Orthogonal Matrix Cryptosystem

James Della-Giustina

Towson University - Baltimore, MD

Introduction

We aim to implement a cryptography system proposed by Yeray Santana [4] that utilizes properties of orthogonal matrices in order to encode and decode plain-text.

Orthogonal Matrices

Let's identify the properties of orthogonal matrices that we can exploit in order to replicate the cryptosystem.

For any orthogonal matrix Q:

1. Possesses a full set of distinct eigenvalues λ_i.
2. $\forall \lambda_i \in \mathbb{R}$.
3. Property (1) automatically implies that Q is diagonalizable such that $Q = D\Lambda D^{-1}$ where $\Lambda =$

$$\begin{pmatrix} \lambda_1 & 0 & 0 & \dots & 0 \\ 0 & \lambda_2 & 0 & \dots & 0 \\ 0 & 0 & \lambda_3 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & \dots & \lambda_n \end{pmatrix}$$

Orthogonal Matrices

4. Each column of Q is linearly independent, so $Q^{-1} = Q^T$ iff $|q_i| = 1$.
5. Given Q is orthogonal, then so is the eigenvector matrix D and from Property (3) we have $D^{-1} = D^T$.

All of these properties are established and proven in [5].

Matrix Exponentials

Additionally, we know that for any diagonalizable matrix $Q = D\Lambda D^{-1}$, then the matrix exponential:

$$
\begin{aligned}
e^{Qt} &= I + Qt + \frac{(Qt)^2}{2!} + \frac{(Qt)^3}{3!} + \dots \\
&= D(I + Qt + \frac{(Qt)^2}{2!} + \frac{(Qt)^3}{3!} + \dots)D^{-1} \\
\therefore \quad &= De^{\Lambda t}D^{-1}
\end{aligned}
$$

Taylor Expandable Functions

Santana states that for any function $f(Q)$ with a Taylor series expansion and a diagonal matrix Λ, then:

$$f(Q) = D \begin{pmatrix} f(\lambda_1) & 0 & \dots & 0 \\ 0 & f(\lambda_2) & \dots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & f(\lambda_n) \end{pmatrix} D^T$$

Encryption Method

Our aim is construct a crypto-system where our each cipher-text number corresponds to some λ_i where an encryption/decryption process produces vectors as:

$$\texttt{encrypt} = (D^T)^j \begin{pmatrix} f(\lambda_1) & 0 & \dots & 0 \\ 0 & f(\lambda_2) & \dots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & f(\lambda_n) \end{pmatrix} D^j \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \end{pmatrix}$$

$$\texttt{decrypt} = D^j \ \ \texttt{encrypt} \equiv D^j \ (D^T)^j \begin{pmatrix} f(\lambda_1) & 0 & \dots & 0 \\ 0 & f(\lambda_2) & \dots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & f(\lambda_n) \end{pmatrix} D^j \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \end{pmatrix}$$

Our Function

In order to deviate from the pure exponential function used in the paper, we decided to employ the hyperbolic cosine function *cosh* since it is a composition of exponentials by the elementary identity:

$$cosh(x) = \frac{e^x + e^{-x}}{2}$$

Python Implementation

Our objective was to implement Santana's crypto-system with the requirements that our matrices were size at least 7×7, our encoding scheme was distinct from Santana's, and that we used a different bijective function. For our implementation we chose Python for its extensive linear algebra libraries.

```
import numpy as np
import sympy as sym
from numpy import pad
from sympy import *
from sympy.functions import log
from scipy.stats import ortho_group
import itertools
from sympy.matrices import Matrix, diag
```

Python Implementation

First we needed to generate an 8×8 matrix x calling from SciPy's ortho_group function and encode our plain-text into a cipher-text using a Python 'dictionary' datatype appropriately named encoder.

```
x = ortho_group.rvs(8)
textfromfile = '500 Words Essay on Time Essay On Time: Time is very precious...'
# Setup this empty string encodedtext
encodedtext = []
# Change plaintext into 'ciphertext'
encoder = {
    'a': '27', 'b': '28', 'c': '29', . . .
          }
# Change 'ciphertext' into plaintext
decoder = {};
for i in encoder:
   decoder[encoder[i]] = i
```

```
encoder =
  {'a': '27', 'b': '28', 'c': '29', 'd': '30', 'e': '31', 'f': '32',
   'g': '33', 'h': '34', 'i': '35', 'j': '36', 'k': '37', 'l': '38',
   'm': '39', 'n': '40', 'o': '41', 'p': '42', 'q': '43', 'r': '44',
   's': '45', 't': '46', 'u': '47', 'v': '48', 'w': '49', 'x': '50',
   'y': '51', 'z': '52',

   '0': '68', '␣': '72', '2': '56', '3' : '57', '4': '58', '5': '59',
   '6': '53', '7': '54', '8': '55', '9' : '56','␣␣': '71', '!': '61',
   '.': '62', '?': '63', ':': '64', ';': '65', '/': '66','"': '66',
   '\'': '67', '': '69', ',': '70',

   'A': '60', 'B': '2', 'C': '3', 'D': '4', 'E': '5', 'F': '6',
   'G': '7', 'H': '8', 'I': '9', 'J': '10', 'K': '11', 'L': '12',
   'M': '13', 'N': '14', 'O': '15', 'P': '16', 'Q': '17', 'R': '18',
   'S': '19', 'T': '20', 'U': '21', 'V': '22', 'W': '23', 'X': '24',
   'Y': '25', 'Z': '26'}
```

Python Implementation

Since we are working in blocks of 8 characters, we need to make sure the length of the text is evenly divisible by 8. If this condition is not met, we pad the last block of 8 with the number 72, which represents a space.

```
# Reading in our plain text and encoding it
for i in range(0, len(textfromfile)):
    encodedtext.append(encoder[textfromfile[i]])

# Pad the encoded text with 72's so that its divisible by 8
encodedtext_pad = np.pad(encodedtext, (0, 8-len(encodedtext)%8),
'constant', constant_values=72)

# Convert from string to int so that we can use it as arguments for our functions
encodedtext_pad = [int(i) for i in encodedtext_pad]
```

Python Implementation

Using the integer variable j that we use to exponentiate, we construct a diagonal matrix a and vector vect filled with the numbers corresponding to the plain-text.

```
j = 1
# Empty array
plain=[]
for i in range(0, len(encodedtext_pad), 8): # Step sizes of 8
    # Constructing diagonal matrices with the COSH function
  a = diag(.5*(sym.exp(encodedtext_pad[i])+
        sym.exp(-encodedtext_pad[i])),...,)
    # Create a vector with numerical values that correspond to our
    # ciphertext
  vect = Matrix([[encodedtext_pad[i]],
              [encodedtext_pad[i+1]],...)
```

Python Implementation

We then create the `encrypt` and `decrypt` vectors according to the methodology described above. Once we have constructed our `decrypt` vector, we have Python store each exponential term in a new vector called `plain`.

```
  # To create our encrypted vector, use the orthogonal matrix x
encrypt= (x.T**j) @ a @ (x**j) @ vect
  # Decrypt the encrypted vector
decrypt = a @ (x**j) @ vect
  # Pulls out the power of the exponential in each decrypted vector
for i in range(0, len(decrypt)):
    plain += [decrypt[i].atoms(exp)]
  # Run over all j until all of our ciphertext is operated on
j = j + 1
```

Python Implementation

We then need to remove set brackets and take the natural logarithm *ln* of each of these entries in `plain`. Since there are duplicates for each number we work in step sizes of two and at the end convert each remaining entry to a `string`.

```
    # Remove set brackets from plain array and store ln of absolute
    # value of entries, using step sizes of 2 to isolate one value
    # for each entry
merged = list(itertools.chain.from_iterable(plain))
d_text = []
for i in range(0, len(merged),2):
    d_text.append(abs(ln(merged[i])))
    # Convert back from int to string
dec_text = [str(i) for i in d_text]
```

Python Implementation

Next we have an array of single character strings, we want pass this through our `decoder` dictionary and merge the results back into our original plaintext.

```
plaintext = []
plainmerged = []
for i in range(0, len(dec_text)):
    plaintext.append(decoder[dec_text[i]])
plainmerged[0:len(plaintext)] = [''.join(plaintext[0:len(plaintext)])]
```

Which returns exactly the text that we wanted.

Analysis & Security

Arguably the most glaring security flaw here is the use of the symbolic evaluation library `SymPy`.

- We are essentially passing the information we want to conceal in plain sight
- If we were to instead numerically evaluate at every step of the encryption and decryption process, then we would simply have real numbers as entries for both vectors `encrypt` and `decrypt`.
- It may be possible to ask `SymPy` to somehow convert $t \in \mathbb{R}$ to a scalar r times some exponential e^k for $k \in \mathbb{Z}$ such that $t = r \cdot e^k$, but we were unable to find any built in function to accomplish this.

Analysis & Security

The paper does employ a permutation matrix to permute the entries of the encrypted vector in hopes of increasing security. A permutation matrix P is comprised of:

- The basis vectors $e_1, e_2, \ldots, e_n$ as columns but in differing order than that of the identity matrix $\mathbb{I}_n$ (where the orderings are $e_1, e_2, \ldots e_n$ for $\mathbb{I}_n \in \mathbb{R}^n$) [2].
- Since the permutation matrices P are also orthogonal matrices, then we enjoy all the same properties described above.

However, due to both limited time and the potential exploitation of the symbolic package `SymPy`, the permutation matrices P were not implemented.

Analysis & Security

Lastly, the performance of our crypto-system was severely lacking.

- To encode and decode our sample text found in the Appendix, run-time was measured at 6 minutes and 57 seconds which is $\sim$ .84 seconds per word. Removing all `print` statements shaved one minute off the run-time.
- It's generally well known that matrix operations can be carried out extremely fast on computers due to the large amount of work done to optimize such routines (such as `BLAS` [1]).
- We can speculate and claim this poor performance to be an artifact of the symbolic computations, in which extra overhead to compute quantities like the coefficients of each exponential may be unnecessarily long [3].
- We can confirm this speculation since implementing all matrix methods from `Numpy` resulted in run times of less than 1 second, fully indicating that `SymPy` was indeed the culprit behind poor performance.

Conclusion

While Santana does present a novel system based on orthogonal matrices, the text lacks a cohesive explanation for many of the processes that could potentially satisfy the stringent security required for a working crypto-system. Additionally, symbolic computations and poor performance hinder the system to be robust in almost all aspects.

References

[1] *BLAS Support*. URL: https://www.gnu.org/software/gsl/doc/html/blas.html.

[2] Ron Larson. *Elementary Linear Algebra*. 8th ed. Brooks Cole, 2016. ISBN: 1305658000-9781305658004.

[3] *Numeric Computation*. URL: https://docs.sympy.org/latest/modules/numeric-computation.html.

[4] Yeray Cachon Santana. *Orthogonal Matrix in Cryptography*. 2014. URL: https://arxiv.org/abs/1401.5787.

[5] Gilbert Strang. *Linear Algebra and Its Applications*. 4th. Brooks Cole, 2005. ISBN: 0030105676,9780030105678.