIC003):

Submitted on 2021/10/31

Table of Contents:

Admin Documents:

Contribution Table:

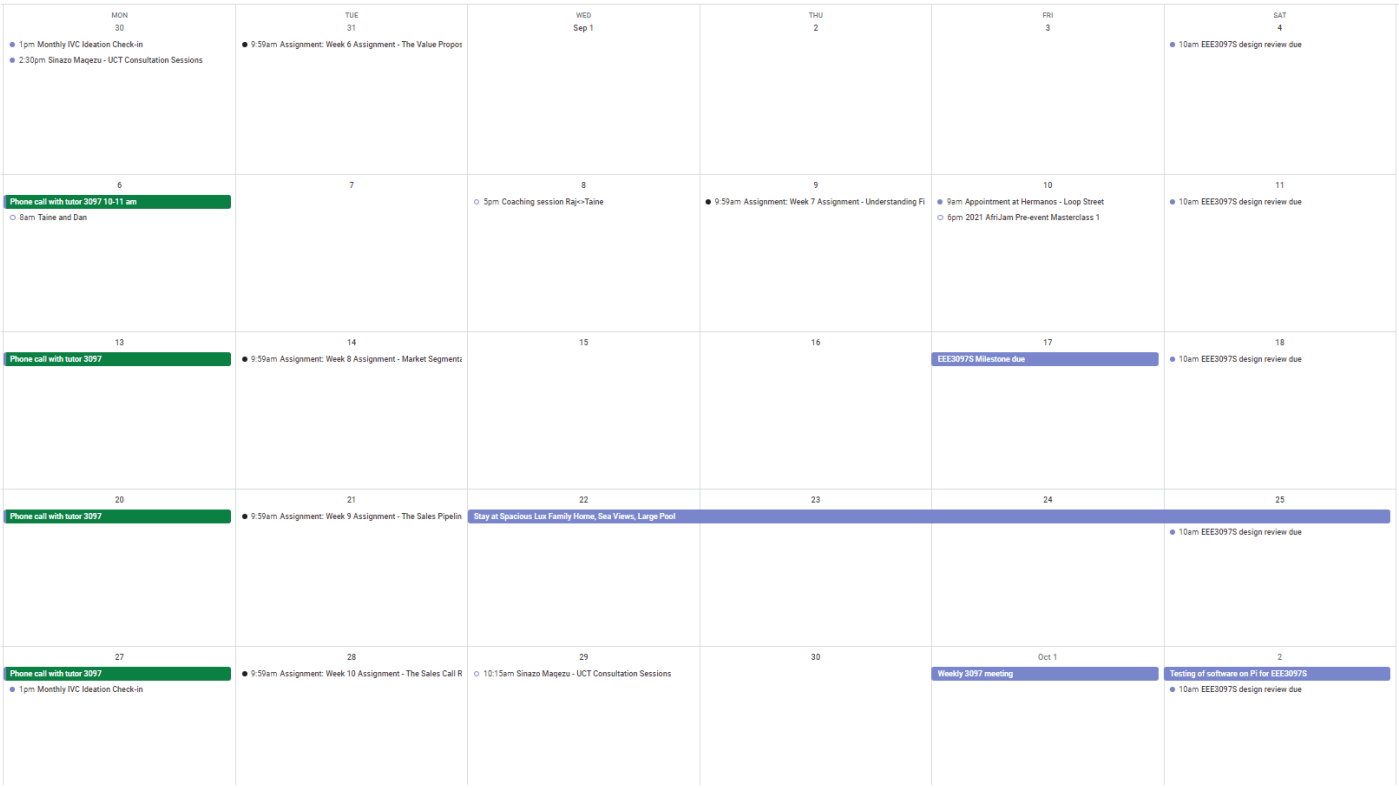
Each group member was assigned a subsystem to investigate, and determine how their respective subsystem will work in the greater design. Taine was assigned the encryption subsystem and Michael was assigned the compression subsystem. For reference , a demo video is linked under results, and can also be found here.

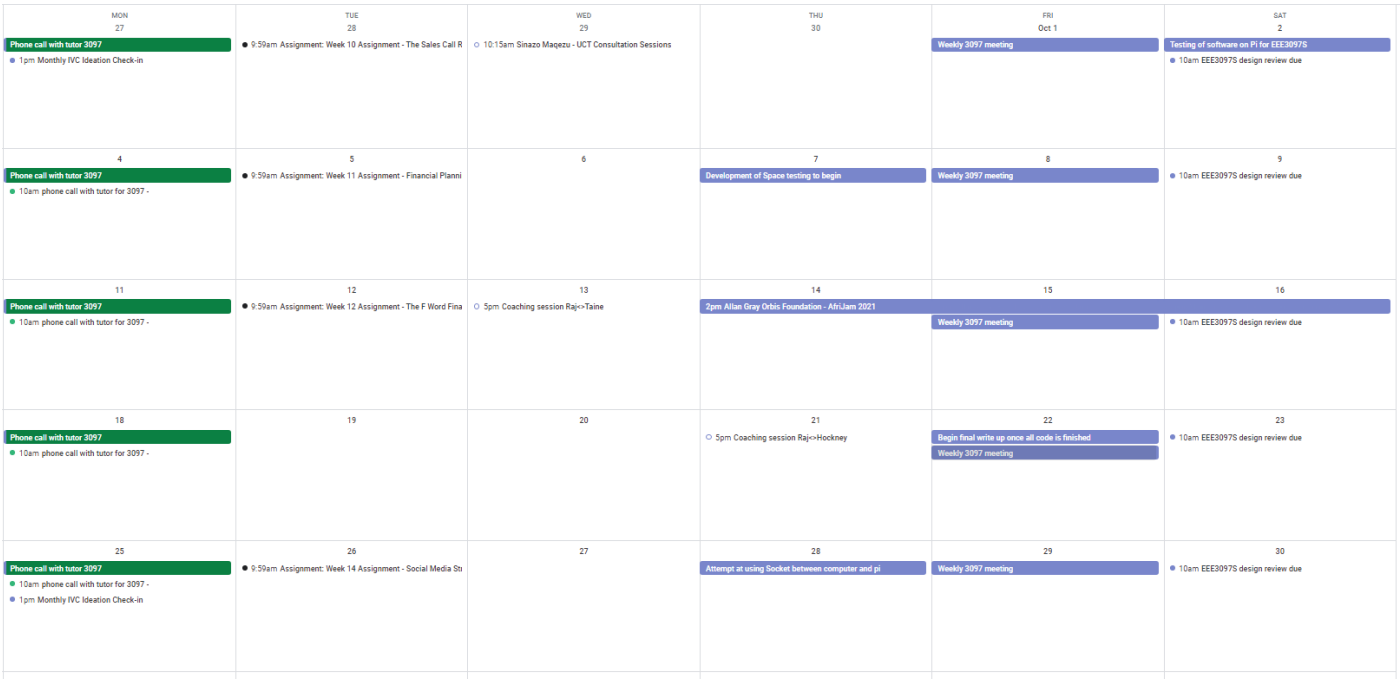
Contributions:	**Contributer:**
Requirement Analysis	Taine
Subsystem design	Michael
Development Timeline	Taine and Michael
Compression/decompression algorithm	Michael
Compression Benchmarking	Michael
Encryption/Decryption algorithm	Taine
Encryption Benchmarking	Taine
System Benchmarking	Michael and Taine
GitHub Admin	Michael and Taine
Report Writing	Michael and Taine
IMU Module	Taine
Experiment Setup	Michael
Results	Michael and Taine
Acceptance Test Procedure (ATP)	Michael and Taine

Project Management Tool:

Through google workspace, google documents are created that allow for all content to be stored in the cloud, as well as allow both team members to edit the document due for the week. Using google calendar, all important dates have been diarized, allowing team members to see deadlines and schedule days to work on the project. If either of us update the calendar, a notification is sent to the other, and their calendar is also updated. This is ideal for remote

working. Below is a screenshot of the projects timeline in its entirety taken off our project management tools calendar:





GitHub:

Click here to be redirected to the newly updated remote GitHub repository containing all relevant compression and decompression algorithms as well as the encryption and decryption algorithms designed by the team members as well as the code needed to operate the IMU for the simulation of the output data. The repository further contains the benchmarking algorithms designed to test each subsystem and the system as a whole, and further contains the output files to which the compressed and encrypted data were sent. The final submission folder contains the original and working files in 1 complete folder. Finally there are 2 folders that have various files that are required to be on the Pi and a laptop respectively, as the team attempted to use sockets to transfer the necessary keys between them. The team was able to transfer the encrypted file and original RSA public key used to encrypt the symmetric key, yet due to time constraints and inexperience in the area could not troubleshoot the final error before decrypting the transferred file

Requirement Analysis:
Interpretation of the requirements

High-level requirements were ascertained based on the description of the project provided by Prof Amit Mishra. These are outlined below:

- Design a system to meet all requirements using engineering design methodology and a project management tool.
- To build and implement code that will encrypt and compress data using a raspberry Pi ARM processor. The data originates from buoys in the South Atlantic that measure wave and ice dynamics. Data is output from an IMU onto an ARM processor located on the buoy.
- The IMU used is the ICM – 20649 but will not be provided. Must design IP without hardware.
- Must be able to extract at least 25% of the Fourier transformed data once compressed. (the lower 25% of the Fourier transforms must be maintained)
- In addition to reducing the data, the processing done by the ARM must be minimized to reduce power usage.

Technical Insights

- Packets of data are sent by the IMU every 10 ms. We are investigating the most efficient way to compress, encrypt and transmit the data from the ARM processor that best fits this IMU parameter.
- The data is stored in 16 bit ADCs before it is sent, looking at raw data provided, each packet produces 29 readings and a timestamp.

Paper Design:

Comparison of available compression and encryption algorithms:

1. Comparison of Encryption Algorithms:

Data encryption translates data into another form, or code, so that only people with access to a secret key (formally called a decryption key) or password can read it. The data is known as ciphertext while it is encrypted and before decryption. The purpose of encryption is to ensure confidentiality (security against 3rd parties reading data) and integrity (security against 3rd party data manipulation) .

To begin, symmetric encryption algorithms were examined and compared. These involve ciphers and a private key made available to all participants. The first kind of symmetric encryption cipher looked at was a stream cipher using the XOR operator on the bits of the plaintext, where each bit is XORed with a binary key, producing a ciphertext stream that can be transported before the same bit key is then applied to the stream to decrypt it. Another example of a symmetric cipher that was analyzed was the caesar cypher, where each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet, before it is transported and decoded through shifting the message back to the original plaintext's position in the alphabet. Both of these algorithms were found to be far too rudimentary, and provide very little communication security, as messages of the same length (which would always be the case for the data transmitted in the project) can be intercepted and the encryption easily solved.

A more complex implementation of symmetrical data encryption that was analyzed was a block cipher. A block cipher takes in a fixed-length input and iteratively encrypts the plaintext again and again using a different key (a "round key") for each round and ultimately outputs a ciphertext of the same length. A different round key may be used for each iteration as depicted below:

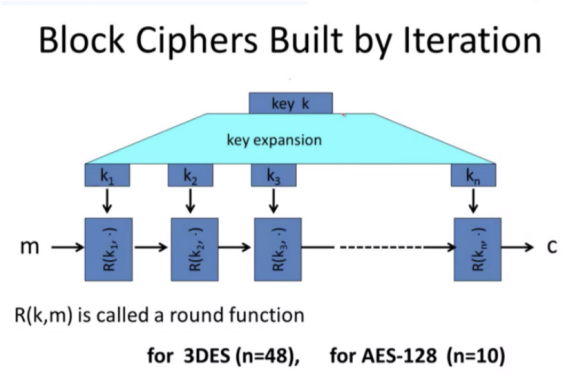


Image 1: Diagram of Block cipher

The issue with each of these algorithms is each requires all participants to have the same key, and is insecure as messages that contain repeated data divulge information on how encryption takes place, threatening security. In addition, each requires time to develop and implement the algorithm, and it is due to these reasons that we have been discouraged from using them.

The next set of algorithms examined were asymmetric encryptions. In such a system, any person can encrypt a message using the intended receiver's public key, but that encrypted message can only be decrypted with the receiver's private key. The algorithm examined is the RSA (Rivest–Shamir–Adleman) encryption algo. This algorithm is secure through the use of prime factorization, which bases the algorithm off the fact that finding factors of large composite numbers is difficult. This algorithm is chosen as not only does it increase security, but also has predefined python libraries which will allow for easier implementation.

2. Comparison of Compression Algorithms:

The following tables display the comparisons of various compression algorithms for both the compression and decompression stages of the process.

Algorithm	Time	Size	Command	Parameters	Comment
none	0m0.934s	939M	tar	cf	tar itself is an *archiving* tool, you do not *need* to compress the archives.
gzip	0m23.502s	177M	gzip	cfz	
gzip	0m3.132s	177M	pigz	c -Ipigz -f	Parallel gzip using pigz 2.4.
bzip2	1m0.798s	134M	bzip2	cfj	Standard bzip2 will only use *one* core (at 100%)
bzip2	0m9.091s	135M	pbzip2	c -Ipbzip2 -f	Parallel bzip2. pbzip2 process used about 900 MiB RAM at maximum.
lz4	0m3.914s	287M	lz4	c -I"lz4" -f	Really fast but the resulting archive is barely compressed. *Worst* compression king.
lz4	0m56.506s	207M	lz4 -12	c -I"lz4 -12" -f	Supports levels -[1-12]. Uses 1 core, and there does not appear to be any multi-threaded variant.
lzip	4m42.017s	116M	lzip	c --lzip -f	v1.21. Standard lzip will only use *one* core (at 100%). **Very slow.**
lzip	0m42.542	118M	plzip	c -Iplzip -f	plzip 1.8 (Parallel lzip), default level (-6).
lzip	1m39.697s	110M	plzip -9	c -I"plzip -9" -f	Parallel lzip at best compression (-9). plzip process used 5.1 GiB RAM at maximum.
xz	5m2.952s	114M	xz	cfJ	Standard xz will only use one core (at 100%). **Unbearably slow.**
xz	0m53.569s	115M	pxz	c -Ipxz -f	Parallel PXZ 4.999.9beta. Process used 1.4 GiB RAM at maximum.
xz	1m33.441s	110M	pxz -9	c -I"pxz -9" -f	Parallel PXZ 4.999.9beta using its best possible compression. pxz process used 3.5 GiB at maximum.
zstd	0m3.034s	167M	zstd	c --zstd -f	zstd uses 1 core by default.
zstd	1m18.238s	117M	zstd -19 -T0	c -I"zstd -19 -T0" -f	-19 gives the best possible compression and -T0 utilizes all cores. If a non-zero number is specified, zstd uses that many cores.

Table 2: Depicts the relative compression performance differences of different compression algorithms.

Algorithm	Time	Command	Parameters	Comments
none	0m1.204s	tar	-xf	Raw tar with no compression.
gzip2	0m4.232s	gzip2	-xfz	
gzip	0m2.729s	pigz	-x -Ipigz -f	gzip is a clear winner if decompression speed is the *only* consideration.
bzip2	0m20.181s	bzip2	xfj	
bzip2	0m19.533s	pbzip2	-x -Ipbzip2 -f	The difference between bzip2 and pbzip2 when *decompressing* is barely measurable
lzip	0m10.590s	lzip	-x --lzip -f	
lz4	0m1.873s	lz4	-x -Ilz4 -f	**Fastest** of them all but not very impressive considering the compression it offers is almost nonexistant.
lzip	0m8.982s	plzip	-x -Iplzip -f	
xz	0m7.419s	xz	-xfJ	xz offers the best decompression speeds of all the well-compressed algorithms.
xz	0m7.462s	pxz	-x -Ipxz -f	
zstd	0m3.095s	zstd	-x --zstd -f	When compressed with no options (the default compression level is 3).
zstd	0m2.556s	zstd	x --zstd -f	When compressed with tar c -I"zstd -19 -T0" (compression level 19)

Table 3: Depicts the relative decompression performance of different compression algorithms.

As depicted above, the performance of Gzip is compared to various other compression algorithms. It should be noted that the Gzip algorithm in its natural implementation has an average runtime of 23.502 seconds when compressing. When compared to the majority of the other algorithms, Gzip's compression time is fast, but not the fastest. Bzip2 has a runtime of approximately 0.7 seconds which is significantly faster than Gzip, however, it is much harder to implement. Gzip can however be made to have a faster

compression time by running it in parallel, bringing the runtime down to 3.132 seconds. When looking at this as a percentage, it is a significant difference.

When looking at the decompression performance Gzip is a clear winner if decompression speed is the only consideration. When taking both the compression and decompression speed into account Gzip is a convincing choice to make in terms of choosings potential algorithms to implement. Furthermore, Gzip is part of the Zlib library which is open source and provides a completely lossless form of data comression. Taking all of these factors into account, Gzip is the compression algorithm of choice for this project.

Feasibility analysis:

Order of Operations on Data:

An important comparison that is undergone within the design process is in deciding whether it is better to encrypt data before compressing the ciphertext, or first compress before encrypting the file. It has been decided to compress the data before encryption, as compression algorithms make use of statistical redundancies in the data. These redundancies could be eliminated if advanced encryption ciphers are used, which will reduce the compression size of the file. This opposes one of the core requirements of the project, as data files will be larger when transported. Thus, compressing it first will reduce data size in file storage as well as transmission. It is important to note that doing this does still allows for side channel attacks on data confidentiality through the use of compression oracles. However, as requirements for the project focus on efficiency over security, this is an acceptable risk.

Python v.s C

C is a structure oriented programming language while python is object oriented. Due to this fact, C is mostly used for the programming and initialising of hardware while python is better suited to software applications. Furthermore, python has fully formed and predefined library functions such as PyCryptoDome which will enable the encryption of the compressed data.

The requirements of this project revolve around the manipulation of data and due to the project being inherently software oriented rather than hardware oriented, Python is the programming language of choice for this implementation.

Possible bottlenecks:

Possible obstacles that may be encountered during the implementation of the compression and encryption algorithms include slow runtime of the algorithms when fed with large sets of data. This problem can be dealt with by implementing a parallelized version of the Gzip algorithm.

This is a solvable problem, however the parallel implementation will be more complex than the original Gzip algorithm.

Another bottleneck is ensuring that the RSA algorithm is capable of converting the required data, as RSA encryption can only handle a maximum amount equal to your key size **(2048 bits** or 3072 bits)**.** Further examination is still required in order to determine if data meets this size, and whether each individual data set that is collected every 10mS is supposed to be encrypted individually and sent or if it is more efficient to group sets together before encrypting and compressing the data for transmission. If this is the case, another form of encryption may be required in order to meet the requirements of the project.

Subsystem Design:

Encryption Subsystem:

The type of encryption used for this project is the RSA encryption, implemented in a python script with the use of the PyCryptoDome library. The algorithm can be used for both confidentiality (encryption) and authentication (digital signature). The module Crypto.PublicKey.RSA provides facilities for generating new RSA keys, reconstructing them from known components, exporting them, and importing them. RSA is one of the most commonly used encryptions. In the event that upon implementation, it is unable to encrypt the data produced in an efficient manner (as many sites have alluded to the fact that it is computationally taxing), it can also be used in conjunction with a symmetric-key algorithm such as EAS, for which there are also python libraries available, and then the symmetric key will be encrypted with RSA encryption.

RSA encryption works under the premise that the algorithm is easy to compute in one direction, but almost impossible in reverse.The first step to RSA encryption involves is to generate keys. This is done through selecting 2 large prime numbers (p and q) and mutiplying them together to get a value n. Once we have *n*, we use Carmichael's totient function:

$$\lambda(n) = Lcm\,(p-1, q-1)$$

And find the lowest common multiple between p-1 and q-1. To create the public key, we choose a random number e (commonly chosen as 65.357 as larger numbers reduce efficiency) and compute the key as $e \mod n$. The cipher text c generated from message m is defined as $c = m^e \mod n$. In order to decrypt the now encrypted message, the private key is required by the recipient. Private keys are comprised of d and n. Where n is known, and the following equation is used to find d:

$$d = 1/e \mod \lambda(n)$$

Messages that were encrypted with the public key can be decrypted using the following formula:

$$m = c^d \bmod n$$

RSA can be used for more than just encrypting data. Its properties also make it a useful system for confirming that a message has been sent by the entity who claims to have sent it, as well as proving that a message hasn't been altered or tampered with.

When someone wants to prove the authenticity of their message, they can compute a hash (a function that takes data of an arbitrary size and turns it into a fixed-length value) of the plaintext, then sign it with their private key. They sign the hash by applying the same formula that is used in decryption (m = cd mod n). Once the message has been signed, they send this digital signature to the recipient alongside the message.

Due to the fact that much of these functions for computing the private and public keys are built into the python library, this encryption protocol offers both security and efficiency in implementing it.

Compression Subsystem:

The type of compression chosen for this project is the Gzip compression (GNU zip) algorithm, mainly due to the fact that Gzip uses the Zlib library.

Zlib is a free, open source software library for lossless data compression and decompression, and contains all the necessary tools needed to compress and decompress the data in such a way that once the process is complete and the data has been both encrypted and decrypted, the data has maintained its original form. Hence, this being a completely lossless process. This ensures accuracy in the findings - an essential part in receiving the data from the ARM processor on the SHARC BUOY in order to perform analysis.

Gzip is based on the deflate algorithm, which is a combination of the LZ77 and Huffman encoding principles. LZ77 is a dictionary based lossless compression algorithm. Its basic idea is to replace an occurrence of a particular sequence of bytes in data with a reference to a previous occurrence of that sequence. On the other hand, Huffman encoding is a statistical compression method. It encodes data symbols (such as characters) with variable-length codes, and lengths of the codes are based on the frequencies of corresponding symbols. Both the LZ77 and Huffman encoding work congruently with one another when compressing and decompressing the data.

From this information, the requirements and specifications of the compression subsystem can be deduced. The fundamental requirement of the compression subsystem is to compress the

raw output data from the SHARC BUOY and decompress the decrypted data in a lossless fashion. This ensures that the raw data maintains its original form so that the results can be interpreted as accurately as possible.

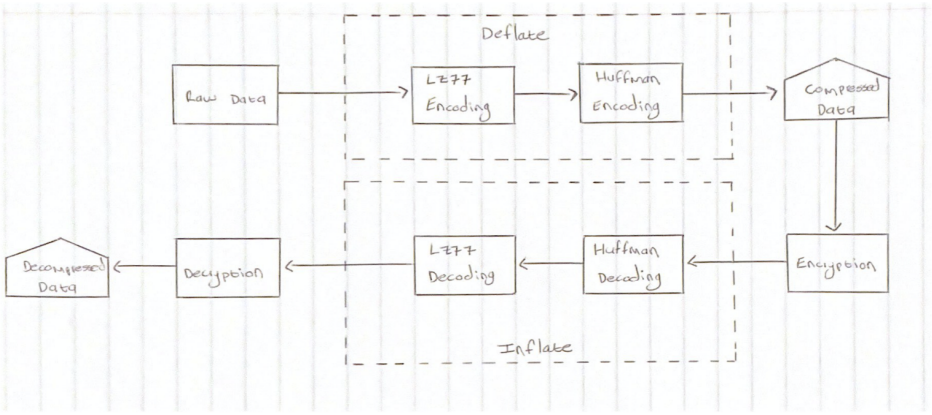
From the above requirements, the compression subsystem specifications can be outlined. In order to achieve the set out user requirements, the Gzip algorithm will have to be implemented meticulously, making sure both the LZ77 and Huffman encoding are used correctly. For the LZ77 encoding, a sliding window needs to be used to examine the input data sequence, and to maintain the historical data that serve as the dictionary. The sliding window consists of a search buffer and a look-ahead buffer. The search buffer serves the purpose of holding the dictionary, while the look-ahead buffer contains the next portion of the input data sequence to undergo encoding. Furthermore, the LZ77 encoding algorithm must implement a length-distance pair. This indicates that each of the next length characters is the same as the exact number of characters behind it in the original raw data stream. In terms of the Huffman encoding there are two steps that are essential in its running. Firstly, a Huffman tree (same structure as a binary tree) from the original data needs to be built. Thereafter, the tree needs to be traversed in order to assign codes to data symbols.

Inter-Subsystem and Inter-Sub-Subsystem interactions:

Various sources expressed different opinions on whether one should encrypt or compress data first. From these sources it was discovered that encryption turns the raw data into high-entropy data, making it indistinguishable from a random stream. This means that encryption shuffles the data in such a way that without the key, one can’t find the patterns within the data necessary to compress it. However, there are methods with much more complex implementations that can allow one to encrypt the data before compressing it, resulting in an increase in security of the data itself.

Although, for this particular project this level of security is not necessary, and the implementation of encryption before compression will lead to complexities out of the scope of this project design. Therefore, it has been decided that the data will be compressed using the Gzip compression algorithm first and only then encrypted. Once encrypted, the compressed encrypted data will be sent to a UCT server where it will be decrypted and then decompressed back into its original raw form.

The following diagram depicts the compression and encryption procedure of the data:



Validation using Simulated Data:

Simulation validation is the process of determining the degree to which a simulation model and its associated data are an accurate representation of the real world from the perspective of the intended uses of the model. In the case of this project, instead of using data directly produced from the Sharc Bouy in the South Atlantic, various .csv files containing the exact data being collected by the Sharc Buoy were provided on which simulation tests were run. This ensured that the compression/decompression and encryption/decryption algorithms would work on real data produced by an IMU (Inertial Measurement Unit) on a Sense Hat B, configured with the Raspberry Pi.

The data being used is all the .csv files containing all of the raw data from 9 different occasions captured by the buoy. For the purpose of testing the functionality of the compression/decompression and encryption/decryption algorithms and the system as a whole, the passing of each of the 9 .csv files into the system was automated. This automation aspect is imperative, as the system was designed in such a way to bring about seamless compression, encryption, decryption and decompression without the need of an individual to have to physically input the incoming files into the system. This results in a much more efficient system and a significantly improved runtime, especially in the case of multiple .csv files incoming at different times of the day in quick succession.

From the previous writeup (milestone 1), it was specified that the zlib library would be best to use for the compression and decompression subsystem (especially considering the type of data

being handled) and Fernet encryption with RSA (Rivest-Shamir-Adleman) encryption on the keys was used. This type of encryption makes use of both symmetric and asymmetric encryption/decryption simultaneously, thus being a form of double encryption, resulting in an even safer and efficient transmission of the compressed data.

Experimental Setup:

Various simulations were run on the data in order to determine/benchmark the responsiveness and efficiency of the system as a whole. In order to gain any understanding of the efficiency of the system as a whole, benchmark tests had to be run on each subsystem respectively.

Compression Benchmarking:

The zlib library has ten different compression levels (0-9) which is an integer input into the second argument of the *compress()* function part of the zlib import. Different levels have different compression performances relating to the compression ratio (ratio of the original raw data size and the compressed size) and the speed at which the compression takes place. Level 0 results in no compression, however does produce the quickest, most efficient results. As one moves up the level towards level 9, the compression ratio improves, meaning the data is compressed more. However, this is done at the expense of the runtime/performance of the algorithm. Therefore, tests had to be run for each incoming .csv file to determine what sacrifices had to be made in terms of the runtime performance and the compression performance in order to find a level that brings about the perfect balance.

This was done by creating a benchmarking python script called *compBenchmark.py* that automates the compression and decompression of each .csv file each with all levels of compression (0-9), and subsequently measured the compression ratio and runtime performance. From this, the optimum level of compression could be chosen for the system.

Encryption Benchmarking:

The fist benchmark revolved around ensuring accurate and reliable implementation of asymmetric encryption through implementing the algorithm and visually comparing output to input. For a more quantitative approach and in order to test the efficiency of the encryption and decryption algorithms, the time module was implemented within the python script of both the encryption and decryption algorithms. The timer was started as the program execution began and ended as the program had reached the end of its execution. Runtime performance is the only metric by which the encryption and decryption algorithms can be measured, as there is no reduction in space complexity across each algorithm.

System Benchmarking:

The system benchmarking/testing could only take place once the subsystem benchmarking was completed, and an optimal compression level was chosen based on the results the *compBenchmark.py* python script produced.

In order to complete the system benchmarking, all the respective python scripts needed to be combined in such a way that they work/run seamlessly and congruently with one another. The incoming data needs to be compressed by the *compression.py* script, and the output data needs to be automatically passed into the encryption algorithm contained within *Encrypt_data.py.* From this point the encrypted data is sent along with its respective RSA encryption keys needed for decryption on the receiving end. The decryption python script, *Decrypt_data.py* then passes the compressed encrypted data into its argument which then decrypts the data and subsequently passes this decrypted but still compressed data to the decompression algorithm contained within *decompression.py* which outputs the data in its original raw form.

For accurate benchmarking on the runtime performance of this system as a whole, a timer is started at the beginning of execution within the main class and ends as the data is decompressed and outputted.

The results of the individual subsystem benchmarking and subsequently the benchmarking of the system as a whole is discussed in the **Discussion of Results** section below.

Discussion and Results:

Compression Benchmarking Results:

The way in which the compression benchmarking was done is mentioned in the section above: '*compBenchmark.py* that automates the compression and decompression of each .csv file each with all levels of compression (0-9), and subsequently measured the compression ratio and runtime performance. From this, the optimum level of compression could be chosen for the system.' As a result, a total of 9 tests were conducted.

The results of this automated testing is shown and discussed below:

```
Looking at file:  2018-09-19-03_57_11_VN100.csv
Raw data size of:  9071140
Compressed size of 0:  12095777 . Time took:  0.04065680503845215 . Ratio is:  1.333435158094793 .
Full compression and decompression time:  0.09084391593933105

Compressed size of 1:  5569145 . Time took:  0.12671518325805664 . Ratio is:  0.6139410261554777 .
Full compression and decompression time:  0.19053316116333008

Compressed size of 2:  5496437 . Time took:  0.1487109661102295 . Ratio is:  0.6059257160621487 .
Full compression and decompression time:  0.2115018367767334

Compressed size of 3:  5255801 . Time took:  0.266632080078125 . Ratio is:  0.5793980690409364 .
Full compression and decompression time:  0.3217442035675049

Compressed size of 4:  5191945 . Time took:  0.2246096134185791 . Ratio is:  0.5723586010137646 .
Full compression and decompression time:  0.2796206474304199

Compressed size of 5:  5080981 . Time took:  0.435870885848999 . Ratio is:  0.5601259599124255 .
Full compression and decompression time:  0.4925661087036133

Compressed size of 6:  5052101 . Time took:  0.566547155380249 . Ratio is:  0.5569422365876836 .
Full compression and decompression time:  0.6243131160736084

Compressed size of 7:  5041445 . Time took:  0.6572229862213135 . Ratio is:  0.5557675220534575 .
Full compression and decompression time:  0.7144951820373535

Compressed size of 8:  5031433 . Time took:  1.0363600254058838 . Ratio is:  0.5546638019036196 .
Full compression and decompression time:  1.092829942703247

Compressed size of 9:  5031433 . Time took:  1.0318970680236816 . Ratio is:  0.5546638019036196 .
Full compression and decompression time:  1.0890049934387207
```

```
Looking at file:  2018-09-19-04_22_21_VN100.csv
Raw data size of:  9067615
Compressed size of 0:  12091077 . Time took:  0.04151296615600586 . Ratio is:  1.3334351976787722 .
Full compression and decompression time:  0.08659911155700684

Compressed size of 1:  5577157 . Time took:  0.13085269927978516 . Ratio is:  0.6150632773888173 .
Full compression and decompression time:  0.19245076179504395

Compressed size of 2:  5504921 . Time took:  0.15042519569396973 . Ratio is:  0.6070969047538961 .
Full compression and decompression time:  0.20982909202575684

Compressed size of 3:  5265521 . Time took:  0.2726128101348877 . Ratio is:  0.5806952544853304 .
Full compression and decompression time:  0.3264758586883545

Compressed size of 4:  5201733 . Time took:  0.22916197776794434 . Ratio is:  0.5736605491080069 .
Full compression and decompression time:  0.28324294090270996

Compressed size of 5:  5093609 . Time took:  0.4418208599090576 . Ratio is:  0.5617363551496176 .
Full compression and decompression time:  0.4978199005126953

Compressed size of 6:  5066609 . Time took:  0.570584774017334 . Ratio is:  0.5587587254200801 .
Full compression and decompression time:  0.6262898445129395

Compressed size of 7:  5056245 . Time took:  0.6630589962005615 . Ratio is:  0.557615756734268 .
Full compression and decompression time:  0.7189249992370605

Compressed size of 8:  5047693 . Time took:  1.0278260707855225 . Ratio is:  0.5566726200880827 .
Full compression and decompression time:  1.0835819244384766

Compressed size of 9:  5047693 . Time took:  1.0266270637512207 . Ratio is:  0.5566726200880827 .
Full compression and decompression time:  1.0833261013031006
```

```
Looking at file:  2018-09-19-06_28_11_VN100.csv
Raw data size of:  9083563
Compressed size of 0:  12112341 . Time took:  0.03977394104003906 . Ratio is:  1.3334350188356705 .
Full compression and decompression time:  0.08943676948547363

Compressed size of 1:  5584853 . Time took:  0.12527227401733398 . Ratio is:  0.6148306562083623 .
Full compression and decompression time:  0.18808722496032715

Compressed size of 2:  5510149 . Time took:  0.1470348834991455 . Ratio is:  0.6066065705714817 .
Full compression and decompression time:  0.207139253616333

Compressed size of 3:  5270269 . Time took:  0.2667989730834961 . Ratio is:  0.5801984309460946 .
Full compression and decompression time:  0.32111191749572754

Compressed size of 4:  5205933 . Time took:  0.22779202461242676 . Ratio is:  0.5731157476421972 .
Full compression and decompression time:  0.2828540802001953

Compressed size of 5:  5097297 . Time took:  0.438647985458374 . Ratio is:  0.561156123428659 .
Full compression and decompression time:  0.49512290954589844

Compressed size of 6:  5070377 . Time took:  0.5700101852416992 . Ratio is:  0.5581925286366154 .
Full compression and decompression time:  0.626471996307373

Compressed size of 7:  5060477 . Time took:  0.663459062576294 . Ratio is:  0.5571026479367183 .
Full compression and decompression time:  0.7207980155944824

Compressed size of 8:  5051833 . Time took:  1.0412399768829346 . Ratio is:  0.5561510389700606 .
Full compression and decompression time:  1.098294973373413

Compressed size of 9:  5051833 . Time took:  1.028306245803833 . Ratio is:  0.5561510389700606 .
Full compression and decompression time:  1.0848610401153564
```

```
Looking at file:  2018-09-19-06_53_21_VN100.csv
Raw data size of:  9080971
Compressed size of 0:  12108885 . Time took:  0.030538082122802734 . Ratio is:  1.3334350478599701 .
Full compression and decompression time:  0.075347900390625

Compressed size of 1:  5583457 . Time took:  0.12566804885864258 . Ratio is:  0.6148524205175856 .
Full compression and decompression time:  0.18610501289367676

Compressed size of 2:  5508197 . Time took:  0.15088987350463867 . Ratio is:  0.6065647605305644 .
Full compression and decompression time:  0.21075677871704102

Compressed size of 3:  5267653 . Time took:  0.26886415481567383 . Ratio is:  0.5800759632422569 .
Full compression and decompression time:  0.323972225189209

Compressed size of 4:  5203109 . Time took:  0.23002123832702637 . Ratio is:  0.5729683532741158 .
Full compression and decompression time:  0.28491711616516113

Compressed size of 5:  5094221 . Time took:  0.4381067752838135 . Ratio is:  0.5609775650643527 .
Full compression and decompression time:  0.49506092071533203

Compressed size of 6:  5067177 . Time took:  0.5693798065185547 . Ratio is:  0.5579994694399971 .
Full compression and decompression time:  0.6256759166717529

Compressed size of 7:  5057365 . Time took:  0.6588101387023926 . Ratio is:  0.5569189682468978 .
Full compression and decompression time:  0.7151060104370117

Compressed size of 8:  5048913 . Time took:  1.024902105331421 . Ratio is:  0.5559882307739998 .
Full compression and decompression time:  1.0812199115753174

Compressed size of 9:  5048913 . Time took:  1.0250258445739746 . Ratio is:  0.5559882307739998 .
Full compression and decompression time:  1.081603765487671
```

```
Looking at file:  2018-09-19-08_59_11_VN100.csv
Raw data size of:  9141936
Compressed size of 0:  12190181 . Time took:  0.038694143295288086 . Ratio is:  1.333435390490592 .
Full compression and decompression time:  0.08681201934814453

Compressed size of 1:  5570297 . Time took:  0.1258070468902588 . Ratio is:  0.6093126226217291 .
Full compression and decompression time:  0.18644499778747559

Compressed size of 2:  5484533 . Time took:  0.14749789237976074 . Ratio is:  0.5999312399474247 .
Full compression and decompression time:  0.20798587799072266

Compressed size of 3:  5250617 . Time took:  0.26334095001220703 . Ratio is:  0.574344099543029 .
Full compression and decompression time:  0.31702494621276855

Compressed size of 4:  5174405 . Time took:  0.22663283348083496 . Ratio is:  0.5660075721378929 .
Full compression and decompression time:  0.28155088424682617

Compressed size of 5:  5063573 . Time took:  0.43186020851135254 . Ratio is:  0.5538841006981453 .
Full compression and decompression time:  0.488523006439209

Compressed size of 6:  5026533 . Time took:  0.5602948665618896 . Ratio is:  0.5498324424935812 .
Full compression and decompression time:  0.616919994354248

Compressed size of 7:  5013625 . Time took:  0.6480221748352051 . Ratio is:  0.5484204877391398 .
Full compression and decompression time:  0.7040798664093018

Compressed size of 8:  5002585 . Time took:  1.0377230644226074 . Ratio is:  0.5472128660712566 .
Full compression and decompression time:  1.0942962169647217

Compressed size of 9:  5002585 . Time took:  1.0334229469299316 . Ratio is:  0.5472128660712566 .
Full compression and decompression time:  1.0896220207214355
```

```
Looking at file:  2018-09-19-09_49_31_VN100.csv
Raw data size of:  9132768
Compressed size of 0:  12177957 . Time took:  0.04554295539855957 . Ratio is:  1.333435492941461 .
Full compression and decompression time:  0.09053182601928711

Compressed size of 1:  5567045 . Time took:  0.12444114685058594 . Ratio is:  0.6095682053896475 .
Full compression and decompression time:  0.18410420417785645

Compressed size of 2:  5483341 . Time took:  0.1464850902557373 . Ratio is:  0.6004029665485864 .
Full compression and decompression time:  0.20705509185791016

Compressed size of 3:  5248545 . Time took:  0.2634420394897461 . Ratio is:  0.574693783965606 .
Full compression and decompression time:  0.31679725646972656

Compressed size of 4:  5174149 . Time took:  0.22617602348327637 . Ratio is:  0.5665477322975904 .
Full compression and decompression time:  0.2806107997894287

Compressed size of 5:  5061337 . Time took:  0.43486523628234863 . Ratio is:  0.554195288876275 .
Full compression and decompression time:  0.4910151958465576

Compressed size of 6:  5025085 . Time took:  0.557715892791748 . Ratio is:  0.5502258460961671 .
Full compression and decompression time:  0.6141777038574219

Compressed size of 7:  5013249 . Time took:  0.643359899520874 . Ratio is:  0.5489298534683023 .
Full compression and decompression time:  0.6999249458312988

Compressed size of 8:  5002581 . Time took:  1.0245780944824219 . Ratio is:  0.547761751968297 .
Full compression and decompression time:  1.080902099609375

Compressed size of 9:  5002581 . Time took:  1.0274369716644287 . Ratio is:  0.547761751968297 .
Full compression and decompression time:  1.0839190483093262
```

```
Looking at file:  2018-09-19-09_24_21_VN100.csv
Raw data size of:  9152768
Compressed size of 0:  12204621 . Time took:  0.04062700271606445 . Ratio is:  1.3334349783584594 .
Full compression and decompression time:  0.08935928344726562

Compressed size of 1:  5560561 . Time took:  0.12862515449523926 . Ratio is:  0.6075277992406232 .
Full compression and decompression time:  0.1892850399017334

Compressed size of 2:  5472365 . Time took:  0.14844584465026855 . Ratio is:  0.5978918071560428 .
Full compression and decompression time:  0.2086639404296875

Compressed size of 3:  5238053 . Time took:  0.26332902908325195 . Ratio is:  0.572291682690963 .
Full compression and decompression time:  0.31713414192199707

Compressed size of 4:  5158381 . Time took:  0.226517915725708 . Ratio is:  0.5635869935739658 .
Full compression and decompression time:  0.28115081787109375

Compressed size of 5:  5047405 . Time took:  0.42983293533325195 . Ratio is:  0.5514621369185803 .
Full compression and decompression time:  0.4861578941345215

Compressed size of 6:  5008697 . Time took:  0.555239200592041 . Ratio is:  0.5472330337663972 .
Full compression and decompression time:  0.6116092205047607

Compressed size of 7:  4995117 . Time took:  0.6395089626312256 . Ratio is:  0.5457493296017117 .
Full compression and decompression time:  0.6956968307495117

Compressed size of 8:  4982485 . Time took:  1.0314960479736328 . Ratio is:  0.544369200661483 .
Full compression and decompression time:  1.0877997875213623

Compressed size of 9:  4982485 . Time took:  1.0319139957427979 . Ratio is:  0.544369200661483 .
Full compression and decompression time:  1.0880401134490967
```

```
Looking at file:  2018-09-19-11_55_21_VN100.csv
Raw data size of:  9140323
Compressed size of 0:  12188029 . Time took:  0.041944026947021484 . Ratio is:  1.3334352626269335 .
Full compression and decompression time:  0.08761024475097656

Compressed size of 1:  5545445 . Time took:  0.12722468376159668 . Ratio is:  0.606701207386216 .
Full compression and decompression time:  0.1876087188720703

Compressed size of 2:  5452609 . Time took:  0.1506791114807129 . Ratio is:  0.5965444547200356 .
Full compression and decompression time:  0.21053171157836914

Compressed size of 3:  5207165 . Time took:  0.2609517574310303 . Ratio is:  0.5696915743568362 .
Full compression and decompression time:  0.3144567012786865

Compressed size of 4:  5134169 . Time took:  0.22998499870300293 . Ratio is:  0.5617054233203794 .
Full compression and decompression time:  0.2844579219818115

Compressed size of 5:  5013841 . Time took:  0.4325399398803711 . Ratio is:  0.5485408994846244 .
Full compression and decompression time:  0.4888792037963867

Compressed size of 6:  4967425 . Time took:  0.5503358840942383 . Ratio is:  0.5434627419621822 .
Full compression and decompression time:  0.6065738201141357

Compressed size of 7:  4953761 . Time took:  0.6371331214904785 . Ratio is:  0.5419678276139694 .
Full compression and decompression time:  0.6931440830230713

Compressed size of 8:  4940453 . Time took:  1.0463287830352783 . Ratio is:  0.5405118615611286 .
Full compression and decompression time:  1.1023638248443604

Compressed size of 9:  4940453 . Time took:  1.076395034790039 . Ratio is:  0.5405118615611286 .
Full compression and decompression time:  1.132538080215454
```

```
Looking at file:  2018-09-19-12_20_31_VN100.csv
Raw data size of:  9158748
Compressed size of 0:  12212597 . Time took:  0.04177522659301758 . Ratio is:  1.3334352031522212 .
Full compression and decompression time:  0.09146595001220703

Compressed size of 1:  5541453 . Time took:  0.1257762908935547 . Ratio is:  0.6050448161691969 .
Full compression and decompression time:  0.1861281394958496

Compressed size of 2:  5447109 . Time took:  0.1491389274597168 . Ratio is:  0.5947438449010717 .
Full compression and decompression time:  0.2094731330871582

Compressed size of 3:  5196493 . Time took:  0.25925302505493164 . Ratio is:  0.5673802794879824 .
Full compression and decompression time:  0.312877893447876

Compressed size of 4:  5121037 . Time took:  0.22935724258422852 . Ratio is:  0.5591415988298838 .
Full compression and decompression time:  0.28387999534606934

Compressed size of 5:  4998833 . Time took:  0.4246490001678467 . Ratio is:  0.5457987270749234 .
Full compression and decompression time:  0.48108911514282227

Compressed size of 6:  4948073 . Time took:  0.5467178821563721 . Ratio is:  0.5402564848383207 .
Full compression and decompression time:  0.6028039455413818

Compressed size of 7:  4933277 . Time took:  0.6468329429626465 . Ratio is:  0.5386409801863749 .
Full compression and decompression time:  0.7033360004425049

Compressed size of 8:  4918277 . Time took:  1.0627200603485107 . Ratio is:  0.5370032017476625 .
Full compression and decompression time:  1.1187691688537598

Compressed size of 9:  4918277 . Time took:  1.1214489936828613 . Ratio is:  0.5370032017476625 .
Full compression and decompression time:  1.1777441501617432
```

It can be seen that each 9 tests were run on each .csv file denoted at the top of each test. It can further be seen that it makes no sense to use a compression level of 0 as no compression whatsoever takes place even though this level does produce the quickest runtime. In fact, when looking at the compression ratio, it can be deduced that the decompressed file is larger than the original raw data file (seen by the fact that the compression ratio is larger than 1).

When looking on the opposite end of the spectrum, level 9 produces the best overall compression ratio however has a runtime that is significantly longer than all other levels. Thus, when looking at all the compression ratios produced and runtimes, the best compression level is level 6. Furthermore, level 6 compression is the default level when no second parameter is chosen, which is further testament to how this is the level that has the ideal balance between compression ratio and runtime performance.

Encryption Benchmarking Results:

The first benchmark that was tested was to see if asymmetric encryption had successfully been achieved. Upon attempting the implementation of RSA, the team discovered that it is only possible to encrypt files with a byte size equal to that of the private key, a feature that is unscalable and therefore undesirable. Thus, in order to ensure that the security benefits offered by asymmetric encryption were still achieved, a new strategy was adopted were Fernet (symmetrical) encryption was applied to the given data before its key was hen encrypted using an RSA public key, this ensures that only the owner of the private key is able to access and decrypt the original encrypted data, as they will be the only ones capable of discovering the cipher. Further exploration will be required into how the public keys will be exchanged across devices shall be investigated in future.

In order to investigate the run time of the encryption and decryption algorithms, the time library was used to test execution time of the algorithm over each of the original files as well as each of the compressed files. Each program was executed for both the compressed and original data 5 times, and the average runtimes for each were tabulated below:

File Name	Normal Runtime (s)	Compressed FIle Runtime (s)
2018-09-19-03_57_11_VN100	0.198864	0.129625
2018-09-19-04_22_21_VN100	0.214785	0.163206
2018-09-19-06_28_11_VN100	0.163930	0.133643
2018-09-19-06_53_21_VN100	0.172807	0.123669
2018-09-19-08_59_11_VN100	0.181712	0.127630
2018-09-19-09_24_21_VN100	0.172376	0.127685

2018-09-19-09_49_31_VN100	0.167899	0.153588
2018-09-19-11_55_21_VN100	0.194921	0.129654
2018-09-19-12_20_31_VN100	0.173863	0.166554

Decryption Runtimes:

File Name	Normal Runtime (s)	Compressed FIle Runtime (s)
2018-09-19-03_57_11_VN100	0.191185	0.161568
2018-09-19-04_22_21_VN100	0.183558	0.198857
2018-09-19-06_28_11_VN100	0.166305	0.153796
2018-09-19-06_53_21_VN100	0.155419	0.119736
2018-09-19-08_59_11_VN100	0.161690	0.1194896
2018-09-19-09_24_21_VN100	0.165598	0.128911
2018-09-19-09_49_31_VN100	0.199707	0.141125
2018-09-19-11_55_21_VN100	0.2576878	0.164818
2018-09-19-12_20_31_VN100	0.248496	0.1502571

For each average above, the data conveys that the runtime of the encryption algorithm is always faster for the compressed data set. This is as expected as there are less characters to iterate through.

System Benchmarking Results:

To perform a benchmark on the system as a whole, a main script was created, consisting of 2 functions, one meant for the sending side of the data transfer, that compresses and encrypts the input file, and another that would be on the receiving end to decrypt and then decompress the original files. We are able to successfully implement both the send and receive side algorithms, yet further developments need to be undergone to ensure proper implementation of the sending and receiving of private keys used for multiple file transfers. Using the time module, the runtimes for the send function and receive function were recorded for each and tabulated:

File Name	Send Side function (s)	Receive Side function (s)
2018-09-19-03_57_11_VN100	1.32733	1.749505
2018-09-19-04_22_21_VN100	1.24379	1.639162
2018-09-19-06_28_11_VN100	1.45622	1.484783
2018-09-19-06_53_21_VN100	1.2887511	1.020137
2018-09-19-08_59_11_VN100	1.2310569	1.154045
2018-09-19-09_24_21_VN100	1.309508	1.654062
2018-09-19-09_49_31_VN100	1.365545	1.239503
2018-09-19-11_55_21_VN100	1.2887511	1.05045
2018-09-19-12_20_31_VN100	1.258875	1.353736

As conveyed above, the system was successfully able to send these large amounts of data within acceptable time rates, as well as decrypt them into the original data set as desired.

Validation using a IMU:

Hardware based validation is only completed once simulation based validation has proven to be successful, and the attention of the project can shift to the implementation of hardware (‘real life’) application. In the case of this project, the simulation based validation, as mentioned above, was done by running pre-existing .csv files provided through the compression/decompression and encryption/decryption subsystems separately, and then through the system (combination of compression/decompression and encryption/decryption algorithms) as a whole. Thereafter, the same tests can be run on data being produced by the IMU part of the Sense Hat B, and modify the algorithms in each subsystem in such a way that it interacts with the IMU and the data it produces. This type of hardware validation shows that the designed algorithms will be able to withstand real-life/realistic volumes of data, and adequately compress and encrypt incoming data from the Sharc Buoy itself.

The Inertial Measurement Unit (IMU) Module:

The IMU provided for the sake of the project is the Sense Hat B developed for the raspberry Pi, built with multi powerful sensors onboard,such as the ICM20948 (3-axis accelerometer, 3-axis gyroscope, and 3-axis magnetometer) which can detect movement, orientation, and magnetism. The SHTC3 digital temperature and humidity sensor, LPS22HB barometric pressure sensor and TCS34725 color sensor, identifies the color of nearby objects. Despite being slightly

different to the MPU-6050 found on the actual SHark buoy located in Antarctica, not only in terms of physical dimension and in the number and type of readings produced, it has still served as an integral addition in order to test the implementation of the compression and encryption algorithms in a manner that is more aligned with how data would be collected on the buoy itself.

In order to ensure that the testing applied to this IMU is similar to that of the buoy, steps were taken in order to ensure that the data produced by the sense hat resembled the data given to us at the beginning of the semester. As such, a python script was created that constantly printed out the temperature , humidity , roll , pitch, yaw and cartesian acceleration and rotation, as well as wrote each reading (taken every 10 mS) to a .csv file. An if statement was included that would close the file once 100 000 lines of data were appended to it. This was roughly the same size as the data used in preliminary testing, and due to the nature of compression, it was meritorious to send larger groups of data at a time, as compression finds repetition within the set and reduces it. Validation tests were necessary in order to determine if the module worked correctly. These included:

- Ensuring there were physical responses when the hat was plugged in
- Executing the script and adjusting the horizontal and vertical position of the Raspberry pi, before examining the readings of the cartesian acceleration readings during the time of movement
- Executing the script and comparing the temperature and humidity reading to the expected values given by commercial sources such as the weather app (while not completely accurate, it did provide a basic indication for humidity, but was found to be inaccurate for temperature, accredited to the excess heat radiating off of the raspberry Pi)
- Executing the script and placing a hand over the sense hat in order to see whether the temperature reading increased positively over time

Executing the script and rotating the Raspberry Pi, while watching the change in value of the pitch and yaw readings.

Experiment Setup:

Overall Functionality:

As stated above, the IMU (Inertial Measurement Unit) contained within the Sense Hat B simulates the exact data provided by the Sharc Buoy. Furthermore, the data obtained when running the python script necessary for the operation of the IMU is exactly the same units of data used when testing the developed compression and encryption algorithms as subsystems and as a system as a whole. Therefore, in order to test the accuracy of the type of data being outputted by the IMU and stored within its respective files, the Raspberry Pi along with the Sense Hat B had to be put into a situation whereby it measured and outputted a change in the Roll, Pitch, Yaw, Acceleration, Gyroscope and Magnetic, where the latter three measurements

all had sub-measurements in the X, Y and Z directions. Due to the nature of the way in which the SHTC3.py and ICM20948.py python scripts were written, the IMU captured the data of the various measurements described above and displayed its results in the terminal every 10ms. This is exactly the same rate at which measurements were taken on the Sharc Buoy itself, enabling even further accuracy in the Acceptance Test Procedures and the experiment setup as a whole.

It is to be noted that the system isn't only comprised of the Raspberry Pi and the compression and encryption algorithms that run on it, but also the IMU producing the data to be compressed and encrypted. In the previous Acceptance Tests, the compression and encryption algorithms were tested by inputting preset data of the same nature of the data outputted by the IMU in this experiment. Thus, the best way to test the overall functionality of the system is to simulate the measurements produced by the IMU using the SHTC3.py and ICM20948.py python scripts. The SHTC3.py and ICM20948.py python scripts were combined into one python script that printed readings into the terminal. Each time a new reading was taken it would write it to a .csv file as a comma separated list. Once the csv file reached 100 000 lines it would automatically close itself and a new file would be opened. The simulated data is then sent to a .csv file, stored in the same format in which the preset data was stored, and then finally passed through the compression and encryption subsystems where it was assessed whether or not the output resembles the .csv file input containing the simulated data. The file can be found here for reference.

As the scripts were executed on the raspberry pi, an important aspect of the acceptance test was ensuring that the runtime of the compression and encryption algorithm was still acceptable for file sizes of large size, as this part of the data transportation would take place in the Pi.

Lastly, when determining/benchmarking the responsiveness and efficiency of the system as a whole various simulations had to be run using the data outputted by the IMU. In order to gain any understanding of the efficiency of the system as a whole, benchmark tests had to be run on each subsystem respectively - shown below.

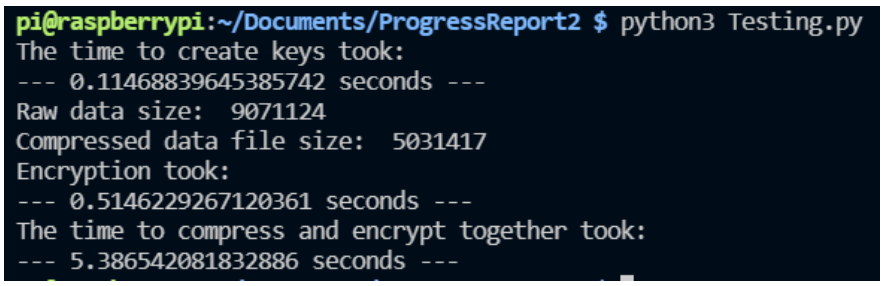
Discussion and Results:

Overall System Results:

The system was successful in meeting all of the validation testing steps outlined above, and it can be concluded that the SenseHat interacts correctly with the Pi. A demonstration video of the execution of the python scripts was uploaded to google drive and can be found here. As mentioned in the experiment setup, it was important to ensure that the Send Data script (that generates the necessary keys, encrypts and compresses the data) could execute on the Raspberry Pi, and was still effective for large sets of data. In order to prove this to be true, one of the sample CSV files was downloaded onto the Pi, and the average runtime for 5 executions was taken and found to be 5.446 seconds. As the 130000 lines of data in the CSV file

represented over 30 minutes of sampling from the IMU, despite the fact that this runtime was far longer than when executed on a laptop, it can still be viewed as acceptable in being able to execute efficiently.

Example output:



Consolidation of ATP's and Future Plan:

In Milestone 1, it was decided that running performance tests on both the encryption and compression algorithms and running performance tests on each algorithm individually and comparing runtime to refresh rate is necessary for adequate benchmarking. It was further discussed that the ultimate acceptance test would be to implement the algorithms on the raw data provided and to see that this data is transmitted safely and securely from the RPi used in simulation to a PC and that the data received is in its original raw form. This was done after the benchmarking of the compression and encryption algorithms individually took place and the system as a whole was tested and produced the raw data in its original form with no sign of any loss in the data. However, in the case of this submission, the data passed into the compression and encryption subsystems originated from the .csv file produced by the IMU and the combined SHTC3.py and ICM20948.py python scripts.

It was also discussed that an ATP needed to be run in order to determine if it's more efficient to encrypt and compress each individual data packet produced every 10 milliseconds by the IMU or whether it's more efficient to store groups of the data before encryption and compression. This ATP was only partially met as while the team has seen through empirical evidence that it makes sense to send larger groups at a time, the optimal size of the data being sent has still not been discovered, as adequate tests for this were not designed in time for this submission, however it is planned for this specific test to take place in further iterations of the design when improvements are made to the existing compression and encryption algorithms.

Furthermore, the system has never been tested for data sizes larger than the .csv files dealt with during these benchmark tests. Therefore, it would be useful to run further testing on the system as a whole to evaluate how the time complexity works for significantly larger data sets.

After further analysis into how 2 computers (or in this case a Pi and a computer can communicate), another ATP being examined by the team is to see if we can successfully transfer the necessary keys between 2 devices through the use of sockets, and to test whether our algorithm can successfully double encrypt a file on 1 device using the RSA public key from another, and successfully decrypt the data on the receiving device. In order to do this, the team looked at using python sockets to connect the raspberry pi and a laptop together, with the goal to be that the laptop would act as a server, listen for the client (the PI) to connect, before sending the pi an RSA public key that has been generated. The pi would then be able to use that key to encrypt the symmetric key used to encrypt the data, before then sending the encrypted and compressed data through the socket back to the Server (laptop). This was done as a means to demonstrate that our double encryption algorithm worked not only in practice (as previously conveyed in our earlier demo), but could go a step further and show how the keys could also be transported between devices.

As the team were able to achieve the original goals of implementing encryption, compression, decryption and decompression locally, we attempted to use python socketing as a means to transport the necessary keys used between the client (raspberry Pi) and server (Taines laptop) in order to fully demonstrate double encryption (with Taines laptop generating an RSA key pair, sending that key to the Pi, which would encrypt the symmetric key, send the encrypted key and encrypted and compressed data to the laptop, that would then decompress it and decrypt it) Unfortunately due to time constraints and a lack of understanding in encoding, the team was able to successfully transfer the keyfiles (validated through comparing md5sums of the files being sent between devices) but faced a padding error that prevented decryption. This will be solved easily in future iterations of the algorithm.

ATP Requirement:	**Was the requirement met?**
Compression runtime ATP.	This requirement was met in the benchmarking process.
Compression level ATP.	This requirement was met in the benchmarking process.
Encryption/decryption runtime ATP.	This requirement was met in the benchmarking process.
System runtime ATP.	This requirement was met in the final benchmarking process.
Individual packet testing.	This requirement was not met in the benchmarking design, however will be met in further iterations of the design.

Use of RSA Public Key on another device.	This requirement was not met in the benchmarking design, however will be met in further iterations of the design.

Conclusion:

This project entailed designing and implementing both a compression/decompression and encryption/decryption algorithms that will take in data incoming from the Sharc Buoy measurement device in the South Atlantic. This was a particularly daunting task for us students, as it was the first time we were handed a project whereby the implementation had to be completely handled by the students, and the scope of the algorithms used and the way in which they were implemented into the system as a whole was up to us. There were walls that were hit and had to be overcome over the course of this semester and the development of this compression/encryption system. Some of these difficulties faced entailed choosing not only an adequate form compression and encryption to be used, but algorithms that would enable us to go the extra mile, and design an implementation that would perform its duty as efficiently and safely as possible (‘safely’ referring to the secure encryption of the IMU data).

The project began with the paper design whereby each subsystem (compression and encryption) was divided between the two group members. As mentioned above, the scope in terms of the compression and encryption that can be used is very broad. This meant that each group member had to extensively research their respective subsystems, in order to gain a general understanding of how it could be implemented using python and in such a way that it interfaces with a Raspberry Pi, and further conclude which types of compression and encryption would be best to use. As mentioned in the report above, it was eventually decided that Z-lib compression would be used, due to its simple implementation and efficient results, and RSA Double Encryption would be implemented to encrypt the compressed incoming data.

Thereafter, the compression and encryption system as a whole was designed and tested using simulated data. The simulated data was provided in the form of a .csv file containing all the relevant pieces of data currently being collected by the Sharc Buoy. After various tests, such as investigating the ideal compression level to use based on execution time and compression size and showing that the encryption subsystem was successfully able to send large amounts of data within acceptable time rates, as well as decrypt them into the original data set as desired, the results proved that the system was ready for hardware based validation with the use of a Sense Hat B containing an IMU similar to the one used on the Sharc Buoy. In the case of this type of hardware validation testing, the only notable difference compared to that of the simulation validation is that the data being passed into the system is produced by the IMU on the Sense Hat B, attached to the Raspberry Pi. The data produced by the IMU is sent straight to a .csv file, displaying the exact same data in the same format as the provided simulated data. The system was able to successfully compress, encrypt, decrypt and then decompress the IMU

module data. At this point of the project, as a group we had successfully implemented a working system.

This project proved to be extremely valuable in not only giving us experience in figuring out a real-life engineering scenario using tools we had no previous knowledge of (compression and encryption), but also opened our eyes to the process of implementing a successful/efficient solution to any problem we may face in the future. Both group members have vastly expanded their knowledge on the subject of z-lib compression and RSA Double Encryption, and how to put together various subsystems into one system that works in a seamless manner.

References:

- *Understanding zlib* (2021). Available at: https://www.euccas.me/zlib/ (Accessed: 25 October 2021).

- *Python zlib Library Tutorial* (2017). Available at: https://stackabuse.com/python-zlib-library-tutorial/ (Accessed: 25 October 2021).

- How do you compress a string, a. and Pietzcker, T. (2011) *How do you compress a string, and get a string back using zlib?*, *Stack Overflow*. Available at: https://stackoverflow.com/questions/4845339/how-do-you-compress-a-string-and-get-a-string-back-using-zlib (Accessed: 25 October 2021).

- zlib TypeError: a bytes-like object is required, n., Барашков, B. and Rublev, R. (2018) *zlib TypeError: a bytes-like object is required, not 'str'*, *Stack Overflow*. Available at: https://stackoverflow.com/questions/51348407/zlib-typeerror-a-bytes-like-object-is-required-not-str/51374073 (Accessed: 25 October 2021).

- *How to convert strings to bytes in Python* (2021). Available at: https://www.educative.io/edpresso/how-to-convert-strings-to-bytes-in-python (Accessed: 25 October 2021).

- *Sense HAT (B) - Waveshare Wiki* (2021). Available at: https://www.waveshare.com/wiki/Sense_HAT_(B) (Accessed: 25 October 2021).

- Python, R. (2021) *Socket Programming in Python (Guide) – Real Python*, *Realpython.com*. Available at: https://realpython.com/python-sockets/ (Accessed: 25 October 2021).

- *SHARC Buoy Robust firmware design for a novel, low-cost autonomous platform for the Antarctic Marginal Ice Zone in the Southern Ocean (2021). Available at: https://vula.uct.ac.za/access/content/group/d1ec1eeb-17ac-4770-a46b-86feee48f0d6/11.%20Additional%20resources/Jamie_MSc_Thesis.pdf (Accessed: 25 October 2021).*

- Altshuler, M. and de Buys, T., 2021. *EEE3097S Milestone 2 Submission Taine de Buys (DBYTAI001) and Mike Altshuler (ALTMIC003):*. Milestone 2. Cape Town, p.1 - p.12. (Accsessed: 25 October 2021).

- Altshuler, M. and de Buys, T., 2021. *EEE3097S Milestone 1 Submission Taine de Buys (DBYTAI001) and Mike Altshuler (ALTMIC003):*. Cape Town: UCT, pp.1-10.

- Altshuler, M. and de Buys, T., 2021. *EEE3097S Milestone 3 Submission Taine de Buys (DBYTAI001) and Mike Altshuler (ALTMIC003):*. Cape Town: UCT, pp.1-17.

- *Verification and Validation of Simulation Models* (2013). Available at: https://www.mitre.org/publications/systems-engineering-guide/se-lifecycle-building-blocks/other-se-lifecycle-building-blocks-articles/verification-and-validation-of-simulation-models (Accessed: 31 October 2021).

- Security?, I., Moore, R. and Mellor, J. (2010) *Is it better to encrypt a message and then compress it or the other way around?* Available at: https://stackoverflow.com/questions/4399812/is-it-better-to-encrypt-a-message-and-then-compress-it-or-the-other-way-around (Accessed: 4 September 2021).

- https://www.techopedia.com/definition/884/data-compression (Accessed: 4 September 2021).

- *RSA Encryption Implementation in Python* (2021). Available at: https://www.pythonpool.com/rsa-encryption-python/#:~:text=Encryption-,What%20is%20RSA%20Encryption%20in%20python%3F,key%20and%20the%20private%20key. (Accessed: 4 September 2021).

- *Cryptography 101: Data Integrity and Authenticated Encryption* (2020) (Accessed: 4 September 2021).

- https://blog.appcanary.com/2016/encrypt-or-compress.html (Accessed: 4 September 2021).

- https://security.stackexchange.com/questions/19969/encryption-and-compression-of-data (Accessed: 4 September 2021).

- https://linuxreviews.org/Comparison_of_Compression_Algorithms (Accessed: 4 September 2021).

- https://www.educba.com/c-vs-python/ (Accessed: 4 September 2021).