Elementary Number Theory Honors Project: Cryptography

Joe Cloud

November 28, 2016

1 Introduction

Today's computer security relies on encryption algorithms such as RSA and AES. RSA, as covered in chapter 5 of 'Number Theory Through Inquiry' is based on concepts we've developed in class. I wrote a web application that acts as an encryptor/decryptor based on theorems from the book. I will cover the details of how I implemented it, the proofs behind, and additional details such as the code itself and challenges I ran into in implementation.

2 Number Theory

2.1 Public Key Cryptography

This project is built upon the concepts of Public Key Cryptography (PKC) which exploits a computational limitation- factoring is a computationally difficult task. PKC is a system that uses two keys, a public and private key (which are related by two primes- denoted as p and q). The public key can be used to encrypt a message but only the private key can decrypt it.This provides a clever way of masking data- which becomes ciphertext upon encryption. RSA is a popular encryption method based on this idea of a public key that allows secure messaging to take place across a network and then decrypted once it is processed by a server internally.

2.2 Proofs

5.1. Theorem. *If p and q are distinct prime numbers and W is a natural number with (W, pq) = 1, then* $W^{(p-1)(q-1)} \equiv 1 \pmod{pq}$.

5.2. Theorem. Let p and q be *distinct primes*, k be a *natural number*, and W be a *natural number less than pq*. Then

$$W^{1+k(p-1)(q-1)} \equiv W(modpq).$$

5.3. Theorem. *Let p and q be distinct primes and E be a natural number relatively prime to* $(p-1)(q-1)$. *Then there exist natural numbers D and y such that*

$$ED = 1 + y(p-1)(q-1).$$

5.4. Theorem. *Let p and q be distinct primes, W be a natural number less than pq, and E, D, and y be natural numbers such that ED = 1 + y(p − 1)(q − 1). Then*

$$W^{ED} \equiv W(modpq).$$

2.3 Implemented scheme

Let $p = 2^{3217} - 1$ and $q = 2^{4253} - 1$, both of which are Mersenne primes.
also $plaintext = p$ and $ciphertext = c$
Therefore $\varphi(pq) = (p-1)(q-1)$, and $n = pq$, the RSA key is 7470 bits.
Encryption scheme:

1. Plaintext file passed through to server.
2. Server decodes plaintext files and converts each character to numeric representation.
3. Numeric plaintext is then passed through encryptor function.
4. Encryptor function randomly picks e between 0 and $\varphi(pq)$ that is relatively prime to $\varphi(pq)$.
5. Encryptor function determines ciphertext, $c = p^e(modn)$
6. c and e are returned to user in form of files.

Decryption scheme:

1. Ciphertext file and key passed to server.
2. Server decodes ciphertext and key from utf-8.
3. Ciphertext and key passed to decryptor function.
4. Decryptor determines what d by Extended Euclidean Algorithm.
5. Decryptor decodes ciphertext by $decoded\_message = c^d(modn)$
6. Decrypted message returned to user in form of file.

3 Programming

3.1 Overview

The application is mainly written in Python and HTML. Flask/Flask-WTF were instrumental in connecting the web page to the program running on the back-end handling input/output of file. Using a home server that I have routed through a localhost tunnel linked to a website- I was able to make the python publicly accessible. This was a factor upon deciding on using Python/Flask as the two combined would make bringing the program to fruition in a much more practical way.

3.2 Technical Challenges

There were a few challenges in implementing an RSA-like algorithm. The biggest challenge was figuring out how to allow users to input files with general text as input as opposed to relying on numeric input (this significantly limited the functionality). This was later solved by insuring that the files were properly decoded and converted to a utf-8 numeric equivalent of the characters. Once processed this way, the encrypted output consisted of a string of numbers that would then be easier to input later when it was time to decrypt.
Another challenge was the length of an encryptable string, this was later resolved when the base primes were changed to larger values. This is made obvious by one of the requirements of the algorithm (which is w $<$ pq).
Another issue was figuring out how to practically implement the outputs in a way that would be easy to generate as well as access once processed. This was remedied by redirecting users to separate pages for both encryption and decryption of files.
An issue that impacted the first revision was a bug when converting the characters to a binary numeric representation, the binary string would sometimes have a leading 0, which isn't hugely important for the text, but is important when converting back to a human readable format after processing as the missing 0 would offset blocks in the conversion. In more detail- lets take the following example (these values are made up for the purposes of the example).
Let's say the letter X = 0100, X is in a string of other letters, so it looks something like this

010010101001010101. The 4 Most Significant Bits (MSB) represent X. When processed as an integer, Python discards the leading 0, which leaves us with 10010101001010101. This becomes an issue after processing and when attempting to convert back as the missing 0 offsets the rest of the conversion process, you are then converting a different value depending on the direction of the conversion (MSB to LSB, or LSB to MSB).

3.3 How to run the program

The appendix contains a brief overview to running the program, here are more detailed steps.

3.3.1 Executing the code locally

To run the code on a Linux-based machine:

```
$ pip3 install --upgrade pip #insures pip is updated
$ # This will install the necessary 3rd party packages
$ pip3 install flask
$ pip3 install flask-wtf
$ pip3 install WTForms
$ # Download the code from the github repository, be sure to be in the save directory
$ git clone https://github.com/binarysaurus/Number-Theory-PKC-Project.git
$ cd Number-Theory-PKC-Project
$ python3 app.py
$ # Navigate to localhost:12345 on your web browser!
```

3.3.2 Accessing the web server

In order to use the current working site, please visit `crypto.joe.cloud`

3.4 Quick script that tries to hack the key

A Wrote a short prime generator to use in factoring the original prime. This is unlikely to be done by the end of next semester (strictly generating the primes necessary). It is unlikely this section will see any expansion. Brute-forcing 768 bit keys takes an immense amount of resources let alone 7470 bit.

4 Concluding Remarks

The discussion from the following subsections will have a lot of related ideas as they build upon each other greatly.

4.1 What was learned

A lot was learned about every aspect of the project, the Number Theory, the programming and languages, as well as the Markup in HTML and LaTeX.
There were challenges in representing the data in a way that it could easily be processed- this was easily the most challenging aspect of the project.
A lot was learned, and in that a lot that can be improved has been discovered.

4.2 How can it be improved

There are some fundamental issues with my implementation of the project. The project relied on encryption/decryption of the files being handled server-side, which although alleviates resource hogging on client machine for computing encrypted data- also means that at some point the data must be transmitted in plaintext to the server, since the server is not (currently) running on HTTPS, this plaintext attachment could be intercepted in transmission on unsecured networks.
In a ideal situation, the application would be run internally to prevent this risk- or at the minimum make use of HTTPS. Performance is a serious issue for larger files. Since the RSA key is so large it can take minutes for multi-line files to process, as well as the files that are generated by encrypted

the document are orders of magnitude larger than the plaintext documents.
There are two quick solutions to this, either limit the file size of documents that can be encrypted to a size that makes processing time reasonable, or- decrease the key size. The issue with the latter option is that it also restricts the length of a single line of a plaintext document (documents are encrypted line by line), since $w < pq$. The user interface can be significantly improved. A cleaner, more intuitive design would greatly polish the project.

4.3 Future goals

This project has made me much more curious to the inner workings of computer encryption as well as how to exploit vulnerabilities in it- particularly through weak implementations of familiar concepts from Number Theory, the use of small primes, and what other issues can arise from it.
In continuing to develop the project and refining my skills in all topics necessary- I'd like to do the following:

- Switch to HTTPS for secure transfer.
- Solve processing time issue for large files.
- Improve UI, condense everything to one page.
- Provide some live details as encryption is in process, this is functional via the CLI but it would be neat to have it work on the page as well.
- Refactoring the code to clean up remnants from tests, and work on catching possible glitches (it is error prone and minimal error handling was incorporated).
- Allow users to choose an encryption method, more control on things such as key to use, primes, etc.

5 Appendix

5.1 System Requirements

In order to access the online web application, a user will need a browser with internet access. To run the program on a personal machine, preferably a *nix based system with Python installed, as well as Flask, Flask-WTF, and WTForms.

5.2 Accessing the server

To access the web page, please navigate to `crypto.joe.cloud` In the case that the server is not accessible, please contact Joe Cloud. To test the program, input a number and click the first submit button, this will generate the key and cipher text that can be used later to decrypt. In current revisions you must clear the input box for encryption before decrypting.

5.3 Code

5.3.1 Main Python program

```
#This is the main app.py file, this runs the webserver

from fractions import gcd
from random import randint, randrange, getrandbits
import binascii
import string
from flask import Flask, render_template, flash, request, url_for, make_response, Response
from flask.ext.wtf import Form
from wtforms import Form, TextField, TextAreaField, validators, StringField, SubmitField
from wtforms.validators import Required
from itertools import repeat
from functools import reduce


DEBUG = True
app = Flask(__name__)
app.config.from_object(__name__)
app = Flask(__name__, static_url_path='/static')
app.config['SECRET_KEY'] = '7d441f27d441f27567d441f2b6176a'




#p and q are mersenne primes.
p = 2**3217-1
q = 2**4253-1
n = p * q
phi = (p-1)*(q-1)

def encryptor(m):
    e = randint(0,phi)

    while gcd(e, phi) != 1:
        e = randint(0,phi)

    c = pow(m,e,n)

    print("ORIGINAL BEFR: ", m)
    print("CIPHERTEXT ENC : ", c)

    return c, e

def decryptor(c, e):
    extend_ea = xgcd(e, phi)
    d = extend_ea[1]%phi
    decode = pow(c,d,n)

    print("ORIGINAL TEXT IN: ", decode)

    return decode

def is_prime(num):
    if num > 1:
        for i in range(2,num):
            if (num % i) == 0:
```

```
                return False;
                break
        else:
            return True;
    else:
        return False;

# EEA from:
# https://goo.gl/J1yvGF

def xgcd(b, n):
    x0, x1, y0, y1 = 1, 0, 0, 1

    while n != 0:
        q, b, n = b // n, n, b % n
        x0, x1 = x1, x0 - q * x1
        y0, y1 = y1, y0 - q * y1

    return  b, x0, y0

@app.route('/umessage', methods=["POST"])
def encryptmessage():
    file = request.files['data_file']
    if not file:
        return "No file"
    f_contents = file.readlines()
    ciphered = []
    keyed = []
    for line in f_contents:

        line = line.decode('utf-8')
        intbin_line = int(text_to_bits(line))
        print(intbin_line)
        print('now using encryptor function\n--------\n')
        enc_line = encryptor(intbin_line)
        ciphered.append(enc_line[0])
        keyed.append(enc_line[1])
    print('GENERATED CIPHER TEXT\n')
    encrypted_file = open('static/encrypted_text.txt', 'w')
    for cipher in ciphered:
        print(cipher)
        encrypted_file.write("%s\n" % cipher)
        print('\n')
    print('GENERATED KEYS\n')
    key_file = open('static/keys.txt', 'w')
    for key in keyed:
        print(key)
        key_file.write("%s\n" % key)
        print('\n')

    return render_template('download_enc.html')

@app.route('/dmessage', methods=["POST"])
def decryptmessage():
    cipher_text = request.files['data_file']
    key_text = request.files['key_file']
    if not cipher_text or not key_text:
        return "must upload both the cipher text, and the key file"
```

```
    ciph_cont = cipher_text.readlines()
    key_cont = key_text.readlines()
    assert len(ciph_cont) == len(key_cont), "Files have been tampered with"
    dec_list = []
    decrypted_file = open('static/decrypted_text.txt', 'w')
    for i in range(len(ciph_cont)):
        ciph_line = ciph_cont[i].decode('utf-8')
        key_line = key_cont[i].decode('utf-8')
        print(ciph_line)
        print(key_line)
        dec_line = decryptor(int(ciph_line), int(key_line))

        print(dec_line)
        dec_dec_line = text_from_bits(str(dec_line))
        print(dec_dec_line)
        decrypted_file.write("%s" % dec_dec_line)

    return render_template('download_dec.html')

@app.route("/", methods=['GET', 'POST'])
def main():
    form = ReusableForm(request.form)

    print(form.errors)

    if request.method == 'POST':
        if (request.form['str_encrypt']):
            str_encrypt=request.form['str_encrypt']

            print(str_encrypt)

            if form.validate():

                encrypted_data = encryptor(int(str_encrypt))
                flash('Encrypted Text: '+str(encrypted_data[0]))
                flash('Key:            '+str(encrypted_data[1]))
            else:
                flash('All the form fields are required. ')
        elif(request.form['str_decrypt']):
            str_decrypt=request.form['str_decrypt']
            str_key = request.form['str_key']

            print(str_decrypt)

            if form.validate():
                decrypted_data = decryptor(int(str_decrypt), int(str_key))
                flash('Decrypted Text: '+str(decrypted_data))
            else:
                flash('All the form fields are required. ')

    return render_template('index.html', form=form)



class ReusableForm(Form):
    str_encrypt = TextField('Number to encrypt :')
    str_decrypt = TextField('Encrypted text :')
```

```
    str_key     = TextField('Key of Encrypted :')


def text_to_bits(text, encoding='utf-8', errors='surrogatepass'):
    bits = bin(int(binascii.hexlify(text.encode(encoding, errors)), 16))[2:]

    return bits.zfill(8 * ((len(bits) + 7) // 8))

def text_from_bits(bits, encoding='utf-8', errors='surrogatepass'):
    n = int(bits, 2)

    return int2bytes(n).decode(encoding, errors)

def int2bytes(i):
    hex_string = '%x' % i
    n = len(hex_string)

    return binascii.unhexlify(hex_string.zfill(n + (n & 1)))

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=12345, debug=True)
```

5.3.2 HTML code for browsers

```
<!-this is the rendered form that constructs the main page when navigating to the site.->
<!DOCTYPE html>
<html lang="en">

<body>
        <h1>MATH 3307 - HONORS PROJECT - JOE CLOUD</h1>
        <h2>Fall 2016 - Section 001 (85596) - Epperson </h2>
        <p> This program will encrypt and decrypt simple text files.
    Keep in mind due to the strength of the key that files with multiple
    lines will potentially take 20-30 seconds to encrypt and decrypt.</p>
        <p> For the following file selector, choose a text file you'd like
    to encrypt, then submit.</p>
    <form action="/umessage" method="post" enctype="multipart/form-data">
        <input type="file" name="data_file" />
        <input type="submit" />
    </form>
    <p> To decrypt a file, please upload the ciphered text in the first box,
    and the key in the second box, then submit. </p>
     <form action="/dmessage" method="post" enctype="multipart/form-data">
        <input type="file" name="data_file" />
        <input type="file" name="key_file" />
        <input type="submit" />
    </form>
    <br><br>



<p> Below is the first version of the program, which can only encrypt numbers.
Please be aware that you must clear the 'Number to encrypt'
box before submitting text to decrypt. </p>
    <form action="" method="post">
        {{ form.csrf }}

        <div class="input text">
            {{ form.str_encrypt.label }} {{ form.str_encrypt }}
```

```
        </div>

        <div class="input submit">
            <input type="submit" value="Submit" />
        </div>

        <div class="input text">
            {{ form.str_decrypt.label }} {{ form.str_decrypt }}
        </div>
        <div class="input text">
            {{ form.str_key.label }} {{ form.str_key }}
        </div>

        <div class="input submit">
            <input type="submit" value="Submit" />
        </div>
    </form>

    {% with messages = get_flashed_messages(with_categories=true) %}
    {% if messages %}
    <ul>
        {% for message in messages %}
        <li>{{ message[1] }}</li>
        {% endfor %}
    </ul>
    {% endif %}
    {% endwith %}

</body>
</html>
```

References

[1] David Marshall, Edward Odell, and Michael Starbird. *Number Theory Through Inquiry*. Mathematical Association of America, 2006.

[2] Burt Kaliski. *The Mathematics of the RSA Public-Key Cryptosystem*. RSA Laboratories
www.mathaware.org/mam/06/Kaliski.pdf

[3] Wikibooks: Extended Euclidean Algorithm
en.wikibooks.org/wiki/Algorithm_Implementation/Mathematics/Extended_Euclidean_algorithm