DESIGNING AN EASILY MODIFIABLE CIPHER FOR EDUCATIONAL PURPOSES

Michael Jeffords*
M.S. Program in Computer Science, Rivier College

Keywords: *Cryptography, Cryptanalysis*

Abstract

People like solving puzzles. Most people prefer those puzzles to have visual clues that can lead to the solution. This paper describes a pair of easily implemented ciphers that can be used to teach the basics of cryptanalysis using character frequency analysis. Each cipher builds from the previous and the implementer can choose to keep contextual clues such as spacing and capitalization. These ciphers are designed to both hide letter frequencies and to be broken easily. The pitfalls of developing one's own ciphers are described along the way. The hope is that this approach could be used to excite students about the field of cryptanalysis and steer them toward open standards of encryption.

1 General Introduction

In 1919 Gilbert Vernam received a patent for a "Secret Signaling System" (US Patent 1,310,719) that combined characters from a paper tape with that of a plaintext message using the exclusive or (XOR) operation for use in telegraph systems of the day [1, 2]. If two users have access to the same key (in this case the paper tape) they can encrypt or decrypt a message using the same technique which makes this a symmetrical encryption algorithm. As of October 18, 2011, fifteen other US patents reference Vernam's original work [3].

After creating a set of algorithms for the purposes of doing basic character frequency obfuscation, I presented my initial results to my advisor, Professor Vladimir Riabov. He liked the approach I was taking and asked if I had done any research regarding similar algorithms. Since then I have searched the descriptions of several types of ciphers and found that the Vernam cipher was the most similar to my work. The character encoding of the time is different but the basic methodology is the same.

It should be noted that this type of cipher is still used in an age of modern computing. For example, RC4 is a relatively strong stream cipher created by Ron Rivest of RSA and is more advanced than the ciphers I created herein [6]. However, RC4 has been shown to be very weak if used improperly. Bruce Schneier in his "Schneier on Security" blog entry dated January 18, 2005 describes the following regarding a flaw in the way that Microsoft implemented RC4 for its Office products: "One of the most important rules of stream ciphers is to never use the same keystream to encrypt two different documents. If someone does, you can break the encryption by XORing the two ciphertext streams together. The keystream drops out, and you end up with plaintext XORed with plaintext -- and you can easily recover the two plaintexts using letter frequency analysis and other basic techniques" [4].

ISSN 1559-9388 (online version), ISSN 1559-9396 (CD-ROM version).

2 Early Attempts at Obscuring Letter Frequency

In my first attempts at studying letter frequency, I was interested in directly modifying a monoalphabetic cipher. I wanted to include a larger set of characters in the frequency distribution hoping that I could obscure frequencies by manipulation of letter combinations. I decided to experiment with adding a subset of additional characters (numbers 0-9, space, comma, period, end of line). Figure 1 below shows the character frequencies of "The Declaration of Independence". We should note the similarity of the standard letter frequencies for the English language. E and T are the most frequent and are located at the beginning. X, Q and Z are infrequent and located toward the end. The frequency of space denoted by <SP> is the most common. A simple incorporation of space <SP> into the cipher would not throw off a trial and error iterative attack for very long.

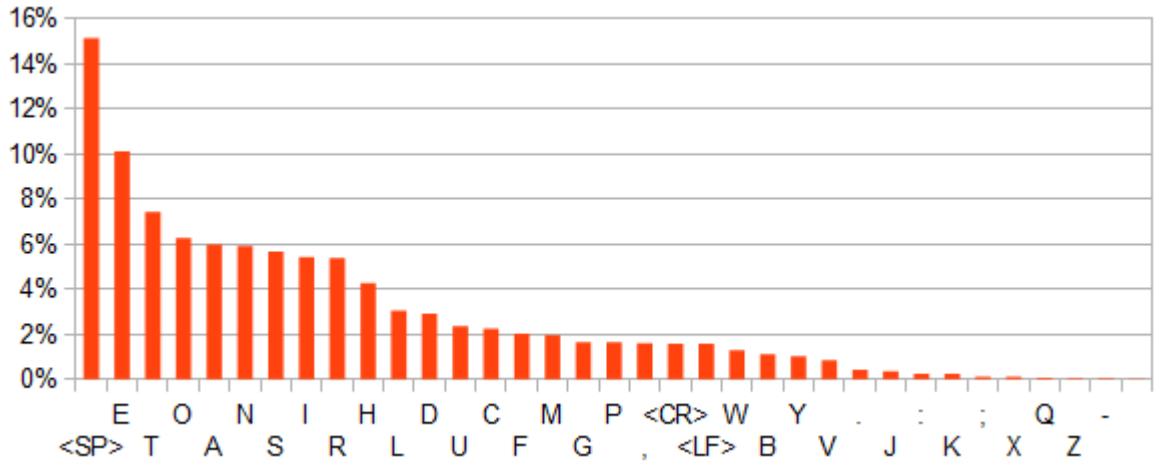

Figure 1: Letter Frequency of *The Declaration of Independence* including punctuation

Next I decided to check the frequency of double letter combinations to see if that in itself would change the playing field and result in a reasonable method of hiding. After reminding myself that having double letters would result in a very large substitution set (26*26= 676 pairs not including punctuation), I decided to be more selective about pairing. Figure 2 shows one attempt at this. I wanted to use space <SP> to help obscure E but found in Figure 1 that it is considerably more frequent than E. I decided to try making digraphs from space and all the combinations of letters and punctuation and numbers – see Figure 2. Note: It makes E really obvious and just stretches out the remainder of the frequencies but does nothing to change the overall shape of the graph. Rather than prepending space I probably would have had a better result by adding it to the last letter of words as E is one of the most common last letters. This might bring <SP>, E and "E<SP>" into better obscurity. However, I did not want to peek ahead for every letter in my algorithm. It also became apparent that any sorts of tricks using this type of method would still leave large portions of the frequency graph completely intact. This would snowball into having to make several tweaks to any algorithm. For example, I also tried encoding the word "The" as it is the most common word and is a large contributor to the frequency of T and E. If I flattened <SP>, T and E then the next letters would still be open to attack if the algorithm was ever made known. My hope was to find a simple solution and a simple algorithm.

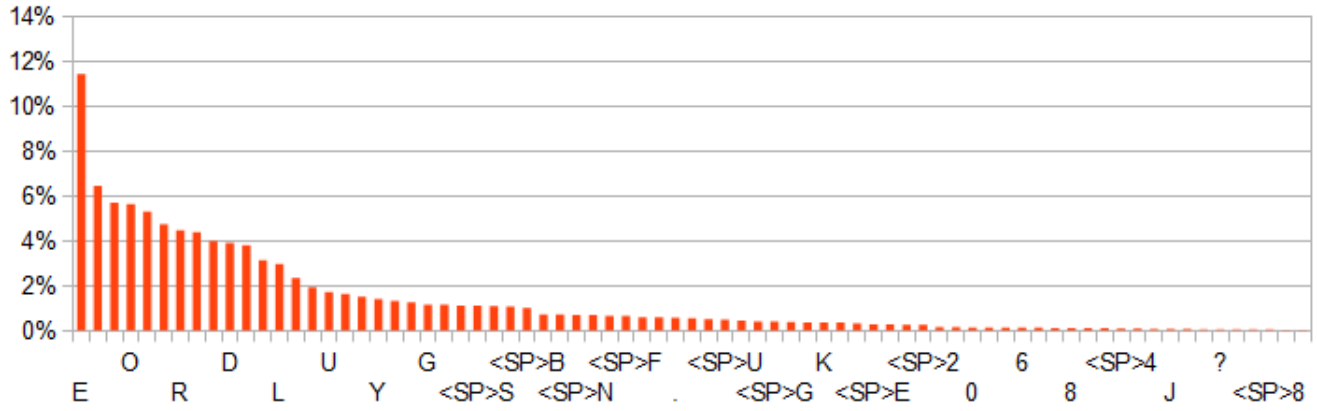

Figure 2: Letter Frequency when trying to eliminate the spike created by Space <SP> by adding categories for letter combinations of <SP>A, <SP>B, etc.

3 Studies of ASCII and UTF-8 Character Sets

I overcame the limitations I had previously encountered by looking at the ASCII and UTF-8 tables to determine what was actually being encoded for characters. I thought I remembered from ASCII encoding that there was an offset between upper case and lower case letters such that simply changing one bit could change any letter from upper case to lower case and vice versa.

Here is an example using the letter B showing this feature:

```
's' in binary 01110011
'S' in binary 01010011
```

Note how the least significant 5 bits are the same. With 5 bits you can define 32 values and ASCII and UTF-8 use this type of encoding for 4 distinct groups. The 3 high bits determine the group as you can see in Figure 3:

- 000 are all control characters
- 001 are mostly numbers and punctuation
- 010 are mostly upper case letters
- 011 are mostly lower case letters.

Since we want a cipher that can convert plain text to human readable cipher text we can only modify characters 32 to 127. We will have to avoid altering the control characters to keep characteristics of a text file. Furthermore, since the characters we are most interested in dealing with (a-z and A-Z) are defined by the low 5 bits we will not try to do anything tricky to use more than 5 bits.

Our requirements for the simple cipher are now as follows:

- Plaintext should be convertible to readable ciphertext
- Ciphertext should keep capitalization
- Ciphertext should have the option to keep spacing in tact.
- We can manipulate the low 5 bits of characters 32 to 127.

0 to 31		32 to 63		64 to 95		96 to 127	
character	ASCII UTF-8 binary	character	ASCII UTF-8 binary	character	ASCII UTF-8 binary	character	ASCII UTF-8 binary
control	00000000	<SP>	00100000	@	01000000	`	01100000
control	00000001	!	00100001	A	01000001	a	01100001
control	00000010	"	00100010	B	01000010	b	01100010
control	00000011	#	00100011	C	01000011	c	01100011
control	00000100	$	00100100	D	01000100	d	01100100
control	00000101	%	00100101	E	01000101	e	01100101
control	00000110	&	00100110	F	01000110	f	01100110
control	00000111	'	00100111	G	01000111	g	01100111
control	00001000	(	00101000	H	01001000	h	01101000
control	00001001	)	00101001	I	01001001	i	01101001
control	00001010	*	00101010	J	01001010	j	01101010
control	00001011	+	00101011	K	01001011	k	01101011
control	00001100	,	00101100	L	01001100	l	01101100
control	00001101	-	00101101	M	01001101	m	01101101
control	00001110	.	00101110	N	01001110	n	01101110
control	00001111	/	00101111	O	01001111	o	01101111
control	00010000	0	00110000	P	01010000	p	01110000
control	00010001	1	00110001	Q	01010001	q	01110001
control	00010010	2	00110010	R	01010010	r	01110010
control	00010011	3	00110011	S	01010011	s	01110011
control	00010100	4	00110100	T	01010100	t	01110100
control	00010101	5	00110101	U	01010101	u	01110101
control	00010110	6	00110110	V	01010110	v	01110110
control	00010111	7	00110111	W	01010111	w	01110111
control	00011000	8	00111000	X	01011000	x	01111000
control	00011001	9	00111001	Y	01011001	y	01111001
control	00011010	:	00111010	Z	01011010	z	01111010
control	00011011	;	00111011	[	01011011	{	01111011
control	00011100	<	00111100	\	01011100	|	01111100
control	00011101	=	00111101	]	01011101	}	01111101
control	00011110	>	00111110	^	01011110	~	01111110
control	00011111	?	00111111	_	01011111	control	01111111

Figure 3: ASCII/UTF-8 binary values for the encoding of the first 128 characters

4 Development of the Simple Cipher for ASCII and UTF-8 character sets

Based on the knowledge that the low 5 bits represented letter and the upper three bits of each character represented the 32 character subset, I took the approach that some simple math using XORs for substitution would be the best approach since performing an XOR twice results in the original value. Thus you get encrypt and decrypt with the same simple algorithm. It will also allow you to preserve control characters such as CR LF and capitalization. It will slightly obscure spaces and nearly completely obscure letters if you perform a different XOR on each character that is read in. Since there are 32 items per set I have chosen to have 32 XOR values and a 32 character cycle. After every 32 characters the cycle begins again using mod 32. For the XOR key values you can cycle through values between 0 (no change) and 31 (11111 flip all bits) for each character in the text (or distribute them randomly). Figure 4 below shows what I was able to achieve when using a perfect randomly generated key. A perfect key in this case consists of an array of 32 values from 0 to 31(11111) dispersed randomly in the key. Here is an example key.

```
int[] key= {18,14,25,6,9,24,7,3,0,16,8,11,22,29,26,1, 30,15,20,5,2,21,17,31,12,23,4,28,13,10, 19,27};
```

If you look at the key, you will see that each value from 0 to 31 is represented in the key. Since each value in the key is 5 bits the entire key could be represented by 160 bits or 20 bytes. However, we will consider it to have 32 bytes. The key is used in the following pseudocode algorithm the blue lines does all the encrypting/decrypting.

```
Int keyIndex = key[0];     // You can start the key index at any value in this case 18
Int sourceChar = 0;        // source character
Int outputChar;      // ciphertext if plaintext is the input
                     // plaintext if ciphertext is the input
while ( sourceChar != EndOfFile ){
sourceChar = readNextCharacterFromSource();

// if sourceChar is one of first 32 control characters or EOF
// if you want retain spaces as spaces set this value to 64
if ( sourceChar < 64 || sourceChar == EndOfFile)
{
       outputChar = sourceChar;
}
else //encrypt/decrypt character by XORing it with one of the keys
{
       keyIndex = (keyIndex +1)%32; // cycle through the key
       outputChar = sourceChar XOR key[keyIndex];
}
writeToOutput(outputChar);
}
```

When the ciphertext for “The Declaration of Independence” is generated, the frequencies of the letters in the cipher text all have nearly equivalent values. So this algorithm meets the initially desired conditions – simplicity and obscured letter frequency. Here is an example of the first paragraph as it is translated.

Ciphertext working on characters 64-127	Plaintext
@mg Dqmymqb□` Vkzjhjfwi□f y{ uvj Qj|ckirj Xdzowj U}ysfs gm \wdlfwd ... @a ee□□ zqcz} wre|ce nn mq qp}y-rruio}o, mnhl bl| fss □}q agt~xr` h{fz~, rays txmr |hd jzambt{ u} ybvr` E{}fwob |□ir }jfqc|□ yyepdo}zpb| {q`ktc, bu{u nyjlr x□aoh riw uoo}, lyjndic □ap v}t |bvoxcg }h nhhwjnu{x...	The Unanimous Declaration of the Thirteen United States of America ... We hold these truths to be self-evident, that all men are created equal, that they are endowed by their Creator with certain unalienable rights, that among these are life, liberty and the pursuit of happiness...

Ciphertext working on characters 32-127	Plaintext
@mg5Dqmymqb□`;Vkzjhjfwi□f y{:uvj4Qj|ckirj<Xdzowj9U}ysfs0gm6\wdlfwd ;?1 @a<ee□□2zqcz}'wre|ce=nn>mq%qp}y!rruio}o>.mnhl'bl|( fss: □}q%agt~xr`<h{fz~"9rays#txmr6|hd>jzambt{,u}<ybvr`. E{}fwob(|□ir!}jfqc|□?yyepdo}zpb|&{q`ktc$+bu{u> nyjlr1 x□aoh*riw.uoo}+#lyjndic!□ap%v}t?|bvoxcg;}h9 nhhwjnu{x834	The Unanimous Declaration of the Thirteen United States of America ... We hold these truths to be self-evident, that all men are created equal, that they are endowed by their Creator with certain unalienable rights, that among these are life, liberty and the pursuit of happiness...

Figures 4a and 4b: Translation of *The Declaration of Independence*

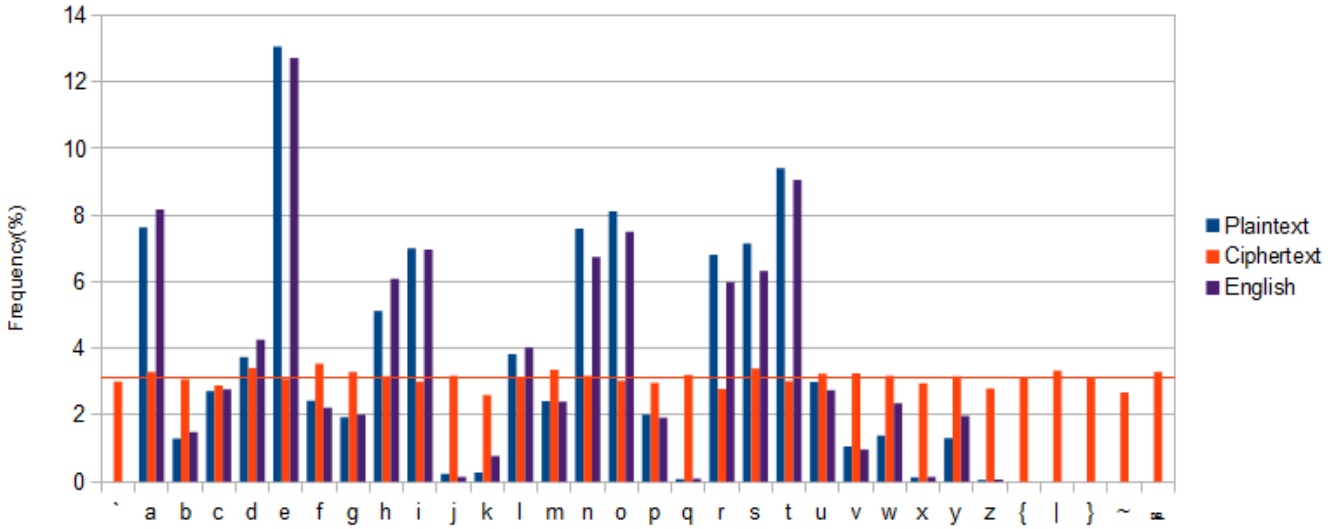

Figure 5: Plaintext and Ciphertext frequencies of characters 96 through 127 within *The Declaration of Independence* (compared to letter frequencies of the English language)

On first look this is a better cipher than a monoalphabetic cipher as it obscures character frequencies. The algorithm preserves control characters such as *carriage return line feed* (CR LF) and *end of file* (EOF). This makes the algorithm very good at preserving overall layout of plain text while obscuring frequencies within each set of 32 characters. However, the contextual clues will allow it to be broken easily.

4 Breaking the Simple Cipher by Exploiting Contextual Clues

The major contextual issue with this cipher can be seen in Figure 6 below. Figure 6 shows the frequencies of all the ASCII/UTF-8 characters and their frequencies for “A Christmas Carol” by Charles Dickens. The total number of spaces is off the chart at 25503 occurrences. The three plateaus are the three sets of characters 32 to 63 space, numbers and punctuation, 64 to 95 mostly upper case, 96 to 127 mostly lower case. The right side of the chart is similar to the subset of characters shown in Figure 5.

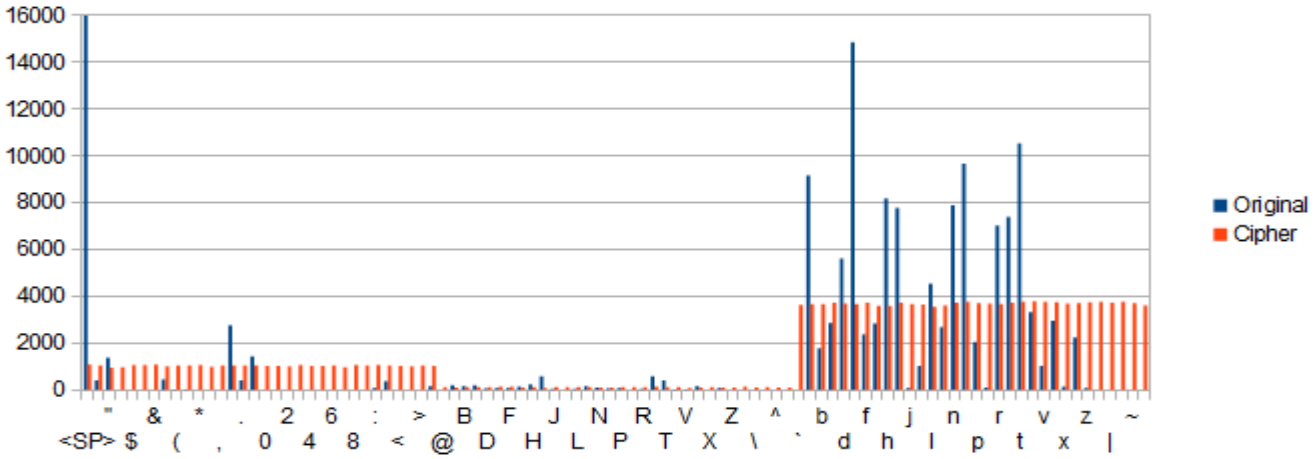

Figure 6: *A Christmas Carol* – Original frequencies versus Ciphertext frequencies

Looking back at Figure 4b you can see all the spaces highlighted in yellow. Each one has a different value and for this reason along with Figure 6 you could programmatically crack this cipher by looking at the distribution of likely space values to determine the 32 byte key. You have over an 80% chance of correctly back calculating any particular key value based on the character count mod 32 if you think that you have a space. For example, the 4th character in figure 4b is '5' which puts it in the group of characters that contains possible spaces. '5' maps to a value of 00010101. If you look back at the key on page 5 you will see that: key[0] + characterCount - 1 = 18 + 4 – 1 = key[21].

By coincidence key[21] also happens to have a value of 21. 10101 in binary = 21 in decimal. Repeating this process for each of the 32 indexes will result in recovering the full key. The sample shown in Figure 4b contains a minimum of 17 unique space values which gives the user more than one half of the bytes in the key.

5 Breaking the Simple Cipher by Exploiting Letter Frequency Weaknesses

If you remove the first group of characters (32 to 63) to avoid the issue noted in the previous section regarding spaces you get rid of one set of issues but you are not hiding any of those 32 characters which include numbers. Knowing that the cycle is only 32 characters you can quickly expose the typical English language letter frequencies by examining every 32 characters offset by N characters to start the count. This will occur if you had a significant number of characters. We can see from "The Declaration of Independence" that it has a sufficient number of characters for the analysis by looking for the highest frequency character at N 2+18 'g' = 17.62% and N 5+18 'z' = 10.04%. This case is slightly more complicated in that with the space previously all we had to do was match the number. In this case we need to figure out what value XORed with our ciphered character equals the encoding for 'e' in plaintext. The low 5 bits of 'e' = 00101. The low 5 bits of 'g' = 00111. The difference is 00010 or 2 which does in fact fall at index 20 in the key. The low 5 bits of 'z' = 11010. The difference is 11111 or 31 which does in fact fall at index 23 in the key. Again you can programmatically crack this cipher by this method.

6 Creating a Slightly Better Cipher by Applying the Key More Randomly

Since the first cipher is just too easy to solve, I wanted to make it a bit more difficult. To do so we will extend the feature from the first cipher that uses the key values to change the start position for each set of 32 characters. This means that the pattern will repeat every 32*32 = 1024 = 10^3 characters instead of every 32 characters. This is still extremely weak. As a comparison the RC4 cipher has a period greater than 10^{100} according to the RSA website [5]. Our new algorithm is similar to the following pseudocode where changes from the original are shown in blue:

```
Int block = 0;              // Each block is 32 characters. This is the block count
Int count = 0;              // This is the count of characters within a block
bool newBlock = false;
Int keyIndex = key[0];      // You can start the key index at any value in this case 18
Int sourceChar = 0;         // source character
Int outputChar;      // ciphertext if plaintext is the input
                     // plaintext if ciphertext is the input
while ( sourceChar != EndOfFile ){
sourceChar = readNextCharacterFromSource();
// if sourceChar is one of first 32 control characters or EOF
// if you want retain spaces as spaces set this value to 64
if ( sourceChar < 64 || sourceChar == EndOfFile)
{
```

```
        outputChar = sourceChar;
}
else //encrypt/decrypt character by XORing it with one of the keys
{
        if ( newBlock ) {
                keyIndex = key[block]; // change the starting index at every new block
                newBlock = false;
        }
        outputChar = sourceChar XOR key[keyIndex];
}
if(count == 32) //increment block and reset count
{
        block = (block + 1)%32;
        count = 0;
        newBlock = true;
}
count++;
keyIndex = (keyIndex +1)%32; // cycle through the key
writeToOutput(outputChar);
}
```

When this is used you receive the following ciphertext from “The Declaration of Independence”:

Ciphertext working on characters 64 - 127	Plaintext
@mg Dqmymqb□` Vkzjhjfwi□f y{ uvj Niw}``g{ Jb~pyi @osz|u wa A}my□~v ... Lw qie| whu{n ihtjgg vz }i tflv-n`t~dp{, v}pk vhp gvu xtl sznwi□e jepcy, x□eh ~{~k xtl bmhxsyi qb zqc`j Cbmjbrh if`m vtmwayf cs{mwjzd`yt ~~ctyy, ffxr yjig□ txmxs {s{ xldp, `~fy□~j s`} ff| ymupuy| y{ i□□dllpbl...	The Unanimous Declaration of the Thirteen United States of America ... We hold these truths to be self-evident, that all men are created equal, that they are endowed by their Creator with certain unalienable rights, that among these are life, liberty and the pursuit of happiness...

Figure 7: Translation of *The Declaration of Independence* using the slightly improved cipher

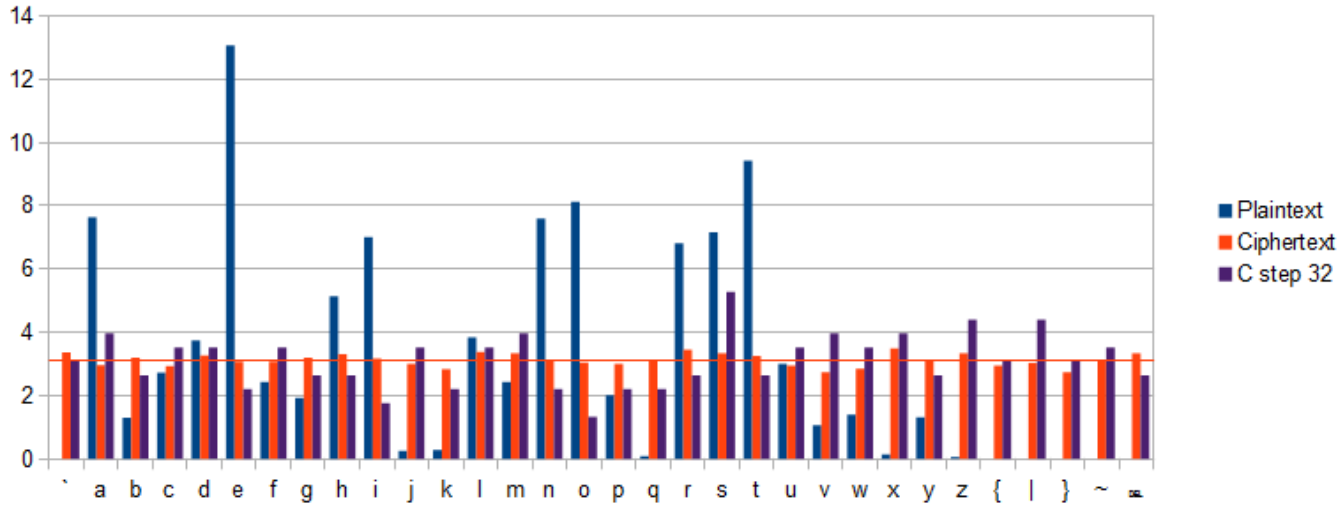

Figure 8: Plaintext vs. Ciphertext frequencies for characters 96-127 in *The Declaration of Independence* using the slightly improved cipher

The outcome of this modification is that the pattern becomes more difficult to see on reasonably sized texts. “The Declaration of Independence” is too small to get decent character frequency obfuscation when looking at every 32 characters. A larger text such as “A Christmas Carol” will result in a more even distribution for the character frequency of every 32nd character. The range is 2.56% to 3.68% for “A Christmas Carol” vs. 1.32% to 5.28% for “The Declaration of Independence”. Prior to this modification however the range was 0% to 17.62% when looking at every 32 characters starting with an offset of 2. Although we could look at every 1024th character and find similar results as the previous it would be an exercise that depended heavily on the size of the text.

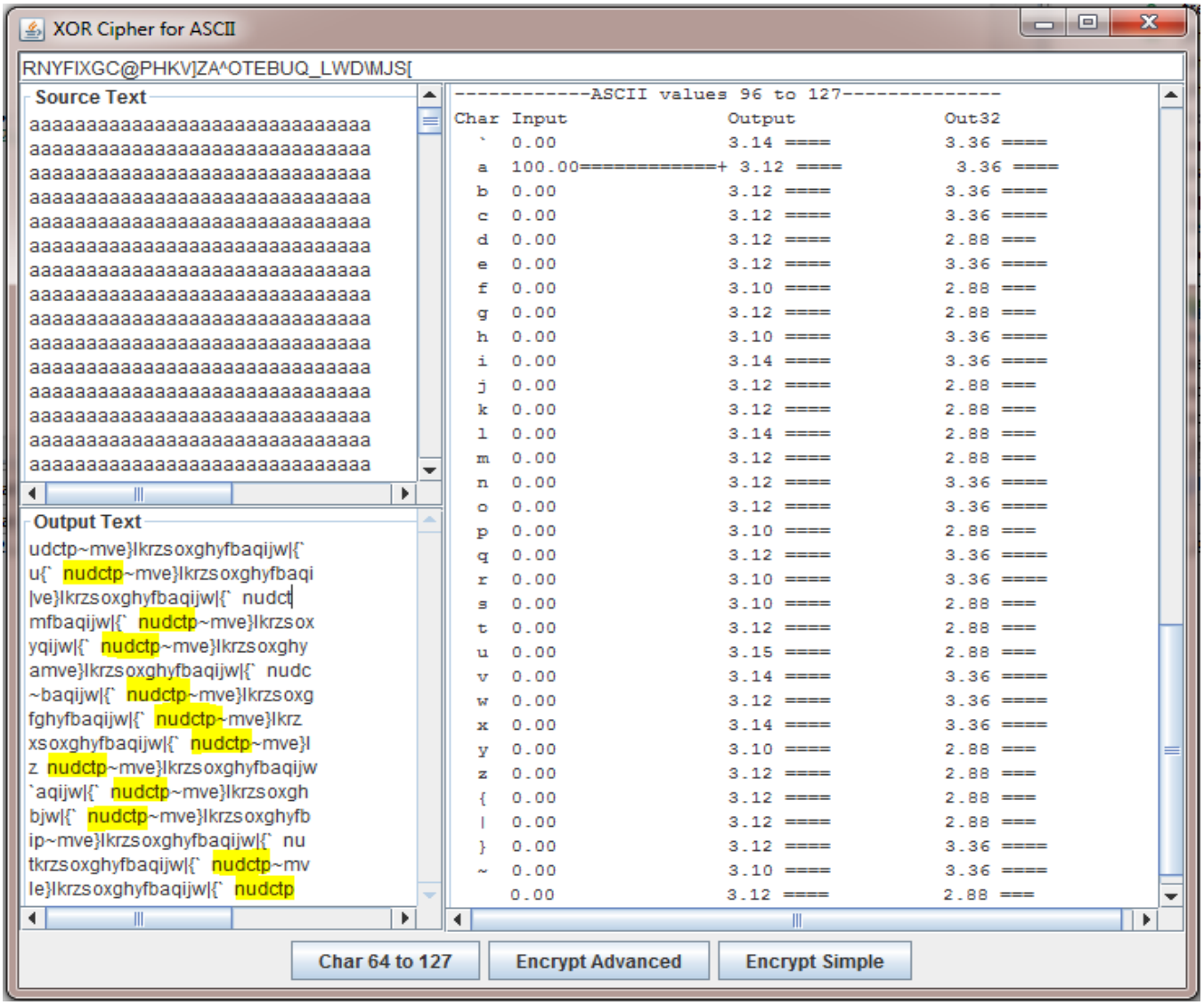

Figure 9: Known plaintext used to try to uncover key by searching out patterns in the ciphertext

As mentioned in the introduction, a stream cipher must use a new key for every encryption. The only way it is safe is if you use a one-time pad (matching the length of the message). The examples here have used a repeated key. The second cipher used a method to break up the 32 byte key but you can see in Figure 9 that the pattern is still mostly intact from one line to the next. If you use the same key you

allow for quick discovery of the key and or algorithm. Another way to break a cipher can occur if you can see a pattern form by looking using a known plaintext. The plaintext in this case is just the character 'A' repeated 31 times per line. It produces a repeated output every 32 lines with obvious patterns forming throughout as can be seen in Figure 9. Even if you tried to hide your algorithm from the users in this case allowing any sort of encryption by the person you are trying to hide it from would quickly lead to the discovery of the algorithm.

Assuming that a user can be trained to treat a stream cipher as a one-time pad, the main weakness with this cipher is still in the key generation. In order to make it more secure we would need to do a much better job at randomly repeating the 32 available bytes. Someone attempting to improve the ciphers described here could easily use a passphrase to create a random set of bytes and then use a rotating transposition cipher to better hide the characters. For an even better approach the RC4 method should be explored even though it was outside the scope of this paper. The RC4 method is clearly described on the web.

7 Conclusions and Remarks

In this discussion we have covered some very basic principles of cryptanalysis. It is very unlikely that an untrained individual would produce a cipher that would be difficult for an expert to crack. I am not an expert in this field. I developed these ciphers rapidly and cracked them myself almost as quickly. Following this pattern of creating your own cipher and exploiting its weakness is the educational message herein. By keeping the ciphers here in readable characters with contextual clues, I hope you found the ciphered messages fairly well hidden to the naked eye while the exploits should have also been easy to see.

Open standards/algorithms have been tested thoroughly by many experts in the field to uncover their weaknesses. Often these algorithms will stand for years as accepted practice until someone discovers a flaw or the compute power increases sufficiently that a brute force attack becomes practical. Although there is a risk with standard algorithms that someone will discover a flaw and does not disclose it, their generally accepted strength is known and established as are the best practices for using them. While it is enjoyable to learn through experimentation, those of us who like to play with these principles should know when to experiment and know when to use generally accepted algorithms.

References

[1] Wikipedia, *Gilbert Vernam*. Retrieved March 17, 2012, from http://en.wikipedia.org/wiki/Gilbert_Vernam

[2] Vernam, Gilbert S., *Secret Signaling System*, US Patent Number 1,310,719, Filing date Sep 13 1918, Issue Date July 22, 1919.

[3] Google Patent History**,** Secret Signaling System. Retrieved March 17, 2012, from http://www.google.com/patents?vid=1310719

[4] Schneier, Bruce, *Microsoft RC4 Flaw*. *Schneier on Security Blog*, January 18, 2005. Retrieved March 17, 2012, from http://www.schneier.com/blog/archives/2005/01/microsoft_rc4_f.html

[5] RSA Laboratories. *What is RC4?* Retrieved March 17, 2012, from http://www.rsa.com/rsalabs/node.asp?id=2250

[6] Wikipedia, *RC4*. Retrieved March 17, 2012, from http://en.wikipedia.org/wiki/RC4

* **MICHAEL JEFFORDS** finished up his coursework for a Masters Degree in Computer Science at Rivier College in December 2012. He lives in Manchester, New Hampshire with his wife and 6 kids. Although he dabbles in cryptography and cryptanalysis, his current efforts focus on cross platform signal processing and image processing through the use of Java and OpenGL ES 2.0.