CHAPTER 5
IMPLEMENTATION AND TESTING

5.1. Implementation

This project uses Arduino IDE for programming and the program is divided by the function to save memory usage. Below is the complete explanation of the programs.

5.1.1. Code for Compression (Modified Huffman)

Here is the code for the Compression process :

```
1. char* StrCompress(char saved[])
2. {
3.     char *s = saved;
4.     char *r, *p;
5.     int count, i;
6.
7.     while (*s)
8.     {
9.         /*Start from first character*/
10.         count = 1;
11. /*Check if the character at the point is the same as following one*/
12.         while (*s && *s == *(s+1))
13.         {
14. /*If yes, then increase the count and increment the pointer to next
    character*/
15.             count++;
16.             s++;
17.         }
18.         if (count > 1) /*If more than one character of a kind is found*/
19.         {
20. /*Assign the value of count to second occurence of a particular
    character*/
21.             *(s - count + 2) = count + '0';
22. /*Using array shift, delete all other character occurences except the
    first and second one*/
23.             for (i = 0; i < count - 2; i++)
24.             {
25.                 p = s + 1;
26.                 r = s;
27.
28.                 while (*r)
29.                     *r++ = *p++;
30.                 s--;
31.             }
32.         }
33.         s++;
34.     }
35.     return saved;
36. }
```

In the compression process, line 1 needed to get the input char array compressed. Then, line 7 is for looping a pointer as long as an inputted char array. Then, in lines 12 until 16, it is declared that if the first index and next index character are the same, the loop will count the number of occurrences. Finally, in line 21 until 30, if the character number of occurrences is more than one, the first character found will append with the total occurrences of said character.

Here is the code for measuring compression time :

```
1.   Serial.println("Compressed String is : ");
2.   unsigned long time1 = micros();
3.   Serial.print(StrCompress(saved));
4.   Serial.println("\nWaktu Compress : ");
5.   Serial.print(micros() - time1);
```

The measurement of compression time is using the micros() function. Total compression time starts at line 2 and will stop after the compression process displaying the time result at line 5.

Here is the code for Decompression process :

```
1.   Serial.println("Decompressed String is : ");
2.   //String comp = "A4B3C2";
3.   String comp(saved);
4.   int len = comp.length();
5.   int x = comp.toInt();
6.   String simpan ="";
7.   String temp;
8.   int temp1;
9.   String decomp = "";
10. unsigned long time2 = micros();
11.    for(i=0; i<len; i++){
12.        if(i % 2 == 0){
13.            simpan = comp.charAt(i);
14.          }
15.          else{
16.            temp = comp.charAt(i);
17.            temp1 = temp.toInt();
18.              for(int x=0; x<temp1; x++){
19.                  decomp = decomp + simpan;
20.                }
21.            }
22.      }
23. Serial.println(decomp);
24. Serial.println("Waktu Decompress : ");
25. Serial.print(micros() - time2);
```

For the decompression process, the total decompression time starts measuring in line 10 before the looping process, and it will stop and show the time result after the loop, as shown in line 26. The decompression process is started in line 11 by looping as long as the length of the compressed input string. Then, in lines 12 and 13, the character will be saved in a simple variable

when the index is odd-numbered. The code line 15 until 19 starts when the index number is even-numbered, then the character will be temporarily saved in the temp variable (line 16). Then in the following line, the temp1 is used to store the converted character to integer value used next for looping. Finally, in line 19, the character will be appended in the decomp variable, as long as the number of occurrences saved in the temp1 variable.

5.1.2. Code for RSA Encryption

Here is the code for RSA Encryption process :

```
1. void encrypt(){
2.     long int pt, ct, key = e[0];
3.     long int k, len;
4.     int i = 0;
5.     len = msg.length();
6.
7.     while(i != len)
8.     {
9.       pt = m[i];
10.        pt = pt - 96; //Untuk mencegah character melebihi batas unsigned
   dan menjaga value tetap dalam range
11.        k = 1;
12.        for(j = 0; j < key; j++){
13.            k = k * pt;
14.            k = k % n;
15.          }
16.          temp[i] = k;
17.          ct = k + 96;
18.          c[i] = ct;
19.          i++;
20.      }
21.      c[i] = -1;
22.      Serial.println("\nThe Encrypted Message Is");
23.      for(i = 0; c[i] != -1; i++){
24.        Serial.print(c[i]);
25.      }
26.      Serial.println("\nThe Encrypted Message In HEX");
27.     for (int i=0; c[i] != -1 ;i++){
28.      Serial.println(c[i]&0xFF,HEX);
29.      }
30.      Serial.println("-----END OF HEX-----");
31.   }
```

In this RSA encryption process, line 2, the key used is E as a public key. Thus, the key used in encryption is E and N. Line 12 until line 15 is used to generate k, which is the modulo of message in integer (message is separated per character block (mi)) as long as the key index. Then, in line 17, k is added back with 96 to convert back to printable ASCII characters. From line 22

until 24, the encrypted text will appear on the serial monitor. Then, to insert the message into the RFID card, line 26 until 28 is used to convert the ASCII text to hexadecimal format.

Here is the code for RSA Decryption Process :

```
1.  void decrypt(){
2.     long int pt, ct, key = d[0];
3.     long int k;
4.     int i = 0;
5.     while(c[i] != -1)
6.     {
7.        ct = temp[i];
8.        k = 1;
9.        for(j = 0; j < key; j++)
10.        {
11.           k = k * ct;
12.           k = k % n;
13.        }
14.        pt = k + 96;
15.        m[i] = pt;
16.        i++;
17.     }
18.     m[i] = -1;
19.     Serial.println("\nThe Decrypted Message Is");
20.     for(i = 0; m[i] != -1; i++){
21.        Serial.print(m[i]);
22.      }
```

For the decryption process, in line 2, the key used is D as a private key. After that, the key used in decryption is E and N. Line 12 until line 15 is used to generate k, which is the modulo of ciphertext in integer (the ciphertext is separated per character block (ci)) as long as the key index. After that, the key D is not shared and used to calculate back from ciphertext to plaintext message.

Here is the code to measure time of RSA encryption and decryption :

```
1.   unsigned long time1 = micros();
2.   encrypt();
3.   Serial.println("\nWaktu Encrypt : ");
4.   Serial.print(micros() - time1);
5.   unsigned long time2 = micros();
6.   decrypt();
7.   Serial.println("\nWaktu Decrypt : ");
8.   Serial.print(micros() - time2);
```

The measurement of encryption and decryption time is using the micros() function. Total encryption time starts at line 1 and will stop after the encryption process displaying the time result at line 4. The decryption process will start measuring in line 5 and will stop after the decryption process is done, displayed in line 8.

5.1.3. Code for AES Encryption

Here is the code for AES encryption and decryption :

```
1. unsigned long time1 = micros();
2.   aes128_enc_single(key, msg);
3.   Serial.print("Encrypted: ");
4.   Serial.println();
5.   Serial.println(msg);
6.   Serial.println("\nWaktu Encrypt : ");
7.   Serial.print(micros() - time1);
8.   Serial.println("\nConvert to HEX :");
9.   Serial.println();
10.   for (int i=0; msg[i] != 0 ;i++){
11.   Serial.println(msg[i]&0xFF,HEX);
12.    }
13.   unsigned long time2 = micros();
14.   aes128_dec_single(key, msg);
15.   Serial.print("Decrypted: ");
16.   Serial.println();
17.   Serial.println(msg);
18.   Serial.println("\nWaktu Decrypt : ");
19.   Serial.print(micros() - time2);
```

In this AES encryption process, in line 1, the encryption time will start measuring in, and in line 7, the total time of encryption will be shown in output after the encryption process was done. After that, the ciphertext will be shown in text format in line 5 and hexadecimal format using the loop at line 11 until 12. Finally, the decryption process time is measured starting at line 13 and will be stopped after the decryption process ends at line 19.

5.1.4. Code for Random Compression

Here is the code for random compression :

```
1. char string[6] = {'A','B','C','D','E','F'};
2. const byte stlen = sizeof(string) / sizeof(string[0]); //menghitung size
   dari array
3. char notes[len+1]; // ditambah 1 untuk NULL
4. unsigned long time1 = micros();
5. void setup() {
6.   Serial.begin(9600);
7.   randomSeed(analogRead(A0));
8.   for (int n = 0; n < 16 ; n++)
9.   {
10.      notes[n] = string[random(stlen)];
11.      notes[n + 1] = '\0'; //untuk terminate string dibagian ujung
12.  }
13. }
```

The code in line 1, inside the array, is characters that determine the random input because the result is expected to be repeated character, so the input is limited. The purpose of limiting this

array is to make the result of compression more consistent. Line 7 is needed to ensure that every time the program runs, the output string is random. The line 8 until line 11 function is to loop the character randomly from the array to create a string with 16 characters.

5.2. Testing

5.2.1. RSA Encryption Testing without Compression

After testing RSA encryption using the same sized bytes of words, in this test that even with the same amount of character as plaintext. The ciphertext result size got an increase in size, as could be seen in the table down below :

Table 5.1. RSA Encryption Testing Comparison

Words	**Plaintext Size (in Bytes)**	**Encrypted Size (in Bytes)**	**Encryption Time (in microsecond)**
about	5	11	89260
brand	5	9	89260
cheap	5	9	89260
drive	5	8	89260
event	5	7	89260
additional	10	20	115248
basketball	10	24	115248
comparison	10	24	115248
determined	10	16	115248
electronic	10	24	115248
Standard Deviation		7.22341870431015	

Table 5.1 above shows that while the size of encrypted text may vary depending on the plaintext, the encryption time itself remains consistent per size of words. The RSA algorithm encrypted text size is closer to the original plaintext size than the AES encryption, indicating that this algorithm is better for encrypting short text. The time needed to complete encryption was consistent across the inputs but significantly slower when compared to the AES algorithm.

5.2.2. AES Encryption Testing without Compression

In this AES testing, the size of text is also increasing after being encrypted. However, interesting enough, the size is increased in a certain amount and is more consistent compared to the RSA algorithm before, as could be seen in the table below :

Table 5.2. AES Encryption Testing Comparison

Words	Plaintext Size (in Bytes)	Encrypted Size (in Bytes)	Encryption Time (in microsecond)
about	5	40	728
brand	5	40	728
cheap	5	40	728
drive	5	46	728
event	5	40	728
atmosphere	10	44	760
basketball	10	46	760
contribute	10	42	760
discovered	10	40	760
enterprise	10	46	760
Standard Deviation		2.7968235951204	

In Table 5.2, we could see that the encryption is increasing to around 40 bytes from any 5 and 10 bytes samples. That indicates that the encryption changes any plaintext input between 1 character until 15 characters into the 16-bit length of the ciphertext. However, the time result is showing that AES is consistently faster than the RSA algorithm. This encryption also has a lower standard deviation that is ≈2.80, compared to RSA, with ≈7.20. This result would be necessary to notice for the later test when combining the encryptions with the compression method.

5.2.3. Encryption Time without Compression

Below is the table of comparison between the two algorithms encryption times, using random input (source: http://www.yougowords.com/16-letters) :

Table 5.3. RSA and AES Encryption Time Comparison

Words	RSA Encryption Time (in microsecond)	AES Encryption Time (in microsecond)
acknowledgements	146428	660
agriculturalists	146428	660
biostatisticians	146428	660
biotechnologists	146428	660
cinematographers	146428	660
counterarguments	146428	660

differentiations	146428	660
diversifications	146428	660
electromagnetics	146428	660
experimentations	146428	660
familiarizations	146428	660
formularizations	146428	660
geochronologists	146428	660
governmentalists	146428	660
hypercatabolisms	146428	660
hospitalizations	146428	660
immunomodulatory	146428	660
insurrectionists	146428	660
microelectronics	146428	660
misapprehensions	146428	660
nationalizations	146428	660
neoconservatives	146428	660
objectifications	146428	660
ophthalmologists	146428	660
personifications	146428	660
photojournalists	146428	660
quadruplications	146428	660
quarterfinalists	146428	660
rationalizations	146428	660
reconfigurations	146428	660
standardizations	146428	660
superimpositions	146428	660
totalitarianisms	146428	660
transplantations	146428	660
ultracentrifuged	146428	660
ultrafiltrations	146428	660
vasoconstrictors	146428	660
videoconferences	146428	660
weatherboardings	146428	660
whatchamacallits	146428	660
MEAN	146428	660

UNIVERSITAS KATOLIK SOEGIJAPRANATA

This test uses words that use 16 characters to form. Whether the RSA and AES are using the exact words and length of the word in this testing, from Table 5.3 above, we could see that the average time for encryption using RSA is 146428 microseconds or 0.146428 seconds. This result is significantly slower than the AES method of encryption, which is only 660 microseconds or

0.00066 seconds. Furthermore, when comparing the AES algorithm results from testing samples using 5 bytes and 10 bytes plaintext, the encryption times are higher than this testing using 16 bytes plaintext. That shows this method only works optimally if the input is the same length as the AES 128-bit key length, which is also 16 bytes in size.

5.2.4. Decryption Time without Compression

The table below is the comparison between the two algorithms decryption times :

Table 5.4. RSA and AES Decryption Time Comparison

Words	RSA Decryption Time (in microsecond)	AES Decryption Time (in microsecond)
acknowledgements	64480	64480
agriculturalists	64480	64480
biostatisticians	64476	64476
biotechnologists	64480	64480
cinematographers	64480	64480
counterarguments	64480	64480
differentiations	64480	64480
diversifications	64476	64476
electromagnetics	64480	64480
experimentations	64480	64480
familiarizations	64480	64480
formularizations	64480	64480
geochronologists	64480	64480
governmentalists	64476	64476
hypercatabolisms	64480	64480
hospitalizations	64480	64480
immunomodulatory	64480	64480
insurrectionists	64480	64480
microelectronics	64476	64476
misapprehensions	64480	64480
nationalizations	64480	64480
neoconservatives	64476	64476
objectifications	64480	64480
ophthalmologists	64476	64476
personifications	64480	64480

photojournalists	64480	64480
quadruplications	64480	64480
quarterfinalists	64480	64480
rationalizations	64480	64480
reconfigurations	64480	64480
standardizations	64480	64480
superimpositions	64480	64480
totalitarianisms	64480	64480
transplantations	64480	64480
ultracentrifuged	64480	64480
ultrafiltrations	64480	64480
vasoconstrictors	64480	64480
videoconferences	64480	64480
weatherboardings	64480	64480
whatchamacallits	64480	64480
MEAN	64479.4	64479.4

Table 5.4 shows that the average decryption time of the two algorithms is the same, 64479.4 ms, between RSA and AES algorithms using the same input variable. That indicates that the limitation of decrypting time(s) is only the system this test was running on (Arduino UNO processing power).

5.2.5. Combining Encryption with Simple Compression Algorithms

Before conducting a test using a combination of Encryption and Compression algorithms, the limitation of the simple compression algorithms program must be stated. This limitation of usage is mainly because of Arduino software limitations. As we could see on the figures below :

```
COM3

Original String is :
iPhone 12 yang diumumkan Apple beberapa pekan lalu masih menyisakan euforia di kalangan pencinta gadget.
Compressed String is :
iPhone 12 yang diumumkan Ap2le beberapa pekan lalu masih menyisakan euforia di kalangan pencinta gadget.
Waktu Compress :
128960
```

Figure 5.1 Compression Test 1 Using Paragraph Input (source : https://tekno.kompas.com/read/2020/10/22/10250047/kelebihan-dan-kekurangan-iphone-12-di-mata-para-pengulas-gadget?page=all#page4)

```
COM3

Original String is :
Sebagai informasi, Starlink merupakan proyek yang dikembangkan SpaceX sejak 2015.
Compressed String is :
Sebagai informasi, Starlink merupakan proyek yang dikembangkan SpaceX sejak 2015.
Waktu Compress :
105040
```

Figure 5.2 Compression Test 2 Using Paragraph Input (source : https://tekno.kompas.com/read/2020/11/05/08240007/internet-buatan-elon-musk-diuji-coba-kecepatan-download-tembus-160-mbps)

Figure 5.1 and Figure 5.2 concluded that everyday words or paragraphs are not adequate when using this program. The program was made to be compatible for it to run on Arduino hardware. This program only supports the compression of repeating characters using techniques called Run Length Encoding. This method is included in the category of lossless compression method, which is a compression method that ensures the data compressed could be decompressed back into its original state without losing any original data in the process.

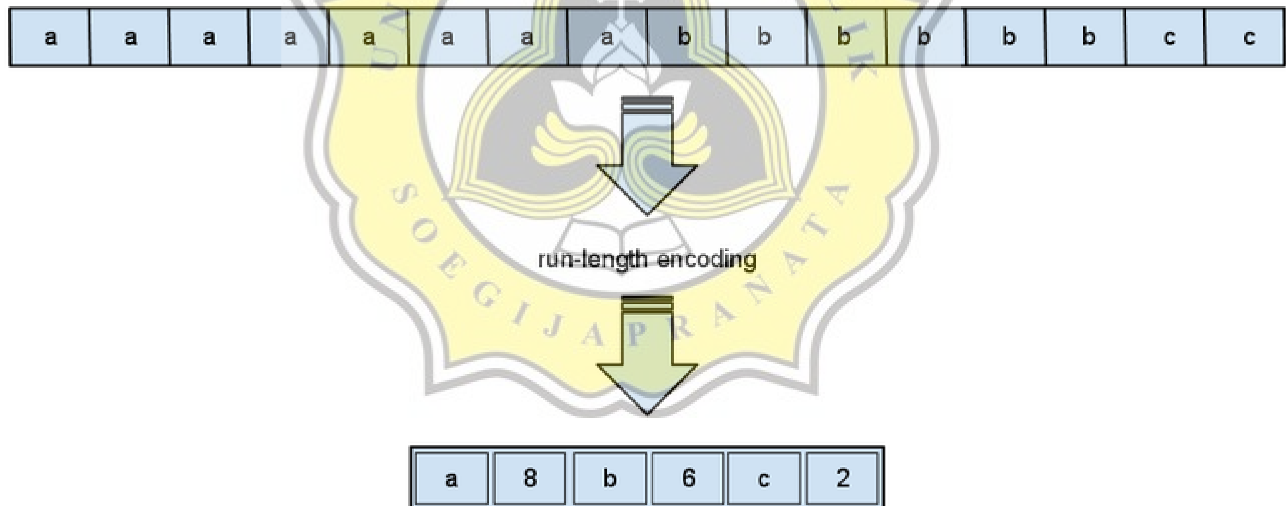

Figure 5.3 Simple Diagram of Run Length Encoding (source : https://thomaslock.blog/2018/01/04/run-length-encoding-tutorial/)

Figure 5.3 above concluded that this method is more effective when the input data are repeatable characters. Because of this, the testing method used in this test is different from the test previously (on RSA and AES test) and using repeatable characters as input is the only viable option because of this reason.

For the procedure of this comparison, first, the input text would be compressed then encrypted after it. However, it is impossible to do the other way around (encrypt text first, then compress after it) because the program would not read the already encrypted version of the text, primarily non-alphabetical characters.

5.2.6. Comparison of RSA and AES with Compression Method

With the knowledge from the previous sub-chapter explanation (see 5.2.4), this test was run using custom-made input of 16 alphabetical characters with random times of occurrence each. This comparison below was run using the same method as explained in sub-chapter 5.2.4. Below is the table time comparison between AES and RSA using compression:

Table 5.5. AES and RSA with Compression Comparison

Input	**AES with Compression Time (in microseconds)**		**RSA with Compression Time (in microseconds)**	
	Compression	***Encryption***	***Compression***	***Encryption***
AAABBBBCCCCCDDDD	29120	660	29120	138108
AABBBBBCCCCCDDDD	29120	660	29120	137068
AAABBBBBBCCCCDDD	29120	660	29120	137068
AAABBBBBBCCCCCDD	29120	660	29120	138108
AAAAAABBBBBBCCDD	29120	660	29120	138108
AAABBCCCCDDDEEEE	31200	672	31200	141236
AAAABBBCCCDDEEEE	31200	672	31200	141236
AAABBCCCCDDDDDEE	31200	672	31200	139148
AAABBBCCCCCDDEEE	31200	672	31200	141236
AABBBBCCCCCDDDEE	31200	672	31200	139148
AAABBBCCDDEEEFFF	33280	660	33280	141236
AABBBBCCCDDEEEFF	33280	660	33280	140188
AABBBCCCCDDDEEFF	33280	660	33280	140188
AAAABBCCDDEEFFFF	33280	660	33280	142268
AABBCCDDEEEEFFFF	33280	660	33280	142268
AABBBCCDDDEEEFFF	33280	660	33280	140188
AABBBBCCDDDEEEFF	33280	660	33280	139148
BBBCCDDDEEEFFFGG	33280	660	33280	140188
BBCCDDDEEEEFFGGG	33280	660	33280	141236
BBBCCCCDDEEFFFGG	33280	660	33280	141236
BBBBCCDDDEEFFFGG	33280	660	33280	139148
BBBBBCCDDEEEFFGG	33280	660	33280	139148

BBBCCDDDDEEEFFGG	33280	660	33280	139148
BBCCDDDEEFFGGHHH	35360	660	35360	141236
BBCCDDDEEEFFGGHH	35360	660	35360	141236
BBBCCCDDEEFFGGHH	35360	660	35360	141236
BBCCDDEEEFFGGHHH	35360	660	35360	142268
BBCCDDEEEFFFGGHH	35360	660	35360	142268
BBBCCDDDEEFFGGHH	35360	660	35360	140188
BBCCDDDEEFFGGGHH	35360	660	35360	141236
BBCCCDDEEFFFGGHH	35360	660	35360	142268
BBCCDDEEFFFGGHHH	35360	660	35360	142268
CCDDEEFFGGHHIIII	35360	660	35360	144348
CCCDDEEFFGGHHIII	35360	660	35360	144348
CCDDEEFFFGGGHHII	35360	660	35360	144348
CCDDEEEFFFGGHHII	35360	660	35360	144348
CCCCDDEEFFGGHHII	35360	660	35360	144348
DDEEEFFGGHHIIJJJ	35360	660	35360	144348
DDEEFFGGGHHIIIJJ	35360	660	35360	144348
DDDEEFFGGHHIIJJJ	35360	660	35360	143308
MEAN	33384	661.5	33384	141100
Standard Deviation		4.019		2109.094

The data from Table 5.5 above show that the inconsistency from RSA algorithms affects the result compared to AES. The RSA got a higher standard deviation which was 2109.094, versus a more consistent AES of 4.019. The average time to run compression plus AES from this data is 34045.5 microseconds, while the average time to run compression plus RSA is 168238.95 microseconds. This result shows that AES is more time-efficient for the Arduino system to run. Interestingly, between the same algorithms compressing the plaintext before encrypting it with RSA is improving in average time, from averaging without compression 146428 microseconds and then 5328 microseconds faster with compression down onto 141100 microseconds (3.63865% faster than without compression). However, for AES, the same could not be said with compression. The average time to complete increases instead 0.227273% from 660 microseconds to 661.5 microseconds.

5.2.7. Compression Effectiveness

The effectiveness of compression also could be seen in Table 5.6 and Table 5.7 below :

Table 5.6. Compression Rate to Time Chart

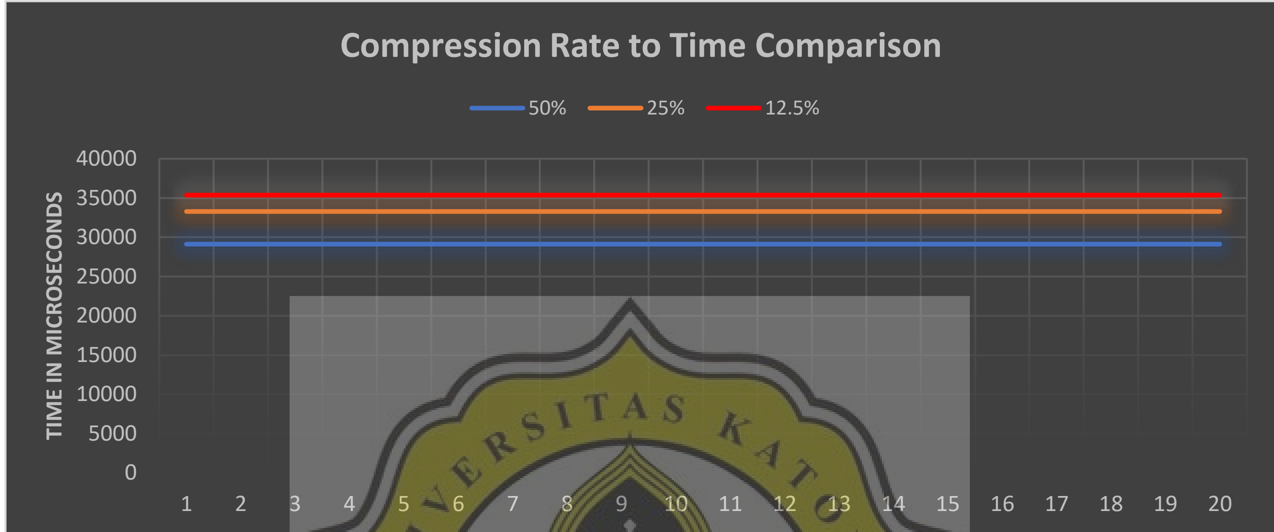

Table 5.7. Compression Rate to Time Data Comparison

Test	Compression Rate		
	50%	**25%**	**12.5%**
1	29120	33280	35360
2	29120	33280	35360
3	29120	33280	35360
4	29120	33280	35360
5	29120	33280	35360
6	29120	33280	35360
7	29120	33280	35360
8	29120	33280	35360
9	29120	33280	35360
10	29120	33280	35360
11	29120	33280	35360
12	29120	33280	35360
13	29120	33280	35360
14	29120	33280	35360
15	29120	33280	35360
16	29120	33280	35360
17	29120	33280	35360
18	29120	33280	35360
19	29120	33280	35360
20	29120	33280	35360

Table 5.6 and Table 5.7 above show that the higher compression rate will make the compression process complete faster in the Arduino system. The time needed to complete a run with eight characters output is 29120 microseconds or 0.02912 seconds, which is 0.00416 seconds faster than 12 character output and 0.00208 seconds faster than 14 character output. These results show that the effectiveness of 50% compression rate is better than 25% compression rate by 14.28% and 12.5% compression rate by 21.43%. Based on previous data time needed to complete non compressed input of 16 characters are 146428 microseconds and the maximum compression rate that is 50% cuts the encryption time down to 137068 microseconds. The conclusion is that encryption time using RSA with compression is up to 6.39% faster than without compression.

The compression time is also affected by input text. Table 5.8 below shows that using a random text input makes the program not compressing optimally because of the limitation of the program in this test. Compressing automatic input is significantly higher than manual input because the program needs to read random input and append the character that the program could not be shortened. The Arduino limitation increases the time to process a higher than the input that is repeated characters. Below is the comparison of manual 12.5% compression rate vs. random input :

Table 5.8. Compression Time Random vs. Manual Comparison

Test	Compression Time in microseconds	
	Manual 12.5%	**Random**
1	35360	40188
2	35360	40188
3	35360	40188
4	35360	40188
5	35360	39148
6	35360	40188
7	35360	39148
8	35360	40188
9	35360	40188
10	35360	40188
11	35360	40188
12	35360	40188
13	35360	39148
14	35360	39148
15	35360	39148
16	35360	39148
17	35360	39148

18	35360	40188
19	35360	39148
20	35360	40188
MEAN	35360	39772

Table 5.8 shows that using manual input, even with the lowest possible output compression, is still faster on average than automatic random input. The difference of 0.004412 seconds will add up when compression is combined with encryption, making the total time even slower than before. Compression time consistency also lowers on random input because every random input generated got a different compression rate.