Cryptography Final Project

INFR 3600U

Jesuloba Egunjobi

Nicolas Bermeo

Kevin Bonilla

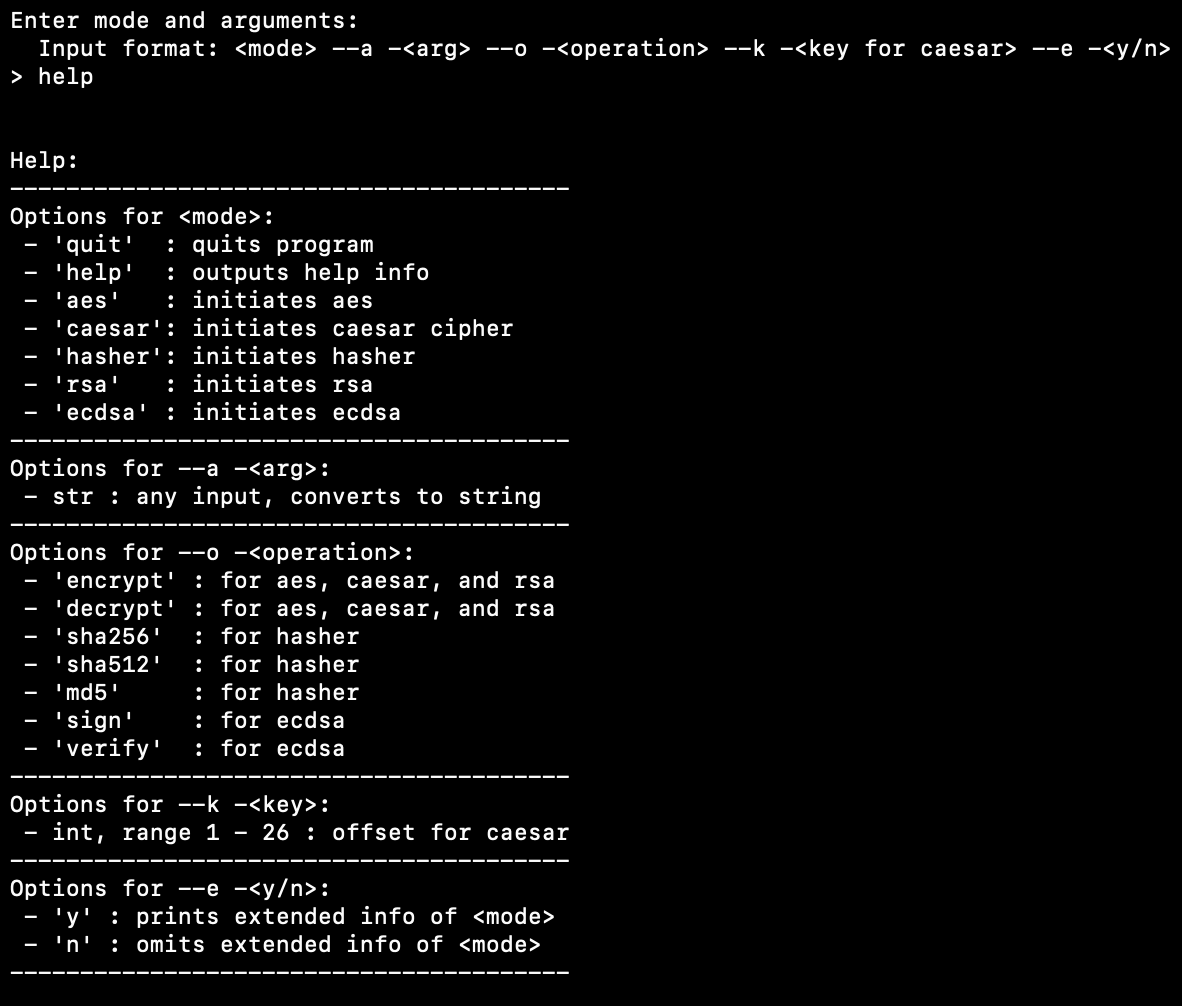
Jennifer Canizaro

# Menu Options

When the program is executed, users are initially greeted with a CryptoFP banner and the *Input Format* of the program. This format is broken down into steps based on the five (5) cryptographic tools at the user’s disposal:

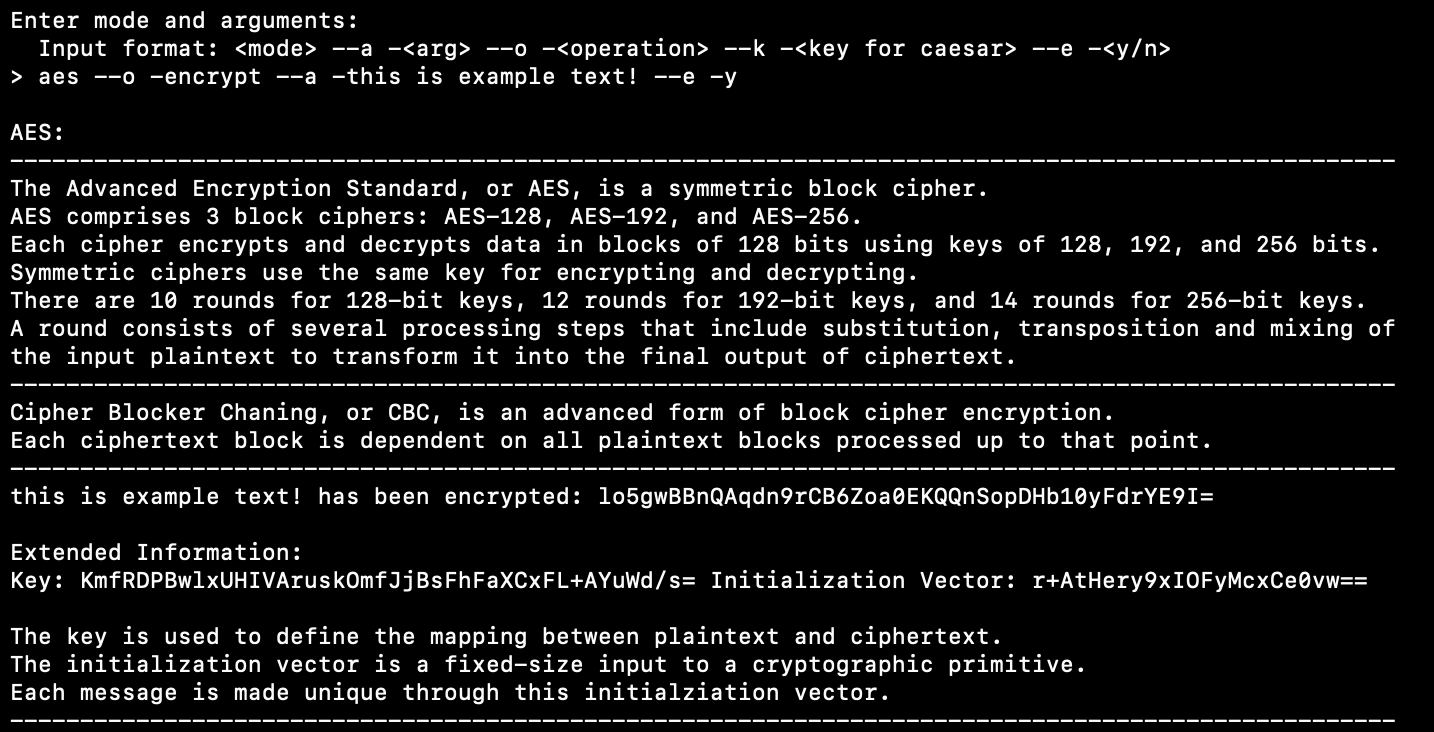
1. *AES*: Advanced Encryption Standard (AES) encryption & decryption
2. *RSA*: Rivest-Shamir-Adleman (RSA) encryption & decryption
3. *Caesar*: A Caesar Cipher
4. *Hasher*: Hashing Algorithms (SHA256, SHA512, MD5)
5. *ECDSA*: Elliptic Curve Digital Signature Algorithm (ECDSA)

Additionally, users can type *quit* as input to terminate the program, or *help* to access the menu below:



# Encryption / Decryption (AES & RSA)

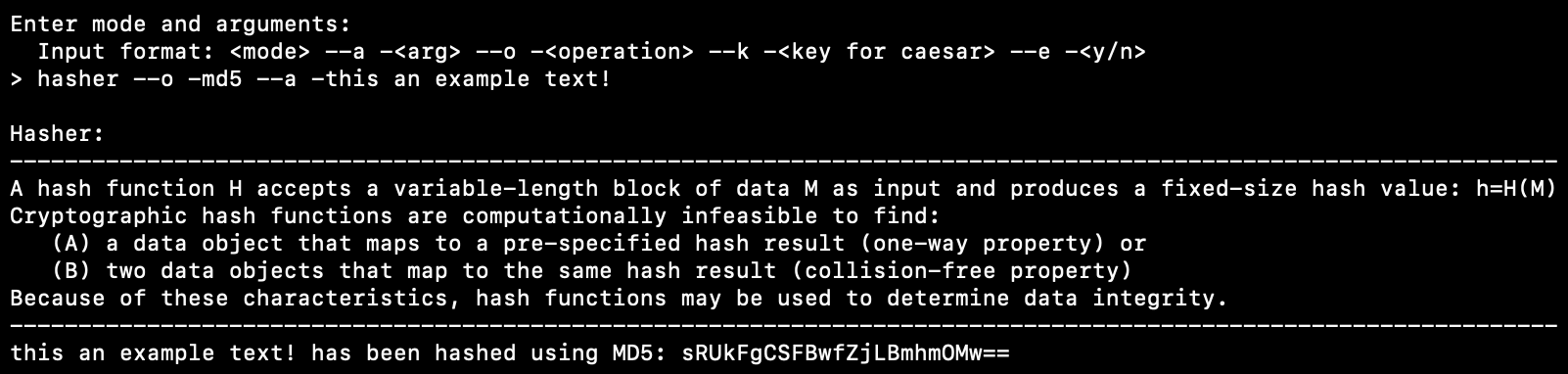
To implement AES or RSA, users must substitute <mode> with the text *aes* or *rsa*. Both are examples of encryption / decryption algorithms, which means that users will be required to enter a string argument to be encrypted or decrypted. In this example, we will encrypt the text “this is example text!” using AES, and by entering yes (--e -y) at the end of the statement, the user will be able to see more information about AES and its calculations. Below is the final input and calculation of AES for “this is example text!”:



PS – RSA generates a keypair when program the start so it can only decrypt ciphertext’s encrypted with the public key generated

# Hasher

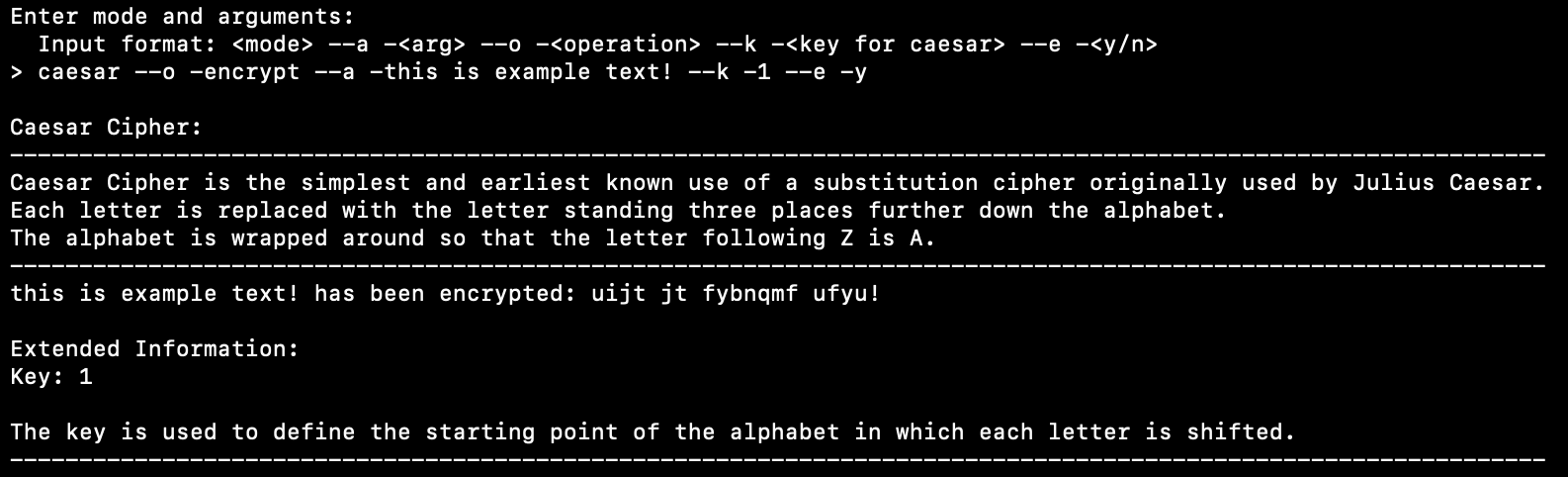
Next on the toolkit is the hashing function. Users have the option to choose from three (3) different hashing algorithms: (1) SHA256, (2) SHA512, (3) MD5. To implement any of these hashes, users must enter a string argument and an operation. As an example, we will use an MD5 hash to hash the message “this is example text!”:



PLEASE NOTE: extended information is not available for any of the hashing functions. All required information is explained with each calculation.

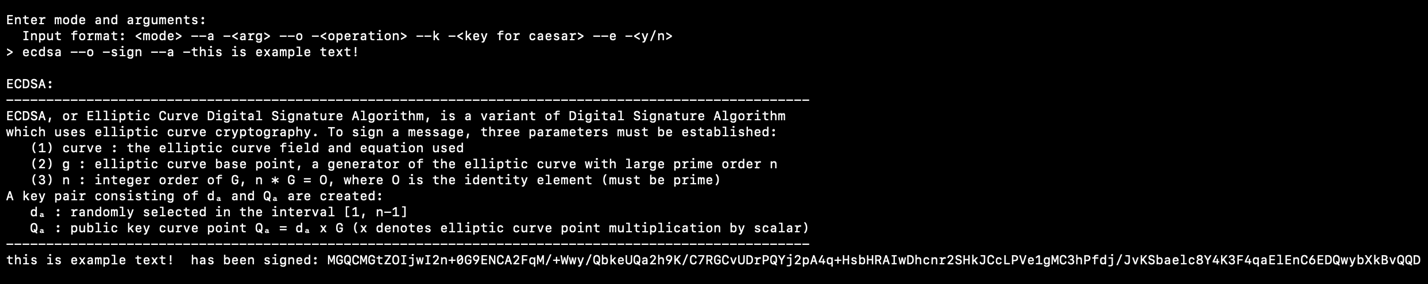
# Caesar Cipher

The Caesar Cipher is the simplest and earliest known use of a cipher. This is used to encrypt and decrypt text using keys, which can be various integer values. Let’s stick with “this is example text!” to test out this function, and we’ll add in a *key* argument this time. Keys for the cipher in this toolkit can be chosen from the range 1 – 26:



# Ellyptic Curve Digital Signature Algorithm (ECDSA)

With ECDSA, users have the option to *sign* or *verify* a string argument. To execute the ECDSA tool, users must substitute the <mode> with *ecdsa* and enter in a string argument (we will continue with the same example text). Next, users must select one of the two aforementioned options, *sign* or *verify*, as an operation, and may choose to include or omit extended information. Below is the last example we will cover in this manual:



PS – ECDSA generates a keypair when program the start so it can only verify signatures signed with the private key generated

# Appendix

# main.py

import argparse

from src.aes import AES

from src.caesar import Caesar

from src.hasher import Hasher

from src.rsa import RSA

from src.ecdsa import ECDSA

from src.desc import DESC

if \_\_name\_\_ == "\_\_main\_\_":

with open("banner.txt") as f:

print(f.read())

rsa = RSA()

ecdsa = ECDSA()

aes = AES()

while True:

parser = argparse.ArgumentParser(description='Script to learn basic argparse')

parser.add\_argument("cmd", choices=["quit", "help", "aes", "caesar", "hasher", "rsa", "ecdsa"], help='Command to execute', type=str)

parser.add\_argument("-a", "--arg", help="data to be encrypted", default=None)

parser.add\_argument("-o", "--operation", choices=["encrypt", "decrypt", "verify", "sign", "sha256", "sha512", "md5"], help="operation to perform", default=None)

parser.add\_argument("-k", "--key", choices=['1','2','3','4','5','6','7','8','9','10','11','12','13','14','15','16','17','18','19','20','21','22','23','24','25','26'], help="key/offset for caesar", default=None)

parser.add\_argument("-e", "--extended", help="extended info on function", default=None)

print('\nEnter mode and arguments:\n Input format: <mode> --a -<arg> --o -<operation> --k -<key for caesar> --e -<y/n>')

results = parser.parse\_args(str(input("> ")).split(' -'))

if results.cmd == "quit":

print("Quitting program...")

break

elif results.cmd == 'help':

print(

"\n\nHelp:\n"

"----------------------------------------\n"

"Options for <mode>:\n"

" - \'quit\' : quits program\n"

" - \'help\' : outputs help info\n"

" - \'aes\' : initiates aes \n"

" - \'caesar\': initiates caesar cipher\n"

" - \'hasher\': initiates hasher\n"

" - \'rsa\' : initiates rsa\n"

" - \'ecdsa\' : initiates ecdsa\n"

"----------------------------------------\n"

"Options for --a -<arg>:\n"

" - str : any input, converts to string\n"

"----------------------------------------\n"

"Options for --o -<operation>:\n"

" - \'encrypt\' : for aes, caesar, and rsa\n"

" - \'decrypt\' : for aes, caesar, and rsa\n"

" - \'sha256\' : for hasher\n"

" - \'sha512\' : for hasher\n"

" - \'md5\' : for hasher\n"

" - \'sign\' : for ecdsa\n"

" - \'verify\' : for ecdsa\n"

"----------------------------------------\n"

"Options for --k -<key>:\n"

" - int, range 1 - 26 : offset for caesar\n"

"----------------------------------------\n"

"Options for --e -<y/n>:\n"

" - \'y\' : prints extended info of <mode>\n"

" - \'n\' : omits extended info of <mode>\n"

"----------------------------------------\n"

)

elif results.cmd == "aes":

if results.operation == "encrypt" and results.arg != None:

aes.setData(results.arg)

aes\_desc = DESC(operation=results.operation, extended=results.extended, extdata1=aes.get\_key(), extdata2=aes.get\_iv(), extdata3=results.arg, extdata4=aes.encrypt())

aes\_desc.AES()

elif results.operation == "decrypt" and results.arg != None:

aes\_desc = DESC(operation=results.operation, extended=results.extended, extdata1=aes.get\_key(), extdata2=aes.get\_iv(), extdata3=results.arg, extdata4=aes.decrypt(results.arg))

aes\_desc.AES()

else:

print("Please pass data to be encrypted / decrypted in the --arg argument")

elif results.cmd == "caesar":

if results.operation == "encrypt" and results.arg != None:

caesar = Caesar(ct=results.arg,key=int(results.key))

caesar\_desc = DESC(operation=results.operation, extended=results.extended, extdata1=caesar.get\_key(), extdata3=results.arg, extdata4=caesar.encrypt())

caesar\_desc.Caesar()

elif results.operation == "decrypt" and results.arg != None:

caesar = Caesar(ct=results.arg,key=int(results.key))

caesar\_desc = DESC(operation=results.operation, extended=results.extended, extdata1=caesar.get\_key(), extdata3=results.arg, extdata4=caesar.decrypt())

caesar\_desc.Caesar()

else:

print("Please pass data to be encrypted / decrypted in the --arg argument and a valid key in the --key argument")

elif results.cmd == "hasher":

if results.operation == "sha256" and results.arg != None:

hasher = Hasher(data=results.arg)

hash\_desc = DESC(operation=results.operation, extended=results.extended,extdata3=results.arg,extdata4=hasher.SHA256())

hash\_desc.Hasher()

elif results.operation == "sha512" and results.arg != None:

hasher = Hasher(data=results.arg)

hash\_desc = DESC(operation=results.operation, extended=results.extended,extdata3=results.arg,extdata4=hasher.SHA512())

hash\_desc.Hasher()

elif results.operation == "md5" and results.arg != None:

hasher = Hasher(data=results.arg)

hash\_desc = DESC(operation=results.operation, extended=results.extended,extdata3=results.arg,extdata4=hasher.MD5())

hash\_desc.Hasher()

else:

print("Please pass data to be hashed in the --arg argument and a valid operation in the --operation argument")

elif results.cmd == "rsa":

if results.operation == "encrypt" and results.arg != None:

rsa.setData(results.arg)

rsa\_desc = DESC(operation=results.operation, extended=results.extended,extdata1=rsa.publicKey(),extdata2=rsa.privateKey(),extdata3=results.arg,extdata4=rsa.encrypt())

rsa\_desc.RSA()

elif results.operation == "decrypt" and results.arg != None:

rsa\_desc = DESC(operation=results.operation, extended=results.extended,extdata1=rsa.publicKey(),extdata2=rsa.privateKey(),extdata3=results.arg,extdata4=rsa.decrypt(results.arg))

rsa\_desc.RSA()

else:

print("Please pass data to be encrypted / decrypted in the --arg argument")

elif results.cmd == "ecdsa":

if results.operation == "sign" and results.arg != None:

ecdsa.setMessage(results.arg)

ecdsa\_desc = DESC(operation=results.operation,extended=results.extended,extdata1=ecdsa.publicKey(),extdata2=ecdsa.privateKey(),extdata3=results.arg,extdata4=ecdsa.sign())

ecdsa\_desc.ECDSA()

elif results.operation == "verify" and results.arg != None:

ecdsa\_desc = DESC(operation=results.operation,extended=results.extended,extdata1=ecdsa.publicKey(),extdata2=ecdsa.privateKey(),extdata3=results.arg,extdata4=ecdsa.verify(results.arg))

ecdsa\_desc.ECDSA()

else:

print("Please pass data to be signed / verified in the --arg argument")

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#aes.py

import os

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

from cryptography.hazmat.backends import default\_backend

from cryptography.hazmat.primitives import padding

from src.helpers import strToBytes, encodeBytes, decodeBytes

class AES:

"""

A class for encrypting and decrypting string data using AES symmetric encryption algorithm

Parameters

----------

data : str

data to be encrypted

"""

def \_\_init\_\_(self, data: str=None) -> None:

self.\_\_key = os.urandom(32)

self.\_\_iv = os.urandom(16)

self.\_\_data = data

self.\_\_cipher = Cipher(algorithms.AES(self.\_\_key), modes.CBC(self.\_\_iv), backend=default\_backend())

"""

Getter for key, iv, data, and cipher

"""

def get\_key(self):

return encodeBytes(self.\_\_key).decode('utf-8')

def get\_iv(self):

return encodeBytes(self.\_\_iv).decode('utf-8')

def get\_data(self):

return self.\_\_data

def get\_cipher(self):

return self.\_\_cipher

def setData(self, data):

self.\_\_data = data

"""

Encrypts provided data

Returns : bytes

-------

ciphertext of encrypted data

"""

def encrypt(self) -> str:

encryptor = self.\_\_cipher.encryptor()

return encodeBytes(encryptor.update(self.\_\_padData()) + encryptor.finalize()).decode('utf-8')

"""

Decrypts provded ciphertext

Parameters

----------

ct : bytes

ciphertext of encrypted data

Returns : str

-------

decrypted data

"""

def decrypt(self, ct) -> str:

decryptor = self.\_\_cipher.decryptor()

dt = decryptor.update(decodeBytes(ct.encode('utf-8'))) + decryptor.finalize()

return self.\_\_unpadData(dt).decode("utf-8")

"""

appends padding to the provided data

"""

def \_\_padData(self):

padder = padding.PKCS7(128).padder()

padded\_data = padder.update(strToBytes(self.\_\_data))

padded\_data += padder.finalize()

return padded\_data

"""

removes appended padding from the data

"""

def \_\_unpadData(self, dt):

unpadder = padding.PKCS7(128).unpadder() #message is now decrypted... but now need to unpad

data = unpadder.update(dt)

unpadded = data + unpadder.finalize()

return unpadded

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#caesar.py

from caesarcipher import CaesarCipher

class Caesar:

"""

A class for bruteforcing a caeser cipher ciphertext

Parameters

----------

ct : str

ciphertext to be bruteforced

"""

def \_\_init\_\_(self, ct, key: int=None):

self.\_\_ct = ct

self.\_\_key = key

"""

Getter for key

"""

def get\_key(self):

return self.\_\_key

"""

Returns : string

-------

encrypted value of ciphertext

"""

def encrypt(self) -> str:

cipher = CaesarCipher(self.\_\_ct, offset=self.\_\_key)

return cipher.encoded

"""

Returns : string

-------

decrypted value of ciphertext

"""

def decrypt(self) -> str:

cipher = CaesarCipher(self.\_\_ct, offset=self.\_\_key)

return cipher.decoded

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#descriptons.py

#Sources referenced:

"""

INFR-3600U Notes

https://thebestvpn.com/advanced-encryption-standard-aes/

https://datalocker.com/what-is-the-difference-between-ecb-mode-versus-cbc-mode-aes-encryption/

https://en.wikipedia.org/wiki/RSA\_(cryptosystem)

https://en.wikipedia.org/wiki/Elliptic\_Curve\_Digital\_Signature\_Algorithm

"""

class DESC():

"""

A class for providing information on each cryptographic function

Parameters

----------

mode : str

function to be explained

operation : str

function operation (encrypy/decrypt)

extended : chr

show extended information

extdata(1-4) : str

external data from other objects to be shown

"""

def \_\_init\_\_(self, operation, extended, extdata1: str=None, extdata2: str=None, extdata3: str=None, extdata4: str=None):

self.\_\_operation = operation

self.\_\_extended = extended

#External data

self.\_\_extdata1 = extdata1 #Ex. for AES = key

self.\_\_extdata2 = extdata2 #Ex. for AES = iv

self.\_\_extdata3 = extdata3 #Ex. for AES = results.arg

self.\_\_extdata4 = extdata4 #Ex. for AES = aes.encrypt() or .decrypt()

"""

Provides information about AES

"""

def AES(self):

print( #General info

"\nAES:\n"

"---------------------------------------------------------------------------------------------------\n"

"The Advanced Encryption Standard, or AES, is a symmetric block cipher.\n"

"AES comprises 3 block ciphers: AES-128, AES-192, and AES-256.\n"

"Each cipher encrypts and decrypts data in blocks of 128 bits using keys of 128, 192, and 256 bits.\n"

"Symmetric ciphers use the same key for encrypting and decrypting.\n"

"There are 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys.\n"

"A round consists of several processing steps that include substitution, transposition and mixing of\n"

"the input plaintext to transform it into the final output of ciphertext.\n"

"---------------------------------------------------------------------------------------------------\n"

"Cipher Blocker Chaning, or CBC, is an advanced form of block cipher encryption.\n"

"Each ciphertext block is dependent on all plaintext blocks processed up to that point.\n"

"---------------------------------------------------------------------------------------------------"

)

if self.\_\_operation == "encrypt":

print( #Encryption info

self.\_\_extdata3 ,"has been encrypted:", self.\_\_extdata4, "\n"

)

elif self.\_\_operation == "decrypt":

print( #Decryption info

self.\_\_extdata3 ,"has been decrypted:", self.\_\_extdata4, "\n"

)

if self.\_\_extended == 'y':

print( #Extended info

"Extended Information:\n"

"Key:", self.\_\_extdata1, "Initialization Vector:", self.\_\_extdata2, "\n\n"

"The key is used to define the mapping between plaintext and ciphertext.\n"

"The initialization vector is a fixed-size input to a cryptographic primitive.\n"

"Each message is made unique through this initialziation vector.\n"

"---------------------------------------------------------------------------------------------------\n"

)

"""

Provides information about Caesar Cipher

"""

def Caesar(self):

print( #General info

"\nCaesar Cipher:\n"

"---------------------------------------------------------------------------------------------------------------\n"

"Caesar Cipher is the simplest and earliest known use of a substitution cipher originally used by Julius Caesar.\n"

"Each letter is replaced with the letter standing three places further down the alphabet.\n"

"The alphabet is wrapped around so that the letter following Z is A.\n"

"---------------------------------------------------------------------------------------------------------------"

)

if self.\_\_operation == "encrypt":

print( #Encryption info

self.\_\_extdata3 ,"has been encrypted:", self.\_\_extdata4, "\n"

)

elif self.\_\_operation == "decrypt":

print( #Decryption info

self.\_\_extdata3 ,"has been decrypted:", self.\_\_extdata4, "\n"

)

if self.\_\_extended == 'y':

print( #Extended info

"Extended Information:\n"

"Key:", self.\_\_extdata1, "\n\n"

"The key is used to define the starting point of the alphabet in which each letter is shifted.\n"

"---------------------------------------------------------------------------------------------------------------\n"

)

"""

Provides information about hasher

"""

def Hasher(self):

print( #General info

"\nHasher:\n"

"-----------------------------------------------------------------------------------------------------------------\n"

"A hash function H accepts a variable-length block of data M as input and produces a fixed-size hash value: h=H(M)\n"

"Cryptographic hash functions are computationally infeasible to find:\n"

" (A) a data object that maps to a pre-specified hash result (one-way property) or\n"

" (B) two data objects that map to the same hash result (collision-free property)\n"

"Because of these characteristics, hash functions may be used to determine data integrity.\n"

"-----------------------------------------------------------------------------------------------------------------"

)

if self.\_\_operation == "sha256":

print( #SHA256 info

self.\_\_extdata3 ,"has been hashed using SHA256:", self.\_\_extdata4, "\n"

)

elif self.\_\_operation == "sha512":

print( #SHA512 info

self.\_\_extdata3 ,"has been hashed using SHA512:", self.\_\_extdata4, "\n"

)

elif self.\_\_operation == "md5":

print( #MD5 info

self.\_\_extdata3 ,"has been hashed using MD5:", self.\_\_extdata4, "\n"

)

if self.\_\_extended == 'y':

print( #Extended info

"Extended Information:\n"

"No extended information available.\n"

"-----------------------------------------------------------------------------------------------------------------\n"

)

"""

Provides information about RSA

"""

def RSA(self):

print( #General info

"\nRSA:\n"

"-------------------------------------------------------------------------------------------\n"

"RSA ia an asymmetric algorithm, which uses two different keys: a public and private key.\n"

"Keys are generated through these steps:\n"

" (1) Two different large random prime numbers p and q are chosen\n"

" (2) The modulus of the public and private keys n is computed through p\*q\n"

" (3) The totient of n is computed through (p-1)\*(q-1)\n"

" (4) An integer e is selected such that 1 < e < totient(n)\n"

" - e is released as the public key exponent\n"

" (5) A value d is computed to satisfy the congruence relation d\*e ≡ 1 (modulus totient(n))\n"

" - d is kept as the private key exponent\n"

"The public key is made of n^e and the private key is made of n^d.\n"

"-------------------------------------------------------------------------------------------"

)

if self.\_\_operation == "encrypt":

print( #Encryption info

self.\_\_extdata3 ,"has been encrypted:", self.\_\_extdata4, "\n"

)

elif self.\_\_operation == "decrypt":

print( #Decryption info

self.\_\_extdata3 ,"has been decrypted:", self.\_\_extdata4, "\n"

)

if self.\_\_extended == 'y':

print( #Extended info

"Extended Information:\n"

"Public Key:", self.\_\_extdata1, "Private Key:", self.\_\_extdata2, "\n\n"

"These keys are used in encryption/decryption through RSA.\n"

"For encryption, message M is turned into a number m smaller than n by using an agreed-upon\n"

"reversible protocol known as a padding scheme. Ciphertext c is computed as c=m^e mod n\n"

"For decryption, m can by recovered from c using private key d: m=c^d mod n\n"

"With m, the original distinct prime numbers can be recovered: m^e\*d ≡ m mod p\*q\n"

"So, c^d ≡ m mod n\n"

"-------------------------------------------------------------------------------------------\n"

)

"""

Provides information about ECDSA

"""

def ECDSA(self):

print( #General info

"\nECDSA:\n"

"----------------------------------------------------------------------------------------------------\n"

"ECDSA, or Elliptic Curve Digital Signature Algorithm, is a variant of Digital Signature Algorithm\n"

"which uses elliptic curve cryptography. To sign a message, three parameters must be established:\n"

" (1) curve : the elliptic curve field and equation used\n"

" (2) g : elliptic curve base point, a generator of the elliptic curve with large prime order n\n"

" (3) n : integer order of G, n \* G = O, where O is the identity element (must be prime)\n"

"A key pair consisting of dₐ and Qₐ are created:\n"

" dₐ : randomly selected in the interval [1, n-1]\n"

" Qₐ : public key curve point Qₐ = dₐ x G (x denotes elliptic curve point multiplication by scalar)\n"

"----------------------------------------------------------------------------------------------------"

)

if self.\_\_operation == "sign":

print ( #Signature info

self.\_\_extdata3 ,"has been signed:", self.\_\_extdata4, "\n"

)

elif self.\_\_operation == "verify":

print( #Verification info

self.\_\_extdata3 ,"is verified:", self.\_\_extdata4, "\n"

)

if self.\_\_extended == 'y':

print( #Extended info

"Extended Information:\n"

"Public Key:", self.\_\_extdata1, "Private Key:", self.\_\_extdata2, "\n\n"

"To sign a message m:\n"

" (1) Calculate e = hash(m)\n"

" (2) Let z be the Lₙ leftmost bits of e, where Lₙ is the bit length of the group order n\n"

" (3) Choose a cryptographycally secure random integer k from [1, n-1]\n"

" (4) Calculate curve point (x1, y1) = k x G\n"

" (5) Calculate r = x1 mod n (If r = 0, go to step 3)\n"

" (6) Calculate s = k^-1(z + r\*dₐ) mod n (If s = 0, go to step 3)\n"

" (7) The signature is the pair (r,s)\n"

"To verify a signature:\n"

" (1) Verify r and s are integers in [1, n-1]\n"

" (2) Calculate e = hash(m)\n"

" (3) Let z be the Lₙ leftmost bits of e\n"

" (4) Calculate w = s^-1 mod n\n"

" (5) Calculate u1 = z\*w mod n and u2 = r\*w mod n\n"

" (6) Calculate the curve point (x1,y2) = u1 x G + u2 x Qₐ (if (x1,y2) = O, signature is invalid)\n"

" (7) The signature is only valid if r ≡ x1 (mod n)\n"

"----------------------------------------------------------------------------------------------------\n"

)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#ecdsa.py

from cryptography.hazmat.backends import default\_backend

from cryptography.hazmat.primitives import hashes, serialization

from cryptography.hazmat.primitives.asymmetric import ec

from cryptography.exceptions import InvalidSignature

from src.helpers import encodeBytes, decodeBytes, strToBytes

class ECDSA:

"""

A class for digital signature using Elliptic curve cryptography

Parameters

----------

message : str

message to be signed / verified

"""

def \_\_init\_\_(self, message: str=None):

self.\_\_message = message

self.\_\_private = ec.generate\_private\_key(

ec.SECP384R1(), default\_backend()

)

self.\_\_public = self.\_\_private.public\_key()

"""

Getter for private and public keys

"""

def publicKey(self):

return self.\_\_public.public\_bytes(

encoding=serialization.Encoding.PEM,

format=serialization.PublicFormat.SubjectPublicKeyInfo

).decode('utf-8')

def privateKey(self):

return self.\_\_private.private\_bytes(

encoding=serialization.Encoding.PEM,

format=serialization.PrivateFormat.TraditionalOpenSSL,

encryption\_algorithm=serialization.NoEncryption()

).decode('utf-8')

def setMessage(self, message):

self.\_\_message = message

"""

Returns : str

-------

signature of a message

"""

def sign(self) -> str:

return encodeBytes(self.\_\_private.sign(strToBytes(self.\_\_message), ec.ECDSA(hashes.SHA256()))).decode("utf-8")

"""

Parameters

----------

sig : str

signature to verify

Returns : bool

-------

True of False depending if verification passes

"""

def verify(self, sig: str) -> bool:

try:

self.\_\_public.verify(decodeBytes(sig.encode("utf-8")), strToBytes(self.\_\_message), ec.ECDSA(hashes.SHA256()))

return True

except InvalidSignature:

return False

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#hasher.py

from src.helpers import strToBytes, encodeBytes

from cryptography.hazmat.backends import default\_backend

from cryptography.hazmat.primitives import hashes

class Hasher:

"""

A class for hashing string data

Parameters

----------

data : str

data to be encrypted

"""

def \_\_init\_\_(self, data):

self.\_\_data = data

"""

Returns : str

-------

SHA256 hash of data

"""

def SHA256(self):

digest = hashes.Hash(hashes.SHA256(), backend=default\_backend())

digest.update(strToBytes(self.\_\_data))

return encodeBytes(digest.finalize()).decode("utf-8")

"""

Returns : str

-------

SHA512 hash of data

"""

def SHA512(self):

digest = hashes.Hash(hashes.SHA512(), backend=default\_backend())

digest.update(strToBytes(self.\_\_data))

return encodeBytes(digest.finalize()).decode("utf-8")

"""

Returns : str

-------

MD5 hash of data

"""

def MD5(self):

digest = hashes.Hash(hashes.MD5(), backend=default\_backend())

digest.update(strToBytes(self.\_\_data))

return encodeBytes(digest.finalize()).decode("utf-8")

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#helpers.py

import base64

"""

converts string data to bytes

"""

def strToBytes(string):

return string.encode("utf-8")

"""

encodes bytes data to base64

"""

def encodeBytes(data):

return base64.b64encode(data)

"""

decodes bytes data from base64

"""

def decodeBytes(data):

return base64.b64decode(data)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

from cryptography.hazmat.backends import default\_backend

from cryptography.hazmat.primitives.asymmetric import rsa, padding

from cryptography.hazmat.primitives import hashes, serialization

from src.helpers import strToBytes, encodeBytes, decodeBytes

class RSA:

"""

A class for encrypting and decrypting string data using RSA asymmetric encryption algorithm

Parameters

----------

data : str

data to be encrypted

"""

def \_\_init\_\_(self, data: str=None):

self.\_\_private = rsa.generate\_private\_key(

public\_exponent=65537,

key\_size=2048,

backend=default\_backend()

)

self.\_\_public = self.\_\_private.public\_key()

self.\_\_data = data

"""

returns generated public key

"""

def publicKey(self):

return self.\_\_public.public\_bytes(

encoding=serialization.Encoding.PEM,

format=serialization.PublicFormat.SubjectPublicKeyInfo

).decode('utf-8')

"""

returns generated private key

"""

def privateKey(self):

return self.\_\_private.private\_bytes(

encoding=serialization.Encoding.PEM,

format=serialization.PrivateFormat.TraditionalOpenSSL,

encryption\_algorithm=serialization.NoEncryption()

).decode('utf-8')

"""

setter for data private variable

"""

def setData(self, data):

self.\_\_data = data

"""

Encrypts provided data

Returns : bytes

-------

ciphertext of encrypted data

"""

def encrypt(self) -> bytes:

return encodeBytes(self.\_\_public.encrypt(strToBytes(self.\_\_data), padding.OAEP(

mgf=padding.MGF1(algorithm=hashes.SHA256()),

algorithm=hashes.SHA256(),

label=None

))).decode('utf-8')

"""

Decrypts provded ciphertext

Parameters

----------

ct : bytes

ciphertext of encrypted data

Returns : str

-------

decrypted data

"""

def decrypt(self, ct):

return self.\_\_private.decrypt(decodeBytes(ct.encode('utf-8')), padding.OAEP(

mgf=padding.MGF1(algorithm=hashes.SHA256()),

algorithm=hashes.SHA256(),

label=None

)).decode("utf-8")