Encryption Algorithms With Linear Algebra and Number Theory

by

Orion Watler

Submitted to the Department of Computer Science and Mathematics
SUNY Purchase in partial fulfillment of the requirements for the degree of Bachelor of Arts

Purchase College

State University of New York

May 17 2021

Sponsor: Knarik Tunyan

Second Reader: Athar Abdul-quader

A thesis presented for the degree in Bachelors of Mathematics and Computer Science

Abstract

Although there are highly sophisticated encryption schemes that have been developed in the past, such as Elliptical Curve Encryption, Secure-Hash-Algorithm and Data Encryption Standard, there is still much to explore in cryptography. The goal of this research is to explore encryption techniques that use methods from linear algebra and number theory, starting at a rudimentary model to more complex algorithms that provide more security. The simpler model, Square Matrix Encryption, relies on the inverse of a square matrix, while a more complex algorithm, Rectangular Matrix Encryption, uses the pseudo-inverse of a rectangular matrix. This paper describes the nuances of both schemes and discusses what makes the latter stronger than the former. This research suggests using a special kind of generalized inverses in encryption that can turn into developing an even stronger encryption scheme. The implementation of all algorithms, which include various linear transformations and modular arithmetic on rational numbers, is done in Java.

Acknowledgements

I would like to thank my professors Dr. Tunyan Knarik and Dr. Athar Abdul-Quader for their immeasurably valuable advice. Without them, I would not be here today and neither would this project. Along with my professors, I have to acknowledge my senior advisor Dr. Irina Shablinsky for putting me on this path in the first place and for ensuring that I stayed on track over the years. My friends and family also supported me in this endeavour so my gratitude goes out to them as well.

Contents

List of Figures

Chapter 1

Introduction

The most abstract description of this specific type of encryption method is that there are two sides, traditionally called Alice and Bob, where Alice is the person who sends an encrypted message and Bob is the person who receives the message and has to decrypt it. It is assumed that both the encrypter and decrypter both posses the same keys bits of information to aide in the cryptography process. There are other encryption models, which will not be discussed further, such as Rivest–Shamir–Adleman, RSA, cryptosystem where Alice would have information unknown to Bob called the private key and both Alice and Bob would have access to shared information called the public key.

In this thesis, we examine two cryptographic methods that involve linear algebra and number theory. In doing so, the strengths and weaknesses of the various methods' security can be analyzed. We will consider the set X, consisting of all lowercase and upper case letters,

digits, special characters like every lowercase letter, uppercase letter, special character, digit and most punctuation marks. There are 89 elements of this set, and so we can put this set in bijection with the set Y of all integers from 0 to 88. We refer to this bijection as $f : X \to Y$. Each method will make use of the function f to map characters in the text that is to be encrypted to distinct integers.

Next, there is a given word that is to be encrypted called plaintext that has n characters. Let p_j be the j^{th} character of the plaintext. So $f(p_j) = q_j$ where $p_j \in X$ and $q_j \in Y$. So there are two vectors $\mathbf{p} = (p_0, ..., p_n)$ and $\mathbf{q} = (q_0, ..., q_n)$. This is where the methods branch off in terms how they each encrypt $\mathbf{q}$ and this is the main area that will be explored. Let $\mathbf{q}'$ be the vector derived from the process of encrypting vector $\mathbf{q}$. Now we can take the inverse of f defined to be $f^{-1} : Y \to X$ such that $f^{-1}(y_i) = x_i$ and then take $f^{-1}(\mathbf{q}') = \mathbf{c}$ where $\mathbf{c}$ is a vector comprised of characters which will go one to make up the ciphertext.

To decrypt the fucntion f is used to convert $\mathbf{c}$ into $\mathbf{q}'$ then reverse whatever specific encryption method used so that $\mathbf{q}'$ is turned back into $\mathbf{q}$. Then take $f^{-1}(\mathbf{q}) = \mathbf{p}$ to get the original set of plaintext characters thus completely reversing the encryption. This is the general form both encryption methods in question take.

Chapter 2

Square Matrix Method

The algorithm takes a plaintext string, its vector representation is denoted $\mathbf{p}$, and a square invertible matrix, called the coding matrix which is denoted A, then use these two objects to encrypt the plaintext into ciphertext. To decrypt, the algorithm takes the ciphertext then uses the inverse of the coding matrix, denoted A^{-1}, to turn the ciphertext back into plaintext.

2.1 Encryption

2.1.1 Create Matrix B

Matrix A can be any non zero square invertible matrix that has a size greater than 4. The width of A is taken, then each character of the plaintext is put in order into vector $\mathbf{p}$. Recall from the introduction the function f. Next is to convert this vector comprised of integers, $f(\mathbf{p})$, into a matrix where its height is equal to the width of A. Let B be the matrix produced from $f(\mathbf{p})$. Matrix B is generated by letting w be the width of A and let n be the number of rows matrix B. By the laws of modular arithmetic [1], set

$$n = \frac{|f(\mathbf{p})| - |f(\mathbf{p})| \mathrm{mod}(w)}{w} + 1.$$

The elements at indices $(j-w)^{th}$ to $(j-1)^{th}$ from $f(\mathbf{p})$ where $j \in \mathbb{N}$ such that $0 < j < |f(\mathbf{p})| - 1$. Then make those vectors the columns of matrix B. If the final column has less than n elements fill the empty spaces with 0s. Thus the vector $f(\mathbf{p})$ has been converted into an n by w matrix B.

Figure 2.1: Possible $f(\mathbf{p})$ vector

Let this example plaintext “Really Good Password I Promise!” and let the corresponding vector $f(\mathbf{p})$ = (80, 41, 37, 48, 48, 61, 0, 69, 51, 51, 40, 0, 78, 37, 55, 55, 59, 51, 54, 40, 0, 71, 0, 78, 54, 51, 49, 45, 55, 41, 17) be used to construct matrix B.

Figure 2.2: An example of matrix A and matrix B

$$\begin{pmatrix} 2 & 3 & 6 & 3 & 5 \\ 8 & 5 & 2 & 8 & 7 \\ 4 & 0 & 5 & 7 & 8 \\ 1 & 6 & 3 & 1 & 5 \\ 6 & 1 & 0 & 7 & 4 \end{pmatrix} \qquad \begin{pmatrix} 80 & 61 & 40 & 55 & 0 & 51 & 30 \\ 41 & 0 & 0 & 59 & 71 & 49 & 0 \\ 37 & 69 & 78 & 51 & 0 & 45 & 0 \\ 48 & 51 & 37 & 54 & 78 & 55 & 0 \\ 48 & 51 & 55 & 40 & 54 & 41 & 0 \end{pmatrix}$$

(a) Matrix A **(b)** Matrix B

2.1.2 Generate Core Ciphertext

Since A is a w by w matrix, the laws of matrix multiplication allow A and B to be multiplied; which then forms a new matrix AB. This matrix AB is converted into a vector by letting each successive column be the respective segments of the new vector. Let the vector which was derived from matrix AB be denoted $\mathbf{q}$.

Figure 2.3: Matrix AB

$$\begin{pmatrix} 889 & 944 & 934 & 955 & 717 & 889 & 60 \\ 1639 & 1391 & 1157 & 1549 & 1357 & 1470 & 240 \\ 1225 & 1354 & 1249 & 1173 & 978 & 1142 & 120 \\ 725 & 574 & 586 & 816 & 774 & 740 & 30 \\ 1049 & 927 & 719 & 927 & 833 & 904 & 180 \end{pmatrix}$$

Figure 2.4: Vector $\mathbf{q}$

Using the integers from the matrix at figure 2.3 let $\mathbf{q}$ = (889, 1639, 1225, 725, 1049, 944, 1391, 1354, 574, 927, 934, 1157, 1249, 586, 719, 955, 1549, 1173, 816, 927, 717, 1357, 978, 774, 833, 889, 1470, 1142, 740, 904, 60, 240, 120, 30, 180)

Claim: $\mathbf{q}$ is unique.

Proof. We are given two distinct plaintexts and their respective character mapped vectors

$\mathbf{v_1}$ and $\mathbf{v_2}$. If the lengths of $\mathbf{v_1}$ and $\mathbf{v_2}$ do not divide the width w of the coding matrix A then assume that the extra 0's have already been appended to $\mathbf{v_1}$ and $\mathbf{v_2}$ to ensure that their respective lengths divide w. Let the length of $\mathbf{v_1}$ be n and the length of $\mathbf{v_2}$ be m. Let B and C be matrices derived from $\mathbf{v_1}$ and $\mathbf{v_2}$ such that their columns are segments of size w from $\mathbf{v_1}$ and $\mathbf{v_2}$ respectively. That is to say, $B = (\mathbf{b_1} \dots \mathbf{b_n})$ where $\mathbf{b_i} \in B$ is the i^{th} segment of size w of vector $\mathbf{v_1}$ and $C = (\mathbf{c_1} \dots \mathbf{c_m})$ where $\mathbf{c_i} \in C$ is the i^{th} segment of size w of $\mathbf{v_2}$. Since the plaintexts are distinct $\mathbf{v}_1 \neq \mathbf{v}_2$. Vector $\mathbf{v_1}$ contains elements from $\mathbf{b}_1 \dots \mathbf{b_2}$ in order from the first element of $\mathbf{b_1}$ to the last, then the first element of $\mathbf{b_2}$ to the last and so on until $\mathbf{b_n}$. Similarly, $\mathbf{v_2}$ contains elements from $\mathbf{c_1} \dots \mathbf{c_m}$ in that order. It follows from there that since column vectors $\mathbf{b_i} \neq \mathbf{c_i}$ then $B \neq C$. Because $B \neq C$ so too does $AB \neq AC$. Expanding the matrix multiplication we see that $AB = A(\mathbf{b_1} \dots \mathbf{b_n}) = (A\mathbf{b_1} \dots A\mathbf{b_n})$. Similarly do the following expansion, $AC = A(\mathbf{c_1} \dots \mathbf{c_m}) = (A\mathbf{c_1} \dots A\mathbf{c_m})$. Each column vector $A\mathbf{b_i} \in AB$ and $A\mathbf{c_i} \in AC$ is turned into single vectors $\mathbf{q_1}$ and $\mathbf{q_2}$ respectively. Both vectors $\mathbf{q_1}$ and $\mathbf{q_2}$ contain elements from the columns of AB and AC in the aforementioned matrix columns to single vector order. Since $AB \neq AC$ it follows that $\mathbf{q_1} \neq \mathbf{q_2}$. Therefore no two distinct plaintexts, even those of the same length, can never generate the same $\mathbf{q}$ vector.

□

In this form, there are a number of integers in $\mathbf{q}$ that exist outside the codomain of f^{-1} which means the character mapping function cannot be used to generate the ciphertext. To solve this problem, recall that $x\mathbf{mod}(m) = y$ implies $x - y = mk$ and that $|X| = |Y| = 89$.

For each $q_i \in \mathbf{q}$, find $q_i \mathbf{mod} m = q_i\prime$ where $\mathbf{q\prime}$ is a vector comprised of integers such that $q_i\prime \in \mathbf{q\prime}$ where $0 \leq \ i \leq |\mathbf{q}|$ [2]. Then use the inverse of f to do $f^{-1}(\mathbf{q\prime}) = \mathbf{c}$ where $\mathbf{c}$ is the vector that contains the ciphertext characters in order.

2.1.3 Key List

For decryption to work the decrypter needs information on how to reverse the modular arithmetic that was done on $\mathbf{q}$ to get $\mathbf{q\prime}$. Let the vector $\mathbf{k}$ contain elements of the form

$$k_i = \frac{1}{m}(q_i - q_i'),$$

where $q_i \in \mathbf{q}$ and $q_i' \in \mathbf{q}'$. The digits of each integer in $\mathbf{k}$ put into a vector $\mathbf{d}$ and the intervals of integer length, in terms of digits, are put into another vector $\mathbf{v}$. The complete output of the ciphertext is a string made up of characters from the first character to the last character from $\mathbf{c}$, then append each digit in the elements of $\mathbf{d}$ to the ciphertext string. Lastly, append each element of $f^{-1}(\mathbf{v})$ to the ciphertext string then append the integers $f^{-1}(|\mathbf{d}|)$ and $f^{-1}(|\mathbf{v}|)$ to the end of the string. That is the ciphertext that gets sent to the decrypter.

Figure 2.5: Following the previous figures, this is $\mathbf{q}\prime$ and the core ciphertext

$\mathbf{q}\prime$ =(88, 37, 68, 13, 70, 54, 56, 19, 40, 37, 44, 0, 3, 52, 7, 65, 36, 16, 15, 37, 5, 22, 88, 62, 32, 88, 46, 74, 28, 14, 34, 47, 68, 17, 13)
$f^{-1}(\mathbf{q}\prime)$ =(Z a F = H r t 6 d a h $ p * C] 3 2 a ^ 9 Z z ~ Z j L " 1 x z ? ! #)
"ZaF=Hrt6dah $p*C]32a^9Zz~ZjL"1xz?!#"

Figure 2.6: Next is the Key List

$\mathbf{k} = \frac{1}{m}(\mathbf{q} - \mathbf{q}\prime)$ = (9, 18, 13, 8, 11, 10, 15, 15, 6, 10, 10, 13, 14, 6, 8, 10, 17, 13, 9, 10, 8, 15, 10, 8, 9, 9, 16, 12, 8, 10, 0, 1, 0, 0, 1)
k gets turned into digit string
d = "9181381110151561010131468101713910815108991612810021 02"

Followed by the interval list:
v = (0, 1, 3, 5, 6, 8, 10, 12, 14, 15, 17, 19, 21, 23, 24, 25, 27, 29, 31, 32, 34, 35, 37, 39, 40, 41, 42, 44, 46, 47, 49, 50, 51, 52, 53, 54)
$f^{-1}(\mathbf{v})$ = (@ $ ^ & < | + 1 2 4 6 8 0 . : ‘ ? ~ _ [a c d e f h j k m n o p q r)
" @$^&<|+124680.,:‘?~_[acdefhjkmnopqr"

Complete Key List

"9181381110151561010131468101713910815108991612810021 02 @$^&<|+124680.,
:‘?~_[acdefhjkmnopqrr]"

Final Out Going Ciphertext

"ZaF=Hrt6dah $p*C]32a^9Zz~ZjL"1xz?!#9181381110151561010131468101713910815108991612810021 02 @$^&<|+124680.,:‘?~_[acdefhjkmnopqrr]"

2.2 Decryption

2.2.1 Parse KeyList from Ciphertext

Given some ciphertext, there is information encoded within it which has to be parsed out. The first two pieces are the last and second to last characters appended to the ciphertext which are the lengths of **v** and **d** respectively. With this information, the ciphertext string can be split up into three strings where the first contains characters from **c**, the second contains digits from **d** and the third contains characters from **v**. Put the characters of those strings into their respective vectors. Use the information about integer intervals, stored in **v**, to then go through **d** to reconstruct **k**.

The following is an example of the key list being separated from the core ciphertext which is then used to reconstruct **k**. The decrypter is given the full ciphertext:

```
"ZaF=Hrt6dah $p*C]32a^9Zz~ZjL"1xz?!#9181381110151561010131468101713910815108991612810021
02 @$^&<|+124680.,:‘?~_[acdefhjkmnopqrr]"
```

Look at its last two characters mapped to their respective integers: 54 and 36. These two numbers tell the decrypter the length of the digit list and the interval list which give him enough information to know where to break the ciphertext into three pieces.

```
core ciphertext = "ZaF=Hrt6dah $p*C]32a^9Zz~ZjL"1xz?!#"
digits = "9181381110151561010131468101713910815108991612810021
02"
```

```
intervals = "@$^&<|+124680.,:‘?~_[acdefhjkmnopqrr]"
```

Put the digit and interval strings into vector form, denoted $\mathbf{d}$ and $\mathbf{v}$, and apply f to those vectors to get

$f(\mathbf{d})$ = (9, 1, 8, 1, 3, 8, 1, 1, 1, 0, 1, 5, 1, 5, 6, 1, 0, 1, 0, 1, 3, 1, 4, 6, 8, 1, 0, 1, 7, 1, 3, 9, 1, 0, 8, 1, 5, 1, 0, 8, 9, 9, 1, 6, 1, 2, 8, 1, 0, 0, 2, 1, 0, 2)

followed up by

$f(\mathbf{v})$ = (0, 1, 3, 5, 6, 8, 10, 12, 14, 15, 17, 19, 21, 23, 24, 25, 27, 29, 31, 32, 34, 35, 37, 39, 40, 41, 42, 44, 46, 47, 49, 50, 51, 52, 53, 54)

The information contained in $f(\mathbf{v})$ tells the decrypter how to merge certain elements in $f(\mathbf{d})$ to recreate $\mathbf{k}$ = (9, 18, 13, 8, 11, 10, 15, 15, 6, 10, 10, 13, 14, 6, 8, 10, 17, 13, 9, 10, 8, 15, 10, 8, 9, 9, 16, 12, 8, 10, 0, 2, 1, 0, 2)

Recall that $\mathbf{c}$ contains all the characters of the core ciphertext. Apply the character mapping function f to the first vector $\mathbf{c}$ in order to get $f(\mathbf{c})$ = (88, 37, 68, 13, 70, 54, 56, 19, 40, 37, 44, 0, 3, 52, 7, 65, 36, 16, 15, 37, 5, 22, 88, 62, 32, 88, 46, 74, 28, 14, 60, 62, 31, 30, 2) which is a vector that is equivalent to $\mathbf{q}\prime$. At this point, there is enough information to reverse the modular operation that was done in the encryption scheme.

2.2.2 Reconstruct Plaintext

Let $k_i \in \mathbf{k}$, $q_i \in \mathbf{q}$ and $q_i\prime \in \mathbf{q}\prime$. By the fact that $q_i \mathbf{mod}(m) = q_i\prime$ implies $q_i - q_i\prime = mk_i$, create a vector that is defined such that each of its elements are of the the form $mk_i + q_i\prime$. Since $mk_i + q_i\prime = q_i$ it is clear that this vector is $\mathbf{q}$.

If $\mathbf{k}$ = (9, 18, 13, 8, 11, 10, 15, 15, 6, 10, 10, 13, 14, 6, 8, 10, 17, 13, 9, 10, 8, 15, 10, 8, 9, 9, 16, 12, 8, 10, 0, 2, 1, 0, 2),

$\mathbf{q}\prime$ = (88, 37, 68, 13, 70, 54, 56, 19, 40, 37, 44, 0, 3, 52, 7, 65, 36, 16, 15, 37, 5, 22, 88, 62, 32, 88, 46, 74, 28, 14, 60, 62, 31, 30, 2) and

$m = 89$ then

$m\mathbf{k} + \mathbf{q}\prime$ = (889, 1639, 1225, 725, 1049, 944, 1391, 1354, 574, 927, 934, 1157, 1249, 586, 719, 955, 1549, 1173, 816, 927, 717, 1357, 978, 774, 833, 889, 1470, 1142, 740, 904, 60, 240, 120, 30, 180) = $\mathbf{q}$

The same algorithm that turned $f(\mathbf{p})$ into matrix B is used to turn $\mathbf{q}$ into the matrix AB. By the associativity of matrix multiplication $A^{-1}(AB) = (A^{-1}A)B = (I)B = B$ where I is the identity matrix [3]. Iterate through the columns of matrix B while putting each element of the i^{th} column into a single vector and lets denote that vector $\mathbf{x}$. By the nature of the algorithm $\mathbf{x} = f(\mathbf{p})$ so that means the i^{th} element of $\mathbf{x}$ equals the i^{th} element of $f(\mathbf{p})$ so $f^{-1}(\ \mathbf{x}) = f^{-1}(f(\mathbf{p})) = (f^{-1} \circ f)(\mathbf{p}) = \mathbf{p}$. Now, the decryption is complete.

Since the decrypter has access to A, he knows its width w and its inverse A^{-1}. With that information, $\mathbf{q}$ = (889, 1639, 1225, 725, 1049, 944, 1391, 1354, 574, 927, 934, 1157,

1249, 586, 719, 955, 1549, 1173, 816, 927, 717, 1357, 978, 774, 833, 889, 1470, 1142, 740, 904, 60, 240, 120, 30, 180) can be broken up into segments of size w, in this case $w = 5$, and those segments will go on to form the columns of AB. That is to say, (889, 1639, 1225, 725, 1049), (944, 1391, 1354, 574, 927), (934, 1157, 1249, 586, 719), (955, 1549, 1173, 816, 927), (717, 1357, 978, 774, 833), (889, 1470, 1142, 740, 904), and (60, 240, 120, 30, 180) will be the column vectors of AB.

$$AB = \begin{pmatrix} 889 & 944 & 934 & 955 & 717 & 889 & 60 \\ 1639 & 1391 & 1157 & 1549 & 1357 & 1470 & 240 \\ 1225 & 1354 & 1249 & 1173 & 978 & 1142 & 120 \\ 725 & 574 & 586 & 816 & 774 & 740 & 30 \\ 1049 & 927 & 719 & 927 & 833 & 904 & 180 \end{pmatrix}$$

$$A^{-1}AB = \begin{pmatrix} 80 & 61 & 40 & 55 & 0 & 51 & 30 \\ 41 & 0 & 0 & 59 & 71 & 49 & 0 \\ 37 & 69 & 78 & 51 & 0 & 45 & 0 \\ 48 & 51 & 37 & 54 & 78 & 55 & 0 \\ 48 & 51 & 55 & 40 & 54 & 41 & 0 \end{pmatrix}$$

The columns of B get put into vector $\mathbf{x}$ = (80, 41, 37, 48, 48, 61, 0, 69, 51, 51, 40, 0, 78, 37, 55, 55, 59, 51, 54, 40, 0,71, 0, 78, 54, 51, 49, 45, 55, 41, 17) and $f^{-1}(\mathbf{x})$ =(R, e, a, l, l, y, , G, o, o, d, , P, a, s, s, w, o, r, d, , I, , P, r, o, m, i, s, e, !) = $\mathbf{p}$.

Chapter 3

Pseudoinverse Matrix Encryption

3.1 Partial Pseudoinverse of a Rectangular Matrix

The pseudoinverse of a rectangular matrix is itself a matrix that mimics the properties of a genuine square matrix inverse. Although completely mimicry is impossible, the pseudoinverse has properties that allow it to be used in ways that similar to the square inverse. There are a variety of pseudoinverses of any one rectangular matrices; the one that is being used here is a modified version of the Moore-Penrose Pseudoinverse called the Partial Pseudoinverse. Consider matrix A which has height h that is less than it's width w and let A^p denote the pseudoinverse of A and it has height w and width h. Given these conditions, the most relevant properties of a the pseudoinverse are $AA^pA = A$, $A^pAA^p = A^p$ and it is unique. The unique quality of this particular pseudoinverse is important for replacing the role the

square matrix inverse had with respect to the level of security it brings.

3.1.1 Partitions List

First construct an identity matrix of dimensions h by h called M. Then make a list of pivot intervals, call it $\mathbf{K}$, such that the list is made up of intervals of integers. The first element of the list is $\{0, \frac{w}{h}\}$, the i^{th} element is $\{\frac{(i-1)w}{h}, \frac{iw}{h}\}$ and the last element is $\{\frac{(h-1)w}{h}, w\}$ where the $\frac{w}{h}$ is integer division and we ignore the remainder. In this list of intervals, there are no gaps, every element is in order and so, necessarily, each interval is unique and the number of intervals equals the number of rows in A.

If we are given the matrix that has width $w = 5$ and height $h = 3$

$$A = \begin{pmatrix} 40 & 52 & 43 & 51 & 58 \\ 56 & 17 & 42 & 60 & 53 \\ 50 & 10 & 59 & 37 & 2 \end{pmatrix}$$

start off at $i = 1$. The first key interval is calculated by first getting this finding a different, temporary, interval $\{\frac{(i-1)w}{h}, \frac{iw}{h}\} = \{\frac{(1-1)5}{3}, \frac{1\times5}{3}\} = \{0, 1\}$. All of the integers starting at the beginning right up until the end are part of the first key interval. In this case the first key interval is $\{0\}$. Next is $i = 2$ and the second temporary interval is $\{\frac{(i-1)w}{h}, \frac{iw}{h}\} = \{\frac{(2-1)5}{3}, \frac{2\times5}{3}\} = \{1, 3\}$ because $\frac{5}{3} = 1$ and $\frac{10}{3} = 3$ in integer division. The second key interval is $\{1, 2\}$. Lastly, when $i = 3$ we follow the same steps by first making a temporary interval $\{\frac{(i-1)w}{h}, \frac{iw}{h}\} = \{\frac{(3-1)5}{3}, \frac{3\times5}{3}\} = \{3, 5\}$ and so the third key interval is $\{3, 4\}$. The final partition list $\mathbf{K} = \{\{0\}, \{1, 2\}, \{3, 4\}\}$.

With the pivot intervals constructing the actual partial pseudoinverse matrix can begin. This is where the name Partial Pseudoinverse comes from, since it differs from Full Row Moore-Penrose Pseudoinverse [4] in how it partially orthogonalizes the rows from R instead of doing the same for each complete row in R.

3.1.2 Produce Pivot Vector

Start the process at row one, which will be called the pivot row, and use the first interval of integers in $\mathbf{K}$, called $\mathbf{k_1}$, to extract elements from the first row and put them into vector that has the same length as $\mathbf{k_1}$. Let this vector be called key row, $\mathbf{c_r}$, and find the norm n of $\mathbf{c_r}$.

Using the same matrix A, let the current pivot row be the first row $\mathbf{v_1} = (40\ 52\ 43\ 51\ 58)$. Since the first element of $\mathbf{K}$ is $\{0\}$, let the key row derived from $\mathbf{v}$ be $\mathbf{c_{r1}} = (40)$. The norm of $\mathbf{c_{r1}}$ is 40.

The alpha vector is next. Initially, let the pivot row $\mathbf{v_1}$ be the first row of A. We will then operate on all the other rows of A. Row two is (56, 17, 42, 60, 53) and row three is (50, 10, 59, 37, 2). The partial row of row two $\mathbf{p_{r2}}$ is found by taking each element that is accessed by the indexes contained in $\mathbf{k_1}$. That is to say, since $\mathbf{k_1} = \{0\}$ then $\mathbf{p_{r2}} = (56)$. The dot product $d_2 = \mathbf{c_{r1}} \cdot \mathbf{p_{r2}} = (40) \cdot (56) = 2240$. The corresponding alpha value is $\alpha_1 = \frac{-d_2}{n} = \frac{-2240}{40} = -56$. Next, get the partial row of row three $\mathbf{p_{r3}} = (50)$, find the dot product $d_3 = \mathbf{c_{r1}} \cdot \mathbf{p_{r3}} = 2000$. And so $\alpha_3 = \frac{-d_3}{n} = -50$. The alpha vector generated where

the first row of A is the pivot row is $\mathbf{a_1}$ = (-56 -50).

In general, if we let i be the index of the current pivot row $\mathbf{v_i}$, a single α is produced by going to any row in A whose index $j \neq i$. Then use $\mathbf{k_i} \in \mathbf{K}$ and the indexes contained in $\mathbf{k_i}$ to produce two vectors. The first is called the key row $\mathbf{c_r}$ which is a sub vector of $\mathbf{v_i}$ and the other is called the i^{th} partial row $\mathbf{p_{ri}}$. Calculate the dot product of $\mathbf{c_r}$ and $\mathbf{p_{ri}}$, call it d, multiply d my -1 and divide it by the norm($\mathbf{c_r}$) = n. The resulting formula is $\alpha_j = \frac{-d}{n}$. Repeat this process starting at the first row whose index is not equal to the index of $\mathbf{v_i}$ to produce a list of alpha values and this list will be the alpha vector $\mathbf{a}$.

With the alpha vector and the norm of $\mathbf{k_r}$ multiply the pivot row in A by $\frac{1}{\text{norm}(\mathbf{k}_r)}$. Then for each $i \neq j$, do row addition such that $\text{row}_j = \text{row}_j + \alpha_j \mathbf{v_i}$. Do this exact operation on the identity matrix M. Repeat these steps such that the next pivot row is row two and the next one is row three and so on until each successive row in A has been the pivot vector one time.

The steps done on A are as follows,

$$A = \begin{pmatrix} 40 & 52 & 43 & 51 & 58 \\ 56 & 17 & 42 & 60 & 53 \\ 50 & 10 & 59 & 37 & 2 \end{pmatrix}$$

$$\mathbf{K} = \{\mathbf{k_1}, \mathbf{k_2}, \mathbf{k_3}\} = \{\{0\}, \{1, 2\}, \{3, 4\}\}$$

$$\mathbf{k_1} = \{0\}$$

$$\mathbf{v_1} = (40,\ 52,\ 43,\ 51,\ 58)$$

$$\mathbf{c_{r1}} = (40) \text{ and } n_1 = \text{norm}(\mathbf{c_{r1}}) = 40$$

$$\mathbf{p_{r2}} = (56) \text{ and } d_2 = \mathbf{c_{r1}} \cdot \mathbf{p_{r2}} = 2240 \text{ and } \alpha_2 = -\frac{d_2}{n_1} = -56$$

$$\mathbf{p_{r3}} = (50) \text{ and } d_3 = \mathbf{c_{r1}} \cdot \mathbf{p_{p3}} = 2000 \text{ and } \alpha_3 = -\frac{d_3}{n_1} = -50$$

$$\mathbf{a_1} = (\alpha_2, \alpha_3) = (-56, 50)$$

Scalar multiplication on A: $\frac{1}{n_1}\text{row}_1$

Row Addition on A: $\text{row}_2 = \text{row}_2 + \alpha_2 * \text{row}_1$

Row Addition on A: $\text{row}_3 = \text{row}_3 + \alpha_3 * \text{row}_1$

Result of operations:

$$A = \begin{pmatrix} 1 & 1.3 & 1.075 & 1.275 & 1.45 \\ 0 & -55.8 & -18.2 & -11.4 & -28 \\ 0 & -55 & 5.25 & -26.75 & -70.5 \end{pmatrix}$$

Repeat the row operations on M which results in

$$M \begin{pmatrix} 0.025 & 0 & 0 \\ -1.4 & 1 & 0 \\ -1.25 & 0 & 1 \end{pmatrix}$$

According to $\mathbf{k_1}$ the elements of the pivot row will be found in (1, 1.3, 1.075, 1.275, 1.45) in indexes contained in $\mathbf{k_1}$. The pivot element, singular in this case, is (1) but the whole pivot row must be the same width as A. So the pivot vector is $\mathbf{p_v} = (1, 0, 0, 0, 0)$.

To construct matrix B we will repeat this process where each row in A is the pivot row one time and let the pivot vector that gets produced at the end of each step be a row in B. All together the step by step process looks like:

$$A = \begin{pmatrix} 1 & 1.3 & 1.075 & 1.275 & 1.45 \\ 0 & -55.8 & -18.2 & -11.4 & -28 \\ 0 & -55 & 5.25 & -26.75 & -70.5 \end{pmatrix}$$

$$\mathbf{K} = \{\mathbf{k_1}, \mathbf{k_2}, \mathbf{k_3}\} = \{\{0\}, \{1, 2\}, \{3, 4\}\}$$

$$\mathbf{k_2} = \{1, 2\}$$

$$\mathbf{v_2} = (0, -55.8, -18.2, -11.4, -28)$$

$$\mathbf{c_2} = (-55.8, -18.2,) \text{ and } n_2 = \text{norm}(\mathbf{c_2}) = 58.69310010554903$$

$$\mathbf{p_{r1}} = (1.3, 1.075) \text{ and } d_1 = \mathbf{c_2} \cdot \mathbf{p_{r1}} = -92.105 \text{ and } \alpha_1 = -\frac{d_1}{n_2} = 1.5692645274208663$$

$$\mathbf{p_{r3}} = (-55, 5.25) \text{ and } d_3 = \mathbf{c_2} \cdot \mathbf{p_{r3}} = 2973.45 \text{ and } \alpha_1 = -\frac{d_1}{n_2} = -50.660980501162534$$

$$\mathbf{a_2} = (\alpha_2, \alpha_3) = (1.5692645274208663, -50.660980501162534)$$

Scalar multiplication on A: $\frac{1}{n_2}\text{row}_2$

Row Addition on A: $\text{row}_1 = \text{row}_1 + \alpha_1 * \text{row}_2$

Row Addition on A: $\text{row}_3 = \text{row}_3 + \alpha_3 * \text{row}_2$

Result of Operations:

$$R = \begin{pmatrix} 1.0 & -0.191912 & 0.588390 & 0.970200 & 0.696022 \\ 0.0 & -0.950708 & -0.310087 & -0.194231 & -0.480465 \\ 0.0 & -6.8362 & 20.959334 & -16.910084 & -46.159155 \end{pmatrix}$$

Repeat the row operations on M which results in

$$M = \begin{pmatrix} -0.012431 & 0.026736 & 0 \\ -0.023852 & 0.017037 & 0 \\ -0.041589 & -0.863150 & 1 \end{pmatrix}$$

The pivot vector elements are (-0.950708, -0.310087) and the complete pivot vector is

(0, -0.950708, -0.310087, 0, 0).

Follow these exact steps where row three is the pivot row to get a third pivot vector (0.0, 0.0, 0.0, -0.343986, -0.938974).

3.1.3 Construct the Pseudo-Inverse

Create a list of pivot vectors where they each have length h by following the steps from 3.1.2. Keep in mind that each time the steps from 3.1.2 are repeated, matrix A is altered and so to is matrix M. Let this list of pivot vectors be the rows for matrix B. Once that is done transpose B. Matrix $A^p = B^T M$. Now the pseudo-inverse has been produced.

Based on the previous steps the following matrices B and M are equal to

$$B = \begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & -0.950708 & -0.310087 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & -0.343986 & -0.938974 \end{pmatrix}$$

$$M = \begin{pmatrix} -0.013266 & 0.009401 & 0.020083 \\ -0.023414 & 0.026132 & -0.010536 \\ -0.000846 & -0.017558 & 0.020342 \end{pmatrix}$$

They are then multiplied together to get

$$A^p = \begin{pmatrix} -0.013266 & 0.0094017 & 0.0200834 \\ 0.022260 & -0.0248441 & 0.010017 \\ 0.007261 & -0.008103 & 0.003267 \\ 0.000291 & 0.006039 & -0.006997 \\ 0.000794 & 0.016486 & -0.019101 \end{pmatrix}$$

3.1.4 Diophantine Equation Model

The following model is a framework on where the pseudoinverse can be implemented. Although it does work as an encryption scheme in its own right, the pseudoinverse is an element of the algorithm that would improve the security of this model.

encrypt

Let matrix R be a rectangular matrix where $RR^pR = R$, $R^pRR^p = R^p$ and R^p is unique which means R has the properties necessary for it to have a pseudoinverse. Given some plaintext and a height minimum h, turn it into a matrix of width h and variable height that depends on the length of the plaintext and let P be that matrix. Multiply matrix P by R and then find the remainder mod 89 of each element in PR. Let this matrix be denoted C.

In this example let

$$P = \begin{pmatrix} 55 & 52 & 41 \\ 37 & 47 & 41 \\ 54 & 0 & 0 \end{pmatrix}$$

$$R = \begin{pmatrix} 40 & 52 & 43 & 51 & 58 \\ 56 & 17 & 42 & 60 & 53 \\ 50 & 10 & 59 & 37 & 2 \end{pmatrix}$$

be the example matrices. Which means that $PR\mathbf{mod}(89)$ is

$$C = \begin{pmatrix} 52 & 14 & 38 & 62 & 88 \\ 42 & 73 & 44 & 14 & 39 \\ 0 & 18 & 72 & 54 & 72 \end{pmatrix}$$

Turn the elements of C into a vector $\mathbf{c\prime}$ by letting the rows of C be the ordered partitions of $\mathbf{c\prime}$. Due to the modular arithmetic from earlier in the process, each integer in $\mathbf{c\prime}$ can be mapped to a character in f^{-1}. Let vector $\mathbf{c} = f^{-1}(\mathbf{c\prime})$ and add the height of C mapped to its respective character in f^{-1} to the end of $\mathbf{c}$. The outgoing ciphertext is a string made up of the characters from $\mathbf{c}$.

Let $\mathbf{c_1}, \mathbf{c_2}, \mathbf{c_3}$ be the rows of C. The ciphertext is made up of characters from $f^{-1}(\mathbf{c_1}), f^{-1}(\mathbf{c_2})$, and $f^{-1}(\mathbf{c_3})$ in that order. This means the outgoing message sent to the decrypter is `"p1bzZfKh1c 5JrJ$"`.

decrypt

The height h of matrix C is stored as the last character of the ciphertext. Map each character of ciphertext to an integer in f and store the results in a vector c. With $f(\mathbf{c})$ and h, reconstruct matrix C.

Since $f(\mathbf{c})$ = (52, 14, 38, 62, 88, 42, 73, 44, 14, 39, 0, 18, 72, 54, 72) and height 3 that allows the decrypter to break $f(\mathbf{c})$ into 3 parts $(52, 14, 38, 62, 88)$, $(42, 73, 44, 14, 39)$, $(0, 18, 72, 54, 72)$. Those three parts are the rows of C.

Find the inverse of R times the transpose of R, which is $(RR^T)^{-1}$, then, to get the right inverse of R multiply R^T, multiply the transpose of R with $(RR^T)^{-1}$. Each element of the right inverse of R is a rational decimal. Since they can be written as fractions, find the common denominator for each fraction then multiply the right inverse of R by that common denominator, all of which is denoted by W'. Let matrix W be W' mod 89. Matrix P' is CW. In P' there are integers and fraction. So, for each fraction in P' set them equal to some variable x mod 89. This equation represents a Diophantine Equation such that if the fraction in question was $\frac{n}{m}$ then the equation would be of the form $n - mx = 89y$ [5]. By the end there should be a solution where x and y are integers. Make use of the solution for x by doing x mod 89. The entry index that x was derived from will be the correct entry to store the result of x mod 89 in P. Turn P back into plaintext.

3.1.5 Pseudoinverse Implementation

The working theory is that where ever R or R^T appears in the Diophantine Equation Model the pseudoinverse of R can be used to add an extra layer of security within the algorithm. Particularly, in the beginning right before the remainder of each element in PR gets calculated, which means C would now be equal to PRR^p instead of PR. Assuming that each element in C is rational, find the common denominator of each number then multiply C by that common denominator to get an integer matrix. The elements of C mod 89 can then be used to generate ciphertext.

If not there then later on in the algorithm, the transpose of R pseudoinverse can be multiplied to the right inverse of R. Assuming the result of that product, $R^pR^T(RR^T)^{-1}$ is a matrix of rational numbers, a common denominator can be found and which can then scale that matrix so it becomes an integer matrix that can be processed into ciphertext.

In either case, the final result is a scheme that is more secure than the Square Matrix Method which highlights why the Partial Pseudoinverse Method was developed. Chief among the security improvements is that the highly vulnerable key list from the Square Matrix Encryption method was done away with. The uniqueness of an encoding key, where A from the Square Matrix Encryption or R from Pseudoinverse Encryption are examples, because if there are multiple ways to decode a message this means the algorithm used to construct that message is extremely weak. It is known that any invertible square matrix has a unique inverse matrix and similarly, a wide rectangular matrix has a partial pseudoinverse that is

also unique. The Partial Pseudoinverse Method saves a lot of computational time because it does not take as many arithmetical operations as calculating a full row Moore-Penrose Pseudoinverse. Lastly, a layer of security is added because in Partial Pseudoinverse Method a random column partition is chosen for each row of R. Also in Moore-Penrose each vector is perpendicular to each other. Whereas in Partial Pseudoinverse, only the part of the vector that correspond to the K parititions lists are perpendicular to each other.

Chapter 4

Conclusion: Weakness Of Any Pure Linear Algebra Scheme

A linear function is defined to be a function $f : V \to W$ whose properties are $f(x+y) = f(x) + f(y)$ and $f(ax) = af(x)$ where V and W are vector spaces, $x, y \in V$ and a is some scalar. To explain what a perfect non-linear function is some preliminaries need to be stated. A function $h : X \to Y$ is considered to be balanced when for each $y \in Y$ the cardinality of the set $x \in X$ such that $h(x) = y$ is equal to $\frac{|X|}{|Y|}$. Given a different function $f : G \to H$, where G and H are additive groups, f is perfect nonlinear if and only if for each $g \in G$ the map $x \mapsto f(g + x) - f(x)$ is balanced which means that $|H|$ divides $|G|$ [6]. The use of perfect nonlinear functions or almost perfect nonlinear functions are used in Data Encryption Standard and other nonlinear ciphers in general because they are resilient

to differential attacks. Notice that differentiation is a linear function. If $D : V \to W$ is the differentiation function where V and W are a vector spaces that contain all differentiable functions then the statement $f, g \in V$, $f'+g' = (f+g)'$ is the same as $D(f)+D(g) = D(f+g)$. Similarly, where r is a scalar $r(f') = (rf)'$ is the same as $D(rf) = rD(f)$, which shows that differentiation is linear. There are a variety of cryptographic attacks that make use of this property, among others, to gather information about the security key that can be used to break a linear encryption scheme.

Ciphers that rely on linearity can be broken much more easily than non linear ciphers [8]. In the case of an encryption scheme that relies almost entirely on linearity, such as the Square Matrix Encryption model, although many of its security flaws were taken care of by the pseudoinverse implementation, a different path can be taken that implements nonlinear processes to add an extra layer of security that is more secure than what is offered by the Pseudoinverse Method. The key list in Square Matrix Encryption leaks far too much information to would be attackers thereby making that the weakest part of the algorithm. To fix this, a non linear cipher can be applied to it to increase security. On the decryption end of the process, the key list parsing steps of the algorithm can remain unchanged in all ways aside from the fact that now there must be a way to decrypt the original key list to move on with the decryption process. This modification to the Square Matrix Encryption algorithm

Chapter 5

Pseudo-Code

Algorithm 1 Square Matrix Encryption: encrypt(String plaintext)

1: Matrix A
2: Turn given plaintext into a matrix B using the width of A and the length of the plaintext
3: *Multiply* matrices A and B
4:
5: Turn the columns of AB into a list called ctNumbers
6: List ctModP
7: **for each** double n in ctNumbers, *add* n mod($prime$) to ctModP
8:
9: String ciphertext
10: **for each** double n in ctModP, *append* $f^{-1}(n)$ to ciphertext
11:
12: List keyList
13: **for each** double c in ctNumbers and double m in ctModP, *add* $(c-m)/prime$ to keyList
14:
15: List digitList
16: **for each** String s in keyList, *iterate* through each character of s and *add* each of those characters to digitList
17:
18: List positions
19: **for each** String s in keyList, *add* pCount to positions list then *increment* pCount+=s.length
20:
21: String lastKey
22: *add* positions[positions.size()-1]+lastKey.length() to positions list
23: Make a new String called digitString
24: **for each** String s in digitList, *append* s to digitString
25:
26: String positionString
27: **for each** int p in positions, *append* $f^{-1}(p)$ to positionString String
28:
29: *Append* digitString, positionString, digitString.length, positionString.length to ciphertext in precisely that order
30: return ciphertext

Algorithm 2 Square Matrix Encryption: decrypt(String ciphertext)

get last character from ciphertext
get 2ndLast character from ciphertext
List cipherCharacters
for $i = 1, 2, \ldots$ ciphertext.length - f(last) - f(2ndLast) - 2 **do**
    Add ciphertext.charAt(i) to cipherCharacters
end for
List digitCharacters
for $i = 1, 2, \ldots$, ciphertext.length() - f(last) - 2 **do**
    Add ciphertext.charAt(i) to positions
end for
List positions
int lim = cipherCharacters.size + digitCharacters.size;
for $i = lim, lim + 1, lim + 2 \ldots$, ciphertext.length() - 2 **do**
    Add f(ciphertext.charAt(i)) to positions
end for
List separate
for $i = 1, 2, \ldots$ positions.size **do**
    String str
    for int k = positions[i]; k <positions[i+1]; k++ **do**
        Append digitCharacters[k] to str
    end for
    Add str to separate
end for
List keyList
for each String s in separate, Parse a double from s then *add* that double to the keyList
List ctModP
for each char c in cipherCharacters, *add* $f(c)$ to ctModP
List ctNumbers
for $i = 1, 2, \ldots$ ctModP.size **do**
    ctNumbers.add(*prime*×keyList[i] + ctModP[i])
end for

Matrix $A^{-1} = A.inverse$
Matrix AB = arrayToMatrix(ctNumbers, A.height)
Matrix B = $A^{-1} \times AB$
String plaintext = matrixToWord(B)
return plaintext

Algorithm 3 Pseudo-Inverse Matrix Generator: pseudoInv(Matrix R)

Matrix I_{MM} is a new Matrix of height $R.height$ and width $R.height$
ListOfLists Keys = generate a list of lists that contain column indices of R in order where there are no repetitions
ListOfLists pivotVectors
for $p = 0, 1, 2, \ldots$ Keys.size **do**
    keyRow is a list of elements from row p of Matrix R at column indices Keys[p]
    n is the norm of keyRow
    List alphas
    for $i = 0, 1, 2, \ldots < R.height$ **do**
        if $p \neq i$ **then**
            partialRow is a list from row i of Matrix R at column indices Keys[p]
            dot is the dot product of keyRow and partialRow
            $\alpha = -1 * dot/norm$
            add α to alphas
        end if
    end for

    multiply row p of Matrix R by $1/n$
    int ac = 0;
    for $i = 0, 1, 2, \ldots R.height$ **do**
        if $i! = p$ **then**
            Perform row addition in Matrix R: alphas[ac] times row $R[p]$ plus row $R[i]$
            ac++;
        end if
    end for
    repeat theses row operations on I_{MM}

    pivotElements is a list of elements from row p or Matrix R at column indices Keys[p]
    List pivotVector. *add* $R.width$ many 0s to this list
    int $i = 0$
    for each int k in Keys[p] **do**
        remove element from pivotVector at position k, add pivotElements[i] to pivotVector at position k then $i++$
    end for
    add pivotVector to pivotVectors
end for

The rows that make up Matrix B are the pivotVectors in order from the previous part
Transpose Matrix B to get Matrix B^T
Matrix $R^+ = B^T \times I_{MM}$
return R^+

Algorithm 4 Pseudoinverse Matrix Method: encrypt(plaintext)

Let h be the standard height
if $plaintext.length \geq 2h + 1$ **then**
    Matrix A is a matrix where it's rows are the ordered partitions of the characters in plaintext where the partition size is h
    $prime$ is the size of the domain and codomain of f
    Matrix P is the transpose of A
    Follow the pseudoinverse algorithm applied to R to get R^+
    Multiply P and R
    Matrix C = PR mod $prime$
    String ciphertext is currently empty
    for $i = 0, 1, 2, \ldots, C.height$ **do**
        for $j = 0, 1, 2, \ldots, C.width$ **do**
            append $f^{-1}(C^i_j)$ to ciphertext
        end for
    end for
    append $(f^{-1}(C.height))$ to ciphertext
    return ciphertext
else
    plaintext is too short
    return null
end if

Algorithm 5 Pseudoinverse Method: decrypt(ciphertext)

char h = last character in ciphertext
int $height = f(h)$
List cNumbers is initially empty
for $i = 0, 1, 2, \ldots,$ ciphertext.$length - 1$ **do**
    add f(ciphertext.charAt(i))
end for
Matrix C = arrayToMatrix(cNumbers, height)
Matrix R^T is the transpose of R
Matrix $rinvR = R^T \times (R \times R^T)^{-1}$
Convert each element of $rinvR$ from double to fraction
Find the common denominator for each fraction in $rinvR$
Let matrix $wrinvR$ be equal to $(common\ denominator) \times rinvR$
Matrix $W = wrinvR$ mod($prime$)
Matrix $P' = C \times W$
In P' there are integers and fractions.
Matrix P is empty
for each element in P' **do**
    if element is a fraction **then**
        Set them equal to some variable x mod $prime$
        Solve the Diophantine Equation $(numerator) - (denominator)x = (prime)y$
        add x mod($prime$) to P at current positon
    else
        add element to P at current position
    end if
end for
Turn P back into plaintext

Algorithm 6 Diophantine Equation: remainderTheorem(double a, double b)

```
1: if b < a then EuclideanAlgorithm(a, b)
2: else EuclideanAlgorithm(b, a)
3:
1: EuclideanAlgorithm (double small, double big){
2: double r = small mod big
3: int d = (small)/(big) where small and big are cast as integers
4: if r == 0 then return big
5: double[] equationBits = r, 1, small, −d, big
6: equationStack.push(equationBits)
7: return EuclideanAlgorithm(big, r)}
```

Algorithm 7 Diophantine Equation: build(double[] p, double[] q)

```
1: if p[4] ≠ q[4] then swap elements at indices 2, 4 and 1, 3 in list p
2: double[] sum = new double[5]
3: for k = 1, 2, 3, ... p.length do
4:     if q[0] == p[k] and k is even then
5:         q[1] = p[1]*q[1];
6:         q[3] = p[1]*q[3];
7:         sum[0] = p[0];
8:         sum[1] = q[3] + p[3];
9:         sum[2] = q[4];
10:        sum[3] = q[1];
11:        sum[4] = q[2]; break
12:    end if
13: end for
14: return sum
```

Algorithm 8 Diophantine Equation: equation(double a, double b, double m)

```
1: StackOfLists equationStack is declared as a global variable
2: If either a or b is 0 exit the method
3: Declare String equ = a+"x -"+b+"="+m+"k"
4: Call remainderTheorem on a, b
5: int i = equations.size-1
6: if i > 1 then
7:     double[] p = build(equationStack.pop(i), equationStack.pop(i-1))
8:     int k = 2;
9:     while equationStack.size > 0 do
10:        p = build(p, equationStack.pop(i-k)
11:        k++
12:    end while
13:    return p
14: end ifElse
15: if i == 1 then
16:    return equationStack.pop(i);
17: end if
18: if i == 0 then exit the method
```

Algorithm 9 arrayToMatrix(List list, int limit)

```
ListOfLists prim is an empty list
int n = (list.size - list.size mod(limit))/limit + 1
int zerosToAdd = limit - list.size mod(limit)
if list.size mod(limit) = 0 then zerosToAdd = 0
int j = 0
for i = 1, 2, 3, ..., n do
    List col is an empty list
    while j < limit * i and j ¡ list.size do
        add list[j] to col
    end while
    if col.size ≠ limit and col.size ≠ 0 then
        int k = 0
        while k < zerosToAdd do
            add 0 to col and k++
        end while
    end if
    if col.size ≠ 0 then
        add col to prim
    end if
end for
Let prim be Matrix A
return the transpose of A
```

Bibliography

[1] N/A, Modular Arithmetic, December 1 2009,

http://www.cs.jhu.edu/ cgarman/ModularArithmetic.html#inverseTable

[2] N/A, Cryptography, December 14 2009,

https://www.cs.jhu.edu/ cgarman/Cryptography.html

[3] Rodriguez A, Cryptography and Linear Algebra, July 25 2014,

ttps://www.nibcode.com/en/blog/1130/cryptography-and-linear-algebra

[4] Asmaa Kanan and Zaleekha Abu Zayd, Using the Moore-Penrose Generalized Inverse in Cryptography, World Scientidic News,

August 5 2020

[5] Weisstein, Eric W. "Diophantine Equation." From MathWorld–A Wolfram Web Resource. https://mathworld.wolfram.com/DiophantineEquation.html

[6] Blondeau C and Nyberg K, Perfect Nonlinear Functions and Cryptography, Scientific Direct,

November 7 2014,

https://www.sciencedirect.com/science/article/pii/S1071579714001208

[7] Davis, James A. and Poinsot, Laurent, "G-Perfect Nonlinear Functions" (2008). Math and Computer Science Faculty Publications. 141. http://scholarship.richmond.edu/mathcs-faculty-publications/141

[8] Blondeau, C., and; Nyberg, K. (n.d.). Perfect nonlinear functions and cryptography. Finite Fields and Their Applications, 32, 120-147. doi:March 2015