

IMC

Independent Mathematical Contractors, Inc.

136 TMCB
Provo, UT 84602

8 September 2020

OCRA Creative Recursive Acronyms, Inc.
485 Primality Way
Provo, UT 84604

Dear OCRAI,

In response to your security requirements, we have developed the [REDACTED] algorithm—we call it the [REDACTED] cipher—to automatically encrypt/decrypt messages between authorized personnel. As mentioned in your letter, a common problem in computer security is the human element, which could easily make a mistake or be coerced into leaking sensitive information to malefactors.

Key Security Principles

To ensure security, we decided it was imperative to approach the issue using two security principles—*confusion* and *diffusion*. Confusion is the principle of altering the encoding of a message. This can be seen in substitution ciphers like the Caesar cipher, or in the substitution box phase of AES. The [REDACTED] algorithm performs confusion by creating a mapping from the original characters to new characters in a way that's dependent on a randomly-generated key, specifically by using the Vigenère cipher (discussed in depth in the Confusion section of this report). Diffusion is the principle of changing one part of the message in a way that changes the entire message as a whole. This can be seen in the use of finite field multiplication in the AES cipher. The [REDACTED] cipher performs diffusion by organizing the plaintext in row-major order in blocks of square matrices, then after the *confusion* is performed, reading the ciphertext from the blocks in column-major order.

Diffusion Part 1 of 2

We first separate the plaintext into chunks of length 16 and fill a sequence of 4x4 matrices. The matrices are filled in row-major order (left to right, top to bottom). We organize it like so:

1. Take the first 4 characters of the plaintext message and assign them to the first row of the first matrix.
2. Repeat step 1 for each row of the matrix.
3. Repeat steps 1-2 for each successive matrix, until all the plaintext characters are in a matrix.
4. Fill any empty cells in the last matrix with random letters.

We will demonstrate the process with the plaintext “*When it comes to cryptography things get complex quickly*”, which has 48 letters. We organize it into three 4x4 matrices, following steps 1-4 above:

W	H	E	N		P	T	O	G		E	T	C	O
I	T	C	O		R	A	P	H		M	P	L	E
M	E	S	T		Y	T	H	I		X	Q	U	I
O	C	R	Y		N	G	S	G		C	K	L	Y

Confusion

The Vigenère cipher is a substitution cipher with a sort of “rolling” key. A key of length n is used to encrypt the message character by character. If the key is shorter than the plaintext, the key is repeated to match the length. Then, the corresponding plaintext and key positions are summed ($\text{mod } 26$). For example, with the plaintext CRYPTO and key BYU, the key is repeated to become BYUBYU, or $[1, 24, 20, 1, 24, 20]$ as integers. Then, each character is shifted by its corresponding value in the repeated key: C is increased by 1 to become D, R+24 becomes P, Y+20 becomes S, P+1 becomes Q, T+24 becomes R, and O+20 becomes I. Notice that upon reaching the end of the key, we “roll” to the beginning of the key and continue our substitution process. Hence, encrypting CRYPTO with the key BYU becomes DPSQRI.

Returning to the previous example, we will now use the Vigenère cipher to “confuse” the characters in our matrices. We begin by converting all the letters into integers. “A” is 0, “B” is 1, and as can be figured, “Z” is 25. Here is what our matrices look like now:

22	7	4	13		15	19	14	6		4	19	2	14
8	19	2	14		17	0	15	7		12	15	11	4
12	4	18	19		24	19	7	8		23	16	20	8
14	2	17	24		13	6	18	6		2	10	11	24

Now we apply the Vigenère cipher to these matrices. The key length should be relatively prime to 16 so that the matrices do not start at the same position of the key. Additionally, the key can be randomly generated, so it is not subject to a dictionary attack. To encrypt our message, we will use the key FNVZEJO, which is $[5, 13, 21, 25, 4, 9, 14]$ as integers.

The first integer in our key is 5. We will use modular addition to map the first cell of the first matrix to a new cell like so: $23 + 5 = 28 \equiv 2 \pmod{26}$. Using modular addition for the next

6 cells gives $8 + 13 \equiv 21 \pmod{26}$, $5 + 21 \equiv 26 \pmod{26}$, $14 + 25 \equiv 13 \pmod{26}$, $9 + 4 \equiv 13 \pmod{26}$, $20 + 9 \equiv 3 \pmod{26}$, and $3 + 14 \equiv 17 \pmod{26}$, using the next 6 integers from the key. For the next cell (15), we use 5 from the key to get $15 + 5 \equiv 20 \pmod{26}$. Repeating this pattern will result in:

1	20	25	12		10	18	18	15		8	2	16	19
12	2	16	19		5	5	2	2		25	10	10	8
25	25	17	23		23	23	16	22		6	4	25	21
23	16	22	11		18	19	13	5		23	9	15	7

The cells can now be translated back into alphabet characters:

B	U	Z	M		K	S	S	P		I	C	Q	T
M	C	Q	T		F	F	C	C		Z	K	K	I
Z	Z	R	X		X	X	Q	W		G	E	Z	V
X	Q	W	L		S	T	N	F		X	J	P	H

Diffusion Part 2 of 2

We will complete the diffusion of the message by extracting each character from the matrices in column-major order (as opposed to row-major as before), starting with the first matrix. We begin by removing the first column [B, M, Z, X] and writing it onto a horizontal line. Repeat this for every column. The resultant ciphertext is:

BMZXUCZQZQRWMTXLKFXSSFXTSQNPCWFIZGXCKEJQKZPTIVH

Decryption

To decrypt, perform the encryption steps in reverse:

1. Organize the ciphertext into 4x4 matrices in column-major order, by matrix.
2. Translate the letters into their 0-indexed numerical representations.
3. Perform modular subtraction using the Vigenère cipher key. For example, the first cell in the first matrix has a 2. Subtract it by the first integer of the key. This results in $2 - 5 = -3 \equiv 23 \pmod{26}$.
4. Translate the numbers into their alphabetic representations.
5. Write each character out in row-major order, by matrix.

This results in the plaintext:

WHENITCOMESTOCRYPTOGRAPHYTHINGSGETCOMPLEXQUICKLY

Insert the spaces in between distinguishable words and the ciphertext is completely decrypted:

When it comes to cryptography things get complex quickly

If a word at the end of the plaintext does not resemble a dictionary word, assume it was random letter padding and discard it.

In Summary

As explained above, the [REDACTED] cipher applies the principle of confusion in tandem with diffusion to protect sensitive information from prying actors. Our algorithm is more secure than a simple substitution cipher because the use of diffusion makes it difficult to discern the order of the Vigenère cipher, and the use of the Vigenère cipher causes greater difficulty to attacks trying to decrypt the message. Letter frequency count is weakened because the attacker must be able to accurately group which letters correspond to which position of the encryption key, and even then the frequency is $1/n$ the size of the message body, leading to a much smaller sampling size for analysis.

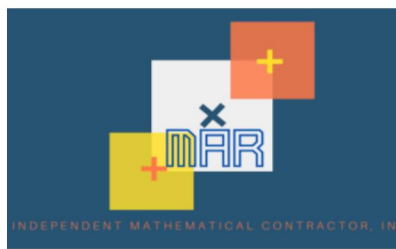
We have determined that through application of the [REDACTED] cipher, the messages sent between employees will be much more secure, and should a recipient error occur again the message shouldn't be immediately discernible. However, our full recommendation would be to restrict messaging between authorized persons participating in the app, such that messages could never leak to an outside actor and thus expose company information to those who would exploit it. We also posit that our encryption could be strengthened by including a transposition within the grid to further diffuse the cipher text and increase the encryption strength, per Kerckhoff's principle of relying upon key secrecy rather than algorithm secrecy.

Yours securely,

[REDACTED]

Founders
Independent Mathematical Contractors, Inc.

Group B



M.A.R an Independent Mathematical Contractor Inc.

Report On Work Performed For OCRAI

Note: This report contains the work done and MAR's solution for the problem given by OCRAI. All work has been done according to the instructions provided by OCRAI. Therefore, this document contains only the information about our company's solution to the specific problem and it should not be applied to any other problems OCRAI might have.

OCRAI's Problem

We, MAR an Independent Mathematical Contractor, received a request for our services from the OCRAI's Vice President of Security on 1 September 2020. On September 3rd, we received a detailed description of the problem that our company has been contracted to solve. On September 8th, we sent OCRAI a confirmation for its request with the information of all the team members that will be working on the problem. We accepted the request with the conditions of complying with all the instructions given by OCRAI and delivering our work by the 11th of September at 5:00 PM. In the same, OCRAI promised full payment (all points) for our service after an inspection of the quality of our work.

OCRAI's problem consists of the use of insecure messages to exchange sensitive company information such as trade secrets. This has caused multiple severe security breaches. The problem originated when OCRAI's employees erroneously used smartphones to communicate and send messages to incorrect recipients. These issues not only represent a threat to OCRAI's profits and future revenue streams, but create a bad image for the company. For this reason, OCRAI requested our services to provide a method for keeping OCRAI data secure.

Our team will provide a nontrivial system of encrypting plaintext English messages. OCRAI will use our system of encryption to encrypt all text messages before they are sent, and decrypt any received messages. OCRAI requested that our solution to its problem should take standard text messages (160 characters) and output ciphertext that can be sent in at most 5 text messages (800 characters). It was also requested that our encryption key should be of limited length due to the limited processing power of mobile devices. Additionally, it was also stated that the ciphertext should be in a form that can be typed with any QWERTY design; this will allow OCRAI's engineers to use our cryptosystem to create a mobile application for its employees. Thus, any messages sent to unauthorized recipients will be unreadable without our encryption system.

How MAR's System of Encryption Works

MAR encryption works by employing the fundamental theorem of arithmetic. Each letter is assigned a unique value between 1-26 as follows:

A	B	C	D	E	F	G	H	I	J	K	L	M
1	2	3	4	5	6	7	8	9	10	11	12	13

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
14	15	16	17	18	19	20	21	22	23	24	25	26

Encryption

First we generate a random string of characters between 5-10 characters. This string serves as the key. The key will be broken up into sections of x letters with the key word repeating as many times as needed to fully encrypt the message. For the following examples, we will split up the key into sections of 4 letters (x=4). This number (x) can be changed as needed. In order to keep the integrity of the key, x must be smaller than the total number of characters in the key and should not be below 4. The following chart shows an example of how the keys will be broken up. For illustration purposes, we will use “tests”.

KEY	test	stes	tste	stst	ests	test	stes	tste	stst	ests
Plaintext Character Index	0	1	2	3	4	5	6	7	8	9

As illustrated above, the first character (indexed as 0) of the plaintext will be encrypted using the key “test”. The second character (indexed as 1) of the plaintext will be encrypted using the key “stes”. Each of these keys can be changed into numerical representations by using the originally stated substitution (A=1, B=2, etc). The following chart shows how the above listed keys would be listed numerically.

KEY	test	stes	tste	stst	ests
Numerically	20, 5, 19, 20	19, 20, 5, 19	20, 19, 20, 5	19, 20, 19, 20	5,19,20,19

This numerical key, that will differ with each letter of the code, will be referred to as the number key. Due to the fundamental theorem of arithmetic, each letter’s assigned number has a unique prime factorization. For example, the prime factorization for twenty-two(22) or “V” is $2*11$ or $2*11*1*1$. Each letter is assigned an x-digit prime factorization based on its

corresponding number. If a number's prime factorization is less than x digits, we fill in with ones (1) until the product reaches x digits. For example two(2) or "B" is $2*1*1*1$. We treat one (1) as a series of four ones ($1*1*1*1$). It is essential that the prime factorization is arranged in order from smallest to largest with the exception of the ones(1). Ones(1) should be at the end of the prime factorization. We will refer to this number as the PTPF (plain text prime factorization)

Each plaintext character is encrypted into x cipher-text characters. Spaces are omitted to further secure and protect this encryption method. The x-character ciphertext is generated using the PTPF and the number key. Each letter in the key is shifted by each number in the plaintext character's prime factorization. The following formula illustrates this:

Let:

x=number of characters in the key (4 in this case)

p=a prime number in the plaintext letter's prime factorization

t=the value of the letter in the number key

$PTPF = p_1p_2...p_x$

Number Key = $t_1t_2...t_x$

A single plaintext character will be encoded into the following set of x numbers

$$((t_1+p_1) - 1) \bmod(26)) + 1, ((t_2+p_2) - 1) \bmod(26) + 1, \dots ((t_x+p_x) - 1) \bmod(26) + 1$$

Each number in the above will then be converted to its corresponding letter and the commas will be removed.

This may be best illustrated in the following example for encoding "B" using the string "tests". For this encoding we will assume that "B" is the first letter of the plain text and therefore uses "test" as its code.

Note: $B = 2 = 2*1*1*1$

	T	E	S	T
Number Key (TEST in this scenario)	20	5	19	20
Corresponding Prime Factor of Plaintext Character	2	1	1	1
$(t_x+p_x) - 1 \bmod(26) + 1$	22	6	20	21
Ciphertext Character	V	F	T	U

Thus, plaintext B becomes VFTU

Decryption

Decryption requires the key and the ciphertext. To illustrate this, we will decrypt the ciphertext UYFT with the key "stes". This assumes that UYTF are the ciphertext characters for the second plaintext letter. We will reverse the above process.

Let:

x =number of characters in the key (4 in this case)
 y =the ciphertext character's number
 t =the value of the letter in the number key
 Key = $t_1t_2...t_x$

	S	T	E	S
Ciphertext Character	U	Y	F	T
Corresponding numerical value	21	25	6	20
Number Key (STES in this scenario)	19	20	5	19
$(y_x - t_x) - 1 \bmod(26) + 1$	2	5	1	1
Value of Plaintext Primes	2	5	1	1

Since $2*5*1*1 = 10$ and $10=J$, Ciphertext UYFT decrypts to plaintext J

Note: By convention $(y_x - t_x) - 1 \bmod(26)$ should be between 0 and 25. Add 26 as necessary to make $(y_x - t_x) - 1 \bmod(26)$ between 0 and 25.

Conclusion

Mar's encryption works by employing a fundamental theorem of arithmetic. An original key, a random string of 5-10 characters, will be broken up into sections of four letters with the key word repeating as many times as needed. Each section becomes a key. Each of these keys can be changed into numerical representations by using MAR's index of the English alphabet which is referred to as the number key. Each letter's assigned number has a unique prime factorization. The prime factorization is arranged in order from smallest to largest. If a number's prime factorization is less than four digits, we fill in with ones at the end of the prime factorization arrangement until the elements of the arrangement reaches the number of letters in the original key, this is what we call PTPF. Each plaintext character is encrypted into four cipher-text characters. Spaces are omitted to further secure and protect this encryption method. The four-character ciphertext is generated using the PTPF and the number key. Each letter in the key is shifted by each number in the plaintext character's prime factorization. Finally, the decryption requires the key and the ciphertext and it consists of reversing the process of encryption. This nontrivial method of encrypting plaintext English messages will prevent OCRAI from experiencing the same major security breach again.

Project 3



September 2020

Introduction

This report presents an implementation of an algorithm using matrices for encryption and decryption of plaintext messages. Our algorithm is capable of encoding plaintext messages of any size and any character set into a ciphertext which can be typed on a standard keyboard.

The security of the algorithm we will present is based on cipher keys. Keys are simple to generate and are made up of a sequence of numbers and can vary in length and be as little as 4 numbers. For this report we will use a 3x3 matrix with 9 numbers, for increased security. Even with knowledge of the encryption algorithm, messages will remain secure as long as the key is not leaked. The key must be distributed to anyone who needs to encode or decode messages. As such, it is imperative that the key be safeguarded.

Explanation and Example

The algorithm uses the knowledge that $A^{-1}(AM) = M$ is true, with A and M being matrices. M is our message, put into a matrix, and A is the encryption key. In our example we will use a 3x3 matrix, but this could be changed to 2x2 if it needed to be simplified for any reason, or 4x4, or any bigger square matrix, to be even more secure. A 3x3 matrix, which can be represented with 9 numbers, is portable, easy to generate, and significantly less vulnerable to brute force attacks than a 2x2 matrix.

Encryption - Step 1

We start with a plaintext message. We will use the short phrase "the quick onyx goblin jumps over the lazy dwarf" as an example. The plaintext message is translated to numbers using the scheme A=1, B=2, C=3... with the space character represented as the number 27. The resulting message after applying this translation is shown below.

20 8 5 27 17 21 9 3 11 27 15 14 25 24 27 7 15 2 12 9 14 27 10 21 13 16 19 27 15 22 5 18 27 20 8 5
27 12 1 26 25 27 4 23 1 18 6

The translation scheme we have presented above is a simple one representing all lower case letters of the alphabet and the space character which we believe is sufficient for sending basic encrypted messages. However, alternative schemes can be used that include other character sets if this is not sufficient. In order to add other characters, each new character must be assigned a unique number. For the purpose of sending readable messages, representing the alphabet and space character is sufficient.

Encryption - Step 2

The translated message is put into a matrix with three rows. Since the original message has 47 characters, the matrix will have 16 columns. Because 47 does not divide equally into 3, we will pad the final row with 0s so that each row has the same length.

$$M = \begin{bmatrix} 20 & 8 & 5 & 27 & 17 & 21 & 9 & 3 & 11 & 27 & 15 & 14 & 25 & 24 & 27 & 7 \\ 15 & 2 & 12 & 9 & 14 & 27 & 10 & 21 & 13 & 16 & 19 & 27 & 15 & 22 & 5 & 18 \\ 27 & 20 & 8 & 5 & 27 & 12 & 1 & 26 & 25 & 27 & 4 & 23 & 1 & 18 & 6 & 0 \end{bmatrix}$$

Encryption - Step 3

We will use the A as our encryption key matrix represented by a string of 9 numbers. The first 3 numbers of the key correspond to the first row of the matrix, the next 3 to the second row, and the last 3 to the final row. This could be any 3x3 matrix as long as it has an inverse, but the following will be used in our example as it has only integers in both the original matrix and its inverse.

$$A = \begin{bmatrix} -1 & 5 & -1 \\ -2 & 11 & 7 \\ 1 & -5 & 2 \end{bmatrix}$$

Encryption - Step 4

We then use the equation $AM = E$ Where E is the encoded matrix. Because of how matrix multiplication works, there is no single number that corresponds to every instance of that letter, and would be near impossible to decipher without the encryption key.

$$E = \begin{bmatrix} 28 & -18 & 47 & 13 & 26 & 102 & 40 & 76 & 29 & 26 & 76 & 98 & 49 & 68 & -8 & 83 \\ 314 & 146 & 178 & 80 & 309 & 339 & 99 & 407 & 296 & 311 & 207 & 430 & 122 & 320 & 43 & 184 \\ -1 & 38 & -39 & -8 & 1 & -90 & -39 & -50 & -4 & 1 & -72 & -75 & -48 & -50 & 14 & -83 \end{bmatrix}$$

Encryption - Step 5

This message is converted to a string of numbers, as shown below.

28 -18 47 13 26 102 40 76 29 26 76 98 49 68 -8 83 314 146 178 80 309 339 99 407 296 311 207 430 122
320 43 184 -1 38 -39 -8 1 -90 -39 -50 -4 1 -72 -75 -48 -50 14 -83

Encryption - Step 6

The next step is to convert the numbers into letters. This will create the ciphertext that is sent. We will use the following algorithm, which is a very similar scheme to the translation we did at the beginning that converted letters to numbers. The algorithm is shown below:

A=1 B=2 C=3 D=4 E=5 F=6 G=7 H=8 I=9 J=0 K=' ' L=-

The cipher text is now:

LADKHAKAAIKGAKLBIKLIKECKLBIKIJKHIKAIKLEKEIKADHKAHIKCBHKBJFKAIJKABJKBHFK
AIJKBHAKCGFKBIDKGFKCAGKCKLHJKLAABKLFFKEFKBDKLCCEKFKLHAKLGJHKHAKDKLCI

Decryption


Once received, the ciphertext is decoded by reversing step 6 of the encryption algorithm so that the ciphertext is back to a list of numbers with spaces in between. The list of numbers is converted back into a matrix by dividing it into 3 equal rows. In this case we know that the numbers will divide evenly because of the padding in step 2. This gives us the matrix E found in Step 4. After that we multiply the inverse, A^{-1} by the ciphertext matrix, E . The inverse of A is shown below:

$$A^{-1} = \begin{bmatrix} -57 & 5 & -46 \\ -11 & 1 & -9 \\ 1 & 0 & 1 \end{bmatrix}$$

Since $E = AM$, $A^{-1}(E) = A^{-1}(AM)$. We know from the start will simply give us M , the message matrix. From this, it is a simple process to pull out the string of numbers and replace them with the corresponding letter (or space). This gives the original message, known only to the sender and the recipient

Conclusion

The algorithm we have presented is ideal for encrypting messages of any using the standard alphabet. Messages encrypted using our algorithm are not vulnerable to others who may discover the algorithm. The cipher key is necessary to encrypt or decrypt messages and cannot be derived from knowledge of the algorithm or by analyzing the cipher text. The matrix encryption algorithm is a secure algorithm for encoding messages without requiring large resources or computation and results in a ciphertext of english characters. We think this method is an ideal solution for secure, fast messaging.



Introduction

We were approached by OCRAI with a request to design a method of encryption that could be applied to ordinary employee communications and other typical text files. This algorithm is meant to be secure and efficient for encrypting and decrypting English messages so that important information is not leaked. Per the request of OCRAI this is a novel non-standard method of encryption developed specifically for their needs.

Methods

The method for encryption which we used is based on the principle that any number has a distinct factorization into prime numbers, up to order. When we receive the data, we first assign each of the 26 letters with the corresponding lowest prime number. The prime numbers used for this are not part of the key, but are instead common knowledge. $A = 2, B = 3, \dots, Z = 101$. Additionally, each of the 10 numerals were assigned the 27-36th smallest prime numbers. Therefore, $0 = 103, 1 = 107, \dots, 9 = 151$. We then break apart the plaintext into “words”, with any non-letter/number serving as a delimiter. As an example, the phrase, “I don’t 100% know if I love you yet, but you’re pretty neat I guess” will break into the words: “I” “don” “t” “100” “know” “if” “I” “love” “you” “yet” “but” “you” “re” “pretty” “neat” “I” “guess”. Note that all delimiters, including spaces and punctuation, will be passed into the ciphertext unaltered.

We then take each word, convert each character of the word into the corresponding prime, and output the product of the characters plus a 5-digit number which serves as our key. As an example, if our key is 12345, then the word “cab” becomes $5 * 2 * 3 = 30$ ($C = 5, A = 2,$

B = 3) and, adding the key, becomes “30 + 12345” = “12375”. The only problem is that the receiver of that word, “30”, can obtain the letters “ABC”, but won’t know the order they need to go in. For that reason, we have added a number in the ciphertext which indicates the proper permutation of the letters. Let the word be organized as a list of addresses, similar to a string. Therefore, in the word “CABBY0”, the letter C is in address 0. The letter A is in address 1. The letter B is in address 2. The letter B is also in address 3. The letter Y is in address 4. The numeral 0 is in address 5. In our ciphertext, we have included a number following each “word” which shows the proper permutation of the characters by showing the addresses of each letter, from the largest corresponding prime number to the smallest corresponding prime number. We also add a 0 at the beginning of the permutation to distinguish it from the “words”.

For example,

We convert “CABBY0” to its primes, “5 * 2 * 3 * 3 * 97 * 103”.

We multiply our primes together, “899190”.

We add our cypher key, “899190 + 12345” = “911535”.

We add our permutation, “0540231”.

We send the message “911535 0540231” to Bob.

Bob breaks the word, 911535, into the primes “2 * 3 * 3 * 5 * 97 * 103”.

Bob then converts those primes into the characters, “A, B, B, C, Y, 0”.

Bob then takes the permutation number (0540231), removes the beginning 0 (540231), and matches the addresses to the characters, “A = 1, B = 3, B = 2, C = 0, Y = 4, 0 = 5”.

Therefore, the order of the letters becomes “C = 0, A = 1, B = 2, B = 3, Y = 4, 0 = 5”.

Bob then has the message, “CABBY0”.

Another example is the sentence, “I 100% know how to decrypt this.”

Which becomes the primes, “23 103 * 103 * 107 31 * 43 * 47 * 83 19 * 47 * 83 47 * 71 5
* 7 * 11 * 53 * 61 * 71 * 97 19 * 23 * 67 * 71”.

Which is sent to Bob as, “12368 00 1147508 0012% 5212378 03210 86464 0210 15682 001
8572295680 04635102 2091154 00321.”

Which becomes, “I 0 010 012% KNOW 3210 HOW 210 OT 01 CDEPRTY 4635102 HIST
0321.”

And is decrypted to finally show, “I 100% know how to decrypt this.”

In summary,

Take the message and convert every character in each word into primes. Multiply those primes together. Add the key, which is as a 5-digit number, to each word. After each number, include the permutation of those letters, which is a list of addresses for each character in the order from largest to smallest. Then send the message. The decrypter subtracts the key from each word, finds the prime factors of each word, and sorts each word into its characters. The decrypter then changes each word into its proper form through the permutation given after each word. This cryptosystem is, of course, not without flaws or weaknesses. The cipher could be strengthened if a slightly longer key was used which would designate a rule as to which prime corresponds to which letter instead of simply using a public listing. As it is, however, we deem that the described method of encryption will be sufficient for all of OCRAI’s data protection needs.

List of Primes:

A 2

B 3

C 5

D 7

E 11

F 13

G 17

H 19

I 23

J 29

K 31

L 37

M 41

N 43

O 47

P 53

Q 59

R 61

S 67

T 71

U 73

V 79

W 83

X 89

Y 97

Z 101

0 103

1 107

2 109

3 113

4 127

5 131

6 137

7 139

8 149

9 151

Group E

Cryptography Project 1



September 11, 2020

We at IMC were approached by your company concerning issues involving internal security breaches. Specifically, you expressed having issues where sensitive information has been sent via text to the wrong recipients. To keep your data and information secure, we have come up with a nontrivial method of encrypting these plaintext English messages. In this report we will go over the encryption process, as well as the mathematical underpinnings which keeps your data safe.

Before defining the theory for the current system, some definitions must be provided. First, a *dynamical system* is (loosely) a function $\phi_t(x) = \phi(t, x)$ mapping $T \times M \rightarrow M$ satisfying $\phi_0(x) = x$ and $\phi_{t_1+t_2}(x) = \phi_{t_2}(\phi_{t_1}(x))$. In other words, a dynamical system provides the flow of each point x under a time map, where the flow begins at x and can be naturally partitioned into subflows. A *chaotic* dynamical system is here defined to be a system which is *topologically mixing* and has *sensitive dependence on initial conditions*. The former means merely that, given two subsets A, B of M , if we flow the points of A long enough (say, for some time N) they will eventually intersect B , and at least one flowed point of A will continue to intersect B for all time after that (formally, $\exists N$ such that, $\forall n > N$, $\phi_n(A) \cap B \neq \emptyset$). We think of the flow as having ‘mixed’ the set A and B .

The latter term, namely *sensitive dependence on initial conditions*, merely means that nearby (but unequal) points get split apart (for at least some time) as the system evolves. Finally, a *discrete* dynamical system is merely a dynamical system such that the time component is given by iteration of some related function $\phi : M \rightarrow M$ (with $T = \mathbb{Z}$ or $T = \mathbb{N}$), i.e. $\phi_0(x) = x$, $\phi_1(x) = \phi(x)$, $\phi_2(x) = \phi(\phi(x))$, and so forth.

With these definitions in mind, the basic encryption idea can be outlined as follows: take some dynamical system with an unknown parameter (representing the key), canonically embed the message into some points of the space domain for the system, flow the system forward for some time, determine the points where the embedded message flowed to, and turn these points back into some encrypted, alphanumeric message. To decrypt the message all one must do is re-embed the ciphertext and flow *backwards* in time until a readable message appears. To prevent precision or floating point errors with decryption one should force the system to be discrete, and to ensure decent encryption the system should be chaotic.

Fortunately, there is a very simple system which satisfies these constraints: *The Discrete Cat Map*, which we will hereafter refer to either as ‘the cat map’ or with the letter Γ . The simple version of the cat map we need here is defined to be the function $(\mathbb{Z}_N)^2 \rightarrow (\mathbb{Z}_N)^2$ given by $\Gamma(x, y) = (2x + y, x + y) \bmod N$ (*Note: throughout the rest of this paper, the letter N will be used only for this purpose*) Importantly, the cat map is known to be chaotic and mixing, though proofs of these would take us far afield in this paper.

The cat map has another wonderful feature which makes it useful for a simple encryption system: it is periodic. What this means is that there exists some $k \in \mathbb{N}$ such that $\Gamma_n(x, y) = (x, y)$. As such, we can keep iterating the system and will get back what we started infinitely many times; this is known as Poincaré recurrence, and k is known as the return time. The importance of this feature is that instead of needing to flow backward in time to return to our original state, we can instead flow forward in time. Interestingly, the return time depends heavily on N , with no simple formula existing. In fact, it is not even true that as N increases the return times increase (for example, $N = 104$ yields $k = 25$, while $N = 124$ yields $k = 15$). What *is* known is that $k \leq 3N$, so the return time cannot be radically higher than N . Because of this, we can view the return time as being something easy to explicitly calculate from N , meaning the cat map has

only the single changing parameter N . With the motivation in the prior paragraph in mind, we will thus let N represent the key for our cryptosystem.

All that remains is to specify a way to embed and extract alphanumeric data into lattice points of $(\mathbb{Z}_N)^2$ and extract/re-embed the encoded message. There are plenty of ways to do this, but the following is the way we have chosen to use: first all spaces, punctuation, and another unnecessary data is removed, leaving only letters, numbers, and symbols. We make everything alphabetic, representing numbers or symbols with a sequence of appropriate letters distinguishing them from the alphabetic text. For example, we might represent the number '2' with the letters 'qqtwoqq', with the pair of double 'q's signaling that the text contained within is a description of the number, though the choice of signifier and encoding can be chosen arbitrarily so long as it is readable and will not be confused with surrounding text. All letters are then made lowercase, and are made numeric in the natural way ($a = 1, b = 2, \dots, z = 26$). What we have now is an ordered collection of integers k_n (where k_n is the integer in the n th position), and to embed these integers as lattice points we map each integer k_n to the point (k_n, n) . At this point we need to view these points as living in $(\mathbb{Z}_N)^2$ for some $N \in \mathbb{N}$. Since our messages will be split into blocks of size at most 160 it suffices to choose $N > 160$.

For the sake of proper cryptosecurity in your forthcoming appliction, the choice of N should be made very large (and should probably be ensured to have a period at least as large as N), though for the sake of testing purposes and demonstration it suffices to use three digit keys (all of which should be greater than 160). (*Note: This also ensures the 800 character requirement is met*). We then flow the system, resulting in a sequence of lattice points. Since $N > 26$, many of these points will have entries far larger than 26, so we need to find a way to (invertibly) encode these points alphanumerically. The way we will do this is by taking considering each lattice point, splitting the first lattice point into single digits, replacing each with the appropriate letter, then concatenating together with the second entry in the lattice point. As an example, the lattice point $(169, 967)$ would be replaced by the alphanumeric string 'afi967' (the digit 0 will be represented by 'z' if it appears). We then concatenate all lattice points, preserving order (see the example below for more details). Since this is clearly invertible, we can later re-embed our encoded message as lattice points.

All the hard work is now done; all that remains is to actually iterate the system and encrypt the data. The amount of time the system is flowed does not matter, so long as the time we flow is not the return time. (*Technical note: for some N the message will be encoded backward at half the return time, so one should not flow to this point either. Nevertheless, for very large N most return times will be quite long, making the choice of these points quite unlikely. If it still occurs, just choose a different amount of time to flow and repeat.*) We now provide a simple example using the text 'Bob ate 2 rolls', with $N = 124$. (*Note: the chosen example message is far shorter than 124 characters, so this choice of N will work here. It was chosen for this example only due to its relatively short period so that a reader could verify the calculations if desired*).

First, the text 'Bob ate 2 rolls' is made alphabetic and lowercase, becoming 'bobateqqtwoqqrolls'.

This string is then made numeric, becoming the ordered list

$\{2, 15, 2, 1, 20, 5, 17, 17, 20, 23, 15, 17, 17, 18, 15, 12, 12, 19\}$

which become ordered lattice points as follows:

$\{[2, 0], [15, 1], [2, 2], [1, 3], [20, 4], [5, 5], [17, 6], [17, 7], [20, 8], [23, 9], [15, 10], [17, 11], [17, 12], [18, 13], [15, 14], [12, 15], [12, 16], [19, 17]\}$

The cat map is then run on each lattice point in order, resulting in the sequence:

$\{[4, 2], [31, 16], [6, 4], [5, 4], [44, 24], [15, 10], [40, 23], [41, 24], [48, 28], [55, 32], [40, 25], [45, 28], [46, 29], [49, 31], [44, 29], [39, 27], [40, 28], [55, 36]\}$

Running the cat map four more times results in the sequence:

$\{[54, 110], [26, 115], [40, 54], [6, 33], [16, 120], [100, 73], [107, 23], [38, 57], [112, 8], [62, 83], [25, 49], [10, 69], [65, 103], [85, 68], [121, 61], [33, 54], [88, 88], [22, 11]\}$

We will use this for our encoded message. We first split and alphabetize the digits of the first entry of each coordinate (recall 0 is mapped to 'z', since it should represent letter before 'a'):

```
{[ed, 110], [bf, 115], [dz, 54], [f, 33], [af, 120], [azz, 73], [azg, 23], [ch, 57], [aab, 8],  
[fb, 83], [be, 49], [az, 69], [fe, 103], [he, 68], [aba, 61], [cc, 54], [hh, 88], [bb, 11]}
```

We then concatenate everything together into a string, yielding:

```
ed110bf115dz54f33af120azz73azg23ch57aab8fb83be49az69fe103he68aba61cc54hh88bb11
```

When we wish to decode this message, we can of course invert the process back until we get the sequence

```
{[54, 110], [26, 115], [40, 54], [6, 33], [16, 120], [100, 73], [107, 23], [38, 57], [112, 8],  
[62, 83], [25, 49], [10, 69], [65, 103], [85, 68], [121, 61], [33, 54], [88, 88], [22, 11]}
```

at which point running the cat map ten more times will return the original sequence

```
{[2, 0], [15, 1], [2, 2], [1, 3], [20, 4], [5, 5], [17, 6], [17, 7], [20, 8],  
[23, 9], [15, 10], [17, 11], [17, 12], [18, 13], [15, 14], [12, 15], [12, 16], [19, 17]}
```

which can be quickly decoded into the original message.

All that remains is to be specific on how messages longer than 160 characters are broken up into blocks. The order taken will be to first make the message entirely alphabetic in the manner above (e.g. removing spaces and replacing numbers and symbols with text equivalents) and to then extract text blocks of length 133 until less than 133 characters remain, making this the last block. The same N is then used on each block for encryption and decryption purposes. The only exception to this rule is when extracting a block of size 133 would only partially grab the text expansion of a symbol (e.g. if the symbol '2' were expanded as 'qqtwoqq' and the 133rd character was the any character of that expanded symbol except for the last 'q'); in this case, the message is to be cut-off just prior to the symbol expansion, so the symbol will be sent in its complete form in the next message. Upon encryption these messages will not exceed 800 characters given the recommendation for a 3-digit choice of N greater than 160, since in this case each coordinate becomes, under iterations the cat map, at most a pair of 3-digit numbers. Since the ciphertext contains at most 133 pairs of 3-character strings, the length of the ciphertext for each block is at most 798 characters.

In summary, we have created a simple encryption system using Arnold's Cat Map. By using this chaotic dynamical system, we create a simple yet decently robust encryption of your sensitive data. Our solution offers a great deal of speed in encryption and decryption, since the encrypted message can be returned to its original state, given the key, merely by repeated iteration of the cat map. Being a chaotic and discrete system, our solution additionally offers increased security and precision over a generic choice of encryption via dynamical systems. With all this in mind, we feel our proposed solution will be more than sufficient to meet your security needs.

Group F

OCRA Creative Recursive Acronyms, Inc.
485 Primality Way
Provo, UT 84604

Dear OCRAI,

The use of text messaging for sending sensitive information can be dangerous as mistakes can be made and attackers will take advantage of this vulnerability. To protect your company's trade secrets, we have developed a cipher with which to encrypt and decrypt your messages. Solving this problem requires taking into account the drawbacks of using the system of a cell phone, which limits the maximum key size and message length used in our cipher.

To successfully keep your messages secure with these limitations, we considered multiple methods an attacker might use and developed the appropriate precautions. A common method of cryptanalysis includes character frequency analysis. To throw off attackers, we chose to use a more randomized method of encrypting characters than a simple substitution cipher. We implemented a Chain Code so that each character undergoes a unique transformation based on both its position in the plaintext and value, rather than simply its value.

However, this is not enough to ensure security for your company's messaging system. Even with our coded text, attackers gain valuable information from the ordering of characters in the message. For example, using a chosen-plaintext attack, eavesdroppers may deduce the key simply by comparing the found plaintext and its associated ciphertext. Therefore, we created a two-stage cipher that utilizes keyed columnar transposition to further randomize the encrypted message and throw attackers off the ordering of the message. This will make the key more useful to the encrypting of the plaintext.

Below is a description of the cipher. We believe that, despite the described limitations, our solution will significantly increase the security of your messaging system and protect your company's sensitive data.

Cipher Overview

Our cipher is constructed from a Chain Code and a keyed columnar transposition. Our cipher takes in a key of 10 unique characters, a plaintext, and produces a ciphertext. At a high level, encryption happens by converting the plaintext into integers (mod 26) using the A0-Z25 mapping. The plaintext is then padded with 'X' characters to a length that is a multiple of 10. The plaintext is then added to the keystream (mod 26) generated by the Chain Code. This result is finally put through a keyed columnar transposition, and the integers are converted back to ciphertext characters using the same A0-Z25 mapping. Decryption follows this same process, just backwards.

Chain Code:

This component takes inspiration from linear-feedback shift register (LFSR) based keystream generators, and functions as follows. Using the ten character key, converted to integers by A0-Z25 mapping, as an initial state (k_0, \dots, k_9), additional keystream characters are generated using the linear recurrence relation...

$$k_{i+10} = k_i + k_{i+1} + k_{i+4} + k_{i+6} + k_{i+9} \pmod{26}$$

This recurrence relation is used to generate a keystream of length equal to the length of the plaintext. This relation bears some resemblance to a LFSR with taps in the 1, 2, 5, 7, and 10 positions. These taps are chosen to increase the periodicity of the keystream.

To encrypt using the Chain Code, simply add the plaintext character to the keystream character mod 26. In other words...

$$c_i = p_i + k_i \pmod{26}$$

This ensures that each plaintext character is mapped to a ciphertext character in a simple, but dynamic way.

To decrypt using the Chain Code, simply add the ciphertext character to twenty-six minus the keystream character mod 26. In other words...

$$p_i = c_i + (26 - k_i) \pmod{26}$$

This decryption works because we are simply adding the additive inverse of the keystream character k_i to each ciphertext character c_i , thus canceling the effect of the key.

Keyed Columnar Transposition:

This component is included as a defense against keystream reconstruction attacks on the Chain Code using predicted or known plaintexts. This transpose works by using the 10 key characters as the key for a simple columnar transposition.

In the keyed columnar transposition, the message is written out in rows of a fixed length, and then read out again column by column, and this column order is chosen depending on the key. For example, the key ZEBRAS is of length 6 (so the rows would be of length 6), and the permutation is defined by the alphabetical order of the letters in the key. In this case, the order would be "6 3 2 4 1 5". Suppose we use the key ZEBRAS and the message WE ARE DISCOVERED. FLEE AT ONCE. We can write this into the grid as follows:

6 3 2 4 1 5
W E A R E D
I S C O V E
R E D F L E
E A T O N C
E Q K J E U

The encrypted message would then be read out as EVLNA CDTES EAROF ODEEC WIREE. To decrypt this, the receiver has to work out the column lengths by dividing the message length by the key length. Then they can write the message out in columns again, then reorder the columns by reforming the key word.

Overall, this cipher is an effective solution due to the ease of implementation, long keystream periods, and Irwin–Hall uniform output distribution. The Chain Code component effectively thwarts any sort of character frequency analysis, due to the effects of the Von Neumann extraction procedure and an additive distribution transformation. Due to simplicity concerns, irregular stepping/clocking, non-linear tapping, and non-linear key state mixing techniques were not included in the Chain Code design. Alone, this means that the Chain Code component is vulnerable to linear cryptanalysis using either a chosen or predicted plaintext, and can be broken using the modified Berlekamp-Massey algorithm. To counter this, we added stage, the keyed columnar transposition, which restricts the ability of an attacker to perform such an attack. This cipher is not without vulnerability, but it offers a simple design and provides modest security given the design restrictions.

Best regards,

Cipher Design Team
IMC

APPENDIX A: Python3 Cipher Demo Implementation

```
import re # for regex
import math # for math purposes

def strip_text(text):
    return re.sub('[^A-Za-z]+' , , text)

def pad_text(text,block_size):
    while((len(text) % block_size) != 0):
        text += "X"
    return text

def string_to_num_array(input):
    input = input.upper()
    result = []
    for x in range(len(input)):
        char = ord(input[x]) - 65
        result.append(char)
    return result

def num_array_to_string(input):
    result = ""
    for x in range(len(input)):
        char = chr(int(input[x] % 26) + 65)
        result += char
    return result

def generate_chain_encryption_key(key, length):
    i = 0
    while (len(key) < length):
        key.append((key[i] + key[i+1] + key[i+4] + key[i+6] +
key[i+9]) % 26)
        i += 1
    return key

def generate_chain_decryption_key(key, length):
    key = generate_chain_encryption_key(key, length)
    for i in range(len(key)):
        key[i] = 26 - key[i]
    return key

def columnar_encrypt(msg, key):
    cipher = ""
    k_idx = 0
    msg_len = float(len(msg))
    msg_lst = list(msg)
    key_lst = sorted(list(key))
    col = len(key)
    row = int(math.ceil(msg_len / col))
    fill_null = int((row * col) - msg_len)
    msg_lst.extend('_' * fill_null)
    matrix = [msg_lst[i: i + col]
        for i in range(0, len(msg_lst), col)]
    for _ in range(col):
        curr_idx = key.index(key_lst[k_idx])
        cipher += ".join([row[curr_idx]
            for row in matrix])
        k_idx += 1
    return cipher

def columnar_decrypt(cipher, key):
    msg = ""
    k_idx = 0
```

```
msg_idx = 0
msg_len = float(len(cipher))
msg_lst = list(cipher)
col = len(key)
row = int(math.ceil(msg_len / col))
key_lst = sorted(list(key))
dec_cipher = []
for _ in range(row):
    dec_cipher += [[None] * col]
for _ in range(col):
    curr_idx = key.index(key_lst[k_idx])
    for j in range(row):
        dec_cipher[j][curr_idx] = msg_lst[msg_idx]
        msg_idx += 1
    k_idx += 1
try:
    msg = ".join(sum(dec_cipher, []))
except TypeError:
    raise TypeError("This program cannot handle repeating letters
in key.")
null_count = msg.count('_')
if null_count > 0:
    return msg[: -null_count]
return msg

def encrypt(plaintext, key):
    plaintext = strip_text(plaintext)
    plaintext = pad_text(plaintext, 10)
    key_array = string_to_num_array(key)
    plaintext_array = string_to_num_array(plaintext)
    chain_code_encryption_key =
generate_chain_encryption_key(key_array, len(plaintext_array))
    ciphertext_array = []
    for i in range(len(plaintext_array)):
        ciphertext_array.append((plaintext_array[i] +
chain_code_encryption_key[i]) % 26)
    ciphertext = num_array_to_string(ciphertext_array)
    ciphertext = columnar_encrypt(ciphertext,key)
    return ciphertext

def decrypt(ciphertext, key):
    key_array = string_to_num_array(key)
    ciphertext = columnar_decrypt(ciphertext,key)
    ciphertext_array = string_to_num_array(ciphertext)
    chain_code_decryption_key =
generate_chain_decryption_key(key_array, len(ciphertext_array))
    plaintext_array = []
    for i in range(len(ciphertext_array)):
        plaintext_array.append((ciphertext_array[i] +
chain_code_decryption_key[i]) % 26)
    plaintext = num_array_to_string(plaintext_array)
    return plaintext

##### testing values below #####
key = "YELZOWSUBA"
encrypted = encrypt("According to all known laws of aviation, there
is no way a bee should be able to fly. Its wings are too small to get its
fat little body off the ground.", key)
decrypted = decrypt(encrypted,key)
print(encrypted,decrypted)
```



1 Introduction

In any business setting, a secure method of communication is required in order for employees within the organization to collaborate and complete a project. The method must be simple enough to implement so that message is still feasible to share without alteration or corruption, while remaining secure from those for whom it is not intended. It is also vital to be able to identify, with a high confidence, where a message comes from and where it goes. The method that we present below does just that; it uses simple mathematics with large digits and prime numbers to encrypt a message in a way that only the sender and recipient are able to read the message and know that it did in fact originate from the sender and was received by the recipient. Furthermore, we outline how OCRAI's in-house engineers will be able to implement this method within the app, thus avoiding any potential human error in encryption and decryption.

2 Encryption Method

We begin by reading in the message as it is. The encryption key is a 10 digit number, where all 10 of the digits are in base 5, and the first digit is not a 0. This means there are $4(5^9)$ possible encryption keys. Furthermore, The maximum encryption key is 4,444,444,444 and our minimum encryption key is 1,000,000,000. Each employee will be assigned a unique encryption key, randomly generated from the above constraints.

We manipulate the data by simultaneously iterating over the string and the encryption key. For each character the message, a number of random ascii letters (upper and lower case letters) are added after the character, where the number of characters corresponds to the current digit in the encryption key. For example if the text message is "Alice wants to send a message to Bob" and the encryption key is 3221420140, after one iteration, the message would look something like:

AdFIlice wants to send a message to Bob

We added three random characters after "A", namely "dFI", since we are at the first character in the message (A) and the first digit in the encryption key (3). After 2 iterations, the message would look something like

AdFIleaiice wants to send a message to Bob

At the 11th iteration, we would be at the character s in the word wants. Since our encryption key is only 10 digits long, we would loop back to the beginning of the key, meaning 3 letters will be added after the s. In general, if we are at the i th character in the string, we are at the $i\%10$ th digit in the encryption key.

Using the same text message and encryption key as above, the message would look something like

AdFileaiDHcpenggW CbwaxnSqfftsGkQ eVtGXok FUSNsMRenqdIOLp aWkf
xpmyXeMsgGersxxagueXHWY tbKfoIT JIBvouKgDbsi

Note: Spaces count as a character in our data manipulation.

The engineers need to design a way for the app to decrypt the messages if they are sent to an employee. Along with a unique encryption key, each employee also has a unique prime number. In order to calculate the range of prime numbers needed, we need the equation

$$\pi(x) \approx \frac{x}{\ln(x)}$$

where $\pi(x)$ is the number of primes less than x . This means we need to find an x such that

$\pi(x)$ is greater than the number of employees. For example, if you have 1,000 employees, $x = 10,000$ is a sufficient number since

$$\pi(10,000) \approx \frac{10,000}{\ln(10,000)} \approx 1,085$$

So choosing primes less than 10,000 will guarantee that each employee has a unique prime.

The decryption key is fairly simple: it is the sender's encryption key multiplied by a by the sender's prime, with the sender's prime and recipient's prime appended on the end. For example, using 3221420140 as the encryption key, 53 as the sender's prime, and 13 as the recipient's prime, the decryption key is 1707352674205313 since $322142014 * 53 = 170735267420$, the sender's prime number is 53 and the recipient's prime is 13. This decryption key provides a way for the app to verify the identity of both the sender and the receiver since each prime is unique to an employee.

To decrypyt the message, the sender's prime and the recipients prime will be removed from the decrypyton key, and then what is remaining will be divided by the sender's prime. For example, our decryption key is 1707352674205313. We remove 53 and 13 to get 170735267420 and then divide 170735267420 by 53, which gives us 322142014, precisely the sender's unique encryption key. Now we use the sender's encryption key to decrypt the message by essentially working backwards, iterating over the encrypted message and the encryption key. The encrypted message is

AdFileaiDHcpenggW CbwaxnSqfftsGkQ eVtGXok FUSNsMRenqdIOLp aWkf
xpmyXeMsgGersxxagueXHWY tbKfoIT JIBvouKgDbsi

At the first iteration, the character we are looking at is A and the digit in the encryption key is 3. Thus, we remove the 3 characters after A to get

AleaiDHcpenggW CbwaxnSqfftsGkQ eVtGXok FUSNsMRenqdIOLp aWkf
xpmyXeMsgGersxxagueXHWY tbKfoIT JIBvouKgDbsi

We continue this process until we reach the end of the message, looping through the encryption key like we did with encrypting the message.

A careful analysis shows that our method of encryption will not produce a message over 800 characters. Suppose we have the maximum number of characters allowed in the text message (160 characters) and the maximum encryption key (4,444,444,444). This means that 160 of the characters will have 4 characters added after them, so the total characters after encryption is 800.

3 Conclusion

As we can see, this method preserves the message, including punctuation, symbols, and the case of the letter, while keeping the encrypted message within the specified parameters. This method also allows a unique prime "identity" to be assigned to each employee, thus allowing sender and recipient verification, as well as a secure way for the recipient to be able to read the message. We believe that once implemented, this app will allow OCRAI's employees to communicate easily and securely, and avoid any future mishaps with leaking secure company data.

Encryption Proposal

1 Introduction

Dear Mr. Andrews,

We are sorry to hear that your company has been struggling with security breaches. We are happy to offer our assistance, and we promise not to charge as much as Dr. Paul Jenkins.

Now, as we understand your problem, you need a system for encrypting text messages sent by your workers. It is imperative that the encryption system we provide can comfortably and securely encrypt a message of up to 160 characters while producing a cipher text no longer than 800 characters.

We propose a system based upon a known method- using an invertible matrix for the encoding key- with a slight twist. As we learn from Kerckhoff's principle, the power of a cipher cannot lie in the method alone, but must also lie in the key used. Our system would allow your company to choose from an infinite number of keys- making attack by brute force impossible- while using a simple computation to enact the key.

In this document we provide a detailed explanation of our method of encryption. We hope that you choose to work with us, and promise the results that you desire.

2 Encryption and Decryption

2.1 The Encryption Process

In order to increase the security of your messages, we propose the following system of encryption. For your convenience, a step by step sample of the encryption and decryption process is included below.

We will start by translating the characters of the message into their ASCII codes, which will instantly make them less available to any outsider. We will then add the number 1 twice between the digits, so that we end up with 4 digit numbers representing each character.

Each 4 digit number will then be converted to a 2x2 matrix by entering each digit into the matrix, from the top left to the top right, and then from the bottom left to the bottom right. After this process, we will have a matrix for each character in the message.

Now comes the part that makes the system truly secure. We use a different 2x2 matrix as the encryption key and multiply (using matrix multiplication) each character matrix by the encryption key matrix. This will produce new matrices from each of the original matrices. The reason this is so secure is because the encryption key could be any 2x2 matrix, and without the inverse of the matrix, we cannot convert back to the original matrices.

We then extract the matrix values from the matrices, and line them up without spaces between them. This creates a seemingly random string of a mixture of positive and negative integers, and that is what an attacker would see upon acquiring any messages sent by company employees.

Decryption, with the inverse of the encryption matrix is simple. We simply divide the digits into groups of 4 digits, put them back into matrix form (from the top left to the top right, and then from the bottom left to the bottom right), and multiply each matrix by the inverse of the encryption matrix (the decryption key). This will produce the original matrices. We then write them in groups of 4 numbers again, remove the 11 from between each group, and convert the ASCII codes back into characters.

Plaintext: To be, or not to be

Convert text to uppercase and remove punctuation: TO BE OR NOT TO BE

Translate to ASCII: 84 79 32 66 69 32 79 82 32 78 79 84 32 84 79

Insert extra ones between each character code: 8114 7119 3112 6116 6119 3112 7119 8112 3112 7118 7119 8114 3112 8114 7119

Convert to matrices: $\begin{bmatrix} 8 & 1 \\ 1 & 4 \end{bmatrix}, \begin{bmatrix} 7 & 1 \\ 1 & 9 \end{bmatrix}, \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix}, \begin{bmatrix} 6 & 1 \\ 1 & 6 \end{bmatrix}, \text{etc.}$

Right multiply by the 2x2 invertible matrix of your choice: $\begin{bmatrix} 8 & 1 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 7 & 1 \\ 1 & 9 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix},$

$\begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 6 & 1 \\ 1 & 6 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}, \text{etc.}$

Multiplication results: $\begin{bmatrix} 8 & -7 \\ 1 & 3 \end{bmatrix}, \begin{bmatrix} 7 & -6 \\ 1 & 8 \end{bmatrix}, \begin{bmatrix} 3 & -2 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 6 & -5 \\ 1 & 5 \end{bmatrix}, \text{etc.}$

Extract matrix values and piece back together: 8-7137-6183-2116-515...

Outcome: To be, or not to be \Rightarrow 8 - 7137 - 6183 - 2116 - 5156 - 5183 - 2117 - 6188 - 7113 - 2117 - 6177 - 6188 - 7133 - 2118 - 7137 - 6183 - 2116 - 5156 - 518

2.2 The Decryption Process

The decryption process is similar. We take the cipher text, rearrange it into matrices, decode it using the inverse of our encrypting matrix, and convert back into letters from the ASCII key characters.

Ciphertext: 8-7137-6183-2116-5156-5183-2117-6188-7113-2117-6177-6188-7133-2118-7137-6183-2116-5156-518

Rearrange as matrices: $\begin{bmatrix} 8 & -7 \\ 1 & 3 \end{bmatrix}, \begin{bmatrix} 7 & -6 \\ 1 & 8 \end{bmatrix}, \begin{bmatrix} 3 & -2 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 6 & -5 \\ 1 & 5 \end{bmatrix}, \text{etc.}$

Right multiply by the *inverse* of your encryption matrix: $\begin{bmatrix} 8 & -7 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} -1 & 1 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 7 & -6 \\ 1 & 8 \end{bmatrix} \begin{bmatrix} -1 & 1 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 3 & -2 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} -1 & 1 \\ 0 & 1 \end{bmatrix},$
 $\begin{bmatrix} 6 & -5 \\ 1 & 5 \end{bmatrix} \begin{bmatrix} -1 & 1 \\ 0 & 1 \end{bmatrix}, \text{etc.}$

Multiplication results: $\begin{bmatrix} 8 & 1 \\ 1 & 4 \end{bmatrix}, \begin{bmatrix} 7 & 1 \\ 1 & 9 \end{bmatrix}, \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix}, \begin{bmatrix} 6 & 1 \\ 1 & 6 \end{bmatrix}, \text{etc.}$

Take out of matrix form: 8114 7119 3112 6116 6119 3112 7119 8112 3112 7118 7119 8114 3112 8114 7119

Remove the extra ones: 84 79 32 66 69 32 79 82 32 78 79 84 32 84 79

Convert from ASCII to letters: TO BE OR NOT TO BE

From here, you can fix casing and replace punctuation however you please.

3 Conclusion

By implementing a method of encryption that requires a key that can be chosen from an infinite number of possibilities, we can guarantee that your messages will be kept private and secure. The system is multi-leveled, implementing the translation of letters to numbers, numbers to matrices, and matrices to different matrices. To the naked eye, the encryption will appear completely undecipherable. Even to the most skilled attacker who is aware of our method of encryption, the prospects of deciphering by brute force are impossible.

Again, we are sorry to hear that your company has been affected by breaches and attacks. Our system of encryption will provide security and peace of mind, so that you can continue to succeed and grow as an organization.

Sincerely,

A solid black rectangular box used to redact a signature.

Group I

Introduction

We have received your request to build a nontrivial cryptographic algorithm that can convert up to 160 characters of plaintext into up to 800 characters of ciphertext, for use in employee communications. Our algorithm uses as its encryption key an integer between 1 and 27, and our algorithm converts each character of plaintext into one character of ciphertext.

Encryption Overview

To encode any specific character in the plaintext, we take the integer value corresponding to the last ciphertext integer and multiply that integer by the encryption key. We take the newly created integer and add the position of the character in the text and the value of the plaintext character. Finally, we take the modulus of that sum and 27, and end up with an integer between 0 and 26, which corresponds to the value of a letter in the alphabet. That corresponding letter is used as the ciphertext character.

So, the steps in order are:

$$\text{cipherValue} = \text{lastCipherValue} * \text{encryptionKey}$$

$$\text{cipherValue} = \text{cipherValue} + \text{plaintextPosition} + \text{plaintextValue}$$

$$\text{finalCipherValue} = \text{cipherValue} \bmod 27$$

To compute the value of a modulo b , compute the following, with a , b , q , and r being integers:

$$a \bmod b = r, \text{ where:}$$

$$r = a - bq, \text{ and } 0 \leq r < |b|$$

The value of a character is its position in the alphabet, starting indexed from 0, with space taking on the value of 26.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

The last ciphertext value is assumed to be 0 for the first character encrypted and is updated after each new character is encoded from left to right. The position of the character in the text is assumed to be indexed starting from 0.

For an example, let us assume that we are encoding the second character of a message, which is the plaintext character 'w', our encryption key is 25, and the ciphertext character value for the first character of the message was an 'e', which has a value of 4. Because this is the second character of the message, it is in position 1, and because it is a 'w', it has a value of 22.

So, in the above steps, lastCipherValue is 4, encryptionKey is 25, plaintextPosition is 1, and plaintextValue is 22.

$$\text{cipherValue} = 4 * 25 = 100$$

$$\text{cipherValue} = 100 + 1 + 22 = 122$$

$$\text{finalCipherValue} = 122 \bmod 27 = 15$$

By following the steps, we discover that we would get a value of 15, so we check our alphabet and find that 'p' has a value of 15. So, the ciphertext assigned is the letter 'p'.

Decryption Overview

To decrypt any specific character, we take the alphabet value of the previous ciphered character, and multiply it by the encryption key, just as we did in the encryption, and then add the ciphered characters position. We then take that sum mod 27, and call this number the decryption value. We take the alphabet value of the current ciphered character and subtract the decryption value, before adding 27. We then take that sum mod 27, and end up with an integer between 0 and 26, which corresponds to the value of a letter in the alphabet. That corresponding letter is used as the plaintext character.

So, the steps in order are:

$$\text{decryptionValue} = \text{lastCipherValue} * \text{encryptionKey}$$

$$\text{decryptionValue} = \text{decryptionValue} + \text{plaintextPosition}$$

$$\text{decryptionValue} = \text{decryptionValue} \bmod 27$$

$$\text{cipherValue} = \text{currentCipherValue} - \text{decryptionValue} + 27$$

$$\text{finalDecipheredValue} = \text{cipherValue} \bmod 27$$

For an example, let us use the same values as we did for the encryption example. We have an encryption key of 25, we are on the second character of the ciphered message, so the position is 1. The current ciphered character is a 'p' and so has a value of 15. The first character of the ciphered message is a 'e', and so has a value of 4.

$$\text{decryptionValue} = 4 * 25 = 100$$

$$\text{decryptionValue} = 100 + 1 = 101$$

$$\text{decryptionValue} = 101 \bmod 27 = 20$$

$$\text{cipherValue} = 15 - 20 + 27 = 22$$

$$\text{finalDecipheredValue} = 22 \bmod 27 = 22$$

By following the steps, we discover that we would get a value of 22, so we check our alphabet and find that 'w' has a value of 22. So, the plaintext is the letter 'w'.

Conclusion

In conclusion, we have presented above our non-trivial cryptographic method to encrypt the data in your employee communications. By converting each plaintext character into a singular ciphertext character, we have kept the length of communications to a max of 160 characters a message, and with a key size of two digits, we believe our method will be of minimal impact on employee devices.

We hope this method meets your specifications, and that you will have a great day.

Sincerely,

A solid black rectangular box used to redact a signature.

Postscript: This report has some parts simplified for ease of explanation and reuse in your application. For better security, position could be assigned starting from a number other than 0, giving us another key to work with, and making the code harder to break.

We at Independent Math Contractors are happy to be trusted with your business. As we understand it, you require a secure method of sending sensitive information over text. You wish to give a normal message in plaintext to be encoded in a ciphertext, which you can send out without fear for its security. To provide you with maximum security, we have combined two well-known methods of encryption with one of our own invention, greatly reducing the possibility of a successful security breach in the near future. For your convenience, we have also created a computer program to perform the encryption so your employees can allocate their time to something more valuable.

The first part of our encryption process includes choosing a 9 letter keyword with no repeating letters. For the examples in this report, we will use the keyword COPYRIGHT. We used the keyword to create a matrix of the alphabet with the keyword in the top row. We then rearranged the columns of the matrix based on the alphabetical order of the keyword (as shown below). A basic substitution cipher was implemented by matching the positions of each letter in the old matrix with the new rearranged matrix. Since 9 does not evenly divide 26, if there was a gap in the matrix, the letter just matches the next available letter in the matrix. In the example below, this means that y=W and z=V.

C	O	P	Y	R	I	G	H	T		C	I	G	H	O	P	R	T	Y
a	b	c	d	e	f	g	h	i		a	f	g	h	b	c	e	i	d
j	k	l	m	n	o	p	q	r		j	o	p	q	k	l	n	r	m
s	t	u	v	w	x	y	z	_		s	x	y	z	t	u	w	_	v

For the second step of encryption, we use an affine cipher. In an affine cipher, letters A-Z are numbered 0-25. To determine how the plaintext (for this stage) letters correspond to ciphertext, we perform the equation $NewValue = \alpha * OldValue + \beta, \text{ mod } 26$, on the value assigned to our plaintext letter. (Mod 26 means we do normal arithmetic, but when we get to the number 26, we start over at 0, like counting on a clock).

To make our affine cipher more difficult to break, we change our values of α and β for each word. For the very first word, the α value equals the numerical value assigned to the first letter of the keyword. However, if the value of α is not relatively prime to the number of characters in the alphabet (in this case 26), then α is reduced (by subtracting 1) until it is relatively prime. If α is zero, then it wraps around to 26 and is reduced from there until it is relatively prime. For our example, with our keyword COPYRIGHT, the first letter is C, with a value of 2. Thus, the α value for the first word is 2. However, 2 is not relatively prime to 26, so it is reduced to 1, which is relatively prime. In a similar manner, β for each word equals the numerical value assigned to the last letter of the keyword. In our example, that is T, with a numerical value of 19, so β is 19. For the next word, we change the plaintext using another affine cipher. For this cipher, instead of using the numbers assigned to the first and last letters of the

keyword, we assign α and β based on the first and last letters, respectively, of the previous word. We continue to use affine ciphers based on the previous word for all remaining words until the entire plaintext sequence has been encoded using various affine ciphers.

Ex. If our “plaintext” from the first step is

Yib bayibm hiapebs rzdhpqw

The first word is coded with affine cipher $NewValue = 1 * OldValue + 19$, as explained above.

$$\begin{array}{r}
 Y I B \\
 24 \ 8 \ 1 \\
 24*1+19 \equiv 17 \pmod{26} \mid 8*1+19 \equiv 1 \pmod{26} \mid 1*1+19 = 20 \pmod{26} \\
 17 \ 1 \ 20 \\
 R \ B \ U
 \end{array}$$

The first word, encoded, is now rbu.

The second word is encrypted using the affine cipher $NewValue = 23 * OldValue + 1$ based on the former previous word yib. Remember that α must be reduced (by subtracting 1) until it is relatively prime to 26.

$$\begin{array}{r}
 B A Y I B M \\
 1 \ 0 \ 24 \ 8 \ 1 \ 12 \\
 24 \ 1 \ 7 \ 3 \ 24 \ 17 \\
 Y \ B \ H \ D \ Y \ R
 \end{array}$$

The second word, encoded, is now ybhdyr.

For the final stage of encryption, we manipulate the text on a word-by-word basis. To retain the original order of the words, we now use the encoded keyword as a kind of marker for each word. Every word will be surrounded by two letters from the keyword. This allows for a plaintext of up to 81 words, but more letters from the keyword could be used to surround the words if the plaintext is longer than 81 words. The first letter at the beginning of the word takes priority in ordering, and the other letters are used to increase the number of possible orderings of a word.

We choose the letters from the keyword thusly: for the first word in our text, we place the first letter of the keyword in front of it. We then place the last letter of the keyword at the end of the word. For the next word, we place the first letter of the keyword at the front of the word, but the second letter of the keyword as the last letter. The third word starts with the first letter and ends with the third letter, and so on, until the end of the keyword is reached. We repeat the same pattern until the keyword has been iterated through once. Then we use the second letter of the keyword at the front of the word and iterate through the keyword as the last letter again. We continue this pattern until all of the words have been labeled. This method could be extended for plaintexts greater than 81 words by adding two letters to the front and end of each word.

Ex. 1. A sequentially ordered set of surrounding letter groups would be as follows:
c-c, c-o, c-p, ... , c-t, o-c, o-o, o-p, ... , o-t, p-c, p-o, p-p, etc.

Ex. 2. If our “plaintext” is rbu ybhdyr, and our keyword is COPYRIGHT, rbu, as the first word in our coded message, becomes **crbuc** (bold added to make encryption steps clearer), and ybhdyr becomes **cybhdyro**.

<u>1</u> RBU <u>2</u>	*underlined numbers indicate positional ordering priority
C O P Y R I G H T	
1 2 3 4 5 6 7 8 9	*numbers below letters indicate letter ordering priority

Arranging alphabetically we get

crbuc cybhdyro

Which happens to be the same word order as in the “plaintext” because there were only two words.

We have now completely encrypted our plaintext. Each of our encryption examples uses less than 81 words. If a significantly longer plaintext needs to be encrypted, this part of the cipher could be extended to cover more words if two or three letters were used before and after each word, but such a strategy would have to be given in the key, which could be done by sending the number of letters before and after each word along with the keyword. Following the contract we received, 1 letter before and after should be plenty, so when there is no number given, we will assume it would be “1.”

To decrypt a message, simply go through the above steps in reverse. First, rearrange the words in order of the first and last letters (based on the keyword). Then decrypt the affine cipher for the first word using the first and last letters of the keyword for α and β . Use the decrypted first word to get α and β for the second word. Continue until the whole text has completed the affine cipher decryption. Then use the keyword to create the alphabet matrix that is the key for the substitution encryption. Use the matrix to solve the substitution encryption for the whole text.

We have a few more notes to present for consideration. First, it would be most secure if the keyword chosen was not a word at all, but rather a scrambled bunch of non-repeating letters. This prevents potential attackers from simply unscrambling the keyword. Second, our method can handle 81 words in plaintext, but if you wish to send longer messages, you can just add a third letter position, follow the same pattern, and notify the recipient of the system change. Third, it is rather painful to go through this entire process by hand for a long message. With that under consideration, we have attached a program file that can perform the encryption for you.

We hope that you are satisfied with this improved way to send your sensitive information over text. The combination of multiple encryption methods provides a secure encryption that is difficult for an attacker to break. Thank you for entrusting us with your safety and good luck encrypting.

Independent Math Contractors



Link to program:

https://drive.google.com/file/d/1fdyYrUlqtBCh06FAiOnLmx7_UvnZthxp/view?usp=sharing

Group K

Independent Mathematical Contractors, Inc.
136 TMCB
Provo UT, 84602
September 11, 2020

OCRA Creative Recursive Acronyms, Inc.
385 Primality Way
Provo, UT 84604

Dear Mr. Andrews,

Thank you for your interest in our team's cryptographic capabilities. We understand the serious implications that our contribution has for OCRAI and aim to effectively preserve the integrity of internal communications. As experts in cryptography, we have developed a unique cryptographic scheme that can be used to encode plaintext messages into ciphertext. This ciphertext is unreadable to a typical observer and difficult for a malignant attacker to decode. Decrypting the ciphertext requires knowledge of a special key and application of the appropriate decoding scheme. Without this key, encrypted messages cannot be understood. This encryption scheme is also rather efficient, requiring no more than five times the storage space of the original, unencrypted message. With the help of our method, OCRAI will be able to maintain safe and secure internal communications without worry for accidental or malicious leakage of confidential details.

Our encryption scheme begins with a key. This key is essential to encoding messages to be sent and decoding received messages. Since this system will be implemented in the phone app that OCRAI is developing, we can embed the key in the app itself. The advantage of this embedding is twofold. First of all, the key need not be known to anyone other than the developers who program it into the app; all other employees of OCRAI can enjoy use of our cryptographic method to send and receive encrypted messages without knowledge of the key. Secondly, this also means that the key does not have to be communicated over any channels beyond the internals of the app itself, which is a compiled software whose source code is automatically obfuscated.

The key is a special number that happens to be the product of exactly three prime numbers. According to the fundamental theorem of algebra, every positive integer greater than or equal to 2 is the product of a unique collection of prime numbers. This means that three pieces of information (one contained in each prime number) can be stored in a single key. There are two constraints on the allowed keys: first, the smallest prime factor of the key must be less than or equal to thirty-nine. Second, the next-to-smallest prime factor of the key must not be two or

thirteen. These are very light constraints that still allow a large number of possible keys. Some valid example keys are $322 = 2 \cdot 7 \cdot 23$, $2431 = 11 \cdot 13 \cdot 17$, and $10759 = 7 \cdot 29 \cdot 53$. Note that due to the non-triviality of factoring large numbers, it is recommended to use keys smaller than 100,000. This does not significantly limit the encryption capabilities of our system. Once the key has been determined, and its factors are known, we assign to the three factors the names a , b , and c . The numbers b and c denote two parameters in an affine cipher, which we will describe in the following paragraph.

An affine cipher gives a rule for how to transform plaintext messages into unreadable ciphertext. We begin by converting each letter in the alphabet the corresponding number of its order in a zero-indexed system (i.e. A = 0, B = 1, C = 2, ..., Z = 25). We call this number r . Once this has been done for each letter in the message, we multiply the number by b and then add c to the result, which we call x . After computing x , we take the equivalent number mod 26 (that is, take the remainder of x after dividing by 26):

$$x \equiv br + c \pmod{26}.$$

Once this has been done, we know that x is some number between 0 and 25, inclusive. A typical affine cipher would simply convert x back to the corresponding letter. We take a different approach to utilize the advantage of a longer ciphertext than plaintext. We first multiply x by a , giving us

$$y = ax.$$

After y has been computed, it is a number with between one and three digits. We can thus represent it with three letters, one corresponding to each digit. For example, if $y = 309$, we would add the string “DAJ” to the cipher. Separation between letters are represented by a single letter whose assigned number is greater than or equal to ten, i.e. and letter between K and Z, and actual spaces between words are represented by two of such letters. These letters can be randomly decided, which significantly increases the difficulty of decryption through typical methods such as frequency analysis. For example, if two two-letter words resulted in y values of 135 and 30 for the first word and 255 and 90 for the second word, the corresponding ciphertext may look like “BDFLDAXQCFFYJF”. Note that the letters “L”, “X”, “Q”, and “Y” in that ciphertext could be replaced by any other letters between K and Z.

Now, since each individual character in the plaintext becomes up to four characters (as y is made of three digits; the extra digit comes from the separation letter between individual characters). Thus for a 160-character message, the corresponding ciphertext will be no longer than 640 characters.

To decrypt the message, we need to know the key. Knowing the key, we factor it into its three prime factors, assigning to a , b , and c the values of the smallest, middle, and largest factors. We then go through the message and convert each letter to its corresponding number. Any sequence of two numbers greater than nine is converted to a space, and any single number greater than nine is discarded. We then divide each number by a to recover x . From here,

decoding the message is identical to decoding any affine cipher. We simply need to solve the equations

$$x_i \equiv br_i + c \pmod{26}$$

or

$$x_i - c \equiv br_i \pmod{26}$$

for each r_i , where i denotes the index of the letter. The fact that b is a prime number that is neither 2 or 13 guarantees that we can “divide” by b in these equations by multiplying each side by the number d for which $db \equiv 1 \pmod{26}$. This gives a solution

$$r_i \equiv d(x_i - c) \pmod{26}.$$

Since we know d , c , and x_i , this gives us a sequence of numbers between 0 and 26 that, when mapped back to the corresponding letter in the alphabet, reveals the original plaintext.

We conclude this by presenting a short example. We first select the key $10759 = 7 \cdot 29 \cdot 53$, so $a = 7$, $b = 29$, and $c = 53$. If our plaintext message is “HELLO THERE”, we first convert these characters to the number sequence $R = \{7, 4, 11, 11, 14, (), 19, 7, 4, 17, 4\}$. Each of these numbers represents a distinct r_i . We have noted the presence of a space with (). We then apply the affine transformation by computing $x_i \equiv br_i + c \pmod{26}$ for each of these numbers. This gives us a resulting sequence $X = \{22, 13, 8, 8, 17, (), 6, 22, 13, 0, 13\}$. Next, we multiply each number in this sequence by a to get the sequence $Y = \{154, 91, 56, 56, 119, (), 154, 91, 0, 91\}$. Finally, we replace each digit with its corresponding letter, insert a random letter from K to Z between each letter, and replace the space with two such letters. One possible resulting ciphertext would be “BFEPJBYFGMFGNBBJQWECVBFEYJBSASJB”. To decode this ciphertext, we remove all letters between K and Z, inserting a space if we find two in a row: “BFE/JB/FG/FG/BBJ EC/BFE/JB/A/JB”. We have inserted a slash between individual letters for clarity, but these will not be kept in the final decrypted message. Replacing each letter with its corresponding numbers and interpreting the result as the digits of up to 3 digit numbers, we recover the sequence Y above. We then divide each of these numbers by a to recover the sequence X . We now need to calculate d by finding the solution to $db \equiv 1 \pmod{26}$. This can be solved using Euclid’s algorithm. Once d has been found, we multiply each element in X by d to recover R . Changing R back to the corresponding letters and keeping note of the space, we recover the original message “HELLO THERE”.

The security of the internal communications of OCRAI can be effectively established and maintained through our solution. With our system, messages can be securely encrypted, sent, and decrypted in a way that electronic interception will yield a ciphertext that is impossible to decode without knowledge of the key. Having the key implemented in messaging software preserves its privacy, which in turn ensures the effectiveness of our scheme. Since the method is based on a simple affine cipher, it does not require a lot of computational power to encode or decode messages, yet its differences from the cipher ensure that the message cannot be decrypted

without knowledge of the key. We firmly believe that the cryptographic system that we have developed and explained will be of great benefit to the advancement of OCRAI.

Sincerely,

A solid black rectangular box used to redact a signature.

Co-founders

IMC

Group L

Dear OCRAI,

In response to the security breaches your company has faced, our team proposes the following three-step encryption process to enable your company to more securely send messages to one another. Our cryptosystem uses a randomly chosen 10-digit number to encode plaintext English into a ciphertext (you could also choose this key yourself, of course). The same key is then used to decode the ciphertext. We will provide examples along the way to illustrate how the cryptosystem might be used.

For our cryptosystem, we use the following alphabet:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, " " (space), "." (period), "," (comma), "'" (apostrophe), "\"" (quote), "?" (question mark), "!" (exclamation point)

We will use these three steps to encrypt the plaintext message:

- 1) Apply an affine cipher
- 2) Reorder the letters
- 3) Add in extra characters

Suppose we had the key 0413277130 and we wanted to encrypt the plaintext:

this is an example of how our cryptosystem works

We begin by performing an affine cipher. The last 4 digits of the key, 7130, tell us how we should use the affine cipher. The first two digits, 71, correspond to α and the next two digits, 30, correspond to β . If α is 0, 43 or 86, the affine cipher will send all characters to the same character. Use 1 for α in these cases. We work with modulo 43 because there are 43 characters in our alphabet. Then, we apply the equation

$$x \rightarrow \alpha x + \beta \pmod{43}$$

where x is the numerical representation of the character to be encoded. For example, the character "t" is represented by 29 because it is the 30th letter of our alphabet, with the first position represented by 0. Then,

$$29 \rightarrow 71(29) + 30 \pmod{43} \equiv 2089 \pmod{43} \equiv 25 \pmod{43}$$

The character corresponding to this number is "P", so we would replace all instances of "t" in our plaintext with "P". When we encode all of the letters of the example plaintext, we obtain

PXI"6I"69T6Z891!GZ6EK6XEN6EAC6MC !PE" "PZ16NECV"

We will refer to this text as ciphertext1.

Next, we will scramble the letters in ciphertext1. To do this, we look at the first two digits of the key, which are 04 in our case. We will call this our “jump number.” We will first number each character in the ciphertext1, starting at 0, as follows:

P	X	I	"	6	I	"	6	9	T	6	Z	8	9	1	!	G	...	V	"
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...	47	48

As we create the scrambled version of ciphertext1, we will begin by taking all of the characters in ciphertext1 that correspond to a number congruent to 0 ($\text{mod } \langle \text{jump number} \rangle$), starting with the smallest and working to the largest. For our example, we would start with all of the numbers congruent to 0 ($\text{mod } 4$), starting with the character corresponding to 0 and ending with the character corresponding to 48. Here is what we have for our ciphertext2 so far:

P698GKNC "ZE

Next, we will put all of the characters congruent to 1 ($\text{mod } \langle \text{the jump number} \rangle$), then keep going in that order until the last set of characters will be all of the characters congruent to ($\langle \text{the jump number} \rangle - 1$) $\text{mod } (\langle \text{the jump number} \rangle)$. So, in our example, the last block of text will be all of the characters with numbers that are congruent to 3 ($\text{mod } 4$). Here is what ciphertext2 will look like when we are done reordering all the characters:

P698GKNC "ZEXIT9Z666! 1CI"616XEMP"6V"6Z!EEACEPN"

You might notice that if the jump number is longer than the length of the message, the letters will not be scrambled. For this reason, we require that the jump number be less than (or equal to) the length of the message by using the $\langle \text{jump number} \rangle (\text{mod } \langle \text{message length} \rangle)$. We also require that the $\langle \text{jump number} \rangle (\text{mod } \langle \text{message length} \rangle)$ not be equal to 0 since ($\text{mod } 0$) is undefined. So if this happens, we use 37 instead.

In other words, if we have a jump number of 6, then we would write every 6th character as our new ciphertext, starting with the first character in the message. When we get to the end of the message, we re-start with the second character in the message and write every 6th character from there on, and so on until we have included every character in the original message. We don't want to jump over the entire message, so if the jump number in the key is longer than the message, divide the given jump number by how many characters are in the message and take the remainder to be the jump number instead. And jumping by 0 characters doesn't make any sense

(because jumping by 1 is already the same as the original order), so if the jump number in the key happened to be 0, we would use 37 instead.

The final part of the encryption process is to insert random characters, thereby creating noise for any attacker hoping to decode the message. We will use the third through sixth digits in the key to determine how many random characters to insert before each character in ciphertext2. In order to ensure that the final ciphertext length will not exceed 800 characters, we can add at most 4 characters before each character in ciphertext 2. So, it is possible that the number from the random number generator (assuming you are using one) will need to be adjusted to account for this by taking the number (*mod* 5). For our example, these digits are “1327”. Since 7 is not between 0 and 4, we would take the remainder from dividing 7 by 5 and use that number instead (this is the same as saying 7 (*mod* 5)). So in our case, our new number would be 2.

The first number in this section (the third number in the key) is the number of random characters to insert before the first character of our ciphertext. So in our example, we would begin by adding 1 random character in front of the first character in ciphertext2. The second number is the number of random characters to insert before the second character. The third number is the number of random characters to insert before the third character. The fourth number is the number of random characters to insert before the fourth character. Then we would start over at the first number to determine how many characters go before the fifth character. And so forth. Here is an example of what ciphertext3 might look like:

RPO586BZ9B08NG.P"KVYN.WC0 0D "H,Z!TEBXJB2I3?T139CZ
2OP6GR6DO67!SJ6 ZR1VECDIMOB"4965S1N6RW3XF5EIJMHP
AAN"5R6EBV""TEJ6X0ZXQ!1EEH!EQ4AZIC7EUCCPQBNN!"

This would be the ciphertext that would get sent by the sender.

To decrypt the message, we need to do the following:

- 1) Remove the random letters
- 2) Unscramble the encrypted letters
- 3) Apply the reverse affine cipher

First, we need to find and remove the extra letters inserted by the sender. Looking at the third through sixth numbers of the key, we can determine how many letters were inserted before each character. Again, if there is a number that is not between 0 and 4 (inclusive), we will need to divide by 5 and take the remainder. In our example, these numbers are “1327.” So, we will begin by removing 1 character from ciphertext3. The next character is part of ciphertext2. Next, we remove 3 characters. The next character is part of ciphertext2. Next, we remove 2 characters.

The next character is part of ciphertext2. Next we remove 2 characters (Since 7 divided by 5 has a remainder of 2). The next character is part of ciphertext2. We continue by repeating this pattern to remove all excess characters.

Next, we need to unscramble the letters. We will begin with a grid, where the number of columns is equal to the “modified jump number”, which is $\langle \text{the jump number given in the key} \rangle \pmod{\langle \text{the length of the message} \rangle}$, or 37 if that number is 0. The number of rows is equal to the length of ciphertext2 divided by the number of columns, rounded up to the nearest whole number. In our example, since the modified jump number is 4 and the length of ciphertext2 is 48, we would have 4 columns and 12 rows. We write the letters of ciphertext2 starting in the upper right hand corner, working down the column, then starting again at the top of the next column and so forth until we run out of letters. Using ciphertext2, which is

P698GKNC "ZEXIT9Z666! 1CI"616XEMP"6V"6Z!EEACEPN"

We would have the following:

P	X	I	"
6	I	"	6
9	T	6	Z
8	9	1	!
G	Z	6	E
K	6	X	E
N	6	E	A
C	6	M	C
	!	P	E
"		"	P
Z	1	6	N
E	C	V	"

Reading from the top right corner across the rows, starting at the next row after the end of each previous row, we have our ciphertext1, which is

PXI"6I"69T6Z891!GZ6EK6XEN6EAC6MC !PE" "PZ16NECV"

If the number of columns does not evenly divide the length of the ciphertext2, the last row will only have a number of cells equal to the remainder when dividing the length of ciphertext2 by the column number. To illustrate how this would work, suppose we had a ciphertext2 that was "TIEGHSSEIASSMA", which has a length of 14 and our jump number was 4. We would have 4 columns (our jump number) and 4 rows (since $14/4 = 3.5$, which rounds up to 4). Since $14/4$ has a remainder of 2, we will only have two cells in the last row. We would still write the letters, starting in the top left corner, working down the column, then go to the next column, as illustrated below:

T	H	I	S
I	S	A	M
E	S	S	A
G	E		

Reading across the rows, the decoded text would be "THISISAMESSAGE".

Finally, we will apply the reverse affine cipher. We will use the equation

$$x \rightarrow \alpha^{-1} \cdot (x - \beta) \pmod{43}$$

Where α^{-1} represents the number such that $\alpha \cdot \alpha^{-1} \equiv 1 \pmod{43}$. In our example, α is 71, and since $71 \cdot 20 \equiv 1 \pmod{43}$, 20 is our α^{-1} . Our β is 30. Then we apply the equation

$$x \rightarrow 20 \cdot (x - 30) \pmod{43}$$

to each character in ciphertext1, which will yield our plaintext message.

(Note: While we chose α in such a way so that α^{-1} will exist, the process of actually finding this number is nontrivial. While we do not go into detail here on how to find α^{-1} by hand, a guess and check method will work, as α^{-1} will always be between 1 and 42, inclusive.)

While this can all be done by hand, we understand that as busy employees and administrators, it may be too time-consuming to encrypt and decrypt all your messages personally, so we are happy to provide you with a computer program that will perform the needed changes for you and

that as an added bonus, generates random characters to add in as well as random encryption keys. This will certainly save you the effort of choosing your own.

We hope that our cryptosystem will be sufficient for your needs.

Yours sincerely,

A solid black rectangular box used to redact a signature.

Independent Math Contractors

Group M

Dear members of the OCRAI board committee:

At our company we hope to support other companies' business endeavors through providing encryption security for sensitive data at an affordable cost. We are happy that you have reached out to us on a review from another one of our clients, and we hope that you will be pleased with the encryption method included in this letter that we have created for your company. Our mission is to enable secure communication through a standard encryption process that relies on the secrecy of the encryption key more than the actual process of encryption, so anyone with malintent will not be able to decipher the message if it is intercepted even if they know the encryption process. Using these methods, we hope to remedy the problems that you brought to us. In your letter, you described that your company has recently had security breaches through company employees accidentally sending text messages to the wrong recipients. In doing so, sensitive company information was released and could have caused serious repercussions. We understand that these messages are meant only for other company employees and need to remain confidential, even in the event that they are sent to the wrong person. These messages need to still be compatible with the mode of transmission (text messages on mobile phones) even in their encrypted state. They also need to remain a reasonable length so it does not take a significantly larger percentage of characters to send the ciphertext. And, of course, they need to be easily decrypted by other company members who are the intended recipients of the text messages through a secure and unique process. We hope our proposal of an encryption system will satisfy all of these needs to your standards.

To clearly illustrate the process of encryption and decryption, we will provide an example at each step. As stated previously, we need a key with which to encrypt and subsequently decrypt the text of the message. It can be any word with twelve or fewer characters. For example, we can choose the word, "Jerusalem" for our key. We adjust all the letters to be lowercase, and with this key, we remove any duplicate letters. Our edited key then becomes "jerusalm". We also need to divide the plaintext message up into segments of 160 characters and apply the following process to each segment. Our example will have fewer than 160 characters, but it will be illustrative enough for our purposes.

We then use the edited key to create a cyclic map of letters and apply this to our plaintext message we want to encrypt. For example, let's say we want to encrypt the phrase, "Joshua has asked me to leave my possessions in the office." We adjust all these letters to be lowercase as well, and then we use the cyclic map to write any j's as e's, any e's as r's, any r's as u's, and so on until any m's are written as j's. After this first step, our altered plaintext (which we will refer to as cipher 1) now reads, "eoahsl hla lakrd jr to mrlvr jy poaaraaiona in thr offier."

We also apply this cyclic map of letters to our alphabet, which also includes spaces (' '), periods ('.'), and commas (',') for both mathematical and security purposes. After this transformation, our alphabet "abcdefghijklmnopqrstuvwxyz .," becomes, "lbcdrfghiekjmnopquatsvwxyz .," which we will refer to as our alphabet prime.

We next create an initial shift index by adding up all of the positions of cyclically rotated letters in our zero-based alphabet. For example, the letter 'j' was originally at index 9, the letter 'e' was originally at index 4, 'r' at 17, 'u' at 20, 's' at 18, 'a' at 0, 'l' at 11, and 'm' at 12. So, our

initial shift is $9 + 4 + 17 + 20 + 18 + 0 + 11 + 12 = 91$. We then take $91 \pmod{29} = 4$ as our initial shift index. The mod 29 is because our alphabet of 26 letters has 3 additional characters appended on to it.

We then take cipher 1 and apply the following operation to it to shift the characters and call the result cipher 2: add the index of the current character in cipher 1, the initial shift index calculated in the previous step, and the position of the current character in the alphabet prime. We then take this number (mod 29) and replace the character with the resulting character at that position in our alphabet prime. For example, in our cipher 1, the first letter is 'e'. So, we take the index of the current character 0, since we are at position 0 in cipher 1, the initial shift index 4, and the position of the current character in the alphabet prime 9, since 'e' is in the ninth position. We then add $0 + 4 + 9 = 13$. Then the first character of cipher 2 is 'n' since that is the letter at the 13th position in our alphabet prime. The second letter in cipher 1 is 'o'. So, we take the current index 1, the initial shift index 4, and the position of 'o' in alphabet prime 14, add them together to get 19, and get 't' as the second character of cipher 2. We continue this process on until we get cipher 2 to be "ntyoe,hajcmpf.wwur sokxedlwglqldww.,pbcwllgp.rajb,wmdridgb".

For the third and final step of the encryption, we parse the cipher 2 in such a way that we pass through and grab all the letters that are equivalent to zero (mod the length of the original key) in order and have that be the beginning of the next message, which will be referred to as cipher 3. We then pass through cipher 2 and grab all the letters that are equivalent to one (mod the length of the original key) and append it to the existing letters on cipher 3. We repeat this process up until we have grabbed all the letters from cipher 2. To best illustrate this concept, we will not only show what cipher 3 becomes, but also use colors to demonstrate the new placement of the letters. In our example, we first go through and cipher 2 and collect all the letters in the locations that are equivalent to 0 (mod 9) since "Jerusalem" has 9 characters. This would be: "nc gpri". We then do the same for those letters that are equivalent to 1 (mod 9), which would be: "tmslbad". Following this pattern, we would get cipher 3 to be "nc gpri tmslbadypocjgofklwbb,.xdl,ewewlwhwdwgmaul.pdjrw,.r". We now show the comparison of cipher 2 and cipher 3 using colors to indicate the position of the letters. Cipher 2: "ntyoe,hajcmpf.wwur_sokxedlwglqldww.,pbcwllgp.rajb,wmdridgb" becomes cipher 3: "nc_gpri tmslbadypocjgofklwbb,.xdl,ewewlwhwdwgmaul.pdjrw,.r".

If the original plaintext message was longer than 160 characters, we will have completed these steps for each of the blocks of text we parsed at the beginning. At this point the encryption is complete, and we send the blocks that are at most 160 characters each in the same order as the plaintext message would have been sent. Each of these steps is reversible given the key, and reversing them will yield the original plaintext message. You will want to decrypt them in blocks and not as a whole with the program.

While this encryption and decryption can be done by hand, we understand that our method can seem quite intricate. To that end, we have created a python code program that will complete the encryption and decryption for the employees. Because the key is the important part of the encryption process, the existence of the program is not a security vulnerability. We will provide the working program to you as a part of our service.

As a general overview of the process we just covered in the letter, given a plaintext message to be encoded, it will be split up into blocks, scrambled according to a key in a Caesar Cipher fashion, shifted by a scrambled alphabet and character location information in a way somewhat similar to an Affine Cipher but with slightly different technicalities, and then parsed into a pattern determined by the key. This multi-step encryption process should address the problem of confidential company information being received by unintended parties, because to any erroneous recipient, it will seem like gibberish. Regarding any party with malintent who is trying to intercept the messages, the messages will be secure as long as the malicious party does not obtain the key. While this was not stated in the original explanation of the problem in your letter to us, it is an added benefit that comes with encryption. The text messages will still be easily transmitted using mobile phones since they use the same characters as the original plaintext message being sent. The messages also remain the same length as the original plaintext, so overflow is not an issue as long as the plaintext messages are not that long themselves. Overall, we expect that this encryption will meet all of your needs and be straightforward enough to use, especially with the aid of the python program. If you have any questions whatsoever, please feel free to reach out to us. We would be more than happy to help you with any concerns.

Best regards,

A solid black rectangular box used to redact a signature.

Python Code:

```
from collections import OrderedDict

def remove_duplicate(str1):
    return "".join(OrderedDict.fromkeys(str1))

def split_text(plaintext, chunk_size):
    return [plaintext[i:i + chunk_size] for i in range(0, len(plaintext),
chunk_size)]

def encrypt(key_data, plaintext_data):
    # turn inputs into strings
    key = remove_duplicate(str(key_data).lower())
    plaintext = str(plaintext_data).lower()
    key_length = len(str(key_data))

    # create a cyclical pattern to cycle some letters through
    cycle = key + key[0]

    # Cycle through the key letters once
    cipher1 = ""
    for char in range(0, len(plaintext)):
        location = cycle.find(plaintext[char])
        if (location != -1):
            cipher1 += cycle[location + 1]
        else:
            cipher1 += plaintext[char]

    alphabet = "abcdefghijklmnopqrstuvwxyz ." # a string with 29
    characters, and 29 is prime
    # jumble the alphabet
    alphabet_prime = ""
    for char in range(0, len(alphabet)):
        location = cycle.find(alphabet[char])
        if (location != -1):
            alphabet_prime += cycle[location + 1]
        else:
            alphabet_prime += alphabet[char]
    cipher2 = ""

    start_position = 0
    for char in range(0, len(key)):
        start_position += alphabet_prime.find(key[char])

    for char in range(0, len(cipher1)):
        location = alphabet_prime.find(cipher1[char])
        cipher2 += alphabet_prime[(start_position + location + char) % 29]

    cipher3 = ""
    for i in range(0, key_length):
        cipher3 += cipher2[i:key_length]

    return cipher3
```



```

def decrypt(key_data, ciphertext_data):
    # turn inputs into strings
    key = remove_duplicate(str(key_data).lower())
    ciphertext = str(ciphertext_data).lower()
    key_length = len(str(key_data))

    # find out how many extra characters there are past a multiple of
    key_length
    bonus_chars = len(ciphertext) % key_length
    num_iters = len(ciphertext) // key_length

    plain1 = ""
    extra_cycle = 0
    if (bonus_chars > 0):
        extra_cycle = 1
    for i in range(0, num_iters + extra_cycle):
        character_number = i
        if (i != num_iters):
            plain1 += ciphertext[i]
            for j in range(1, key_length):
                character_number += num_iters
                if (j <= bonus_chars):
                    character_number += 1
            plain1 += ciphertext[character_number]
        else:
            for j in range(0, bonus_chars):
                character_number = i + j * (num_iters + 1)
                plain1 += ciphertext[character_number]

    # create a cyclical pattern to cycle some letters through
    cycle_prime = key + key[0]
    cycle_length = len(cycle_prime)
    # reverses the cycle for the decryption cycle
    cycle = cycle_prime[cycle_length::-1]

    alphabet = "abcdefghijklmnopqrstuvwxyz ." # a string with 29
    characters, and 29 is prime
    # jumble the alphabet
    alphabet_prime = ""
    for char in range(0, len(alphabet)):
        location = cycle_prime.find(alphabet[char])
        if (location == -1):
            alphabet_prime += alphabet[char]
        else:
            alphabet_prime += cycle_prime[location + 1]

    plain2 = ""

    start_position = 0
    for char in range(0, len(key)):
        start_position += alphabet_prime.find(key[char])

    for char in range(0, len(ciphertext)):
        location = alphabet_prime.find(plain1[char])
        position = (location - start_position - char) % 29

```

```
    plain2 += alphabet_prime[position]

plain3 = ""
for char in range(0, len(plain2)):
    location = cycle.find(plain2[char])
    if (location != -1):
        plain3 += cycle[location + 1]
    else:
        plain3 += plain2[char]

return plain3
```

```
encryption = encrypt("key goes here", "plaintext goes here")
print(encryption)

decryption = decrypt("key goes here", "ciphertext goes here")
print(decryption)
```

Introduction

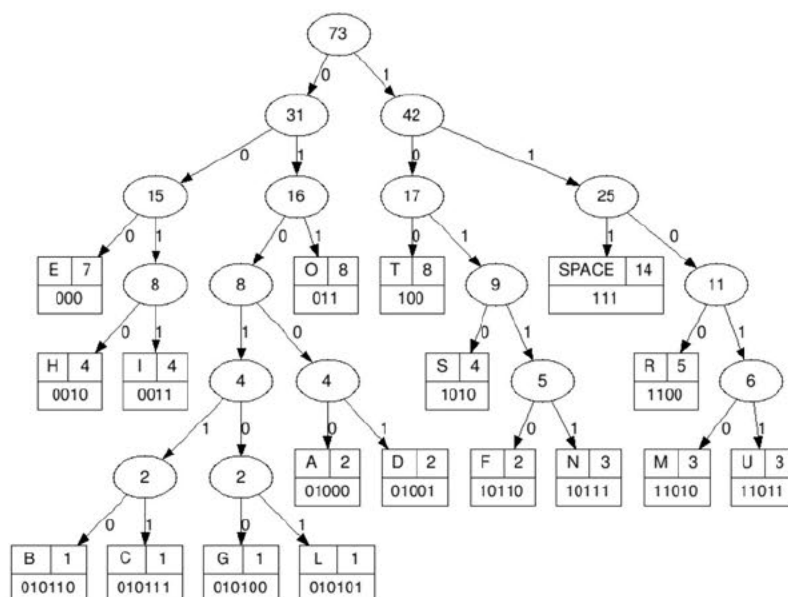
We in the [REDACTED] team are grateful for the opportunity to design a cryptosystem for OCRAI Creative Recursive Acronyms, Inc. We are glad to help you address the problem of security breaches. An ounce of prevention is truly worth a pound of cure, and we hope our cryptosystem will be a great help to you.

Our cipher system, modestly named the [REDACTED] Huffman encryption, will help address your problem by encoding messages in a way that makes it very difficult to crack without knowing the key, which will help your company effectively protect it's messages and private information.

Cipher System

The cipher system that we've prepared for you is based on a system known as Huffman Compression. Typically, a text message is stored as a series of seven-digit codes of 0's and 1's, known as ASCII. For example, the letter A is represented as 1000001, while the character "^" is represented as 1011110. Huffman Compression recognizes that some things, like letters, are used far more frequently than other things like dollar signs and asterisks. In Huffman Compression, more frequent characters in a text are represented with shorter codes, while infrequent characters are represented by shorter codes.

To create these shorter codes, Huffman Compression creates a frequency tree. To show an example of this kind of tree, we'll compress the phrase "OUR MEETING IS SCHEDULED FOR FOUR IN THE ROOM AT THE BOTTOM OF THE STAIRS". To start, we count the number



of times each character appears in this phrase. Some characters, like C and B, appear only once, while T appears 8 times and a space is used 14 times. Once we know the frequency all of the characters appear, we can combine low-frequency characters into higher frequency pairs. For example, while C and B only appear once each, there are two characters in the message that are either C or B, so they have a combined frequency of two. G and L appear once each, so they also have a combined frequency of two. C, B, G, and L

all then have a combined frequency of 4. We can combine lower frequency characters or groups

together until they've all been combined into a single tree, as shown in this image. In the case of ties for difference frequencies, the characters appear left to right in alphabetical order, as is the case with B, C, G, and L.

Once the frequency tree has been constructed, we can create new codes for each character by following the path to the character in the tree, with left branches being represented by 0's and right branches being represented by 1's. For example, the letter E has the code 000, because to find it we move left three times in the tree, while U has a code of 11011, because we move right twice, then left, then right two more times. We then replace all of the characters in our message with the codes; for example, "OUR" would be replaced with 011 11011 1100, as the codes for O, U, and R are 011, 11011, and 1100 respectively in this tree.

Once we have replaced the entire message with the codes from our frequency tree, we can turn the codes back into letters using the codes from this table for the final step of the

A	000 000	Q	010 000	g	100 000	w	110 000
B	000 001	R	010 001	h	100 001	x	110 001
C	000 010	S	010 010	i	100 010	y	110 010
D	000 011	T	010 011	j	100 011	z	110 011
E	000 100	U	010 100	k	100 100	0	110 100
F	000 101	V	010 101	l	100 101	1	110 101
G	000 110	W	010 110	m	100 110	2	110 110
H	000 111	X	010 111	n	100 111	3	110 111
I	001 000	Y	011 000	o	101 000	4	111 000
J	001 001	Z	011 001	p	101 001	5	111 001
K	001 010	a	011 010	q	101 010	6	111 010
L	001 011	b	011 011	r	101 011	7	111 011
M	001 100	c	011 100	s	101 100	8	111 100
N	001 101	d	011 101	t	101 101	9	111 101
O	001 110	e	011 110	u	101 110	."	111 110
P	001 111	f	011 111	v	101 111	""	111 111

cipher. The table gives each of the letters and numbers, as well as the space character, a six-digit code. We can use this table to break up our message in 6-digit codes, padding with 1's at the end so the total number of digits is divisible by six. So, for our combined code of 011110111100 for "OUR", the first 6 digits (011 110) would be replaced with the letter "e", as the code for "e" is 011 110. We would follow this pattern for the entirety of the message. As an example, the first 24 digits of our message after converting to 0's and 1's is the following: 011110 111100 111110 100000. When converting this using our table, we get "e80g". We would send this encoded message by text to the recipient via text, at which point the

recipient would be able to turn this encoded message back into the regular text.

In a regular Huffman Compression, the frequency tree is attached to the message, as there's no way to turn the text "011110111100" back into our original message of "OUR" without the frequency tree. In our encryption system, we will only send a list of character frequencies with the message instead of the tree, so that there's no way that someone intercepting the message can decipher the message. The recipient will have a key that tells them which frequencies belong to each character. For example, the message we send could begin with 4 5 3 2. If the recipient's key began with IRMD, it would signify that the frequencies of I, R, M, and D would be 4, 5, 3, and 2 respectively. Each person would have a unique key made of all of the characters a person could type shuffled together. Because this key wouldn't be sent with the message, anyone that intercepted the message would only see a series of numbers and the ciphertext, which no indication how to build the correct frequency tree.

Each key should be a randomized list of all of the valid characters that the people communicating want to be able to use. For example, the dollar sign can only be used if it is included in the key. For this reason, we recommend that all characters available on a standard keyboard be used in the key. However, because every character in the key requires more padding at the beginning of the message, the encryption system allows for smaller keys. One of the biggest advantages of having larger keys is the added security; most keyboards have 100+ characters, with the number of keys being equal to the number of characters raised to the power

of the number of characters. In the case of 100 characters, there are 100^{100} keys, or 10^{200} . In addition, because each pair of keys can include any characters, including emoticons, it is very difficult to break this code by guessing what characters are based on their frequencies, as no hacker can know what possible characters are included in the code.

Conclusion

The [REDACTED]-Huffman encryption system, which works by building a character frequency tree for each message and encoding the characters by their position on the tree, will help your company encrypt messages. For longer messages, because it is based on a compression algorithm, it may even have the benefit of compressing your data as well. The encryption will help secure company data and prevent accidental data leakage from employees. You can rest secure knowing that your data is protected and secure.

Notes: you may find the tool: <https://www.csfieldguide.org.nz/en/interactives/huffman-tree/> helpful in developing a Huffman tree from a given text string

Group O

Cryptographic System for OCRAI

11 September 2020



Introduction: The problem we are faced with is a serious one. Our business competitors have gained potential access to our company's classified information via mis-sent text messages. Our goal with this report is to explain the solution our team has come up with in order to gain control over these security breaches of the past and prevent them from reoccurring in the future. Human error can be a serious cause of security problems, and our goal is to mitigate that as much as possible. By creating a secure encryption system where company messages are sent, we will prevent the accidental data leaks that can happen when someone mistakenly sends information to someone it was not meant for.

Encrypting in our system: Our nontrivial encryption method encrypts each single letter to another letter, in sequence through the whole text. It uses the original letter as well as where that letter is positioned in the text. This yields extra strength over simpler methods that always translate a given letter to another given letter, no matter where the original letter appears in the text. As such, our system is more resistant to letter frequency analysis¹ attacks.

Our random key is two positive integers: call them **a** and **b**. The greatest common divisor between **a** and 27 must be 1.

¹ An example of frequency analysis is as follows: "E" is very common in English text, so an attacker looks at the most common symbol in the encrypted text and assumes it is "E"

The first step in the encryption process is to convert all the letters (or spaces) into integers, according to the following table. If a letter is lower-case, find its upper-case version in the table. Discard any characters from the input text that are not in the table.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	(space)
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

Then, each cipher letter's integer value c is computed with the formula

$$c = p * a + b + i \pmod{27}$$

where p is the original letter's integer value, a and b are from the key, and i is the position in the text (starting from 0 for the first letter and increasing by 1 for each letter.) Mod 27 means that the final value of c is actually the remainder after dividing by 27.

After all c values are computed, find the corresponding letters in the table above and write them in sequence. Note that some "letters" will be a space, for the value 26.

For reference, "hello" with the key $a=5$, $b=7$ encrypts to "PBKLA" (note there are no repeated letters, despite the repeated "L" in "hello".)

Here is a brief summary of the operations to encrypt "hello":

H	E	L	L	O	(change the letters to uppercase)
↓	↓	↓	↓	↓	(translate from the table)
7	4	11	11	14	
↓	↓	↓	↓	↓	(compute with the mod 27 formula)
15	1	10	11	0	
↓	↓	↓	↓	↓	(translate back to letters)
P	B	K	L	A	

If the resulting ciphertext is longer than the limit for a text message, split it into multiple messages in sequence. We assume that the cellular system will deliver the messages in order.

Decrypting in our system: If a message consists of multiple text messages, begin by combining the ciphertexts into one long ciphertext. Decryption proceeds similarly as encryption: convert all letters (or spaces) to the corresponding integers using the same table. Then compute each plaintext letter p with the formula

$$p = (c - b - i) * A \pmod{27}$$

where c is the encrypted letter's integer value, b is from the key, i is the index as before, and A is the multiplicative inverse, mod 27, of the key's a . We won't waste the board members' time with a full explanation of the multiplicative inverse, but a short way to put it is that if one calculates $a * A$ and then takes the remainder mod 27, the result will be 1 if the values are correct. The inverse can be calculated in SageMath with `inverse_mod(a, 27)`. For reference, for $a = 5$, $A = 11$. The "mod 27" part works as explained above. After all integer values have been calculated, convert them to letters (or spaces) as in the table above and put them together in sequence to obtain the plaintext message.

Conclusion: Our solution to the security breach issue is invaluable to the future of our informational security. By using the index of the plaintext letter for the encryption, we add another layer of protection against those who are attempting to gain access to our company's private information.

Dear OCRAI,

Thank you for your letter. We realize that, especially in these times of competition and of computing progression, the need for secure cryptosystems becomes increasingly important and needed. We commend your ability in identifying the vulnerabilities within your process. Mobile phone usage has long been a security vulnerability, and we hope that our proposed cryptosystem will satisfy your needs as a company.

In order to properly encrypt and secure the data, we have endeavored to find a cryptosystem and a corresponding algorithm in order to code the messages, per the company's request. The method we describe here has been constructed to be a combination of the Vigenère cipher and the Hill Cipher. We hope that the duality of this cipher will help prevent attempts to decode the messages by those who are not meant to read them, and we will outline the reasons why we believe it will be so.

To encrypt using our process, we must first obtain the first part of a key which is 4 to 6 letters long. We convert these letters to numbers (by assigning a=0, b=1, and so on until z=25). This algorithm will then roll through each letter of the message (omitting spaces) and shift the letter by that number of letters in the alphabet. An alternative way of calculating this, is assigning each letter of the plaintext to a number (using the system just discussed) and add the key's number to that letter. We use the following table in our example.

A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	2	3	4	5	6	7	8	9	10	11	12
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
13	14	15	16	17	18	19	20	21	22	23	24	25

For example, if the key is "OCRAI", then the corresponding shift factor is [14, 2, 17, 0, 8]. Let us have a plaintext that begins with "Four score and seven..." We now convert this sequence to their corresponding numbers, which gives us

5 14 20 17 18 2 14 17 4 0 13 3 18 4 21 4 13

We then add each key to each letter, adding the first letter of the key to the first letter of the plaintext, second to second, and so on, and once we reach the end of the key (in this case, after 5 letters), we begin again. We note that if our resulting number is greater than or equal to 26, we subtract 26 from it. We also note that if the message has an odd number of letters, we will add a random letter to the ciphertext at the end of the message, which is necessary for the next step in our cipher. So our resulting ciphertext will be based on (where R is a random number chosen to make the number of letters even).

19 16 1 17 0 16 16 8 4 8 1 5 9 4 3 18 15 R

We note that multiple times in our cipher, we had to subtract 26 in order to keep the answer between 0 and 25.

We now explain the second part of the 2-step cipher. We now manipulate the cipher using Linear Algebra to further encrypt our data. The second part of the key should be 4 letters long, which will correspond to 4 numbers. In our case, let our full key be OCRAIFIRD. These last 4 numbers will represent a matrix, which will look like

$$\begin{matrix} c & d & 5 & 8 \\ e & f & 17 & 3 \end{matrix}$$

We then take numbers from our previous ciphertext, two at a time, and multiply them by the matrix. So, given letters a and b, and numbers in our 2 x 2 matrix c, d, e, f, we can multiply them, and the resulting formula is

$$a*c+b*e \quad \text{and} \quad a*d+b*f$$

two numbers which will replace what we previously had in our ciphertext. Using our previous example and key, this will give us (using S in computations using R):

3 18 8 7 12 22 8 22 0 4 12 23 9 6 9 0 S S

Giving us the resulting ciphertext

dsgjmwiwaemxjgjaSS

We note that in our choice of a key, the first 4-6 letters are completely arbitrary, but the last 4 (for the matrix) must be a matrix that is invertible (modulus 26). The way that we can ensure that a matrix is invertible (mod 26) is seeing if $\gcd(26, (cf-de))=1$.

In order to decode, we must find the invertible matrix of the key, and follow the same process in reverse (multiplying two elements at a time to the inverted matrix and subtracting the first key amount).

We have performed a frequency analysis, and have discovered that in this process, the frequency of the letters is very similar, eliminating most possibility to decrypt using that method. Eliminating spaces also eliminates the ability to use word length to determine the message. The double layer of encryption adds an extra layer of security, making it difficult to test all of the combinations of the two keys. The Vigenère cipher has the weakness of being able to test all the combinations of cipher lengths, and from that be able to determine the key. The duality of this cipher covers that weakness. The Hill cipher alone has the vulnerability of a plaintext attack, and by having a piece of plaintext and the corresponding cipher, the matrix key is easily obtained. Again, this duality covers this vulnerability.

For your convenience, we have included a Python file that automates the process of encryption and decryption.

Sincerely,

[Redacted Signature]

Independent Mathematical Contractors, Inc.

```

#super_cipher.py
import sys
import re
import numpy as np

def apply_shift_cipher(digittext, key, shift_len):
    for i in range(len(digittext)):
        alpha = key[i % shift_len]
        digittext[i] = (digittext[i] + alpha) % 26
    return digittext

def multiply_pairs(digittext, matrix):
    output = []
    for i in range(len(digittext) // 2):
        pair = np.array([digittext[2*i], digittext[2*i+1]])
        output_pair = np.matmul(pair, mat) % 26
        output.extend(output_pair.tolist())
    return output

def digits_to_string(output_in_digits):
    output = ""
    for d in output_in_digits:
        output += chr(d + 97)
    return output

args = sys.argv
if len(args) != 5:
    print("Usage: python super-cipher.py <encode/decode> <key> <input-filepath> <output-filepath>")
    exit()
key = args[2]
# Note, the first part of the key can be the same for decoding and encoding, but the last
# 4 digits must be the inverse matrix mod 26 for decoding
in_filepath = args[3]
out_filepath = args[4]

input_text = ""
with open(in_filepath) as f:
    input_text = f.read()
# Only keep letters, and make everything lowercase
input_text = re.sub(r"[^a-zA-Z]", "", input_text).lower()

# Turn letters into list of digits 0-26
digittext = [ord(c) - 97 for c in input_text]

# Make key and matrix nice formats
key = [ord(c) - 97 for c in key]
shift_len = len(key) - 4
mat = np.array([[key[shift_len], key[shift_len + 1]], [key[shift_len + 2], key[shift_len + 3]]])

```

```

# Pad message if odd number of characters
if len(digittext) % 2 == 1:
    digittext += [np.random.randint(0, 26)]

if args[1] == "encode":
    shift_output = apply_shift_cipher(digittext, key, shift_len)
    output_in_digits = multiply_pairs(shift_output, mat)
else:
    matmul_output = multiply_pairs(digittext, mat)
    output_in_digits = apply_shift_cipher(matmul_output, [-k for k in key], shift_len)
output = digits_to_string(output_in_digits)
print(output)
with open(out_filepath, "w+") as f:
    f.write(output)

#char_freq = {}
#for c in output:
#    char_freq[c] = char_freq.get(c, 0) + 1
#print(char_freq)

```

Group Q

MATH 485 - CRYPTO PROJECT

1. INTRODUCTION

We are glad to inform you that Independent Math Contractors, Inc has completed the project as described in your previous letter. We have created an entirely new cipher for your exclusive use, to prevent any further embarrassment caused by careless employees. Rest assured we have had our top cryptological math experts work tirelessly on this cryptosystem since receiving your message. The system described herein will encumber any third party's attempt to understand what your internal communication is without the key. Below, we will give an overview of how the system works, and provide a short example of how to encrypt a message.

2. THE CIPHER

2.1. Theory. Consider the function $f_n : \mathbb{Z}_{26} \rightarrow \mathbb{Z}_{26}$ defined by $f_n(x) = x^n$. Since f_n is defined on \mathbb{Z}_{26} , we note that we are working (mod 26) in all computations. To obtain a decryptable cipher, we wish to choose values for n that give a bijective function. This will give us a unique mapping from the English alphabet to itself, allowing us to reverse the encryption process. After numerical experimentation, the only values n for which this function is bijective on \mathbb{Z}_{26} are $n = 1, 5, 7, 11$. For $n \in \{1, 5, 7, 11\}$ we see that the values for f are given by

alphabet	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
$f_1(x) = x^1 \pmod{26}$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
$f_5(x) = x^5 \pmod{26}$	0	1	6	9	10	5	2	11	8	3	4	7	12	13	14	19	22	23	18	15	24	21	16	17	20	25
$f_7(x) = x^7 \pmod{26}$	0	1	24	3	4	21	20	19	18	9	10	15	12	13	14	11	16	17	8	7	6	5	22	23	2	25
$f_{11}(x) = x^{11} \pmod{26}$	0	1	20	9	10	21	24	15	18	3	4	19	12	13	14	7	22	23	8	11	2	5	16	17	6	25

We notice that these functions perform a series of transpositions (swapping two elements with each other) that are distinct (we don't swap 2 with 3 and then 3 with 6). For example, note that $f_5(2) = 6$ and $f_5(6) = 2$. So, this gives that $f_n^2(x) = (f_n \circ f_n)(x) = f_n(f_n(x)) = x$ for $n \in \{1, 5, 7, 11\}$. Hence $f_n = f_n^{-1}$. We also note that if we add a constant shift to these functions to get $g(x) = f_n(x) + s$ for some $s \in \mathbb{Z}_{26}$, then the inverse function is given by $g^{-1}(x) = f_n(x - s)$ since

$$(g \circ g^{-1})(x) = g(g^{-1}(x)) = g(f_n(x - s)) = f_n(f_n(x - s)) + s = x - s + s = x$$

and

$$(g^{-1} \circ g)(x) = g^{-1}(g(x)) = g^{-1}(f_n(x) + s) = f_n(f_n(x) + s - s) = f_n^2(x) = x$$

2.2. Key. Our cryptosystem requires a key to be used for encryption and decryption. The basic structure of the key is a tuple of a permutation of $\{1, 2, 3, 4\}$ and a number in the interval $[1, 25]$, concatenated together. So, the key is 5-6 characters long, where the first four digits are a permutation of $\{1, 2, 3, 4\}$ and the last 1-2 digits are an integer in the interval $[1, 25]$. 14321, 34222, 123413 are all valid keys.

This key is used to define the order of which encryption/decryption functions to use and an offset.

Suppose that the key has the form $\text{key} = d_1 d_2 d_3 d_4 d_5 d_6$, where $d_1, d_2, d_3, d_4 \in [1, 4]$, $d_5 \in [1, 9]$, and $d_6 \in [0, 5]$ are integers. Let s be the integer formed by concatenating the digits d_5 and d_6 together. Note that d_6 is optional and need not be included in the key. Then we choose our encryption functions to be an ordering of the functions $g_n(x) = f_n(x) + s$ with corresponding decryption functions $g_n^{-1}(x) = f_n(x - s)$ depending on the values of d_k for $k \in \{1, 2, 3, 4\}$. This ordering is explained more in section 2.4.

Since there are 4 ways to pick each of d_1, d_2, d_3, d_4 and 25 ways to pick the pair $d_5 d_6$, we have that there are $4 \cdot 4 \cdot 4 \cdot 4 \cdot 25 = 4^4 \cdot 25 = 6400$ possible keys.

2.3. Inputs and Alphabet. Once we have obtained our encryption/decryption functions using the key, we can encrypt each letter of the plaintext. Note that the only acceptable inputs to our cipher are a letter in the alphabet, without case. Spaces or other punctuation are skipped by this algorithm; they are not present in the output.

We are using a zero-based representation of the alphabet, meaning 0 corresponds to **a**, and 25 corresponds to **z**. This maps the alphabet into \mathbb{Z}_{26} .

2.4. Transformation. The basic equations used to encrypt are:

$$(1) \quad f_n(x) = x^n$$

$$(2) \quad g_n(x) = f_n(x) + s$$

Where x is the number to encrypt (according to the alphabet defined above) and s is the number at the end of the key (s described in section 2.1), and n is one of 1, 5, 7, or 11.

We determine the order of which $g_n(x)$ to use to encrypt a plaintext character in a cyclical pattern given by the first four characters of the key. The first four characters of the key define a simple mapping to which $g_n(x)$ to use.

- 1 in the key maps to $g_1(x)$
- 2 in the key maps to $g_5(x)$
- 3 in the key maps to $g_7(x)$
- 4 in the key maps to $g_{11}(x)$

In other words, if the first few characters of plaintext to encrypt are $c_1, c_2, c_3, c_4, c_5, c_6, \dots$, and the key is 423111, the pattern of which $g_n(x)$ to use is $g_{11}(c_1), g_5(c_2), g_7(c_3), g_1(c_4), g_{11}(c_5), g_5(c_6), g_7(c_7), g_1(c_8)$ repeating until the end of the message.

2.5. Example. As an example, suppose that we wish to encrypt the plaintext **CRYPTO** and we choose the key 132417.

For readability, we use the plaintext letters as input to the functions g_n rather than their representatives in \mathbb{Z}_{26} . Because of the pattern in the key, we use g_1 to encrypt the first plaintext letter, g_7 to encrypt the second plaintext letter, and so forth.

$$g_1(\text{C}) = f_1(\text{C}) + 17 = 19 = \text{T}$$

$$g_7(\text{R}) = f_7(\text{R}) + 17 = 8 = \text{I}$$

$$g_5(\text{Y}) = f_5(\text{Y}) + 17 = 11 = \text{L}$$

$$g_{11}(\text{P}) = f_{11}(\text{P}) + 17 = 24 = \text{Y}$$

$$g_1(\text{T}) = f_1(\text{T}) + 17 = 10 = \text{K}$$

$$g_7(\text{O}) = f_7(\text{O}) + 17 = 5 = \text{F}$$

Hence the final encrypted message is **TILYKF**. If we were to decrypt this same message, all that is needed are the decryption functions given by $g_1^{-1}(x) = f_1(x - 17), g_7^{-1}(x) = f_7(x - 17), g_5^{-1}(x) = f_5(x - 17)$, and $g_{11}^{-1}(x) = f_{11}(x - 17)$. Using g_1^{-1} to decrypt the first ciphertext letter, g_7^{-1} to decrypt the second ciphertext letter, and so forth, we see that the plaintext letters are

$$g_1^{-1}(\text{T}) = f_1(\text{T} - 17) = 2 = \text{C}$$

$$g_7^{-1}(\text{I}) = f_7(\text{I} - 17) = 17 = \text{R}$$

$$g_5^{-1}(\text{L}) = f_5(\text{L} - 17) = 24 = \text{Y}$$

$$g_{11}^{-1}(\text{Y}) = f_{11}(\text{Y} - 17) = 15 = \text{P}$$

$$g_1^{-1}(\text{K}) = f_1(\text{K} - 17) = 19 = \text{T}$$

$$g_7^{-1}(\text{F}) = f_7(\text{F} - 17) = 14 = \text{O}$$

Thus the decrypted message is the original plaintext we started with, namely **CRYPTO**.

3. CONCLUSION

In this report we have demonstrated a cipher that will allow OCRA Creative Recursive Acronyms, Inc to add a layer of privacy to their employee's messages, hampering and inadvertent leaks of confidential proprietary information. The cipher accepts a key of length 5 or 6 characters and a message of any length, and uses modular exponentiation to hide the plaintext. The keys and output used by this cipher have an easy-to-read representation that can be typed on any keyboard.

As demonstrated, this cipher will obfuscate any misplaced sensitive company information. Any third party who sees sensitive company information or trade secrets in a leaked employee's message will need to know the cipher key used to encrypt it. This will reduce the chance of any more embarrassment to the company or harm to your future revenue streams.

Group R

You have reached out to us to create an encryption method to transmit sensitive information while maintaining a relatively small size for encrypted text. Additionally, the text must also only use characters available on standard keyboards. We have devised a system that is both difficult for third parties to break and results in a very minimal increase in size from plain text to cipher text.

The basis of the system is matrix multiplication. The message is first transformed into a matrix, then (left) multiplied with an invertible matrix given by an encryption key. To decrypt the message, the matrix is (left) multiplied by the inverse of the matrix given by the encryption key.

We start by generating a square matrix based on the contents of the message [1]. We define the size as being the square root of the character length of the message rounded up to the nearest whole number. We extend the length of the message to match the length of the created matrix, if necessary, by appending tilde (~) characters to the end as padding. From here, we fill in the matrix right to left, top to bottom, with the ASCII value of each character in the message.

We then need to create an invertible matrix that we will use to encrypt the message matrix [2]. This matrix must be invertible so that we can undo its multiplication when decrypting the message.

We know by the Invertible Matrix theorem that a matrix is invertible if it is row equivalent to the identity matrix. This means that we can start with the identity matrix and modify it with multiple successive elementary row operations while preserving its invertibility.

In order to derive this matrix from the key, we first iterate over each row n of the identity matrix and multiply it by the n th digit in the key. We wrap around to the beginning of the key if we reach row numbers greater than the length of the key.

For further modification, we then perform a series of pivot operations, which involve adding a multiple of one row to another row. We grab pairs of digits from the key left to right, and for each pair, we generate three numbers: a starting row number, a destination row number, and a multiplier. We multiply the row referenced by the starting row number by the multiplier, and add the result to the destination row.

We then take the message matrix and multiply it with the key matrix to produce the encrypted matrix. The encrypted matrix is converted to a text format by placing a hex representation of each digit in the matrix followed by a space separator, with the entries written left to right, top to bottom [3].

We now have our ciphertext. Decryption is done by performing the above process in reverse: put the received message into a matrix, multiply by the inverse of the key matrix generated from the key, and read out the ASCII values of the resulting matrix right to left, top to bottom.

Code Appendix

[1] Generate matrix from text

```

def text_to_matrix(plain_text: str) -> np.ndarray:
    side_length = math.ceil(math.sqrt(len(plain_text)))
    text_matrix = np.zeros((side_length, side_length), dtype=np.uint32)
    # make the text have as many symbols as the array has elements
    padded_text = plain_text + ''.join(['~' for _ in range(side_length ** 2 - len(plain_text))])

    text_index = 0
    for i in range(side_length):
        for j in range(side_length):
            text_matrix[i][j] = ord(padded_text[text_index])
            text_index += 1

    return text_matrix

```

[2] Generate matrix from key

```

def key_to_matrix(key: str, n: int) -> np.ndarray:
    """
    Creates an invertible matrix from the provided string.
    """
    key_matrix = np.identity(n, dtype=np.uint32)

    new_key = ''
    for i in range(n):
        new_key += key[i % len(key)]

    for i, key_element in enumerate(new_key):
        key_matrix[i] *= ord(key_element) % 32

    for i in range(0, len(new_key)-1, 2):
        print(new_key[i:i+2])
        row1, row2, x = substring_to_numbers(new_key[i:i+2], n)
        key_matrix[row1] += (x % 32) * key_matrix[row2]

    return key_matrix

```

```

def substring_to_numbers(sub_str: str, n: int):
    """
    Substring should contain two characters
    :return: Two numbers in range(n)
    """
    num1 = ord(sub_str[0]) % n
    num2 = (5 * ord(sub_str[1])) % n
    num3 = (ord(sub_str[0]) + ord(sub_str[1])) % n
    print((num1, num2, num3))
    return num1, num2, num3

```

[3] Convert encrypted matrix to text

```

def matrix_to_text(matrix: np.ndarray) -> str:
    text = ''
    for row in matrix:
        for num in row:
            text += num_to_str(num)
    return text

def num_to_str(num: int) -> str:
    return hex(num)[2:] + ' '

```

MATH 485 Section 1

9 September 2020

OCRAI Message Encryption Report

Given the constraints for your secure messaging system, we have determined that the best method of encrypting and decrypting messages between employees is an approach similar to what is known as a *One-time pad*. The approach is a key-based algorithm that effectively obfuscates the message and any patterns that may allow an attacker to decipher the message.

The encryption algorithm takes a *plaintext* message and uses a predetermined or calculated *key*, shared between both the sender and receiver, to encrypt the *plaintext* into *ciphertext*, then decrypt it back into *plaintext* for the receiver. To begin, the algorithm takes the *plaintext* message and removes any spaces or punctuation, and for simplicity, capitalizes all characters. For example, the *plaintext* message, “*Hello, World!*”, is converted into the *plaintext* message, “*HELLOWORLD.*”

Hello, World! → *HELLOWORLD*
Initial Message Modified Message

This prevents an attacker from being able to make any guesses or inferences from the *ciphertext* based on patterns in the language of the message.

With the *plaintext* message refactored, the algorithm then uses the predetermined *key* to encrypt the updated *plaintext*. The algorithm accomplishes this by shifting each character in the *plaintext* by the number of characters determined by the corresponding character in the *key*.

Before we can make sense of this, we give each character of the alphabet a number:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

These numbers are used to determine how many shifts will be performed, so $A = 0$ shifts, $B = 1$ shift, $C = 2$ shifts, etc. We define a “shift” as an increment in the alphabetic value of the character. For example, the letter A shifted three times becomes the letter D . If the shift goes passed the letter Z , we loop back around to A and continue the shifting. So, the letter Y shifted three times becomes the letter B .

We then use the *key* to determine how many characters to shift each character of the *plaintext*. Using the example *key*, “*PASSWORD*”, we determine that we must shift the first character of the *plaintext* 15 times (since $P=15$), the second character 0 times ($A=0$), the third character 18 times ($S=18$), and so forth. Because the messages are typically longer than the *key*, we start back at the beginning of the *key* when the end is reached and there is more *plaintext* to encrypt. We can visualize this by lining up the *key* with the *plaintext* as follows:

Plaintext:	<i>H</i>	<i>E</i>	<i>L</i>	<i>L</i>	<i>O</i>	<i>W</i>	<i>O</i>	<i>R</i>	<i>L</i>	<i>D</i>
Key:	<i>P</i>	<i>A</i>	<i>S</i>	<i>S</i>	<i>W</i>	<i>O</i>	<i>R</i>	<i>D</i>	<i>P</i>	<i>A</i>

Now we can see how the *key* is used to determine how many characters to shift the *plaintext* to give us the *ciphertext*:

Plaintext:	<i>H</i>	<i>E</i>	<i>L</i>	<i>L</i>	<i>O</i>	<i>W</i>	<i>O</i>	<i>R</i>	<i>L</i>	<i>D</i>
# of shifts:	+15	+0	+18	+18	+22	+14	+17	+3	+15	+0
=Ciphertext:	<i>W</i>	<i>E</i>	<i>D</i>	<i>D</i>	<i>K</i>	<i>K</i>	<i>F</i>	<i>U</i>	<i>A</i>	<i>D</i>

So, the *plaintext* message “*Hello, World!*” encrypted with the *key* “*PASSWORD*” yields the *ciphertext* “*WEDDKKFUAD*.”

This process can be simplified mathematically by thinking of the *plaintext* and *key* in terms of their numbers:

$$c_i \equiv p_i + k_j \pmod{26}$$

where p_i is the numerical value of the i^{th} character in the *plaintext*, k_j is the numerical value of the j^{th} character in the *key*, and c_i is the numerical value of the i^{th} character in the *ciphertext*.

Decrypting the *ciphertext* is very straightforward. Using the shared *key*, the reverse of encryption is applied to the *ciphertext*. The *key* is aligned with the *ciphertext* and then the characters are shifted back based on the *key* values:

Ciphertext:	<i>W</i>	<i>E</i>	<i>D</i>	<i>D</i>	<i>K</i>	<i>K</i>	<i>F</i>	<i>U</i>	<i>A</i>	<i>D</i>
Key:	<i>P</i>	<i>A</i>	<i>S</i>	<i>S</i>	<i>W</i>	<i>O</i>	<i>R</i>	<i>D</i>	<i>P</i>	<i>A</i>
# of shifts:	-15	-0	-18	-18	-22	-14	-17	-3	-15	-0
=Plaintext:	<i>H</i>	<i>E</i>	<i>L</i>	<i>L</i>	<i>O</i>	<i>W</i>	<i>O</i>	<i>R</i>	<i>L</i>	<i>D</i>

Mathematically, this can be described using the same symbols as before with the following equation:

$$p_i \equiv c_i - k_j \pmod{26}$$

There are a few other security factors to consider, such as how the *key* will be shared between users and the security of that *key*, but from an encryption standpoint and the requirements provided, this encryption algorithm will provide the security the company needs to safely transmit messages.

Progressive Key Cipher Encryption

Introduction

Information governs the world around us; it is used in marketing, strategy planning, product development, business management, job hiring, and even restaurant recommendations. This in turn makes information-based security breaches a major threat to individuals and organizations alike. Our goal for this project is to minimize the damage dealt to any party in the event of a zero-day attack or an unintentional leak of information.

How Our System Works

We call our cryptosystem the “Progressive Key Cipher”. We start with the key $K = K_1K_2K_3 \cdots K_n$, which is an alphabetic string of length $2 \leq n \leq 10$. Let P be a string of alphabetic plaintext. For our purposes we will ignore non-alphabetic characters—we encrypt the plaintext as if they were not there. Define the bijective mapping μ from the English alphabet to the integers modulo 26 by assigning each letter to the equivalence class of the number representing its position in alphabetical order (e.g. $\mu(A) = 0, \mu(B) = 1, \mu(C) = 2, \dots, \mu(Z) = 25$). We now outline the procedure by which we encode the plaintext using the key:

1. Break P into chunks of characters $P_1, P_2, P_3, \dots, P_k$ of length $n - 1$. It is okay if the last chunk does not have $n - 1$ characters.
2. Apply the key to the first chunk $P_1 = C_1C_2C_3 \cdots C_{n-1}$ by mapping each character C_i to the character $\mu^{-1}(\mu(C_i) + \mu(K_i))$. So we shift the i 'th character of the chunk by the i 'th letter of the key.
3. Now that we have applied the key to the first chunk, we use the n 'th character of the key to alter the key for the second chunk. To do this, we add the position of the final character in the key to the position of each character in the key (including the final character) and reduce modulo 26. In other words, C_i becomes $\mu^{-1}(\mu(K_i) + \mu(K_n))$. Set K equal to the new key.
4. Apply K to the next chunk of plaintext as in step 2, and afterward generate the new key using the process described in step 3.
5. Repeat step 4 until you have encrypted all of the chunks.

Let E be a string of Ciphertext and let K be the original key. Decryption is essentially the above process, but reversed:

1. Break E into chunks of $n - 1$ characters as before.

2. Decrypt the first chunk $E_1 = C_1C_2C_3 \cdots C_{n-1}$ by subtracting the corresponding K_i alphabetic position from the C_i alphabetic position. In other words, plaintext C_i will be equal to $\mu^{-1}(\mu(C_i) - \mu(K_i))$.
3. Generate the key for the next chunk exactly as in step 3 of the encryption process, and decrypt the next chunk using the new key following the process in step 2.
4. Repeat until all chunks are decrypted.

A Worked Example

To demonstrate our cryptosystem, we will work through a simple example. Suppose our key text is "CODE", and our plaintext is "CRYPTO FUN". We begin by breaking the plaintext into chunks of 3 letters. We have "CRY-PTO-FUN". We start with the first chunk, "CRY". Adding the position values of corresponding letters in the key "COD", we obtain cyphertext "EFB" for our first chunk. The following table outlines these computations in greater detail.

Plaintext Character	C	R	Y
Position in Alphabet (0-25)	2	17	24
Key Character	C	O	D
Position in Alphabet (0-25)	2	14	3
New Position in Alphabet (0-25)	$2 + 2 = 4 \text{ (mod 26)}$	$17 + 14 = 5 \text{ (mod 26)}$	$24 + 3 = 1 \text{ (mod 26)}$
Cyphertext Character	E	F	B

Now we change the key. To change the key, we look at the last letter of the key, "E", which has position 4 in the alphabet. We then add this to the position of the first 3 letters to obtain our new, shifted key, "GSHI".

Key Character	C	O	D	E
Position in Alphabet (0-25)	2	14	3	4
New Position in Alphabet (0-25)	$2 + 4 = 6 \text{ (mod 26)}$	$14 + 4 = 18 \text{ (mod 26)}$	$3 + 4 = 7 \text{ (mod 26)}$	$4 + 4 = 8 \text{ (mod 26)}$
New Key Character	G	S	H	I

Now we apply the new key to the next chunk, "PTO". Adding the corresponding alphabetic positions of "GSH", we obtain the encrypted chunk "VLV".

Plaintext Character	P	T	O
Position in Alphabet (0-25)	15	19	14

Key Character	G	S	H
Position in Alphabet (0-25)	6	18	7
New Position in Alphabet (0-25)	$15 + 6 = \mathbf{21} \pmod{26}$	$19 + 18 = \mathbf{11} \pmod{26}$	$14 + 7 = \mathbf{21} \pmod{26}$
Cyphertext Character	V	L	V

To make the key for the next chunk, we add the position of the last letter ("I") of the new key, "GSHI", to the position of all the letters in the key.

Key Character	G	S	H	I
Position in Alphabet (0-25)	6	18	7	8
New Position in Alphabet (0-25)	$6 + 8 = \mathbf{14} \pmod{26}$	$18 + 8 = \mathbf{0} \pmod{26}$	$7 + 8 = \mathbf{15} \pmod{26}$	$8 + 8 = \mathbf{16} \pmod{26}$
New Key Character	O	A	P	Q

Then we apply this key, "OPAQ", to the next chunk, "FUN". That gives us "TUV". Hence, our encrypted ciphertext is "EFBVLV TUV". To decrypt this text, simply reverse the operations shown in the table.

Conclusion

Our cryptosystem aims to minimize the risk of a security breach or leakage by preventing an unauthorized agent from extracting important information quickly. The pattern is difficult to detect, and is resistant to low-level frequency analysis attacks. Unless a more sophisticated attack is employed, an unauthorized agent will not be able to break the cipher in a short amount of time. Changing the key frequently will further ensure the confidentiality of the encrypted data. Thus, we can confidently say that this achieves our goal for this cryptosystem—to minimize the damage dealt to any agent at the event of a zero-day attack or an unintentional leak of information. It is imperative to recognize the danger of putting the safeguard of information solely on one system, so we strongly recommend the use of other security measures in addition to our cryptosystem.

Professor Jenkins

MATH 485

10 September 2020

Encryption Report

Dear OCRAI, we understand the need for an encryption system for your company. We know that keeping private company information is very important in today's day and age of getting ahead of the competition. Especially when this privacy is reliant upon the employees, things can be easily leaked to the public and rival business. We have put together an encryption method that you can apply to your company's mobile application to use for peer-to-peer encryption with your employees. This system allows for punctuation and special symbols and preserves the case of each letter.

Our cryptosystem requires a private key composed of any letters, numbers, or special symbols available on a standard computer or mobile phone. Per your specifications, the key does not need to be of great length. Although you can technically use as little as one character, we recommend 6-10 characters for maximum security. This key can be set by an administrator as often as desired; setting a randomized key daily will increase encryption security. There will be no need to memorize this key as it is only being used by the encryption method for encrypting messages sent that day. The keysetter only needs to provide a novel key daily and the cryptosystem will handle the rest.

The system works by first converting the characters of the plaintext to their ASCII Decimal representations. For the curious, these can be found at <https://www.ascii-code.com/>. The column marked "DEC" contains the decimal for the character in the corresponding row. We

leave the first and last characters of the message alone, but then iterate through the plaintext message (starting at the second character) and the pass key at the same time. The decimal value of the first character in the key is either added to or subtracted from the decimal value of the second letter in the message, mod 127. If the decimal value of the first character of the message is even, the values are added. If it is odd, the value is subtracted. This is why the first character of the message is left unchanged by encryption. Continuing with this idea, the decimal value of the second character in the key is added to the third character of the message, mod 127, and so on. The pattern continues like this, starting again with the first character of the key when we reach the end of the pass key. We now multiply the resulting decimal value by the value given by the last character of the message, mod 127. This is why this last character of the plaintext is also left untouched by the encryption. These values are needed again for decryption. One issue with using ASCII decimals is that the values from 0-32 correspond to unprintable characters. To remedy this, if the final values fall in that range, our system adds 200 to them to bring them into the range of printable characters while still maintaining a one-to-one relationship from plaintext to ciphertext. These final decimal values are then converted back into characters to complete the encryption process.

We chose to perform calculations in mod 127 because 127 is a prime number, allowing us to multiply these decimal values by any factor mod 127. Also, all upper and lower case letters, numbers, and punctuation marks commonly used fall below 127.

To represent our algorithm mathematically, we collect an ASCII number, A , from the private key, and an ASCII number, B , from a character, X , in the plaintext. We call the ASCII numbers of the first and last characters of the message C and D respectively. Furthermore, we

define a function *parity(x)* that returns a 1 if *x* is even, and a -1 if *x* is odd. Then we illustrate our encryption as follows:

$$\text{encrypted}(X) = (B + \text{parity}(C) * A) * D \pmod{127} = Y$$

$$\text{If } Y < 33: Y = Y + 200.$$

Then *Y* is converted back to a character.

With all of this in mind, decryption can be performed quickly as long as you know the private key. If we consider an encrypted character with ASCII number, *Y*, we perform decryption with the following:

$$\text{If } Y > 200: Y = Y - 200$$

$$\text{decrypted}(Y) = Y * D_{\text{inv}} + \text{parity}(C) * A \pmod{127} = B.$$

Where *D_{inv}* is the modular multiplicative inverse of *D*. From here, *B* is converted back to the character *X* to complete decryption.

In conclusion, our encryption method effectively obfuscates any and all data that may find itself fallen into the wrong hands. It preserves case and allows for special symbols and punctuation. We believe that setting a unique key daily and hiding the encrypting variables within the plain text will greatly increase security in intercompany communication. You shouldn't have to worry about the security of your sensitive data; you can rely on our cryptosystem to safely encrypt your communications and keep them away from prying eyes.