Spring 2012

The Mathematics of Cryptography & Data Compression

Jennings Anderson
Carroll College, Helena, MT

Follow this and additional works at: https://scholars.carroll.edu/mathengcompsci_theses

Part of the Applied Mathematics Commons, and the Mathematics Commons

Recommended Citation
https://scholars.carroll.edu/mathengcompsci_theses/27

This Thesis is brought to you for free and open access by the Mathematics, Engineering and Computer Science at Carroll Scholars. It has been accepted for inclusion in Mathematics, Engineering and Computer Science Undergraduate Theses by an authorized administrator of Carroll Scholars. For more information, please contact tkratz@carroll.edu.
Abstract

This honors thesis focuses on Cryptography, Data Compression, and the link between the two in modern applications.

Beginning with the notion that a link could exist due to the similar structure of the general methods, chapters individually explore the processes.

An analysis of classical cryptography starts in the classical age and describes methods used until World War II. Shifting to modern computer implementations, the call for National standards introduced a new generation of consumer computer based cryptographic methods that needed to be strong enough for world-wide use.

Switching focus to data compression, compression methods from the 1950s through current day are explored and demonstrated.

Ultimately, the original question is answered by accepting that a link does exist, but not in the form of a combination it was originally thought to be. Rather, when used in series: compression then encryption, the compression adds to the overall security of the data and yields a smaller encrypted file.
This Thesis for Honors Recognition has been approved for the
Department of Mathematics, Engineering, and Computer Science

Philip B. Roe  4/10/12
Director  Date

Mel R.  4/10/12
Reader  Date

William Jones  4/10/12
Reader  Date
# Contents

**Introduction**  
1

## I Classical Cryptographic Systems

1 **Shift Cypher**  
3

2 **Affine Cypher**  
4

3 **Substitution Cypher**  
6

4 **Frequency Analysis - A form of Attack**  
6

5 **Vigenère Cypher**  
9

6 **One Time Pad**  
10

7 **Block Cypher: Hill Cypher**  
11

## II Modern Cryptographic Systems

1 **DES**  
14

   1.1 **Design**  
15

   1.2 **Cracking DES**  
16

      1.2.1 **Theoretical Attacks: Linear & Differential Cryptanalysis**  
16

      1.2.2 **Brute Force Attacks**  
17

   1.3 **Triple DES**  
18

2 **AES**  
19

   2.1 **Design**  
19

      2.1.1 **AES S-Box & Finite Fields**  
20

   2.2 **The Strength & Implementation of AES**  
22

3 **Public Key Encryption - RSA**  
24

   3.1 **Design of RSA**  
24

   3.2 **An Example of the RSA Algorithm**  
24

   3.3 **Theoretical One Time Pad Application of RSA**  
27

## III Data Compression

1 **Background**  
28

   1.1 **Principles of Compression**  
28

   1.2 **Lossless & Lossy**  
29
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Evaluation of Compression</td>
<td>29</td>
</tr>
<tr>
<td>3 Huffman Coding</td>
<td>31</td>
</tr>
<tr>
<td>3.1 Analysis</td>
<td>33</td>
</tr>
<tr>
<td>3.2 Entropy Analysis</td>
<td>33</td>
</tr>
<tr>
<td>4 Arithmetic Coding</td>
<td>34</td>
</tr>
<tr>
<td>5 Dictionary Methods</td>
<td>37</td>
</tr>
<tr>
<td>5.1 Ziv-Lempel</td>
<td>37</td>
</tr>
<tr>
<td>5.1.1 LZ77</td>
<td>37</td>
</tr>
<tr>
<td>5.1.2 LZ78</td>
<td>41</td>
</tr>
<tr>
<td>5.1.3 LZW</td>
<td>42</td>
</tr>
<tr>
<td>6 Data Compression Today</td>
<td>44</td>
</tr>
<tr>
<td>IV Cryptography &amp; Data Compression Together</td>
<td>45</td>
</tr>
<tr>
<td>1 The Two Processes</td>
<td>45</td>
</tr>
<tr>
<td>1.1 A Hybrid Attempt – Proof by Counterexample</td>
<td>47</td>
</tr>
<tr>
<td>2 Hash Algorithms</td>
<td>47</td>
</tr>
<tr>
<td>3 Combining the Two in Lossy Compression</td>
<td>48</td>
</tr>
<tr>
<td>4 The Future of Data Compression / Cryptography</td>
<td>49</td>
</tr>
<tr>
<td>5 Conclusion</td>
<td>50</td>
</tr>
<tr>
<td>References</td>
<td>51</td>
</tr>
<tr>
<td>Appendix</td>
<td>53</td>
</tr>
</tbody>
</table>
Foreword

It’s hard to believe that four years can go so quickly. When I started at Carroll, I was still in ecstatic awe from learning that there was actually a field of study dedicated to Computer Science – I had always figured that computers would forever be my hobby, I did not know that one could make a study plan out of being a self-proclaimed nerd.

Studying math alongside computer science has given me a far deeper understanding of computer technology and the ability to address and analyze a computer application to any problem. For this, I would like to thank all of my Carroll Mathematics and Computer Science professors over the last 4 years.

Furthermore, I would like to specifically thank my readers, Gillian Glaes and Mark Parker for reading the chapters of this thesis and advising changes to make it more interesting and read easier.

Ultimately, I thank Phil Rose for four years of advising support, team leadership in the ACM, and dedicated advising in this final year. This thesis would not have been possible without our weekly meetings. He loaned me many textbooks and helped me work through some of the examples included in these pages.

With these professor’s support, I have learned a tremendous amount about data compression and cryptography. Additionally, without these professors, this Honors Thesis would not have been possible,

Thank You.


\section*{Introduction}

The more I have studied Computer Science, the more I see that it is really a form of applied Mathematics. The two are often paired in University departments, and many students choose to study both for a deeper understanding of the fundamental processes of computers.

As technology advances, the capabilities of computers increase dramatically. Simply put, these capabilities are just mathematical functions that run faster and manipulate more data.

To better my understanding of computer technology, I chose to study the mathematics behind two long time (and forever) implemented computer processes: Cryptography and Data Compression.

\section*{Cryptography \& Compression}

Cryptography is the process of designing systems to protect data \cite[p2]{20}. Before the security and integrity of data was handled by the computer, confidential messages underwent different forms of encryption and decryption that, while outdated today, are still mathematically challenging and interesting to study.

The second area that interests me as a basic computer process is data compression. In an age where companies report that their consumer data is worth more than their physical company assets \cite{9}, there is a definite need to focus on how we store data. More data requires more available space on the computer hard disk. Methods of compression have always existed with computers and are consistently improving to allow for more data to fit on a single disk. Today, the common computer user interacts with compressed files everyday.

\section*{Goal}

Specifically, I set about this thesis looking to identify a commonality in the traits of both processes. While exact methods of cryptography vary, the process can be summarized as:

\begin{align*}
\text{Encryption} & \quad \text{Decryption} \\
\text{Plain Text} & \quad \rightarrow \quad \text{Encrypted Data} & \quad \rightarrow \quad \text{Plain Text}
\end{align*}

This over simplification of cryptography parallels the process of Data Compression:

\begin{align*}
\text{Compression} & \quad \text{Decompression} \\
\text{Plain Text} & \quad \rightarrow \quad \text{Compressed Data} & \quad \rightarrow \quad \text{Plain Text}
\end{align*}
These two processes serve different purposes in computing, yet compression yields a file that is incapable of being read until it undergoes a decompression process: This seems like a subtle form of cryptography.

The similarities in these methods pose the question: Is there commonality or an overlap? That is:

*Is there a method of data compression that offers cryptographic strength, or a form of cryptography that yields encrypted data of smaller size than the original plain text?*

Should a compression method that requires a key to be decompressed be impressively difficult to decompress without the key, then one may conclude that data compression is an appropriate form of cryptography.

Conversely, if there is no form of data compression that offers cryptographic strength, then it is interesting enough that there exists no overlap in these two processes that share similar methods. This suggests that there is still development opportunities to connect these fields.

**Approach**

To investigate these two processes, I will examine the history and development of each separately in the following manner:

1. **Classical Cryptography**: The origins of cryptography and the methods used up to World War II.

2. **Modern Cryptography**: Cryptography today as implemented in the digital age.

3. **Data Compression**: History and development of compression methods from mid-20th century to today.

Ultimately, I will conclude by showing the interaction between the two in modern computing, address the question proposed above, and discuss what the future holds for this topic.
Part I

Classical Cryptographic Systems

Classical cryptography includes methods of concealing plain text that go back at least to ancient Rome and continued to be used up to and during the world wars. With the dramatic improvements in computer technology since World War II, these methods no longer offer security. However, their development, implementation, and mathematical aestheticism is worthy of study.

To study these methods, we must first define the link between a plain text message and its mathematical equivalence. Classically, this is done by assigning each letter of the English alphabet a numerical value between 0 and 25, such that $A = 0$, $B = 1$, $C = 2$, $D = 3$, ... $Y = 24$, $Z = 25$. These systems also incorporate the modulus operation, making the number space $[0, 25]$. Therefore, all equations in this section will be done modulus 26. For example, if an operation yields 27, it will be reduced as such:

$$27 \mod 26 \equiv 1$$

This number represents the character $B$. This can be simply thought of as the alphabet wrapping from $Z \rightarrow A$.

I begin by presenting, demonstrating, and analyzing some of the better known classical cryptographic systems:

1 Shift Cypher

The shift cypher is the simplest method of cryptography. It forms an encrypted text string of the same length as the original message. The encrypted string appears jumbled and unintelligible. The cypher simply maps each plain text letter to a letter a fixed distance ahead in the alphabet (wrapping from $Z$ to $A$). Julius Caesar is known to have used a shift cypher with a shift value of 3 when exchanging messages with his military; therefore, today this particular cypher is called a Caeser Cypher [20, p.13].

The basic algorithm for a Shift Cypher is:

$$\text{PlainText}_i + k \mod 26 = \text{EncryptedText}_i$$

(1)

Where $k$ is a consistent shift value and $i$ represents a character in the string. Recall the mod 26 operation assumes that each letter is given a numeric value: $A = 0$, $B =$
1, $C = 2$, and so forth. An example with the string “Hello World” and a shift value of 3 shows this:

<table>
<thead>
<tr>
<th>Plain Text</th>
<th>H E L L O W O R L D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encrypted</td>
<td>K H O O R Z R U O G</td>
</tr>
</tbody>
</table>

In this example, the encrypted text is incomprehensible to the casual reader, but should the reader know the shift value, the plain text can be recovered by simply subtracting the shift value from each character in the encrypted text.¹

**Analysis**

To encrypt, only one mathematical operation must be performed on each character in the string: $(i + k \mod 26)$. Decryption follows the same equation, one just subtracts the key value, so it too only requires as many steps as their are characters. If, however, the shift value is unknown, there are only 25 possible decrypted solutions of the encrypted text. This means that to a casual reader who is unaware of the encryption technique, the message is secure, but if an interceptor suspects a shift cypher, a third party need only try 25 shift values to obtain the original message. Worse yet, if the interceptor knows a single letter of the plain text, then they can find the key in one step [20, p.14]. Today, with computers performing millions of processes a second, this method offers no security, but when first implemented thousands of years ago before computers, it held merit.

## 2 Affine Cypher

The affine cypher adds another dimension to the shift cypher. Before the shift value is added, the plain text character is multiplied by a fixed coefficient. Each plain text character now maps to a different encrypted character in the same character space, however the difference is not constant among adjacent letters. The algorithm is summarized as:

$$PlainText_i \times a + b \mod 26 = EncryptedText_i$$

(2)

Where $a$ is the coefficient by which each character is multiplied, $b$ is the shift value and $i$ represents a character in the string.

¹Source code and a computer implementation of this algorithm can be found in the appendix.
For example:

<table>
<thead>
<tr>
<th>Plain Text</th>
<th>H E L L O</th>
</tr>
</thead>
<tbody>
<tr>
<td>▼ 5x + 9  mod 26</td>
<td></td>
</tr>
<tr>
<td>Encrypted</td>
<td>S D M M B</td>
</tr>
</tbody>
</table>

### Decryption

The decryption of an affine cypher encryption requires the multiplicative inverse of $a$. For this reason, possible values of $a$ are limited to those satisfying:

$$\gcd(a, 26) = 1$$

To decrypt, the multiplicative inverse is substituted for $\frac{1}{a}$.

**Example:** The given sample, $y = 5x + 9 \mod 26$ is decrypted by solving for $x$:

$$x = \frac{1}{5}(y - 9)$$

Where $y$ is the encrypted text character and $x$ is the plain text character. However, because calculations in this equation are $\mod 26$, 21 is substituted for $\frac{1}{5}$ [20, p.14]. This is valid because 21 is the multiplicative inverse of 5 $\mod 26$:

$$5 \times 21 \equiv 1 \mod 26$$

In this manner, $x = 21y - 189 \mod 26$. Plugging in the encrypted characters above $(S, D, M, M, B)$ as $y$ yields the original message *HELLO*.

### Analysis

The affine cypher is slightly more secure than a simple shift cypher because the shift value varies with each character. However, the shift value for any particular character will always be the same. Therefore, if a third party can determine just two characters in the original message, typically done by frequency analysis, then the shift value can be solved for in two equations with two unknown variables. This is more secure than a shift cypher,

---

2This can also be determined from the Group $Z_{26}$ [3]
3Source code available in appendix
4Frequency Analysis is a common cracking technique to be discussed later.
which only required learning one original plain text character, but was readily broken even before computer technology, so is no longer used in practical application [20, p.14].

3 Substitution Cypher

Commonly found in newspapers, substitution cyphers do not follow a mathematical function, but rather letters are arbitrarily chosen to substitute for others consistently throughout the text. This eliminates the ease of decrypting the message because no mathematical pattern exists to map a plain text character to the corresponding encrypted text character. However, a frequency analysis can usually break this easily, and that is exactly what one does, aware or not, when solving one of these in the Sunday paper.

Analysis

A substitution cypher is theoretically more secure than an affine cypher because there is no pattern in how the characters map between the plain text and encrypted text, but for applied use, a key that showed the mapping would need to be sent in addition to the encrypted text. This is true for all of these methods, but a key that contains a single mathematical formula is shorter than a full character mapping, a potential security vulnerability in transport.

4 Frequency Analysis - A form of Attack

Each of these cyphers is extremely susceptible to frequency analysis. This cracking technique simply tabulates the frequencies of letters in a given language and compares the encrypted text to these character appearance rates [20, p 25]. For example, in English, the most common letters and their frequency (as a decimal) are:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>.127</td>
</tr>
<tr>
<td>t</td>
<td>.091</td>
</tr>
<tr>
<td>a</td>
<td>.082</td>
</tr>
<tr>
<td>o</td>
<td>.075</td>
</tr>
<tr>
<td>i</td>
<td>.070</td>
</tr>
<tr>
<td>n</td>
<td>.067</td>
</tr>
<tr>
<td>s</td>
<td>.063</td>
</tr>
<tr>
<td>h</td>
<td>.061</td>
</tr>
<tr>
<td>r</td>
<td>.060</td>
</tr>
</tbody>
</table>

Table 1: The most common letters in English & their frequencies [20, p.24]

This table shows that on average, the letter e composes 12% of written English. The shift cypher and the affine cypher are weakest because only one or two characters need to be identified for a mathematical formula to be uncovered. A substitution cypher removes this convenience, but is still susceptible, it requires examining each character,
not determining a pattern. Here is an example of frequency analysis:

Assume that the following code was intercepted:

```
XZDRHUYQHUKBYQBYQDYQTZUNBKVIBVUHNZOKWYQHYQHTNHKKTHUWYQZTBNHKKTQHIBYZ
MBLDHOWBXMXPBUNYQLDUTNQZTLZUUHVVYXZDXZDKYNBVUMBOLVQHIBHLOBHYBOOBTE
ZUTVMKVXYQHUXZDRHUEZTVXMKXGQYQPZXZDNBBEGZOTHUYVHLOXZDRTDOTBYQBP....
```

The total message is 877 characters. It is unknown how it was encrypted, but analysis shows that the most common letters are:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>103</td>
</tr>
<tr>
<td>Y</td>
<td>80</td>
</tr>
<tr>
<td>H</td>
<td>75</td>
</tr>
<tr>
<td>Z</td>
<td>72</td>
</tr>
<tr>
<td>U</td>
<td>69</td>
</tr>
</tbody>
</table>

Following Table 1, it seems likely that B=e, Y=t, H=a, and so forth. If we start by assuming this is a shift cypher and B=e, this implies that it was encrypted with a shift value of 29 because:

\[
\frac{1}{B} + \frac{29}{\text{shift value}} \mod 26 = \frac{4}{e}
\]  

(3)

Plugging in this shift value yields the plain text:

ACGUKXBTK....

It is clear that this is not the shift value, so we can rule out a simple shift cypher (or at least we know \( B \neq e \)). Going one step further would assume that it was encrypted with an affine cypher of the form \( y = a \ast x + b \). With the two values, B=e, and Y=t, this can be approached as two equations with two unknowns:

\[
\begin{align*}
1 &= a \ast 4 + b \mod 26 \\
24 &= a \ast 19 + b \mod 26
\end{align*}
\]  

(4)

This can be written in matrix form:

\[
\begin{bmatrix}
a \\
b
\end{bmatrix}
\begin{bmatrix}
4 & 1 \\
19 & 1
\end{bmatrix}
= 
\begin{bmatrix}
1 \\
24
\end{bmatrix} \mod 26
\]  

(5)

The first step is to find the inverse of the coefficient matrix in \( \mod 26 \). Using Mathematica\(^5\), this is solved to be:

\[
\begin{bmatrix}
19 & 7 \\
3 & 24
\end{bmatrix}
\]  

(6)

\(^5\)Mathematica Command: Inverse[{{4,1},{19,1}},Modulus->26]//MatrixForm
Using the inverse matrix, the two variables can be solved as follows:

\[
\begin{align*}
    a & \equiv 19 \times 1 + 7 \times 24 \mod 26 \\
    b & \equiv 3 \times 1 + 24 \times 24 \mod 26
\end{align*}
\]  

(7)

This evaluates to \(a = 5\) and \(b = 7\). This leads to the possibility that the original message was encrypted with the equation:

\[
EncryptedText_i = PlainText_i \times 5 + 7 \mod 26
\]  

(8)

Where \(i\) represents each character in the message. To decrypt this, one must use the multiplicative inverse of \(5 \mod 26\), which is 21 (as shown in section 2). When this cypher text is decrypted using these values, it reads:

YOU CANNOT HANDLE THE TRUTH. WE LIVE IN A WORLD THAT HAS WALLS . . .

Or, when broken apart and punctuated:

\[You\ can't\ handle\ the\ truth!\ We\ live\ in\ a\ world\ that\ has\ walls . . .\]

which is Jack Nicholson’s monologue response to Tom Cruise in the film \textit{A Few Good Men} [2].

It should be noted that to create this example, I simply searched the Internet for movie quotes and then plugged this into an affine cypher.\(^6\) This corroborates the frequency analysis method in that it worked for this randomly intercepted message. It so happened with this example that the message matched the high frequency letters in order. But in the event that this does not happen, the order of the high frequency letters may be rearranged until intelligible data is reached. In order to do this effectively, one can establish a counting diagram, noting which characters appear adjacent to others and comparing pairs of letters with known frequencies, such as the letter \(n\) is uniquely preceded \(80\%\) of the time by a vowel, or by identifying common pairs such as \(sh, th,\) and \(qu\) [20, p.25].

To obscure basic frequency analysis, cypher text must lose the property that each character represents the same plain text character. To this point, I have shown only cyphers that exhibit this property, next we examine cyphers that use separate, independent keys to create this obscurity.

\(^{6}\)Source code available in appendix
5 Vigenère Cypher

The simplest of independent key-based ciphers, the Vigenère Cypher was developed in the sixteenth century and used up to the 1900s [20, p.16]. To begin encryption, the sender chooses a secret key, typically much shorter than the message. This is called a vector. For this example, we’ll choose the word CRYPTO as the vector. Numerically, this vector is represented as:

\[
\begin{array}{c c c c c c}
< & C & R & Y & P & T & O > \\
< & 2 & 17 & 24 & 15 & 19 & 14 > \\
\end{array}
\]

Next, the message and the vector are aligned, repeating the vector until the lengths are equal. The cypher text is created by shifting each character of the message by the corresponding value in the vector:

<table>
<thead>
<tr>
<th>Message</th>
<th>M</th>
<th>E</th>
<th>E</th>
<th>T</th>
<th>I</th>
<th>N</th>
<th>P</th>
<th>A</th>
<th>R</th>
<th>I</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Vector</td>
<td>2</td>
<td>17</td>
<td>24</td>
<td>15</td>
<td>19</td>
<td>14</td>
<td>2</td>
<td>17</td>
<td>24</td>
<td>15</td>
<td>19</td>
</tr>
<tr>
<td>Cypher Text</td>
<td>O</td>
<td>V</td>
<td>C</td>
<td>I</td>
<td>B</td>
<td>B</td>
<td>R</td>
<td>P</td>
<td>X</td>
<td>L</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{array}{c c c c c c c c c c}
\text{mod 26} \\
\end{array}
\]

The power of this method is that the same plain text character is not always represented by the same character in the encrypted text.

Analysis

The Vigenère Cypher is less susceptible to the basic frequency analysis as in section 4, but far from unbreakable. Since it is known that there is a finite vector length used to generate the key, a more involved frequency analysis can yield the length of vector, and ultimately the vector itself by comparing the cypher text to known frequency values. It is not as straightforward as character-based substitution, but is still easily broken [20, p.23].

Next, we examine a method that looks similar, but uses a random key instead of a repeating vector. This small difference in the key creates a cypher text that truly can never be broken, regardless of technological advances. No amount of time nor computing power can break the next cypher, not now, not in a 100 years.
6 One Time Pad

The one time pad has a unique cryptographic quality in that it cannot be broken. It is not that it is difficult to crack, nor that it takes a long time, but rather that the cypher text contains no information about the key other than its length [20, p.39]. The method was presented in 1918 by Gilbert Vernam and Joseph Mauborgne. The basic principle is to create a random key the same length as the plain text message and then add the two together, character by character:

\[
\begin{array}{cccccccccc}
\text{message} & H & E & L & L & O & W & O & R & L & D \\
\text{key} & + & S & G & S & E & G & F & A & I & O & T \\
\text{Cypher Values} & 25 & 10 & 3 & 15 & 10 & 1 & 14 & 25 & 25 & 22 \\
\text{Interpreted as letters:} & Z & K & D & P & K & B & O & Z & Z & W \\
\end{array}
\]

Similar to the Vigenère method, and different from the shift, affine, and substitution ciphers, it is important to note that each plain text \(L\) translated to a different cypher text character: \(D, P, Z\), respectively; however, this was completely random: there is an equal probability that these characters could translate to any other character, or not even change at all. This makes the form of frequency analysis in section 4 worthless on this encrypted text.

Decrypting the message is straightforward: simply subtract the key from the cypher text. When implemented in a computer, the message and key are simply bit strings that undergo an **EXCLUSIVE OR** operation for encryption; this is then just done again between the cypher text and key to decrypt [20, p.39].

For a one time pad to be truly unbreakable, the key must be generated randomly and never used again (hence the name ‘one time’). This creates difficulties in the practice of using a one time pad: a truly random key would require rolling a 26-sided dice for every character in the key, or finding some other way to generate 26 random integers [20, p.39]. If the key is not truly random, or based on a pseudo random number generator (such as a computer), then there exists the possibility to regenerate the same key.

Once the random key is created, it must be sent in full (unlike the Vigenère cypher where just the vector need be sent) to the recipient along with the message. They can of course be sent separately, but without the key, the cypher text is unbreakable, so the pair must be together at both encryption and decryption for the message ever to be successfully encrypted and decrypted, because neither can ever be regenerated.

That said, during the Cold War, the leaders of the United States and the U.S.S.R. reportedly used one time pads for messages on the ‘hot line’ between Washington D.C.
and Moscow [20, p.40]. So, while they are not used frequently, their nature of being unbreakable makes them a candidate for incredibly high risk message passing [17, p.298].

In summary, one time pads boast the ability to be completely uncrackable, but require a random key of equal length to the original message to be present at encryption and decryption, and an absolutely secure key transmission which makes for large overhead in practical use.

7 Block Cypher: Hill Cypher

The Vigenère Cypher and One Time Pad differ from the previous cyphers because a single character in the cypher text does not always correspond to the same decrypted character. In the case of the One Time Pad's random key, this makes it truly unbreakable; however, the Vigenère cypher is still susceptible to frequency analysis due to the repeating vector key (as noted in section 5).

A block cypher adds further complexity by including interaction between letters in a given block of letters. These blocks are of equal length to the key - similar to the Vigenère cypher, where the blocks are the length of the vector, but different in that the Vigenère blocks do not contain interaction between the letters. The result of this interaction is that changing one letter in the plain text will change multiple letters in the cypher text [20, p.33]. It is important to introduce block cyphers here because modern cryptographic systems are forms of block cyphers.

The Hill Cypher is named after Lester Hill who invented the method in 1929; interestingly, it was not implemented much in its time, but rather is well known because it represents the first implementation of linear algebra in cryptography [20, p.32]. Here is an example of a simple Hill Cypher:

First, choose a block size, a good example size is \( n = 3 \). Next, let \( M \) be an \( n \times n \) with entries are taken \( \text{mod} \ 26 \):

\[
M = \begin{pmatrix} 2 & 3 & 5 \\ 7 & 11 & 13 \\ 17 & 19 & 24 \end{pmatrix}
\]

(9)

The only requirement for the entries in \( M \) is that for decryption to work, the determinant of \( M \) and 26 must be relatively prime, that is:

\[
\gcd (\det (M), 26) = 1
\]

(10)
This property must be satisfied because decryption will require $M^{-1}$, which can only exist (mod 26) if equation 10 is true. In this case, $\det(M) = -77$ and $\gcd(-77, 26) = 1$, so the two are relatively prime.

To encrypt, the message: (meet at nine am), we first break it into blocks of length $n$:

<table>
<thead>
<tr>
<th>Plain Text</th>
<th>M</th>
<th>E</th>
<th>E</th>
<th>T</th>
<th>A</th>
<th>T</th>
<th>N</th>
<th>I</th>
<th>N</th>
<th>E</th>
<th>A</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numerically</td>
<td>12</td>
<td>4</td>
<td>4</td>
<td>19</td>
<td>0</td>
<td>19</td>
<td>13</td>
<td>8</td>
<td>13</td>
<td>4</td>
<td>0</td>
<td>12</td>
</tr>
</tbody>
</table>

Then, these blocks undergo matrix multiplication (mod 26) with $M$:

Block 1:

\[
\begin{pmatrix} 2 & 3 & 5 \\ 7 & 11 & 13 \\ 17 & 19 & 24 \end{pmatrix} \begin{pmatrix} 12 \\ 4 \\ 4 \end{pmatrix} = \begin{pmatrix} 16 \\ 0 \\ 0 \end{pmatrix} \pmod{26}
\]

Using this same equation, blocks 2, 3, 4 are: (23, 2, 5) (17, 10, 13) (4, 6, 22), respectively.

<table>
<thead>
<tr>
<th>Plain Text</th>
<th>16</th>
<th>0</th>
<th>0</th>
<th>23</th>
<th>2</th>
<th>5</th>
<th>17</th>
<th>10</th>
<th>13</th>
<th>4</th>
<th>6</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numerically</td>
<td>Q</td>
<td>A</td>
<td>A</td>
<td>X</td>
<td>C</td>
<td>F</td>
<td>R</td>
<td>K</td>
<td>N</td>
<td>E</td>
<td>G</td>
<td>W</td>
</tr>
</tbody>
</table>

The cypher text therefore reads: QAAXCFRKNEGW. To decrypt, we must first compute the inverse of $M$ modulus 26:

First, find the inverse of the matrix, $M$:

\[
\begin{pmatrix} 2 & 3 & 5 \\ 7 & 11 & 13 \\ 17 & 19 & 24 \end{pmatrix}^{-1} = \begin{pmatrix} -1 \\ 53 \\ -54 \end{pmatrix} \begin{pmatrix} 17 & 23 & -16 \\ 53 & -37 & 9 \\ -54 & 13 & 1 \end{pmatrix}
\]

Since the determinant of $M$ is -77, we must find the inverse of $-77 \pmod{26}$. This can be computed using the Group\(^7\): $Z_{26}$:

\[
1 \equiv -77 \pmod{26}
\]

\[
1^{-1} \text{ in } Z_{26} = 1
\]

Therefore, we can replace $\left(\frac{-1}{77}\right)$ with $1 \pmod{26}$, which means simply reducing each entry

---

\(^7\)From Gallian, Chapter 2 [3]
of the inverse matrix modulus 26:

\[
M^{-1} = \begin{pmatrix}
17 & 23 & 10 \\
1 & 15 & 9 \\
24 & 13 & 1
\end{pmatrix} \mod 26 \tag{13}
\]

To decrypt, we multiply the cypher blocks by the decryption matrix (reducing \( \mod 26 \)):

\[
(16, 0, 0) \begin{pmatrix}
17 & 23 & 10 \\
1 & 15 & 9 \\
24 & 13 & 1
\end{pmatrix} = (12, 4, 4) \mod 26 \tag{14}
\]

Recall that \((12, 4, 4)\) was the first plain text block representing \textit{MEE}, in the word \textit{MEET}. Using this inverse matrix, the rest of the cypher text is decrypted in the same manner.

**Analysis**

Suppose the message was changed to tell the receiver to meet in the evening instead of the morning. This would change the \textit{AM} to \textit{PM} in the original message. The plain text block 4 would therefore change:

\((4,0,12) \rightarrow (4,15,12)\) where the \(A = 0\) changes to a \(P = 15\)

The encrypted value of this block is then:

\[
(4,15,12) \begin{pmatrix}
2 & 3 & 5 \\
7 & 11 & 13 \\
17 & 19 & 24
\end{pmatrix} = (5, 15, 9) \mod 26 \tag{15}
\]

which yields the cypher block \textit{FPM}. Before, block 4 was \textit{EGW}. Though only one letter was changed in the plain text, the entire block of the cypher text was changed.

This property makes block ciphers less susceptible to frequency analysis, especially as the size of the blocks increase [20, p.37]. This is the foundation of modern computer cryptographic systems, to be discussed next.
Part II

Modern Cryptographic Systems

With the implementation of the computer came an increase in both the need for security in digital transactions and the ability to provide more secure, and hence complex, encryption methods. Today, we implement these cryptographic systems in our computers, tablets, and phones when we connect to secure servers such as banks or any on-line transaction that involves sensitive information. These methods are standardized and used internationally so that secure world-wide connectivity can exist across the web. Standardized cryptography began when the National Bureau of Standards put a request to the public for a single, robust standard algorithm in 1973 [20, p.97].

In this section, I will describe the present and past standards for modern computer encryption systems and discuss their designs; it should be noted that the more complexity in design and understanding, the more secure the system. Therefore, I do not step through these algorithms in the same manner as Part I because it becomes very obvious why these methods were not implemented before the modern computer: they are too complex to perform by hand; I will, however, discuss certain aspects of their design and the mathematics that they require.\footnote{If there ever was a place to say: “Details omitted for National Security”, this is it, but alas, the details of these systems are well documented and published publicly.}

1 DES

The technology superpower, International Business Machines (IBM), submitted a cryptographic algorithm under the codename LUCIFER to the National Bureau of Standards in 1974. By 1977, after further tweaking from the National Security Administration (NSA), a modified implementation of LUCIFER named the \textit{Data Encryption Standard} (DES) was deemed the new encryption standard [4]. However, with the NSA’s involvement in the development, many have feared since the inception that a government back door exists in the DES algorithm. A back door would allow a single party to easily decrypt any cypher text without the key - comparable to a fear of ‘Big Brother’. Furthermore, others conversely believed that the NSA’s involvement ensured an IBM back door would have been cleared out [20, p.98]. These conspiracies lasted 13 years until a theoretically possible method of cracking DES known as differential cryptanalysis was presented in 1990; defending themselves from making it look like a back door, IBM released details of the
DES design that were built to be resistant to this type of attack [20, p.98]. This release calmed the conspiracy theories about hidden back doors in DES. When adopted, it was only to be used for 10 - 15 years, but in practice DES was stronger than expected and went through a review process every 5 years until 1999 [21, p.95].

1.1 Design

DES is a complex, symmetrical 64-bit block cypher (recall the details of the Vigenère Cypher, a block cypher). The strength of DES lies within the shear computing power that is required to brute force a solution. There are $2^{56}$ possible keys that must be tried to decrypt the cypher text in common DES. It generates this large key space by using a 56-bit key. Interestingly, this key is actually 64 bits long, but 8 bits are used for parity checking, so only 56 bits provide security [4]. The large key space is the reason that DES remained the standard for 20 years: At the time, specialized computer technology capable of breaking this would cost $20$ million [4]. Today, this cost seems negligible for specialized technology, but at the time, this was a purely theoretical machine, infeasible in construction. Walking through a DES encryption by hand quickly shows why this method was not implemented before computers, but I will present the basic outline of the DES.

DES is called a symmetrical encryption method because the same key is used for encryption and decryption [17, p.296]. It uses the Data Encryption Algorithm (DEA) which takes 64 bit blocks of plain text and breaks them into 32 bit halves: left and right [4]. This is more specifically called a Feistel Cypher [21, p.95]. DES undergoes 16 rounds of the Feistel function which uses a table known as an S-box to substitute the bit string in a specific manner:\footnote{Example DES f Function from Stinson book [21, p.97]}

1. Given one of 32-bit halves from the original 64-bit block, an expansion function creates a 48-bit block in a specific manner (noting that 16 redundant bits have been added).

2. The 48-bit string is then broken apart as 8 different 6-bit strings.

3. An S-box is used to create a 4 bit block from each of the 6-bit strings in the following manner:

   Suppose the input string was 011011, one of the 6-bit strings. The first and last bit are concatenated to represent the row (in this case, 01, which represents a binary
1 – referencing row 1. The middle 4 bits represent the column (in this case, 1101 calls column 13). This is shown in the figure on the following page.  

<table>
<thead>
<tr>
<th>$S_5$</th>
<th>Middle 4 bits of input</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111</td>
</tr>
<tr>
<td>01</td>
<td>0001 0010 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111</td>
</tr>
<tr>
<td>10</td>
<td>0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111</td>
</tr>
<tr>
<td>11</td>
<td>0011 0000 0001 0010 0100 0101 0110 0111 0100 0101 0110 0111 0000 0001 1110 1111</td>
</tr>
</tbody>
</table>

Figure 1: DES S-Box $S_5$, with resulting 4 bit output string from the input string 011011 highlighted.

There are 8 different S-boxes involved for each 6-bit string, for the DES encryption, these S-boxes are available to the public.

4. Ultimately, the result of the S-box substitutions is a 32-bit string that represents one half of a 64-bit block of plain text. This completes one round of DES.

The final step (after 16 rounds of S-Box substitutions) is an inverse of the original permutation that yields a block of complete cypher text [17, p.296]. The final block is still 64 bits, so the message text and the cypher text have the same length [4].

The main concept to be gleaned from this explanation of DES is that the relationship between the original plain text and the cypher text is convoluted to such a degree that the key cannot be extracted from the cypher text (as was the case in Part I), but rather must be searched for, exhaustively through the entire $2^{56}$ possible keys [21, p.101].

1.2 Cracking DES

The most straightforward method of breaking a DES encryption is a brute force attack (an exhaustive key search). This requires checking all possible keys until one successfully decrypts the message [4]. This was the initial strength of DES and the reason that it remained the standard for the next 20 years.

1.2.1 Theoretical Attacks: Linear & Differential Cryptanalysis

Two other forms of attack are theoretically possible, but infeasible: differential cryptanalysis and linear cryptanalysis. Differential cryptanalysis was accounted for in the DES

---

design, but not published at the time DES was adopted because general knowledge of this cracking method did not exist outside of the IBM research team (this was the root of conspiracy issues surrounding DES) [21, p.100]. Linear cryptanalysis is a more efficient attack on DES, but requires tremendous overhead that makes it infeasible to use in cracking of real messages. In 1994, the inventor of linear cryptanalysis, Matsui, used $2^{43}$ different cypher texts that were all DES encrypted with the same key to launch an attack to find the key; it took 10 days, but the attack worked and reproduced the secret key [21, p.101]. This threat to DES is overlooked because it took Matsui 40 days to generate the cypher texts used in his attack [21, p.101]. For this method to work on an actual intercepted message, $2^{43}$ different cypher texts generated from the same key would have to be intercepted, which is over 250,000 times the size of the collection at the Library of Congress\footnote{Over 34.5 Million books and other printed material according to loc.gov/about/facts.html}. These facts reinforce the notion that there is only one semi-feasible manner to break DES: brute force.

### 1.2.2 Brute Force Attacks

From early on, the theoretical notion that a special purpose machine could be built to break it was always to haunt DES [21, p.101]. The original Lucifer system proposed by IBM (from which DES was built) included a key that was 128 bits long; however, as discussed above, DES only uses a 56-bit key [21, p.101]. With major improvements in technology, 1993 saw plans for an improved DES key-search machine that could try 16 encryptions simultaneously. With 5760 specialized chips, a $100,000 machine could be built to crack DES in one and one-half days - for $1,000,000, DES could be cracked in a mere 3.5 hours; however, this machine was never actually built [21, p.101]. The idea that a single computer could be built for a reasonable price was intriguing enough for people to get creative.

In 1997, the RSA Data Security Company released a DES encrypted message with a $10,000 prize to whoever submitted the cracked key that was used in encryption [20, p.120]. Just five months later, Rocke Verser submitted the cracked key and decrypted message that read: ‘Strong cryptography makes the world a safer place.’ Verser created a program that shared the workload over the Internet with thousands of other computers, a process known as Distributed Computing. In his particular program, users that downloaded the program agreed to the terms that if their computer cracked the winning key, they would be receive 40% of the cash prize. Remarkably, only 25% of the full key space had to be searched before the key was found [20, p.120]. This marked the beginning of the end for
DES as the National Institute of Standards and Technology (NIST, formerly known as the NBS who had originally called for DES 20 years earlier) asked for public submissions to replace DES [20, p.127].

When the contest ran again in 1998, a company named 'Distributed Computing Company' launched a project that searched nearly 85% of the entire key space in just 39 days: This dramatic improvement in computer technology poked larger holes in the security of DES [20, p.120].

Also in 1998, with a budget of $200,000, a team known as the Electronic Frontier Foundation built a computer called DES Cracker [20, p.120]. Nicknamed 'Deep Crack,' this DES-cracking specific machine was capable of testing nearly 90 billion keys a second and could test all possible keys in an average of 4.5 days [19]. One design aspect of this machine that made it so efficient was its ability to shrink the key space based on the characters it returned. Since the encrypted strings were built of 8-bit characters (the manner that most computers interpret characters today), the machine would reject decrypted messages whose first two letters did not contain relevant information (such as nonprintable characters) - this reduced the possible 8-bit values to only 1 of 69 instead of 254; this scaling was done on the specialized chip, so it ran very efficiently [20, p.121].

A year later, in an event called 'DES Challenge III', Deep Crack joined a network of 100,000 distributed computers worldwide and was able to test keys at a rate of 245 billion per second, cracking a DES key in just 22 hours [21, p.101].

Since the $2^{56}$ key space was now exhaustively search-able in a matter of hours, a proposed solution was to find a way to increase the size, this lead to the implementation of Triple-DES.

1.3 Triple DES

Triple-DES uses the standard DES algorithm, but performs it multiple times with different keys. It can be done in two ways [20, p.122]:

1. Using 3 Keys: Encrypt with $K_1$, Encrypt with $K_2$, Encrypt with $K_3$

2. Using 2 Keys: Encrypt with $K_1$, Decrypt with $K_2$, Encrypt with $K_1$

Provided that $K_1 \neq K_2 \neq K_3$, these methods increase the key space to $2^{112}$. Cryptography expert Bruce Schneier claims, “there isn’t enough silicon in the galaxy or enough time before the sun burns out to brute-force triple-DES” [19].

While Triple-DES is relatively secure, NIST was still looking for a replacement. There was a request to make a more versatile algorithm that could be run easily on commercially
available processors and should be capable of working with larger blocks. Additionally, the new algorithm must accept much larger keys between 128 and 256 bits [20, p.127]. The new algorithm was dubbed the Advanced Encryption System, or AES.

2 AES

Often referred to as the Rijndael Encryption, the Rijndael proposal was one of five finalists that were chosen from 15 submissions to replace DES in 1998 [20, p.127]. After a lengthy review process that was noted for its International involvement, since the developers were from Belgium, Rijndael was adopted as the Advanced Encryption Standard in 2001 [21, p.102]. The Belgian creators were Joan Daemen and Vincent Rijmen (Rijndael is coined from the combination of their two last names). The multiple-year open review of Rijndael by NIST dismissed some initial fears that the algorithm was foreign [6]. Ironically, some believe that the Belgian origin actually sped up the review process as it eliminated possible U.S. government back doors [6].

Rijndael was ultimately selected for proving proficient in 3 different areas: security, cost, and algorithm characteristics [21, p.102]. The cost in this context is the amount of computer processing power required while algorithm characteristics include the running requirements and flexibility across different systems.

2.1 Design

Similar to DES, AES is a block cypher that undergoes an iterated process over multiple rounds; however, different from AES it uses blocks of 128 bits instead of 64 and the number of rounds is dependent on the size of the key. An AES key can be either 128, 192, or 256 bits; these different sizes call for the total encryption to include 10, 12, and 14 rounds, respectively [21, p.103].

The basic outline of AES is broken into 4 layers [20, 129]:

1. **ByteSub Transform**: A specific layer to guard against differential and linear crypt-analysis, using an S-Box.

2. **ShiftRow Transform**: Creates greater mixing of the bits to further diffuse the encrypted text from the plain text.

3. **MixColumn Transform**: Another diffusion step involving matrix multiplication in the finite Galois field, \( GF(2^8) \).\(^{12}\)

\(^{12}\)Described in section 2.1.1.
4. AddRoundKey: Performs a logical EXCLUSIVE OR between the result of step 3 and the key for that round of encryption.

2.1.1 AES S-Box & Finite Fields

To better understand a major structural difference between DES and AES, let’s look at how the AES S-Box operation is different from DES.

Working in bytes, the Rijndael algorithm calls on the finite field, $GF(2^8)$ which by definition creates a unique finite field with 256 elements [3, p.382].

To use the $GF(2^8)$ field, Rijndael translates every byte to a polynomial in the following manner: Break down an incoming byte into 8 bits: $b_7b_6b_5b_4b_3b_2b_1b_0$, then represent the byte as a polynomial like this [20, p.88]:

$$b_7X^7 + b_6X^6 + b_5X^5 + b_4X^4 + b_3X^3 + b_2X^2 + b_1X + b_0$$  \hspace{1cm} (1)

For example, the byte 10001001 would be represented with the polynomial: $X^7 + X^3 + 1$. In the Rijndael algorithm, all operations within this Field are ultimately executed modulus the following irreducible 8-degree polynomial [21, p.104]:

$$X^8 + X^4 + X^3 + X + 1$$

This ensures that the resulting polynomial exists in the form of equation 1, which can be represented as a sequence of 8 bits. The AES algorithm uses only this particular polynomial for this purpose [15].

The specific AES S-Box is generated from $GF(2^8)$ in the following manner [20, p.132]:

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
\end{pmatrix} \times \begin{pmatrix}
y_0 \\
y_1 \\
y_2 \\
y_3 \\
y_4 \\
y_5 \\
y_6 \\
y_7 \\
\end{pmatrix} + \begin{pmatrix}
1 \\
0 \\
0 \\
0 \\
1 \\
1 \\
0 \\
0 \\
\end{pmatrix} = \begin{pmatrix}
z_0 \\
z_1 \\
z_2 \\
z_3 \\
z_4 \\
z_5 \\
z_6 \\
z_7 \\
\end{pmatrix}$$  \hspace{1cm} (2)

To use the S-Box, a given byte is first transformed to a polynomial as in equation 1. Next, its inverse is computed in $GF(2^8)$ and the inverse is put into equation 2 as the
column vector \( y \). The entry in the S-Box is the resultant \( z \) column vector, as the byte:
\[
\begin{pmatrix}
    z_7 & z_6 & z_5 & z_4 & z_3 & z_2 & z_1 & z_0
\end{pmatrix}
\] [20, p.132].

The purpose of the Diffusion vector in equation 2 is to ensure that a given input will never yield the same output, nor the exact complement of the input [20, p.133]. The complete AES S-Box that is used can be found in figure 2 below:

![S-Box](image)

Figure 2: Complete AES S-Box, as built from equation 2. Image scanned from [20, p.130]

The purpose of using the Finite Field described here is so that the S-Boxes used in AES can be generated algebraically; this is different from DES where the S-Boxes were series of random substitutions [21, p.104]. This property is incredibly useful in decryption of AES because the matrices generated by this finite field are invertible, so running Rijndael backwards with the inverse matrices produces a straightforward decryption [20, p.134].

It is also important to note that in actual implementation of AES, the first step, the ByteSub Transform will just do a table lookup on the table in figure 2 by splitting an incoming byte into two 4 bit halves, the first 4 bits representing the row and the last 4 bits representing the column (similar to DES separating the first and last bit from the middle 4 in a 6-bit string), but it is neat to show how the entries in the table are algebraically generated in equation 2 [20, p.129].

The remaining steps in the 4 layers of AES include matrix multiplication operations in the field \( GF(2^8) \), these functions serve to further diffuse the bits and confuse the relationship between the plain text and the cypher text [20, p.130]. Similar to DES, it becomes
very clear when analyzing each internal step of the algorithm why this cryptographic system did not exist before modern computer technology; it is far too involved to ever be done efficiently by hand. Next we look at public response to AES.

### 2.2 The Strength & Implementation of AES

The long review process of AES that culminated in the successful adoption of Rijndael 2001 made AES approved for government use. Additionally, this change made DES only valid for legacy systems [6].

Nobody claims that Rijndael is unbreakable. Unofficial sources have claimed that crack attempts may have been successful up to 6 or 7 rounds\(^{13}\), but since 128-bit AES uses a 10 rounds, it is still very secure, even on the low end of possible key space. Estimates from NIST claim that if a computer can break DES in one second, it would take nearly 150 trillion years for the same computer to crack a 128-bit AES key [6]. For reference, at that projected rate, it would take this computer \(10^{52}\) years to crack a 256-bit AES key. The Big Bang\(^{14}\) is estimated to have occurred just \(1.37 \times 10^{10}\) years ago, so AES is considered secure enough for now.

To further visualize the number of computations cracking an AES key requires, consider that there are an estimated \(10^{23}\) snowflakes that fall worldwide in an average year; the smallest AES key space is \(2^{128}\), meaning that there are 1,000,000,000,000,000,000 times more possible AES keys than snowflakes in the entire world\(^{15}\)!

This notion makes data security a reality, as NIST is quoted to say,

> “Even with future advances in technology, AES has the potential to remain secure well beyond 20 years” [6].

### Summary of DES, AES in Use

DES and AES are both symmetric algorithms that use the same key (either 56, 128, 192, or 256-bits long) in encryption and decryption. They run fast and efficiently on most modern computers, and AES offers current, modern levels of unbreakable security. However, to encrypt and decrypt data, the user must have the key. The secure transfer of a key remains an untouched topic in this analysis.

\(^{13}\)Unofficial claim from various Blogs and Wikipedia contributors

\(^{14}\)According to Wikipedia contributors and general science

\(^{15}\)Estimates as reported by livescience.com
In classical cryptographic systems, the key may have been a known code word, character mapping pattern, or just an equation. In the case of the One Time Pad\(^\text{16}\), the key was a string of random characters of equal length to the message.

With DES and AES, the key is a string of bits that needs to be transmitted securely from the sender to the receiver so that the encrypted message can be decrypted. This secure transmission is the topic of the next modern cryptographic system.

\(^{16}\)Details can be found in Part 1, section 6
3 Public Key Encryption - RSA

While AES is considered secure, there is still a key that must be passed in order to make decryption possible. Transmitting this key in a secure manner can be done through a radically different form of cryptography that is asymmetric [9]. This means that different keys are used to encrypt and decrypt a message.

In 1977, a successful implementation of a previously conceptualized encryption form, now called Public Key Cryptography was presented by Rivest, Shamir, and Adleman [20, p.137]. In their honor, this is called the RSA algorithm. This method is capable of sending a specific message to a receiver without the transmission of a key; in fact, everything that is transmitted is done so openly.

However, this method requires many more resources, making it ideal for transmitting short amounts of information (such as a key), but a poor choice performance-wise for transmitting large blocks of data [9].

3.1 Design of RSA

The RSA algorithm is based on the difficulty of factoring the product of 2 large primes if you do not already know one of the prime factors. By large primes, RSA today uses primes of at least 200 digits in length. The algorithm remains secure because the public key is constructed such that even though the whole world knows exactly how the public key was constructed, they cannot find the private decrypting key [14]. Next, we look at an example RSA implementation to describe this process.

3.2 An Example of the RSA Algorithm

Suppose the secret message to be encoded is HI, first convert it to decimal, as we did with classical cryptographic systems:

“HI” = 08, 09

Where ‘H’ and ‘I’ are the 8th and 9th letters of the alphabet if A = 1, B = 2, and Z = 26.17

1. This message would be encoded as \( m = 0809 \).

2. The intended receiver of the message then creates their public key. He or she does this by choosing 2 prime numbers and computing their product. For this example,

\[ p = 17, q = 23 \]

\[ n = 391 \]

\[ \phi(n) = (17 - 1)(23 - 1) = 312 \]

\[ e = 11 \]

\[ d = 23 \]

The intended receiver needs to send the public key \( (n, e) = (391, 11) \) to the sender of the message.

To send the message, the sender computes

\[ c = m^e \mod n = 0809^{11} \mod 391 = 304 \]

The intended receiver uses their private key \( d \) to recover the message

\[ m = c^d \mod n = 304^{23} \mod 391 = 0809 \]

17This is a special case where \( A \neq 0 \) because we will be using the entire string as a number. If the string started with ‘A’ - then this character would be lost in the computation.
we’ll choose 2 different 2 digit primes (23 and 43) - Using 200 digit primes as would be done in an actual implementation would be difficult to show on paper:

\[ 23 \times 43 = 989 \]

In common notation, \( p = 23, q = 43, \) and \( n = 989. \)

3. Next, the receiver selects a value: \( e, \) such that \( \gcd (e, (p - 1)(q - 1)) = 1: \)
   (This is required so that \( d \) in step 4 is sure to exist [20, 140])

\[
\gcd (e, 924) = 1, \quad \text{e.g.} \quad e = 5
\]

4. Compute \( d, \) such that \( de \equiv 1 \mod ((p - 1)(q - 1)): \)
   This inverse of \( e \) can be shown to be 185, and is known to exist because of how \( e \) was chosen, therefore:

\[
185 \times 5 \equiv 1 \pmod{924} \quad d = 185
\]

Currently, the following values exist:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>43</td>
<td>989</td>
<td>5</td>
<td>185</td>
</tr>
</tbody>
</table>

5. The message receiver now publishes the values of \( n \) and \( e \) as an ordered pair of the form \((n, e)\) while keeping \( p, q, d \) secret. This is the whole idea and namesake behind public key: these values are published to a public key directory that future message senders can access to send encrypted data, in this case, the pair \((n, e)\) make a person’s public key.

6. To encrypt, the message sender computes: \( c \equiv m^e \pmod{n} \) where \( c \) is the encrypted message:

\[
127 \equiv 0809^5 \pmod{989}
\]

The encrypted text in this example is 127. This is now sent to the message receiver. It is important to note there are considerations such as block length that are omitted here for simplicity.
7. The message is decrypted by the receiver with the formula: \( m \equiv c^d \mod n \)

\[
m \equiv 127^{185} \pmod{989} \equiv 809
\]

Since 80 is out of the character space, it is determined the message cannot be '80', '9', so it must be padded with a leading 0 to become: 08,09, so the deciphered text reads, 'HI.'

**Analysis**

Intercepting this particular cypher text and knowing the public pair \((n, e)\) would be trivial to break, but the shear computing power required to factor a 400-digit long \(n\) makes this method secure. Years of investigation into RSA have highlighted some vulnerabilities, but none have rendered RSA vulnerable given today’s technology and knowledge of factoring [17, p.313].

Of the potential attacks, the most notable is a timing attack discovered in 1995 by Paul Kocher (now President of Cryptography Research).\(^{18}\) This attack involves very sensitive timing of the system while it is performing a series of decryptions; Kocher was able to extract information about the decryption exponent, \(d\), by doing this. This was the first of its kind and today Kocher is credited with inventing the Timing Attack. Since then, it has become simple to implement systems that resist RSA timing attacks [17, p.314].

Ultimately, RSA is a widely accepted cryptographic system that has proved secure since its inception; as computer efficiency improves, users just need to choose larger primes to remain secure. Because it is so wide-spread as the asymmetric counterpart to AES encryption, there is a lot of effort put towards discovering new vulnerabilities, but its time tested security continues to drive its practical implementations [17, p.314].

When it comes to modern security standards, we have now seen the approved, recommended cryptographic algorithms for transmitting sensitive data: confidential data is encrypted with the Rijndael algorithm, known as the Advanced Encryption Standard (AES) using a secret key of 128, 192, or 256-bits. This key is then sent using the public key infrastructure utilizing the RSA algorithm. The intended message recipient receives the AES key that was created with their public key, so they first decrypt the AES key through the RSA algorithm, and obtain the AES key to decrypt the cypher text. An unintended recipient is capable of intercepting a jumbled message in the form of AES-encrypted cypher

\(^{18}\)Cryptography.com
text and the RSA encrypted key for the cypher text. Solving either of these intercepted pieces is generally considered impossible for at least the next 20 years\textsuperscript{19}, and after that, still most unlikely, so this process is considered secure by modern day standards.

There is one more interesting theoretical method that could potentially be truly unbreakable, presented next in closing for this part.

\section*{3.3 Theoretical One Time Pad Application of RSA}

Looking back at classical cryptographic systems, the One Time Pad in part 1, section 6 is truly unbreakable, but requires an unrepeatable random bit generator. A variation on this idea incorporating RSA was developed by Maurer, Rabin, Ding, and others that relies on RSA's strength to sidetrack and delay an intercepting party in message passing [20, p.40].

The idea requires a random bit generation source that both the sender and receiver can access at the same time, for example, a satellite. The satellite would produce a random bit pattern (as close to random as possible with a computer) at speeds that would not allow for intercepting machines to cache the entire sequence of random bits (this is the 'theoretical' part - but with enough computing power on the satellite, this is possible).

The sender would next determine a method to sample bits from the Satellite (for example: record a bit every 3 seconds); he or she would then use the RSA algorithm to send this small sampling formula to the receiver. The two parties would then record the random bit string according to the sampling formula until they built a random bit string of equal length to the message.

The sender then uses this random bit string as the key in a one time pad encryption of the message, and sends the cypher text. The receiver has sampled the key already and can use it to decrypt the cypher text.

By the time a third party has successfully broken the RSA-encrypted sampling formula (an infeasible task), the particular bits that were generated by the satellite at the time of original sampling are long gone, and by the (theoretical) satellites design, there can exist no database of what was generated, so reproduction of the key is impossible; therefore, the cypher text that was intercepted is actually unbreakable! [20, p.40].

\textsuperscript{19}As quoted by NIST in [6]
Part III

Data Compression

1 Background

While the history does not start as early as classic cryptography, the first notions of data compression were developed in 1838, which for technology could still be considered ancient: This early form was Morse Code [22]. The idea incorporated is very simple: shorter codes existed for the common English letters such as ‘e’ and ‘t’ [22]. The idea was improved in the mid twentieth century when Information Theory was introduced by Claude Shannon\(^{20}\) in 1949 [22]. This involved substituting smaller codewords for message content blocks based on probability.

In 1952, Huffman Coding optimized the practice of assigning smaller codewords to blocks with high probabilities of occurrence in a text [16, p.492]. This idea was revisited in the 1970s when it was realized that the assignment of smaller codewords need not be based on probability of blocks occurring, but rather codewords should be assigned dynamically, with text characteristics unique to a given piece of data [22].

This presentation of Data Compression will look specifically at the popular, influential methods of data compression that have been designed and implemented since the 1970s. First, let us look at exactly what exactly data compression does.

1.1 Principles of Compression

While computer technology will always improve and consistently give consumers access to faster computers and larger storage drives, the advantages of smaller files will always be evident:

- Requires less space on disk than an uncompressed file.
- Will transfer faster than an uncompressed file.

In the beginning of the digital age, standard storage capacity was very limited; large file sizes were cumbersome and difficult to handle. Today, while storage capacity is consistently increasing and large capacity is very affordable, the number of digital files is growing rapidly as well. For these reasons, compression methods for digital files have existed since the early days of computing [22].

\(^{20}\)More information in section 2.
1.2 Lossless & Lossy

There are two overarching forms of data compression: Lossless and Lossy. The difference is the ability to get back the exact original contents of the file. At first, one may argue that compression need be lossless and return the exact original contents to be an effective method, but in the case of image, video, and audio files, the omission of many bytes goes unnoticed by the human audience. Text files (typed documents, source code) cannot afford to lose any bytes in file compression, or critical functionality of the document may be damaged [16, p.492].

While there is much to explore in the realm of Lossy compression, this study will focus on Lossless methods on text files. This makes these methods more of a parallel to cryptographic methods in the previous chapters, which required the decrypted text to exactly match the original plain text.

To begin analysis of text compression methods, a measuring method must be declared. Similar to the level of complexity of encrypted text, compressed files can be very well compressed or poorly compressed; to compare these, a metric must be defined.

2 Evaluation of Compression

How does one measure compression? A simple manner is to compare the uncompressed size of a message to the compressed size in a ratio [12, p.420]:

\[
\text{Compression Factor} = \left(1 - \left(\frac{\text{compressed size}}{\text{uncompressed size}}\right)\right) \times 100
\]

This formula gives the ability to say that a particular method compressed a message or file by the ‘Compression Factor’.

For more involved evaluation, another field of study known as Information Theory must be examined [21, 54]. Claude Shannon, a researcher at Bell Laboratories in the 1940s worked in this field and developed the idea of Entropy [12, p.430].

According to Douglas Stinson,\(^{21}\) “Entropy can be thought of as a mathematical measure of information or uncertainty, and is computed as a function of a probability distribution” [21, p.54]. To evaluate compression quality, entropy can be used to determine the amount of content in a message; the higher the entropy of a message, the more information the message contains [12, p.430].

\(^{21}\)Author of Cryptography: Theory & Practice [21]
The average Entropy over a message can be computed with the following formula [21, p.5]; [12, p.430]:

\[ H(X) = \sum_{x \in X} -Pr(x) \log_2 Pr(x) \]  

(1)

where \( X \) represents the message and \( H(X) \) is the average symbol entropy in the entire message. \( x \) represents a single character in the message \( X \) while \( Pr(x) \) represents the frequency of \( x \) in the message, or the probability of that letter computed as: \( \frac{\text{Occurrences of } x}{\text{Length of } X} \).

Simplifying this formula, one can find the entropy of a single character (the symbol entropy) by removing the weighted average of the summation and simply looking at each character:

\[ H(x) = -\log_2 Pr(x) \]  

(2)

In this manner, \( H(x) \) represents the minimum amount of bits required to express character \( x \). To find the theoretical minimum for compression, we can take the summation of each symbol entropy in a message \( X \):

\[ \text{Minimum Number of Bits} = \left\lceil \sum_{x \in X} -\log_2 Pr(x) \right\rceil \]  

(3)

We must take the ceiling of this summation because only integers can be represented in simple binary \[12, p.431\].

An example of this summation can be found in section 3.2, an example Huffman Coding analysis.

\[22\] A topic to be discussed further with Arithmetic Encoding
3 Huffman Coding

Huffman coding relies on counting characters in a file and representing the more common characters with shorter character codes[16, p493]. Different codes for letters are best represented as a binary tree.

Huffman Encoding Tree Example

Let us encode the word Mississippi. First, letters can be tabulated and sorted by frequency:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Count</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>1</td>
<td>$\frac{1}{11}$</td>
</tr>
<tr>
<td>P</td>
<td>2</td>
<td>$\frac{2}{11}$</td>
</tr>
<tr>
<td>S</td>
<td>4</td>
<td>$\frac{4}{11}$</td>
</tr>
<tr>
<td>I</td>
<td>4</td>
<td>$\frac{4}{11}$</td>
</tr>
</tbody>
</table>

Table 1: Frequencies of Letters in MISSISSIPPI

The Huffman encoding algorithm then begins by grouping frequencies into pairs, and representing them on a Binary Tree:

The first step is to separate the highest frequency from the rest, in our case, the I and S have the highest frequency, so from the order above, we start with the I and separate it from the rest of the letters, grouping the others to form MPS* (Where the * denotes multiple letters in that frequency):

**Step 1:** MPS* $\left[\frac{7}{11}\right]$, I $\left[\frac{4}{11}\right]$

A tree can then be started with the following branches:

```
  0
 / \ 1
MPS* I
```

The tree is formed by simply putting the high frequency letter on the branch with label 1 and the group of lower frequency letters on branch 0. Next, MPS* is separated into two branches: The single highest frequency (S) and the rest, MP*:

**Step 2:** MP* $\left[\frac{3}{11}\right]$, S $\left[\frac{4}{11}\right]$
Adding this to our tree in the same manner as above shows:

Finally, MP* is broken into $M$ and $P$ and attached to the tree following the same algorithm, putting higher frequency $P$ on the branch with label 1 and $M$ on 0:

**Step 3**: $M \begin{pmatrix} \frac{1}{11} \\ \frac{1}{11} \end{pmatrix}$, $P \begin{pmatrix} \frac{2}{11} \\ \frac{2}{11} \end{pmatrix}$, $S \begin{pmatrix} \frac{4}{11} \\ \frac{4}{11} \end{pmatrix}$, $I \begin{pmatrix} \frac{4}{11} \\ \frac{4}{11} \end{pmatrix}$:

To obtain the character codes, the tree is traversed from top to bottom until the character is reached. The codes are tabulated below:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>S</td>
<td>01</td>
</tr>
<tr>
<td>P</td>
<td>001</td>
</tr>
<tr>
<td>M</td>
<td>000</td>
</tr>
</tbody>
</table>

**Table 2**: Huffman Encoding of MISSISSIPPI
With this encoding scheme, the word Mississippi can be written in 21 bits as:

\[
\begin{array}{cccccccccccc}
M & I & S & S & I & S & I & P & P & I \\
000 & 1 & 01 & 01 & 1 & 01 & 01 & 001 & 001 & 1 \\
\end{array}
\]

It is important to note that if using the previous tree, this string can only be interpreted in one manner: to represent the word Mississippi. Even though these character codes are of different length, a parser can only interpret 000 as an 'M' because there are no codes of 00 or 0 to cause confusion. This is a feature of binary trees.

### 3.1 Analysis

So how good was this compression of the word Mississippi? A quick manner of assessing it is to count the total number of bits that would otherwise be required. Since there are 4 different letters in the word: M, I, S, P we require 4 different symbols to fully represent the word. This means 2 bits to represent each letter:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>00</td>
</tr>
<tr>
<td>I</td>
<td>01</td>
</tr>
<tr>
<td>S</td>
<td>10</td>
</tr>
<tr>
<td>P</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 3: 4 Symbol binary Representation of MISSISSIPPI

Using this table, we would write the word as:

\[
\begin{array}{cccccccccccc}
M & I & S & S & I & S & I & P & P & I \\
00 & 01 & 10 & 10 & 01 & 10 & 10 & 01 & 11 & 11 & 01 \\
\end{array}
\]

This representation, requiring 2 bits per character uses 22 bits. The previous method only used 21 bits. Using the compression factor formula from section 2, we can say that we achieved a compression factor of 4.545%. This is a trivial improvement in this simple word, but can offer significant improvement in a larger document with more characters.

### 3.2 Entropy Analysis

Recalling the notion of total symbol entropy in the previous section, we can determine the theoretical minimum number of bits in the perfect compression on the word MISSISSIPPI
with the following equation:

$$\text{Minimum Number of Bits} = \left\lceil \sum_{x \in X} - \log_2 Pr(x) \right\rceil$$  \hspace{1cm} (4)$$

Where $x$ represents a character in $X = \text{MISSISSIPPI}$ and $Pr(x)$ is the frequency of character $x$ from Table 1.

We must remember to take the ceiling of this minimum bit number. In the case of MISSISSIPPI, this looks like:

$$\sum_{x \in \text{MISSISSIPPI}} - \log_2 Pr(x) = 20.05 \rightarrow \lceil 20.05 \rceil = 21$$

So, while the compression factor was under 5%, it still achieved the absolute minimum number of possible bits to represent this string. It must be remembered that this was for a single word, in the case of a full length document, modern compression rates are 3:1 [22].

## 4 Arithmetic Coding

To achieve better compression than Huffman coding, arithmetic coding utilizes decimal numbers to represent text. Not limited by the bounds of integer expression in computers, which requires rounding all decimal numbers to the nearest whole integer to be represented as bits, Arithmetic Coding accepts real numbers [12, p.436].

Arithmetic coding works by defining a range for each character based on its frequency. To see how it works, we’ll encode the short word, MEET. Arithmetic Coding is fast on a computer, but tediously slow by hand, so we choose the word MEET because it has 4 total letters, but 3 unique letters, so it can show compression.

First, compute the probabilities and intervals of each character, the lowest possible interval for a character to occupy in MEET is $\frac{1}{4}$ (because there are 4 characters). A character’s interval length in this word equivalent to $\frac{1}{4} \times \text{Total Occurrences}$ [12, p. 436]:

<table>
<thead>
<tr>
<th>Character</th>
<th>Probability</th>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>0.25</td>
<td>0.00 – 0.25</td>
</tr>
<tr>
<td>E</td>
<td>0.50</td>
<td>0.25 – 0.75</td>
</tr>
<tr>
<td>T</td>
<td>0.25</td>
<td>0.75 – 1.00</td>
</tr>
</tbody>
</table>

Table 4: Arithmetic Coding Intervals for MEET
To encode the string, the total interval, $[0 - 1]$, is divided repeatedly and the character high and low interval values are added to the encoded value in the order that the encoder reads each letter. This process is difficult to follow by hand, but the pseudocode exists in figure 2 below:\footnote{Image from [12, p437]}

\begin{verbatim}
ALGORITHM Arith_Code (Message)
  HiVal ← 1.0 /* Upper limit of interval. */
  LoVal ← 0.0 /* Lower limit of interval. */
  WHILE (more characters to process)
    Char ← Next message character
    Interval ← HiVal - LoVal
    CharHiVal ← Upper interval limit for Char
    CharLoVal ← Lower interval limit for Char
    HiVal ← LoVal + Interval * CharHiVal
    LoVal ← LoVal + Interval * CharLoVal
  ENDMETHOD
  OUTPUT (LoVal)
END Arith_Code
\end{verbatim}

Figure 2: Pseudo Code for Arithmetic Coding

Following this psuedo-code, the proceeding table is generated with the referenced charLoVal, charHiVal, LoVal, and HiVal variables:

\begin{center}
\begin{tabular}{|c|c|c|c|c|}
\hline
Character & Interval & charLoVal & charHiVal & LoVal & HiVal \\
\hline
M & 1.0 & 0 & 0.25 & 0.00 & 1.0 \\
E & 0.25 & 0.25 & 0.75 & 0.0625 & 0.1875 \\
E & 0.125 & 0.25 & 0.75 & 0.19375 & 0.15625 \\
T & 0.0625 & 0.75 & 1.00 & 0.140625 & 0.15625 \\
\hline
\end{tabular}
\end{center}

Table 5: Arithmetic Coding of MEET

The final LoVal is the encoded message, in our case, MEET encodes to 0.140625. The details of the arithmetic encoding algorithm, while difficult to follow on paper, are easily implemented in a computer, it reduces to simple multiplication and addition operations across the entries in the table (as can be seen in the pseudo-code). Decryption is similar, just in reverse, looking to match successively divided ranges with character high and low values in frequency. The important quality of arithmetic coding is that it does not represent probabilities (real numbers) with integer values, but rather can represent them exactly with decimal values [12, p.236].

It must noted, however, that error checking is critical in arithmetic encoding compression/decompression algorithms. Computers see arithmetic encoded data as numbers in
floating point representation - this leads to potential problems of underflow in addition and sometimes missing a zero condition [12, p.439]. As long as error checking is accounted for, arithmetic encoding obtains better compression ratios because it can represent exact probabilities.

While arithmetic coding produces near perfect compression, it is far slower than Huffman Encoding due to extra steps involved when dealing with floating point numbers in the computer. Often times, speed is a primary concern in data compression, so limiting the number of times an input message is scanned from beginning to end for frequency and repetition counts can provide the most desirable improvements in a compression method [12, p.439].

Compression algorithms known as dictionary methods offer this improvement.
5 Dictionary Methods

5.1 Ziv-Lempel

The Ziv-Lempel method, or LZ as it is commonly called is a dictionary compression method; the principle is to substitute repeated words or phrases in a text with a reference to the location in the text where the word or phrase was first introduced [12, p.439]. Jacob Ziv and Abraham Lempel published the first version in 1977, which is known today as LZ77 [12, p.439].

5.1.1 LZ77

LZ77 or LZ1 is unique because it uses the text as its own dictionary, not requiring an external reference structure. This is often called pointer-based encoding because the text contains pointers to previous sections [22]. If the parser sees a string that it has previously encoded, it simply writes a pointer to go back to that section, then copies the proper number of characters to repeat the string [16, p494].

The method operates with two structures: The text window and a look-ahead buffer. In practice, these span several kilobytes of data. The look ahead buffer reads the text to be encoded while the text-window holds the encoded data, which acts as the dictionary. If data in the look-ahead buffer matches data found in the text-window, then a dictionary pointer to the location of the first instance of this text is written instead of a literal character [12, p.441]

The basic idea of pointer-based encoding can be seen in the following example:\(^{24}\)

Using Mississippi as an example again, we can see the power of repetition in LZ77:

1. The text window is empty and the entire word is in the look ahead buffer:

   ![Mississippi](image)

2. The first letters are read into the text box:

   ![Mississippi](image)

\(^{24}\)[16, p494]: Example taken from Encyclopedia of Computer Science
3. The buffer now contains a phrase that already exists in the text box, \textit{ISS}. This is written in the text box as a pointer to the first time that this phrase occurs in the text box, and how long the phrase is. The pointer has the form: \((x, y)\), signifying to go back \(x\) characters and write \(y\) characters (or in this case, go back to \(I\) and write \textit{ISS} before moving on):

\[
\begin{align*}
\text{MISS}(3,3) & \quad \text{IPPI} \\
\text{text box} & \quad \text{lookahead buffer}
\end{align*}
\]

4. The next character, \(I\) already exists in the text box. Furthermore, it already has a pointer referencing it. As the word is decompressed, \(I\) will be the first character to be written because it occurs at the pointer offset 3; therefore, the pointer can be incremented as such to include \(I\): \(^{25}\)

\[
\begin{align*}
\text{MISS}(3,4) & \quad \text{PPI} \\
\text{text box} & \quad \text{lookahead buffer}
\end{align*}
\]

The word is finally encoded as:

\[
\text{miss}(3,4)\text{ppi}
\]

It should be noted here that double letters such as \textit{ss}, \textit{pp}, or the two different \(i\) characters that exist in the encoding could be written as pointers with this algorithm, but a single character is rarely encoded because it is a single byte and this method is more concerned about phrases of multiple characters.

When \textit{miss}(3,4)\textit{ppi} is decompressed, the 3 tells the parser to look back 3 characters (to the first ‘i’) while the 4 tells the parser to copy the next 4 characters (‘issi’) \cite[p494]{16}. This can be seen by walking through the steps of decompression:

1. Decompression starts with this:

\[
\begin{align*}
\text{miss}(3,4)\text{ppi} & \rightarrow \\
\text{compressed} & \quad \text{parser} \\
\text{012345678910} & \quad P,C
\end{align*}
\]

\(^{25}\)This is explained in greater detail in decompression
The $P$ in the third row represents the current pointer index and the $C$ represents the cursor (where the next character is to be placed). The second row shows position, and the top row will be the plain text.

2. When the parser sees plain characters, it simply writes them as literal characters:

```
(3,4)ppi  \rightarrow  \begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array}
```

3. The pointer $(3, 4)$ tells the parser to move the pointer back 3 characters in the plain text (to $i$ at position 1), and append the character it finds there, decrementing the pointer. The cursor increments to position 5:

```
(3,3)ppi  \rightarrow  \begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array}
```

4. Now the pointer reads $(3, 3)$, so it looks back 3 from the current cursor at position 5, to position 2, writing an $s$ in position 5, decrementing the counter and incrementing the cursor:

```
(3,2)ppi  \rightarrow  \begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array}
```

5. Similarly, the cursor decreases to $(3, 1)$ and writes the another $s$:

```
(3,1)ppi  \rightarrow  \begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array}
```
6. With the pointer \((3, 1)\), the parser looks 3 characters back from the cursor \((7 - 3 = 4)\), and writes the \(i\) it finds there, ultimately decrementing the pointer to \((3, 0)\):

\[
\begin{array}{cccccccccccc}
\text{m} & \text{i} & \text{s} & \text{s} & \text{i} & \text{s} & \text{s} & \text{i} & \text{-} & \text{-} & \text{-} \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array}
\]

The compressed pointer is \((3, 0)\) and the parser writes in the row, resulting in the compressed string:

\[
(3,0)\text{ppi}
\]

7. Finally, the parser copies the remaining plain text, \(ppi\):

\[
\begin{array}{ccccccccccccc}
\text{m} & \text{i} & \text{s} & \text{s} & \text{i} & \text{s} & \text{s} & \text{i} & \text{p} & \text{p} & \text{i} & \text{i} \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\end{array}
\]

This particular example is interesting to walk through because when the parser starts, there are only 3 characters before the pointer, but the first letter it copies is the ‘i’, so once it makes its first copy, it will then copy the whole string: ‘issi’, which will complete the word [16, p494].

Another word that would be encoded similar is \(\text{banana} \rightarrow \text{ban}(2,3)\). Where the pointer tells the parser to look back 2 characters (to a), then copy the next 3 characters, which will be \(\text{ana}\) after the first a is written.

Particular analysis of this method is difficult to reduce to a single byte count ratio because of how the data is stored: Different implementations of LZ77 use various syntax to denote literal characters from pointers, so the exact byte count varies [12, p.442].

One aspect to notice is that compression ratios are not consistent: any particular method does not guarantee a particular compression ratio because it is completely dependent on the given text to be compressed. In this example, the 11 character word ‘Mississippi’ is reduced to 7 characters and a pointer. The word ‘documentary’ is also 11 characters in length, but will not compress because there is no repetition.

For improvement on LZ77, Ziv and Lempel published a method the following year (1978) known today as LZ78 [12, p.442]
5.1.2 LZ78

The improvement in LZ78 was the move away from a text-window of fixed length. With LZ77, when the text-window was full, it had to be written to disk and cleared - even if the next block of text was perfect repetition of the previous, the window was cleared and the process restarted.

Therefore, the larger the text-window is, the higher the level of compression can potentially be; conversely, the size of the text-window will always bound the level of compression that can achieved.

LZ78 uses a structure called a trie (pronounced “try”) instead of a text-window to encode data. The encoded data is then written as references to the data in the trie. A trie is a n-ary tree, which allows for each node to have multiple child nodes [12, p.746]. An example of a trie is found in figure 3 below.²⁶

![Figure 3: A trie of types of birds](http://www.nctvu.cn/kejian/sjjg/10.3.htm)

The trie in figure 3 shows the unique entries of bird types. For example, bluebird and bunting are found under the top level node b, then separated at level 2, between l and u, as the second characters. Similarly, thrasher and thrush exist on the branch t → h → r, then separate by a and u as the fourth character.

LZ78 compression slowly builds this trie as it parses the data. It is assumed that the top level node contains the entire alphabet, so encoding starts on the second level.

For example, the first time the parser sees the word “gull”, it adds u under the top level node, g. The second time it sees the word, it adds l under “gu”, if it sees the word

²⁶Image from http://www.nctvu.cn/kejian/sjjg/10.3.htm
a third time, the full string "gull" is added to the trie. If it comes across this word again in the text, it simply outputs a pointer to this node on the tree and decompression will print the word "gull" in place of the pointer [11].

It can be seen in this example that the entire tree is built as the algorithm parses the data once; this is an improvement in speed over statistical methods that require knowing character frequencies.

Since each node can potentially contain the entire alphabet in child nodes, the restricting text-window limitations of LZ77 are removed; however, this raises concern of a potential compression flaw in LZ78: extra-large pointers [12, p.443].

If pointers in the trie become too large, then they may require more bits than simply writing the literal characters to disk, this defeats the original purpose of compression and was addressed in a next generation known as LZW.

5.1.3 LZW

In 1984, Terry Welch presented LZW (Lempel-Ziv-Welch) encoding, which had strict regulations on the generation of pointers within the LZ78 trie [12, p.443]. Welch achieved this by declaring a trie of 4096 entries; the first 256 entries in the dictionary refer to ASCII characters (regular bytes), the rest of the dictionary is used for longer strings. [13],[10].

LZW is used in some of today’s common formats: TIFF, PDF, and even implemented in hardware on modems [12, p.443]. This optimized compression algorithm was developed when Welch was an employee of Sperry Computer Corporation, which is now Unisys [12, p.443]. With corporate ownership, the LZW compression scheme is responsible for the following well-known quibble in computer history.

GIF Royalty Debate

It is important to note a difference here between compression and encryption: The level of compression data achieves is irrelevant to National Security; therefore, data compression is not standardized and methods are not distributed openly to the public like encryption.

Most compression methods exist within compression programs that are available to all computer users and distributed freely; additionally, specific file formats are often built on a particular compression scheme.

The Graphical Interchange Format (GIF) image file type uses the LZW compression algorithm making it an efficient, lossless image file [12, p.443]. This prompted a legal ownership dispute over the use of GIF because Welch was an employee of a Unisys company at the time he developed LZW [12, p.443].
Due to its strong compression ratio with LZW, GIF images spread rapidly as they were ideal for the web; Unisys exercised their right to patent their product and requested royalties each time GIF was used by a high-volume user or a service provider. This request was not well received among the web-community\(^{27}\) and some users have chosen to boycott the GIF format. Legal issues over the GIF image format were a component in the quick development of the new format, PNG (portable network graphics) that uses a different compression algorithm. PNG files are very prevalent on the web today and are open source. Most believe that GIF will soon disappear and PNG will represent the main lossless web image format \([12, \text{p.444}]\)

PNG

In a PNG file, the information is reduced by Huffman Coding and then a LZ77 compression occurs with a 32 kilobyte text-window. PNG is mostly used on the web due to its strong compression ratio – typically a 5% - 25% improvement over GIF. Additionally, users can choose between a faster or better compression ratio when creating the file \([12, \text{p.444}]\).

The notion of combining the compression algorithms discussed in this chapter brings us to modern lossless compression methods.

\(^{27}\text{Akin to the debate with the current open source movement.}\)
6 Data Compression Today

First, the data compression algorithms analyzed in this chapter were chosen for their flexibility and general applied use; the field of data compression is massive and ubiquitous with today’s technology. However, much of the implemented compression today requires specialized hardware; the telephone communication network is an example of this type of hardware implemented lossy compression [11]. The algorithms focused on here are implemented with common computer hardware on the average user’s machine.

Briefly, all commonly-used compression programs use a combination of either LZ77 or LZ78 with a statistical compression (Huffman or arithmetic); simply mixing and matching different methods is not advised for good compression, but fine tuned combinations of these methods create everyday compressed file formats that the end-user is familiar with, such as: .zip [13].

It may seem obvious, but when all bits do not need to be reconstructed perfectly, compression can become far more efficient. This is unreasonable for textual data, but lossy compression schemes that exist for digital images, music, and movies can offer compressions typical compression ratios of 20:1 [22]. However, their structure is entirely different: searching instead for data that can be discarded and unnoticed to the human audience, not redundant data that can be expressed more efficiently.

Now that we have seen how general lossless data compression works, let's conclude this analysis of cryptography and data compression by looking at a combination of the two processes.
Part IV

Cryptography & Data Compression Together

1 The Two Processes

At the beginning of this analysis, I posed the question:

Due to the apparent similarities in the structure of the methods, is there a commonality in data compression and cryptography? Is there a feasible manner in which the two are or can be combined, creating an efficient method of combined compression and encryption?

Having presented the development and methods of both fields, we may now address this question.

The main similarity between data compression and cryptography exists within the compressed or encrypted file: To a human observer, this file is unreadable data, and therefore useless until it undergoes a reversing process: either decompression or decryption.

However, it quickly becomes apparent that we are comparing apples to oranges. The priorities of each process are very different, as seen below:

<table>
<thead>
<tr>
<th>Cryptography</th>
<th>Data Compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. Practicality (Availability)</td>
<td>2. Convenience (Speed)</td>
</tr>
<tr>
<td>3. Convenience (Speed)</td>
<td>3. Security</td>
</tr>
</tbody>
</table>

Figure 1: Cryptography vs. Data Compression Prioritized List

Here, one can see that a combination method would require some form of sacrifice or compromise in security or compression factor - ultimately detrimental to the end result.

That is, if data is important enough to need incredibly sophisticated encryption, then the contents are important enough that the file size is unimportant. If security is being
sacrificed for better a better compression ratio, then the security of the data was never a main concern.

Therefore, we can conclude that the two processes, despite their similar structure, do not aim to effect the same result, so their direct combination is unnecessary. The immense complexity in modern cryptographic methods leaves no room to attempt to compress data - additionally, if a compression method was somehow injected into the encryption algorithm, there would be a known structure and pattern to the cypher text – which completely undermines the original encryption.

However, this brings to light another question, the question that really should have been asked:

*How can data compression and cryptography be effectively combined to offer maximum security within the smallest file?*

We can also answer this question, and the result is far more interesting.

Revisiting the individual priorities of each process, it can be shown that the two processes both can and should be used in series. In a review of the Pretty-Good-Protection routine, researchers were less successful in cracking encryptions that were first compressed [7].

Recall that (lossless) data compression works by reducing redundancies in the file. The resulting compressed file does not emulate typical character frequencies, as redundant characters have been extracted. Encrypting a compressed file can actually yield more security than simply encrypting the plain text.

Additionally, it should be noted that good encryption is a lengthy process, compressing a file first provides convenience [18, p226].

It should be noted here that compression must occur first; since a strong encryption method eliminates patterns and redundancies in the cypher text, encrypted files have horrible compression ratios.

In this manner, Bruce Schneier recommends that users follow these steps in secure data transfer:[28]

```
Compress → Encrypt → Error Control ⇒ Error Control → Decrypt → Uncompress
```

Figure 2: Data Compression and Cryptography together

[28]Figure from Schneier [18, p.226]
Error Control in figure 1 simply ensures that the message on the receiving end perfectly matches the original message; if the encryption is strong, a small discrepancy in the received data will not allow for proper decryption.

This concept of compression before encryption, though seemingly trivial and obvious in hindsight, is important for computer users to understand to effectively utilize both processes.

1.1 A Hybrid Attempt – Proof by Counterexample

During my early understanding of these processes, I was under the impression that a hybrid method must certainly exist. If I found no published form in readings, I would perhaps suggest a simple model. Now, I see exactly why this concept failed, and can use it to show the separate natures of these processes.

This hybrid would consist of a dictionary method of compression in which the dictionary (the key to decompression) was then transmitted separately, akin to a key being transmitted separately from the encrypted text. This could simply be an implementation of LZ78 or LZW where the trie is transmitted separately. At first, this seems like a form of cryptography, the file has been altered and requires a form of reassembly to be viewed; however, only repetitive content has been extracted from the text, so much of the original document will be transmitted as plain text.

To solve this, the text must still undergo some further form of encryption before being transmitted. At this rate, encryption and compression both still occur, but just at different times, on independent sections of the data.

This example of a combination method highlights the failure in this concept and corroborate the earlier conclusion that when used in series, compression then encryption offers maximum performance.

2 Hash Algorithms

It should be known that there exist techniques that are often used in computer security that yield files of a specific length - which can be a form of compression: A hash function takes an input of variable length and through the algorithm produces a value of fixed length in a given range [20, p182]. Mathematically, a hash function looks like:

\[ f : A \rightarrow B \text{ where } |A| > |B| \]  (1)
where $f$ is a function that maps $A$ to $B$ and $B$ contains less elements than $A$. Therefore, multiple objects in $A$ may represent the same object in $B$. This is called a collision [9]. The quality of a hash algorithm is measured by the methods ability to minimize collisions and given a wide variety of data sets ($A$), scatter the result in the most uniform and unpredictable manner over $B$.

They are cryptographically significant because they are designed to be one-way functions such that taking a value in $B$ and identifying the original input from $A$ that generated it is computationally infeasible [20, p.182].

Hash algorithms are typically used to verify that a given piece of data has integrity; that is, with a good hash algorithm, a minor change in the input creates a very different output. For this reason, hashes are commonly used to create a digital signature [20, p182].

Hash functions offer a different type of service in security from cryptography, but are worthy of mentioning because they create fixed length data from variable length input.

3 Combining the Two in Lossy Compression

As mentioned briefly earlier, compression techniques change drastically when the data does not need to be reconstructed perfectly. Appropriate only for audio, video, and images, lossy compression techniques can achieve typical rates of 20:1 compression by discarding information that goes unnoticed to the human observer [22].

A publication from the Graduate School of the University of Aizu proposed a study of possible integration of data compression and cryptography after noting the two fields are typically studied separately [8]. Their method relies on an image compression method called k-PCA. In this compression, the image is broken down into sub blocks and then redundancies found among the sub blocks are discarded. Encryption occurs immediately after compression, but on just the principle components of the compression, not the entire compressed file. It is noted that AES would be a secure encryption to use in this method [8].

Their method is similar to the Hybrid Attempt proposed earlier. The idea is that without the key to reconstruct the blocks and identify which redundant sections were removed, the image is useless. The key, therefore, is all that needs to be encrypted because the compression scheme manipulates the original data to such a degree that it is useless without the decompression key. However, this form applies specifically to image files.

Furthermore, the researchers mention a separate image compression/encryption scheme
suggested in the 1990s that works with lossless compression; specifically images, but still lossless data. However, the downside of this apparent scheme is that the level of compression is inconsistent depending on the complexity of the data [8]. This corroborates the dangers of compromising when aiming to compress and encrypt with a single function.

The research conducted by the University of Aizu in [8] shows implementable examples of my previous theorized notion of combined compression/encryption, but also points out the dangers and quality sacrifice that occurs, further encouraging one to separate the processes and compress before encryption.

Next, I’ll finish my analysis with a glimpse into the future of this field.

4 The Future of Data Compression / Cryptography

A major buzz word in computer science research and development right now is Quantum. The field of quantum cryptography uses modern physics to ensure randomness and security. Real implementations of quantum key distribution have already been developed and tested [18, 557]. Quantum computers are under development as well and will dramatically change what is thought possible and impossible in the computing realm.

Quantum cryptography operates on quantum mechanics which claim that particles never exist in a single place. Rather, particles are defined by probabilities of being in one place or another. When a scientist observes a particle, he or she only measures its position, not its velocity. Furthermore, due to “fundamental uncertainty . . . the very act of measuring it destroys any possibility of measuring the other quantity” [18, 555].

To apply this to cryptography, two people can transmit photons across a network and record their positions as coming through specific polarized filters. What comes across the network is not necessarily secure, but what is recorded will be. If the network is tapped, then the third malicious observer is recording the photon’s position as well. The simple act of this third party observing the photon forever changes the stream and the original sender and receiver will be able to tell that the line is insecure [18, 555].

Due to the random nature of the photon’s position, the recorded string can be considered random and secure and used as a symmetrical key or as a one-time-pad for absolute security. The implications of quantum cryptography will forever change the transfer of confidential data. If an eavesdropper is incapable of obtaining the data (bounded by laws of physics), then security, appears to be a guaranteed reality. AES, Triple-DES, RSA all only offer security in the amount of time it would take to crack the data - not a physical guarantee.


5 Conclusion

This analysis of data compression and cryptography in the digital age answered the original question, *Is there an efficient combination method of encryption and compression?* with a simple answer: *It is self-defeating to compromise.*

The two processes have different priorities. Combining them forces compromise and desegregation in respective quality. However, evidence and convenience both suggest that when the processes are done in series: files are first compressed and then encrypted, the processes can work off each other, yielding a better result.

Running these processes in series requires no further effort from the user as the topics discussed here were never presented in end-user form. When the common user interacts with these methods, they typically take the form of a check box in a save-dialog box. Checking the ‘Compress’ or ‘Encrypt’ box will cause the program to run the methods discussed here.

Additionally, while this information was new to me, application programmers have known for long that compression before encryption is always more effective, so checking both boxes will typically result in the proper sequence.

**Following Material**

1. Bibliography

2. Appendix - Java Source Code used for various examples and understanding in writing this thesis.
References


Page 51


Appendix

For a better understanding and to generate models for explanation, I wrote some of the encryption algorithms discussed in Java. The source code for each encryption method or frequency analysis counter can be found here. Note that most of these methods are done modulus 128 to account for ASCII, as opposed to modulus 26 as discussed in the examples.

Shift Cypher

//This class shifts the ASCII value of the Byte Array in the Tank.

//Requirements
//   >>Must have data in the inputBytes Array.

class ShiftCypher{

   //Variables
   private byte shiftValue;
   private FuelTank tank;

   //Constructor
   public ShiftCypher(FuelTank tank, int shiftValue){
      this.shiftValue = (byte) shiftValue;
      this.tank = tank;
      System.out.println("Shift Cypher Initialized "+
            "with shift value: " + shiftValue);
   }

   //Running the Function
   public void run(){
      //Encrypt
      System.out.println("Calling Encrypt");
      encrypt();

      //View Bytes
      //System.out.print("Encrypted Bytes: ");
      //tank.printBytes();

      //View Chars
      //System.out.print("Encrypted Chars: ");
      //tank.printBytesToChars();

   }
}

A 53
// Decrypt
System.out.println("Calling Decrypt");
decrypt();

System.out.println("Finished Decrypt");

// View Bytes
// System.out.print("Decrypted Bytes: ");
// tank.printBytes();

// View Chars
tank.printBytesToChars(1000);
}

/****************** AVAILABLE METHODS ******************/

// Encryption Algorithm
public void encrypt(){
    byte encryptArray[] = new byte[tank.byteArray.length];
    for (int i=0; i<tank.byteArray.length; i++){
        int encrypt = (tank.byteArray[i] + shiftValue) % 128;
        if (encrypt < 0){
            encryptArray[i] = (byte) (128 + encrypt);
        }else{
            encryptArray[i] = (byte) encrypt;
        }
    }
    tank.byteArray = encryptArray;
}

// Decryption Algorithm
public void decrypt(){
    byte decryptArray[] = new byte[tank.byteArray.length];
    for (int i=0; i<tank.byteArray.length; i++){
        decryptArray[i] = (byte) ((tank.byteArray[i] - (shiftValue)));
        if(decryptArray[i] < 0){
            decryptArray[i] = (byte) (128 + decryptArray[i]);
        }
    }
    tank.byteArray = decryptArray;
}

}// end class
Affine Cypher

import java.lang.*;
import java.math.*;
public class AffineCypher{

    //Variables
    private int a, b;
    private FuelTank tank;

    //Constructor
    public AffineCypher(FuelTank tank, int a, int b){
        this.a = a;
        this.b = b;
        this.tank = tank;
        System.out.println("Affine Cypher Initialized +" +
            "with function value: " + a + "x + " + b);
    }

    //Encryption Algorithm
    public void encrypt(){
        byte encryptArray[] = new byte[tank.byteArray.length];
        for (int i=0; i<tank.byteArray.length; i++){
int encrypt = (tank.byteArray[i]*a + b) % 128;
encryptArray[i] = (byte) encrypt;
}
tank.byteArray = encryptArray;

//Decryption Algorithm
public void decrypt(){
byte decryptArray[] = new byte[tank.byteArray.length];
int c = getModInverse(a);
System.out.println("C = "+c);
for (int i=0; i<tank.byteArray.length; i++){
decryptArray[i] = (byte) (c*(tank.byteArray[i]-b));
if (decryptArray[i] < 0){
decryptArray[i] = (byte) (128 + decryptArray[i]);
}
}
tank.byteArray = decryptArray;

//Using Java Classes to get Modular Inverse of this value.
private int getModInverse(int a){
BigInteger bigInt = new BigInteger(Integer.toString(a));
BigInteger modVal = new BigInteger(Integer.toString(128));
System.out.println("Big Int (value of a): " + bigInt);
BigInteger value = bigInt.modInverse(modVal);
return value.intValue();
}

//end class

One Time Pad

import java.util.*;
public class OneTimePad{

private FuelTank tank;
byte[] randomBytes;

//Main Constructor
public OneTimePad(FuelTank tank){
this.tank = tank;
randomBytes = new byte[tank.byteArray.length];
public void run(){
makeRandomBytes();
//System.out.println("Random Bytes are here...");
printRandomBytes();
tank.printBytesToChars();
encrypt();
tank.printBytesToChars();
tank.printBytes();
decrypt();
tank.printBytesToChars();
}

private void encrypt(){
for(int i=0; i<randomBytes.length; i++){
tank.byteArray[i] = (byte) ((tank.byteArray[i]+randomBytes[i]));
}
}

private void decrypt(){
for(int i=0; i<randomBytes.length; i++){
tank.byteArray[i] = (byte) ((tank.byteArray[i]-randomBytes[i]));
}
}

private void makeRandomBytes(){
for(int i=0; i<randomBytes.length; i++){
randomBytes[i] = (byte) ((Math.random() *1000) %128);
}
}

private void printRandomBytes(){
for(int i=0; i<randomBytes.length; i++){
System.out.print(randomBytes[i] + " ");
}
}

Main Driver & Helper Class
This was required for all File I/O
import java.util.*;
//This is the driver class for thesis applets
class YellowCab{

    /**------------------------ GLOBAL VARS ------------------------**/
    public static final String INPUT_FILE_PLAINTEXT = "PlainText/input.txt";
    public static FuelTank tank;

    /**------------------------ LOCAL VARS ------------------------**/

    public static void main (String args[]){

        //Initialize a Tank (A plaintext tank for now)
        tank = new FuelTank(INPUT_FILE_PLAINTEXT);
        tank.pump();

        //How about a Shift Cypher?
        //ShiftCypher shifty = new ShiftCypher(tank, 3);
        //shifty.run();

        //How about an Affine Cypher? WARNING: The first integer, a must
        //be relatively prime with 128.
        //AffineCypher affine = new AffineCypher(tank, 5, 9);
        //affine.run();

        //One Time Pad
        //OneTimePad once = new OneTimePad(tank);
        //once.run();

    } //end main

} //end class

import java.io.*;

class FuelTank{

    /*------------------------ GLOBAL VARS ------------------------*/
    private File inputFile;

    public byte[] byteArray;

    //INPUT
    public String inputStrings[];
    public byte inputBytes[];
public int inputInts[];

//OUTPUT / Modified
public byte outputBytes[];

/*-------------------------------- MAIN CONSTRUCTOR --------------------------------*/

public FuelTank(String inputFileLocation)
{
    this.inputFile = new File(inputFileLocation);

    //Give status of current file in this tank
    System.out.println("Constructor Initialized with file: "+ inputFileLocation);
}

/////////// AVAILABLE DATA STRUCTURES TO READ IN FILE /////////////
//Internal Driver
public void pump()
{
pumpToByteArray();
    //printBytes();
}

/* First Method: Create a ByteArray from file */
public void pumpToByteArray()
{
    try
    {
        //create FileInputStream object
        FileInputStream fIn = new FileInputStream(inputFile);
        this.byteArray = new byte[(int)inputFile.length()];
        fIn.read(byteArray);
    }
    catch(FileNotFoundException e)
    {
        System.out.println("File not found" + e);
    }
    catch(IOException ioe)
    {
        System.out.println("Exception while reading the file " + ioe);
    }
}

/////////// AVAILABLE METHODS TO SEE WHATS STORED IN THE TANK /////////////
public void printBytes(){
    // Print Bytes (Decimal Values)
}
for(int i=0; i<byteArray.length; i++){
    System.out.print(byteArray[i]);
    System.out.print(" "); //Pad with a space.
}
System.out.println("\n");

public void printBytesToChars(){ // Print Bytes >> Chars
    for(int i=0; i<byteArray.length; i++){
        System.out.print((char)byteArray[i]);
        System.out.print(" "); //Pad with a space.
    }
    System.out.println("\n");
}

public void printBytesToChars(int modVal){ // Print Bytes >> Chars
    for(int i=0; i<byteArray.length; i++){
        if (i%modVal==0){
            System.out.print((char)byteArray[i]);
            System.out.print(" "); //Pad with a space.
        }
    }
    System.out.println("\n");
}

public void printBytesToBinaryString(){ // Print Bytes >> Binary String
    for (int i=0; i<byteArray.length; i++){
        int intValue = (Integer.valueOf(byteArray[i]));
        System.out.print(Integer.toBinaryString(intValue) + " ");
    }
    System.out.println("\n");
}

public void printBytesToHexString(){ // Print Bytes >> Hex String
    for (int i=0; i<byteArray.length; i++){
        int intValue = (Integer.valueOf(byteArray[i]));
        System.out.print(Integer.toHexString(intValue) + " ");
    }
    System.out.println("\n");
}

}//end class
Various Code Snippets for character manipulation and frequency analysis

```java
import java.util.*;

public class CharFromInt{
public static void main(String Args[]){

Scanner sc = new Scanner(System.in);

System.out.println("Welcome to Char From Int");
System.out.println("Enter each int, separated by a space. Enter 26 to exit");
int num;
boolean flag = true;
while (flag){
    num = sc.nextInt();
    if (num==26){
        flag = false;
    }else{
        num += 'A';
        char ch = (char) num;
        System.out.println(num-'A' + " = " + ch);
    }
}
}
}

import java.util.*;

public class CharGen{
public static void main(String Args[]){

Scanner sc = new Scanner(System.in);

String word;
System.out.println("Welcome to Char Gen");
System.out.println("Enter each character on a new line, type 'quit' to exit");
boolean flag = true;
while (flag){
    word = sc.next();
    if (word.equals("quit")){
        flag = false;
    }
}
```
else{
    int val = word.charAt(0) - 'A';
    System.out.println(word + " = " + val);
}
}
}

import java.util.*;
import java.io.*;

public class LetterCounter {

    public static void main(String args[])
        throws IOException{
        Scanner sc = new Scanner(System.in);
        String input = "";
        while (sc.hasNext()){
            input += sc.next();
        }

        //input=input.replaceAll(" ", "");
        //input=input.toUpperCase();
        char[] ch = input.toCharArray();
        char[] charArray = count(ch);
        System.out.println("Length = " + charArray.length);
    }

    public static char[] count(char[] ch){
        int[] alph = new int[128];

        for(int i=0; i<ch.length; i++){
            alph[ch[i]]++;
        }

        System.out.println("Done Counting");
        String missing = "";
        for(int i=0; i<alph.length; i++){
            if(alph[i]!=0){
                System.out.println((char) (i)+ " = " + alph[i]);
            } else if (i>='A' && i<='Z'){
                missing += ((char) (i) + " ");
            }
        }
    }
}
```java
import java.util.*;
import java.util.Scanner;

public class Shifter{
    public static void main(String Args[]){

        Scanner sc = new Scanner(System.in);

        String word;
        System.out.println("Welcome to Shifter");
        System.out.println("Enter Input on one line, separated by a space," + "Uppercase letters only. Enter 'quit' to exit");
        System.out.println("Input format: LETTER SHIFT, ex: N 17\n");
        boolean flag = true;
        while (flag){
            word = sc.next();
            if (word.equals("quit")){
                flag = false;
            }
            else{
                int shift = sc.nextInt();
                char ch = (char) ((word.charAt(0)-'A'+shift) % 26+'A');
                System.out.println(word + "+" + shift +"="+ch);
            }
        }
    }
}
```