Cryptography

DP Computer Science SL/HL

Name: ______________________________

Contents

Background

Ever since one human being has had to send a secret message to another, cryptography has been used. Even today, cryptography, or the art of writing secret messages, is at the core of much that we do throughout our daily lives. Online commerce and other forms of secure digital communication would likely be non-existent without the ability to encrypt (and, as necessary, decrypt) information passed between individuals so that a third-party could not intercept and read that information. In this lab, you will be implementing two different cryptographic techniques, including one that is used in modern digital communications.

Historical Encryption Techniques

There have been many cryptographic techniques developed over the years; however, many of the historical methods for encryption can be placed into one of two categories: substitution ciphers or transposition ciphers.

Substitution Ciphers

A substitution cipher is any encryption method which causes one letter in a message to be replaced by another, predetermined one. One such method, called the Caesar Cipher because of Julius Caesar's apparent use of it to encrypt important military messages, prescribes that each letter should be "shifted right" in value by a set number.

Caesar Cipher (Shift = 11)

"MEETAFTERSCHOOL" $\Longrightarrow$ "XPPELQEPCDNSZZW"

Note: "M" becomes "X" because "X" is 11 characters after "M" in the alphabet. Each letter is then shifted according to this same rule, wrapping around back to "A" as necessary.

Decrypting such a message is trivially done if the shift number is known: simply shift the letters to the left in the alphabet, again wrapping around back to "Z" as necessary.

Other substitution ciphers indicate that certain letters should be replaced by set other letters. The Kama Sutra Cipher, for instance, indicates that two keywords should be used to represent letter replacements.

Kama Sutra Cipher (Keywords: *OPULENT*, *DIARYS*)

"MEETAFTERSCHOOL" $\Longrightarrow$ "MRRSUFSRETCHDDA"

Note: Letters found in one word are replaced with their corresponding letters in the other word. Letters which do not appear in either word remain fixed.

Because each letter is replaced with its corresponding letter between the words, decrypting these messages requires the reader to substitute using the same keywords as was used during encryption.

Transposition Ciphers

A transposition cipher does not change any of the letters in the message. Instead, messages are "scrambled" so that the letters appear in a different location than in the original message. One such technique, called the Greek Scytale, involves winding a strip of parchment or leather around a rod of a specific diameter. After writing the message, the strip is then unwound to reveal a seemingly meaningless series of characters.

Scytale (3 letters around, 5 across)

M	E	E	T	A
F	T	E	R	S
C	H	O	O	L

When unwound, this message would reed: "MFCETHEEOTROASL". Note that there are no new letters in the encrypted message; however, the message has been sufficiently scrambled so as to not allow it to be easily read.

Decrypting this message requires the reader to know what size rod was used in its encryption. The message can then be wound around a similarly sized rod and the original message read back.

Evaluating Cryptographic Methods

Although the historical crytographic techniques are interesting, the advent of modern digital computers means they have become extremely vulnerable to attack. Over the last several decades, it has become clear that any new cryptographic techniques must be developed with the ever-increasing computational power of modern devices in mind. *Cryptanalysis* works to find any potential weaknesses in encryption methods. “Breaking” a cryptographic technique usually involves the study and analysis of the specific algorithms used in encrypting a message in the hopes that some method can produce, with high regularity, a decrypted message through the examination and analysis of the encrypted one.

The methods by which cryptanalysis evaluates an encryption algorithm for weaknesses, as well as the many other methods for attacking enciphered text are beyond the scope of this lab; however, it is an ever-growing field of study and one in which some students might find some interest in researching.

It is also important to note that many of the cryptographic techniques employed today rely on the sheer difficulty of “brute-force” decryption or the difficulty in solving complex mathematical problems. Despite this, they are, theoretically, breakable; however, the time it would take to break many of these methods using modern, or even speculated, computers far outpaces the viability of the information they are protecting. In other words, if it takes several thousand, million, or billion years to read encrypted information, the likelihood that it would be useful to do so becomes increasingly slim.

Applications

Question #1: Use a Caesar-Cipher with shift number 7 to encrypt the following message:

"THERE IS NO SCHOOL TODAY"

Question #2: Encrypt the same message in Question #1 using a Greek Scytale that allows for 4 characters to be written around the rod and 5 characters across.
Note: Do not include the spaces between words.

Question #3: Why are the substitution and transposition ciphers decscribed in the background particularly susceptible to attack using modern computers?

Question #4: The famed German Enigma machine used by the German forces in World War II was a modified form of substitution cipher. One big difference between it and the more primitive techniques is that each time a letter was substituted, the substitution sequence would change. In other words, the first letter 'A' might be replaced by a 'T'; however, a future 'A' could be replaced by a 'C', etc. Explain why this makes the encrypted message far more secure than a simple substitution cipher.

Activity #1

Introduction

A "one-time pad" is an historical method for encrypting messages. Using this method, physical pads or notebooks containing encryption keys or methods are distributed to any party that would need to read or write an encrypted message. These encryption keys would then be discarded after every use or after a designated period of time. As recently as World War II, these types of encryption systems were in wide-spread use by military and commercial organizations around the world. Even the famed German Enigma machine used settings that were changed daily in accordance to their own version of a one-time pad.

Although not considered secure enough for highly sensitive information, the following algorithm based on one-time encryption keys can be used to encrypt digital messages. A random key containing a certain number of bits (128 or 256 are common) is first generated and passed to anyone who would need to encrypt/decrypt a message using this method. A digital message is then encrypted by XORing each bit in the message with the corresponding bit in the key. A message that is longer than the generated key can be encrypted by first breaking it into chunks no more than the size of the key, encrypting each chunk, then concatenating all encrypted chunks together. Below is an example using an 8-bit key, a 16-bit key, and a 40-bit message (encoded in Extended ASCII).

Example: *message:* "HELLO", *key:* `10011101`

$$\textbf{HELLO} \Longrightarrow \left\{ \begin{array}{lcl} 01001000 & \oplus & 10011101 & = 11010101 \\ 01000101 & \oplus & 10011101 & = 11011000 \\ 01001100 & \oplus & 10011101 & = 11010001 \\ 01001100 & \oplus & 10011101 & = 11010001 \\ 01001111 & \oplus & 10011101 & = 11010010 \end{array} \right\} \Longrightarrow \textbf{ÕØÑÑÒ}$$

Example: *message:* "HELLO", *key:* `0110111000110101`

$$\textbf{HELLO} \Longrightarrow \left\{ \begin{array}{lcll} 0100100001000101 & \oplus & 0110111000110101 & = 0010011001110000 \\ 0100110001001100 & \oplus & 0110111000110101 & = 0010001001111001 \\ 01001111 & \oplus & 0110111000110101 & = 00100001 \end{array} \right\} \Longrightarrow \textbf{\&p"y!}$$

Note: Because our key is a binary *string*, the chunk of the "HELLO" message that does not contain a full 16-bits is XORed against the *left-most* bits in the string.

Decrypting a message is as simple as XORing it with the same key that was used for encryption, as seen below:

Example: *message:* "ÕØÑÑÒ", *key:* `10011101`

$$\textbf{ÕØÑÑÒ} \Longrightarrow \left\{ \begin{array}{lcll} 11010101 & \oplus & 10011101 & = 01001000 \\ 11011000 & \oplus & 10011101 & = 01000101 \\ 11010001 & \oplus & 10011101 & = 01001100 \\ 11010001 & \oplus & 10011101 & = 01001100 \\ 11010010 & \oplus & 10011101 & = 01001111 \end{array} \right\} \Longrightarrow \textbf{HELLO}$$

Example: *message:* "&p"y!", *key:* `0110111000110101`

$$\textbf{\&p"y!} \Longrightarrow \left\{ \begin{array}{lcll} 0010011001110000 & \oplus & 0110111000110101 & = 0100100001000101 \\ 0010001001111001 & \oplus & 0110111000110101 & = 0100110001001100 \\ 00100001 & \oplus & 0110111000110101 & = 01001111 \end{array} \right\} \Longrightarrow \textbf{HELLO}$$

Note: Because our key is a binary *string*, the chunk of the "&p"y!" message that does not contain a full 16-bits is XORed against the *left-most* bits in the string.

Exercise

Create the following methods in Python:

- The `generateRandomKey()` helper method which will generate a random binary string of the desired length.
- The `binaryToString()`, `stringToBinary()`, and `XORString()` helper methods useful for the encryption and decryption process.
- The `encryptMessage()` method.
- The `decryptMessage()` method.
  Note: This method should generate a new encryption key once a message has been successfully decrypted.

Questions

Question #5: For an encryption method such as this, why is it a good idea to use a key whose bit-length does *not* match the bit-length of your encoding scheme? (i.e., why is using an 8-bit key not desirable for Extended ASCII, an 8-bit encoding scheme?) You might want to use the examples from the introduction as inspiration for this answer.

Question #6: Why is it important that the encryption key being used with this method be kept *private*?

Activity #2

Introduction

In 1977, Ron Rivest, Adi Shamir, and Leonard Adleman, researchers at MIT, publicly described what is known as an assymmetric public-private key cryptosystem. Known as the RSA cryptosystem, this algorithm allows for a published, widely-distributed encryption key. The decryption key is held private and differs from that used to encrypt a message. It relies on two very important key facts: that testing whether or not a number is prime is "easy" and that prime factorization of a number is "hard".

Generating an RSA Key

The algorithm for generating the key for this method involves the choice of two distinct prime numbers. The steps are as follows:

1. Choose two distinct prime numbers, p and q.
2. Calculate $n = pq$.
3. Choose e relatively prime to $(p-1)(q-1)$.
4. Find d so that $ed \equiv 1 \bmod (p-1)(q-1)$.

The *public key* is then (e, n) and the *private key* is (d, n). Note that primality testing, choosing e, and how to calculate d are not part of this lab. A helper method has been provided for you for the calculation of d.

Encrypting a Message using RSA

Once the encryption key is known, encrypting a message is a fairly straight-forward mathematical procedure:

1. Calculate an integer representation, x, for the message.
   Note: For a traditional RSA cryptosystem, x *must be* smaller than n. More specifically, $\lfloor \lg x \rfloor < \lfloor \lg n \rfloor$. If using an encoding scheme, like Extended ASCII, this number can be calculated by concatenating the characters as a binary string, then converting that binary string to its decimal equivalent.
2. Calculate y such that $x^e \equiv y \bmod n$.

The value, y, is the encrypted message.

Decrypting a Message using RSA

The decryption algorithm is essentially the same as that for encrypting the message; however, the private key, d is now used as the exponent.

1. Calculate an integer representation, y, for the encrypted message.
2. Calculate x such that $y^d \equiv x \bmod n$.

x, or the reencoding of x using your encoding system is the decrypted message.

Example

Although practical RSA cryptosystems use very large prime numbers (with, say, 1024 bits), we can examine how it works to encrypt a message with a simplified set of numbers.

Using $p = 17837$, $q = 102881$ allows us to derive $n = 1835088397$, choose $e = 29$, and find $d = 1075670709$. Because $\lfloor \lg 1835088397 \rfloor \approx \lfloor 30.77 \rfloor = 30$, this RSA cryptosystem can encrypt any message containing 29 bits or less.

Encrypting "BYE" (Extended ASCII) using Public Key: $(1835088397, 29)$

$$\textbf{BYE} \Longrightarrow 010000100101100101000101 \Longrightarrow 4348229$$

$$4348229^{29} \equiv 372880434 \bmod 1835088397$$

$$372880434 \Longrightarrow 10110001110011011010000110010$$

Note: Unlike the one-time pad encryption method described and implemented in Activity #1, the RSA cryptosystem often produces an encrypted message with a higher bit-count than the original message.

Decrypting "10110001110011011010000110010" using Private Key: $(1835088397, 1075570709)$

$$\begin{aligned} 10110001110011011010000110010 &\Longrightarrow 37880434 \\ 37880434^{1075670709} &\equiv 4348229 \\ 4348229 &\Longrightarrow 010000100101100101000101 \\ 010000100101100101000101 &\Longrightarrow \textbf{BYE} \text{ (Extended ASCII)} \end{aligned}$$

Hopefully, this example will make clear that, even for simple versions of the RSA encryption method, we will be dealing with very large numbers. In particular, the calculations of 4348229^{29} and $37880434^{1075670709}$ prove problematic for our traditional exponentiation techniques. Luckily, a particularly clever algorithm for handling large exponents in applications such as these has been developed. A helper method has been provided for you which implements the *Modular Exponentiation* algorithm.

Exercise

Create the following methods in Python:

- The `binaryToString()` and `stringToBinary()` helper methods.
  Note: These methods should be identical to the ones from Exercise #1.
- The encryptMessage() method.
  Note: This method should only use the "public key" information: `(e, n)`.
- The decryptMessage() method.

Note: Methods for modular exponentiation and finding modular inverses have been provided for you.

Questions

Question #7: Why are *public key* encryption techniques important in modern communications?

Question #8: What major limitation does the fact that the decimal representation of a message must be less than n impose on sending messages using RSA? How would you overcome this limitation?

Final Analysis

Question #9: *Private key* encryption techniques, such as the Advanced Encryption System (AES), provide significant improvements on processing speed and can overcome the message-length problems RSA faces. Nonetheless, one major weakness, that the private encryption key must be exchanged over potentially unsecure communication lines, persists. Explain how *public key* encryption can be used to overcome this weakness.

Question #10: Private key encryption such as the RSA method described relies on the mathematical idea that multiplication is easy, but prime factorization is hard. Why might this be a weakness in light of recent computing innovations?

Question #11: Which method did you find the most challenging to create? Briefly explain why.

Question #12: What new programming techniques or knowledge did you learn as a result of this lab?

Template Class & Test Cases

Exercise #1

```
"""
Cryptography Lab, Exercise #1 (Template Methods and Test Cases)
These are the template methods and test cases for Exercise #1 of the Cryptography Lab.
Written for the Woodstock School in Mussoorie, Uttarakhand, India.

:author     Jeffrey Santos
:version    1.0
"""

# Variable for storing the current random key.
key = ''

"""
Helper method to generate a random binary string to act as a key.

:param bitLength:   The number of bits the binary string should contain.
:return:            A trandom binary string of bitLength bits.
"""
def generateRandomKey(bitLength):
    randomKey = ''
    # To be implemented...
    return randomKey

"""
Helper method to create an Extended ASCII encoded string from a given binary string.

:param msgBitString:    The binary string to encode.
    Precondition:       msgBitString is a binary string.
:return:                A string representing the binary string encoded in Extended ASCII.
"""
def binaryToString(msgBitString):
    msgString = ''
    # To be implemented...
    return msgString

"""
Helper method to create a binary string from a given string of characters.

:param msgString:       The message to convert to a binary string.
    Precondition:       msgString is encoded in Extended ASCII
:return:                A binary string representing the message string.
"""
def stringToBinary(msgString):
    msgBitString = ''
    # To be implemented...
    return msgBitString

"""
Helper method to XOR bits in a bit string against the corresponding encryption key.

:param  msgBitString:   The bit string to XOR.
    Precondition:       msgBitString is a binary string of length <= the lenght
                        of the key binary string.
    Precondition:       key has already been initialized with a random encryption key.
:return:                The resulting binary string.
"""
def XORString(msgBitString):
    xoredString = ''
    # To be implemented...
    return xoredString

"""
Encrypts the given message using the current encryption key.

:param msg:             The message to be encrypted.
    Precondition:       msg is encoded in Extended ASCII.
    Precondition:       key has already been initialized with a random encryption key.
:return:                The encrypted message encoded in Extended ASCII.
"""
def encryptMessage(msg):
    encryptedMsg = ''
```

```
    # To be implemented...
    return encryptedMsg

"""
Decrypts the given message using the current encryption key.

:param msg:             The message to be decrypted.
    Precondition:       msg is encoded in Extended ASCII.
    Precondition:       key has already been initialized with a random encryption key.
:return:                The decrypted message encoded in Extended ASCII.
    Postcondition:      key contains a newly generated random encryption key.
"""
def decryptMessage(msg):
    decryptedMsg = ''
    # To be implemented...
    return decryptedMsg

key = generateRandomKey(32)

# Test Case #1:
message = "Hello, World!"
encryptedMessage = encryptMessage(message)
decryptedMessage = decryptMessage(encryptedMessage)

print(message)
print(encryptedMessage)
print(decryptedMessage)

# Test Case #2:
# Note: Even though the original message is the same, the encrypted message should look different due to
#       the regeneration of the encryption key after decryption.
message = "Hello, World!"
encryptedMessage = encryptMessage(message)
decryptedMessage = decryptMessage(encryptedMessage)

print(message)
print(encryptedMessage)
print(decryptedMessage)
```

Exercise #2

```
"""
Cryptography Lab, Exercise #2 (Template Methods and Test Cases)
These are the template methods and test cases for Exercise #2 of the Cryptography Lab.
Written for the Woodstock School in Mussoorie, Uttarakhand, India.

:author     Jeffrey Santos
:version    1.0
"""

n, e, p, q = (0, 0, 0, 0)

"""
Helper method to create an Extended ASCII encoded string from a given binary string.

:param msgBitString:    The binary string to encode.
    Precondition:       msgBitString is a binary string.
:return:                A string representing the binary string encoded in Extended ASCII.
"""
def binaryToString(msgBitString):
    msgString = ''
    # To be implemented...
    return msgString

"""
Helper method to create a binary string from a given string of characters.

:param msgString:       The message to convert to a binary string.
    Precondition:       msgString is encoded in Extended ASCII
:return:                A binary string representing the message string.
"""
def stringToBinary(msgString):
    msgBitString = ''
    # To be implemented...
    return msgBitString

"""
Decrypts the message using RSA decryption.

:param  msg:            The message to decrypt.
:return:                The decrypted message.
"""
def decryptMessage(msg):
    decryptedMsg = ''
    # To be implemented...
    return decryptedMsg

"""
Encrypts the message using RSA encryption.

:param  msg:            The message to encrypt.
:return:                The encrypted message.
"""
def encryptMessage(msg):
    encryptedMsg = ''
    # To be implemented...
    return encryptedMsg

"""
Performs modular exponentiation.

:param  base:       The base for modular exponentiation.
:param  exponent:   The exponent for modular exponentiation.
:param  m:          The modulus for modular exponentiation.
:return:            The result of base^exponent mod m.
"""
def modExp(base, exponent, m):
    if exponent == 0:
        return 1
    temp = modExp(base, exponent / 2, m)
    if exponent % 2 == 0:
        return temp
    return base * temp

"""
```

```
Calculates the value, d, such that ed == 1 mod m using Euclid's Extended Algorithm.

:param  e:           The value for which an inverse is requested.
    Precondition:    e < m
:param  m:           The modulus for finding the inverse.
:return:             The value, d, such that ed == 1 mod m.
"""
def calcModInverse(e, m):
    modulus, x, y, buffer, quotient = m, 0, 1, 0, 0

    if m == 1:
        return 0

    while e > 1:
        quotient = e / m
        buffer = m
        m = e % m
        e = buffer

        buffer = x
        x = y - quotient * x
        y = buffer
    if y < 0:
        y = y + modulus
    return y

# Test Case:
(p, q, e) = (1013839, 67866833, 47)
n = p * q

message = "Test"
encryptedMessage = encryptMessage(message)
decryptedMessage = decryptMessage(encryptedMessage)

print(message)
print(encryptedMessage) # 1010100000111010110011111010110110111011110100
print(decryptedMessage)
```