A Code

A.1 Back-end Code

```
# David Budnitsky
# 20453508

import numpy as np
import u20453508_Prac_3_RC4 as RC4


# region helperFunction
def circularRightShift(num, shifts, numBits=64):
    """
    Right circular right-bit shit
    :param numBits:
    :param num:
    :param shifts:
    :return:
    """
    return (num >> shifts) | (num << (numBits - shifts)) & (int(2 **
   numBits) - 1)


def circularLeftShift(num, shifts, numBits=64):
    """
    Left circular bit shift
    :param numBits:
    :param num:
    :param shifts:
    :return:
    """
    return (num << shifts) | (num >> (numBits - shifts))
# endregion helperFunction


# region sha
def sha_Preprocess_Message(inputHex: str) -> str:
    """
    Takes in a hex input and pads it according to the SHA standard
    :param inputHex: Input message as a hex-string, no padding
    :return: The hex-string of the padded input
    """
    messageLen = len(inputHex) * 4
    inputBin = bin(int(inputHex, 16))[2:].zfill(messageLen)

    k = (896 - messageLen - 1) % 1024

    padding = '1' + '0' * k + bin(messageLen)[2:].zfill(128)

    ans = inputBin + padding
    ansLen = len(ans) // 4

    ans = int(ans, 2)
    ans = hex(ans)[2:].zfill(ansLen)
    return ans
```

```
52
53
54  def sha_Create_Message_Blocks(inputHex: str) -> np.ndarray:
55      """
56      Breaks the message into blocks of 1024 bits, 256 hits per block
57      :param inputHex: Preprocessed inputHex hex string
58      :return: np array of hex strings, each with 256 characters
59      """
60      ans = np.array([inputHex[k:k + 256] for k in range(0, len(inputHex),
        256)])
61      return ans
62
63
64  def sha_Message_Schedule(inputHex: str) -> np.ndarray:
65      """
66      Makes the message schedule for a block from the inputHex.
67      The first 16 message schedule pieces use 64-bit (16 hit) pieces of the
         message block
68      :param inputHex: Input hex value to make the 80 message words from.
        This should always have a length of 1024.
69      :return: Array of 80 words
70      """
71      W = [inputHex[k:k + 16] for k in range(0, len(inputHex), 16)]
72      for k in range(16, 80):
73          thisW = [W[k - t] for t in (16, 15, 7, 2)]
74          temp = int(thisW[0], 16) + int(thisW[2], 16)
75
76          x = int(thisW[1], 16)
77          x1 = circularRightShift(x, 1)
78          x2 = circularRightShift(x, 8)
79          x3 = x >> 7
80          temp1 = (x1 ^ x2 ^ x3)
81
82          x = int(thisW[3], 16)
83          x1 = circularRightShift(x, 19)
84          x2 = circularRightShift(x, 61)
85          x3 = x >> 6
86          temp2 = (x1 ^ x2 ^ x3)
87
88          temp = temp + temp1 + temp2
89          temp = temp % int(2 ** 64)
90          temp = hex(temp)[2:].upper().zfill(16)
91
92          W.append(temp)
93
94      W = np.array(W)
95      return W
96
97
98  def sha_Hash_Round_Function(messageWordHex: str, aHex: str, bHex: str,
        cHex: str, dHex: str, eHex: str, fHex: str,
99                              gHex: str, hHex: str, roundConstantHex: str)
        -> tuple:
100     """
101     Performs the Hash round function for SHA. This is seen in figure 11.11
         in the textbook.
102     pdf page 361.
```

```
    :param messageWordHex: Self-explanatory
    :param aHex: Self-explanatory
    :param bHex: Self-explanatory
    :param cHex: Self-explanatory
    :param dHex: Self-explanatory
    :param eHex: Self-explanatory
    :param fHex: Self-explanatory
    :param gHex: Self-explanatory
    :param hHex: Self-explanatory
    :param roundConstantHex: Self-explanatory
    :return: Tuple of new a-h as hex strings, each of 64 bits, 16 hits
    """
    a = int(aHex, 16)
    b = int(bHex, 16)
    c = int(cHex, 16)
    d = int(dHex, 16)
    e = int(eHex, 16)
    f = int(fHex, 16)
    g = int(gHex, 16)
    h = int(hHex, 16)

    wt = int(messageWordHex, 16)
    kt = int(roundConstantHex, 16)
    ch = (e & f) ^ ((~e) & g)
    maj = (a & b) ^ (a & c) ^ (b & c)
    sigma0 = circularRightShift(a, 28) ^ circularRightShift(a, 34) ^
    circularRightShift(a, 39)
    sigma1 = circularRightShift(e, 14) ^ circularRightShift(e, 18) ^
    circularRightShift(e, 41)

    T1 = (h + ch + sigma1 + wt + kt) % int(2 ** 64)
    T2 = (sigma0 + maj) % int(2 ** 64)

    hNew = hex(g)[2:].upper().zfill(16)
    gNew = hex(f)[2:].upper().zfill(16)
    fNew = hex(e)[2:].upper().zfill(16)
    eNew = hex((d + T1) % int(2 ** 64))[2:].upper().zfill(16)
    dNew = hex(c)[2:].upper().zfill(16)
    cNew = hex(b)[2:].upper().zfill(16)
    bNew = hex(a)[2:].upper().zfill(16)
    aNew = hex((T1 + T2) % int(2 ** 64))[2:].upper().zfill(16)

    ans = (aNew, bNew, cNew, dNew, eNew, fNew, gNew, hNew)
    return ans


def sha_F_Function(messageBlock: str, aHex: str, bHex: str, cHex: str,
    dHex: str, eHex: str, fHex: str, gHex: str,
                   hHex: str) -> tuple:
    W = sha_Message_Schedule(messageBlock)

    # Get the round constants as well
    roundConstants = [
        '428a2f98d728ae22', '7137449123ef65cd', 'b5c0fbcfec4d3b2f', '
    e9b5dba58189dbbc',
        '3956c25bf348b538', '59f111f1b605d019', '923f82a4af194f9b', '
    ab1c5ed5da6d8118',
```

```
155         'd807aa98a3030242', '12835b0145706fbe', '243185be4ee4b28c', '550
    c7dc3d5ffb4e2',
156         '72be5d74f27b896f', '80deb1fe3b1696b1', '9bdc06a725c71235', '
    c19bf174cf692694',
157         'e49b69c19ef14ad2', 'efbe4786384f25e3', '0fc19dc68b8cd5b5', '240
    ca1cc77ac9c65',
158         '2de92c6f592b0275', '4a7484aa6ea6e483', '5cb0a9dcbd41fbd4', '76
    f988da831153b5',
159         '983e5152ee66dfab', 'a831c66d2db43210', 'b00327c898fb213f', '
    bf597fc7beef0ee4',
160         'c6e00bf33da88fc2', 'd5a79147930aa725', '06ca6351e003826f', '
    142929670a0e6e70',
161         '27b70a8546d22ffc', '2e1b21385c26c926', '4d2c6dfc5ac42aed', '53380
    d139d95b3df',
162         '650a73548baf63de', '766a0abb3c77b2a8', '81c2c92e47edaee6', '92722
    c851482353b',
163         'a2bfe8a14cf10364', 'a81a664bbc423001', 'c24b8b70d0f89791', '
    c76c51a30654be30',
164         'd192e819d6ef5218', 'd69906245565a910', 'f40e35855771202a', '106
    aa07032bbd1b8',
165         '19a4c116b8d2d0c8', '1e376c085141ab53', '2748774cdf8eeb99', '34
    b0bcb5e19b48a8',
166         '391c0cb3c5c95a63', '4ed8aa4ae3418acb', '5b9cca4f7763e373', '682
    e6ff3d6b2b8a3',
167         '748f82ee5defb2fc', '78a5636f43172f60', '84c87814a1f0ab72', '8
    cc702081a6439ec',
168         '90befffa23631e28', 'a4506cebde82bde9', 'bef9a3f7b2c67915', '
    c67178f2e372532b',
169         'ca273eceea26619c', 'd186b8c721c0c207', 'eada7dd6cde0eb1e', '
    f57d4f7fee6ed178',
170         '06f067aa72176fba', '0a637dc5a2c898a6', '113f9804bef90dae', '1
    b710b35131c471b',
171         '28db77f523047d84', '32caab7b40c72493', '3c9ebe0a15c9bebc', '431
    d67c49c100d4c',
172         '4cc5d4becb3e42b6', '597f299cfc657e2a', '5fcb6fab3ad6faec', '6
    c44198c4a475817']
173     for k in range(0, 80):
174         aHex, bHex, cHex, dHex, eHex, fHex, gHex, hHex =
    sha_Hash_Round_Function(W[k],
175 
                aHex,
176 
                bHex,
177 
                cHex,
178 
                dHex,
179 
                eHex,
180 
                fHex,
181 
                gHex,
182 
                hHex,
183 
                roundConstants[k])
```

```
184
185     ans = (aHex, bHex, cHex, dHex, eHex, fHex, gHex, hHex)
186     return ans
187
188
189 def sha_Process_Message_Block(inputHex: str, aHex: str, bHex: str, cHex:
        str, dHex: str, eHex: str, fHex: str,
190                               gHex: str, hHex: str) -> tuple:
191     f"""
192     Performs sha_F_Function() on the input block then adds a-h to the new
        a-h.
193
194     :param inputHex: Message block
195     :param aHex: Current value of a
196     :param bHex: Current value of b
197     :param cHex: Current value of c
198     :param dHex: Current value of d
199     :param eHex: Current value of e
200     :param fHex: Current value of f
201     :param gHex: Current value of g
202     :param hHex: Current value of h
203     :return: New a-h values
204     """
205
206     oldH = np.array([aHex, bHex, cHex, dHex, eHex, fHex, gHex, hHex])
207
208     newH = sha_F_Function(inputHex, aHex, bHex, cHex, dHex, eHex, fHex,
        gHex, hHex)
209     ans1 = [hex((int(oldH[i], 16) + int(newH[i], 16)) % int(2 ** 64))[2:].
        upper().zfill(16) for i in range(0, 8)]
210
211     ans = tuple(ans1)
212     return ans
213
214
215 def sha_Calculate_Hash(inputHex: str) -> str:
216     """
217     Calculates the hash of the hex string provided.
218     Initialises
219     aHex
220     bHex
221     cHex
222     dHex
223     eHex
224     fHex
225     gHex
226     hHex
227     and then finds the hash.
228
229     You must:
230     initialise a-h
231     preprocess input
232     create blocks
233     find the hash, update a-h for each block
234
235     :param inputHex: Input of any lenght
236     :return:
```

```
    """

    a = "6A09E667F3BCC908"
    b = "BB67AE8584CAA73B"
    c = "3C6EF372FE94F82B"
    d = "A54FF53A5F1D36F1"
    e = "510E527FADE682D1"
    f = "9B05688C2B3E6C1F"
    g = "1F83D9ABFB41BD6B"
    h = "5BE0CD19137E2179"

    inputHex = sha_Preprocess_Message(inputHex)
    messageBlocks = sha_Create_Message_Blocks(inputHex)

    for messageBlock in messageBlocks:
        a, b, c, d, e, f, g, h = sha_Process_Message_Block(messageBlock, a, b, c, d, e, f, g, h)

    ans = a + b + c + d + e + f + g + h
    return ans


def sha_String_To_Hex(inputStr: str) -> str:
    ans = ""
    for char in inputStr:
        temp = hex(ord(char))[2:].upper().zfill(2)
        ans = ans + temp
    return ans


def sha_Image_To_Hex(inputImg: np.ndarray) -> str:
    inputImg = inputImg.flatten()
    ans = ""
    for k in inputImg:
        ans = ans + hex(k)[2:].upper().zfill(2)
    return ans


def sha_Hex_To_Str(inputHex: str) -> str:
    inputBlocks = [inputHex[k:k + 2] for k in range(0, len(inputHex), 2)]
    ans = ""
    for k in inputBlocks:
        k = chr(int(k, 16))
        ans += k
    return ans


def sha_Hex_To_Im(inputHex: str, originalShape: tuple) -> np.ndarray:
    if len(inputHex) % 2 == 1:
        inputHex = '0' + inputHex

    inputBytes = np.array([int(inputHex[i:i + 2], 16) for i in range(0, len(inputHex), 2)])

    inputBytes = np.reshape(inputBytes, originalShape).round(0).astype(dtype=int)
    return inputBytes
```

```
# endregion sha


# region Transmitter
class Transmitter:
    def __init__(self, ):
        return

    def encrypt_With_RSA(self, message: str, RSA_Key: tuple) -> np.ndarray:
        """
        Receives a string of hex characters and encrypts with RSA, 2 bytes at a time. Block sze is 2 bytes, 4 hex characters.

        :param message: Hex string to encrypt. No padding, message will be a multiple of 4 hex chars.
        :param RSA_Key: RSA public key, (e, n)
        :return: 1D int array with encrypted message blocks.
        """
        m_blocks = [int(message[k:k + 4], 16) for k in range(0, len(message), 4)]
        e, n = RSA_Key
        C = [int(i ** e) % n for i in m_blocks]

        C = np.array(C)
        C = C.round(0).astype(int)
        return C

    def create_Digest(self, message) -> str:
        pranks = "this is an easter egg"
        if type(message) == type(pranks):
            inputHex = sha_String_To_Hex(message)
        else:
            inputHex = sha_Image_To_Hex(message)

        temp = sha_Calculate_Hash(inputHex)
        digest = inputHex + temp
        return digest

    def encrypt_with_RC4(self, digest: str, key: str) -> np.ndarray:
        """
        Encrypts the digest with RC4. The key is provided for RC4
        :param digest: M||H
        :param key: RC4 key
        :return:
        """
        cipher = RC4.rc4_Encrypt_String(digest, key)
        return cipher
# endregion Transmitter


# region Receiver
class Receiver:
    def __init__(self, ):
        self.p = 0
        self.q = 0
        self.n = 0
```

```
        self.phi = 0
        self.e = 0
        self.d = 0
        self.publicKey = (0, 0)
        self.privateKey = (0, 0)

    def printRec(self):
        ans =   f"Entered p value: {self.p}\n" \
              + f"Entered q value: {self.q}\n" \
              + f"Calculated n value: {self.n}\n" \
              + f"Calculated phi value: {self.phi}\n" \
              + f"Calculated e value: {self.e}\n" \
              + f"Calculated d value: {self.d}\n" \
              + f"Calculated PU value: {self.publicKey}\n" \
              + f"Calculated PR value: {self.privateKey}\n"
        return ans

    def generate_RSA_Keys(self, newP: int, newQ: int):
        """
        Given the p and q values, find:
            p
            q
            n
            phi
            e
            d
            publicKey
            privateKey

        :param newP: new vbalue to go to p
        :param newQ: new value to go to q
        :return: nothing
        """
        n = newP * newQ
        phi = ((newP - 1) * (newQ - 1))

        if phi > (2 ** 16 - 1):
            e = 2 ** 16 - 1
        else:
            e = phi // 4 - 1

        while np.gcd(e, phi) != 1 and e < phi:
            e += 1

        if e >= phi:
            e = phi // 2 + 1
        while np.gcd(e, phi) != 1:
            e += 1

        d = pow(e, -1, phi)

        PU = (e, n)
        PR = (d, n)

        self.p = newP
        self.q = newQ
        self.n = n
```

```
401         self.phi = phi
402         self.e = e
403         self.d = d
404         self.publicKey = PU
405         self.privateKey = PR
406 
407     def decrypt_With_RSA(self, message: np.ndarray, RSA_Key: tuple) -> str
    :
408         """
409         Will receive an array of ints that make up the ciphertext of the
    RC4 key.
410         Will apply decryption to this key.
411         P = C^d mod n
412         Encryption is done 2 bytes at a time, so I assume that the same
    holds for decryption, hence the .zfill(4)
413         :param message: Int array of values to decrypt
414         :param RSA_Key: Private key for decryption
415         :return: Hex string version of P
416         """
417         P = ""
418         d, n = RSA_Key
419         for block in message:
420             p = int(int(block) ** d) % n
421             p = hex(p)[2:].upper().zfill(4)
422             P = P + p
423         return P
424 
425     def decrypt_With_RC4(self, digest: np.ndarray, key: str) -> str:
426         plaintext = RC4.rc4_Decrypt_String(digest, key)
427         return plaintext
428 
429     def split_Digest(self, digest: str) -> tuple:
430         M = digest[0:-128]
431         H = digest[-128:]
432 
433         ans = (M, H)
434         return ans
435 
436     def authenticate_Message(self, digest: str) -> tuple:
437         M, H = self.split_Digest(digest)
438         h_calculated = sha_Calculate_Hash(M)
439         auth = (H == h_calculated)
440         ans = (auth, M, H, h_calculated)
441         return ans
442 # endregion Receiver
```

A.2 Simulator Code

```
# David Budnitsky
# 20453508

from u20453508_Prac_3_Backend import *
import numpy as np
import string
from PIL import Image

# These values are from https://stackoverflow.com/questions/287871/how-do-i-print-colored-text-to-the-terminal
HEADER = '\033[95m'
OKBLUE = '\033[94m'
OKCYAN = '\033[96m'
OKGREEN = '\033[92m'
WARNING = '\033[93m'
FAIL = '\033[91m'
ENDC = '\033[0m'
BOLD = '\033[1m'
UNDERLINE = '\033[4m'

np.set_printoptions(threshold=np.inf)

stringPrintable = string.printable[0:94]
line = "-" * 13
smallPrimes = [31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
               73, 79, 83, 89, 97, 101, 103, 107, 109, 113,
               127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
               179, 181, 191, 193, 197, 199, 211, 223, 227, 229,
               233, 239, 241, 251, 257, 263, 269, 271, 277, 281,
               283, 293, 307, 311, 313, 317, 331, 337, 347, 349,
               353, 359, 367, 373, 379, 383, 389, 397, 401, 409,
               419, 421, 431, 433, 439, 443, 449, 457, 461, 463,
               467, 479, 487, 491, 499, 503, 509, 521, 523, 541,
               547, 557, 563, 569, 571, 577, 587, 593, 599, 601,
               607, 613, 617, 619, 631, 641, 643, 647, 653, 659,
               661, 673, 677, 683, 691, 701, 709, 719, 727, 733,
               739, 743, 751, 757, 761, 769, 773, 787, 797, 809,
               811, 821, 823, 827, 829, 839, 853, 857, 859, 863,
               877, 881, 883, 887, 907, 911, 919, 929, 937, 941,
               947, 953, 967, 971, 977, 983, 991, 997]
bigPrimes = [1009, 1013,
             1019, 1021, 1031, 1033, 1039, 1049, 1051, 1061, 1063, 1069,
             1087, 1091, 1093, 1097, 1103, 1109, 1117, 1123, 1129, 1151,
             1153, 1163, 1171, 1181, 1187, 1193, 1201, 1213, 1217, 1223,
             1223, 1229, 1231, 1237, 1249, 1259, 1277, 1279, 1283, 1289,
             1291, 1297, 1301, 1303, 1307, 1319, 1321, 1327, 1361, 1367,
             1373, 1381, 1399, 1409, 1423, 1427, 1429, 1433, 1439, 1447,
             1451, 1453, 1459, 1471, 1481, 1483, 1487, 1489, 1493, 1499,
             1511, 1523, 1531, 1543, 1549, 1553, 1559, 1567, 1571, 1579,
             1583, 1597, 1601, 1607, 1609, 1613, 1619, 1621, 1627, 1637,
             1657, 1663, 1667, 1669, 1693, 1697, 1699, 1709, 1721, 1723,
             1733, 1741, 1747, 1753, 1759, 1777, 1783, 1787, 1789, 1801,
             1811, 1823, 1831, 1847, 1861, 1867, 1871, 1873, 1877, 1879,
             1889, 1901, 1907, 1913, 1931, 1933, 1949, 1951, 1973, 1979,
             1987, 1993, 1997, 1999, 2003, 2011, 2017, 2027, 2029, 2039,
```

```
            2053, 2063, 2069, 2081, 2083, 2087, 2089, 2099, 2111, 2113,
            2129, 2131, 2137, 2141, 2143, 2153, 2161, 2179, 2203, 2207,
            2213, 2221, 2237, 2239, 2243, 2251, 2267, 2269, 2273, 2281,
            2287, 2293, 2297, 2309, 2311, 2333, 2339, 2341, 2347, 2351,
            2357, 2371, 2377, 2381, 2383, 2389, 2393, 2399, 2411, 2417,
            2423, 2437, 2441, 2447, 2459, 2467, 2473, 2477, 2503, 2521,
            2531, 2539, 2543, 2549, 2551, 2557, 2579, 2591, 2593, 2609,
            2617, 2621, 2633, 2647, 2657, 2659, 2663, 2671, 2677, 2683,
            2687, 2689, 2693, 2699, 2707, 2711, 2713, 2719, 2729, 2731
            ]


def isPrime(num: int) -> bool:
    if num == 1:
        return False
    if num == 2:
        return True
    for k in range(2, num // 2 + 1):
        if np.gcd(k, num) != 1:
            return False
    return True


receiver = Receiver()
transmitter = Transmitter()
isImage = False
showImage = False
printImage = 0


print(f"{HEADER}Welcome to Dodgy Dave's Dubious Digital Deception.{ENDC}\
    nLet's get started!\n\n")
print("To initialise a secure transmission channel, comply with the
    following instructions:")
print("Please note the following:"
      "\n- The values you enter for p and q must be prime numbers."
      "\n- The values of p and q must multiply to a number greater than
    65535."
      "\n- If you don't want to come up with what is asked for, you can
    simply press enter and we will get our own "
      "default values."
      "\n- When we ask yes or no and you give nothing or invalid input, we
     assume you meant to say no :)"
      "\n- Have fun.")
print(f"{BOLD}Lets get started!{ENDC}\n\nEnter the following:")

p_input = input(f"{OKBLUE}RECEIVER{ENDC} p value, a good choice is 23:")
q_input = input(f"{OKBLUE}RECEIVER{ENDC} q value, a good choice is 3449:")

if p_input:
    p = int(p_input)
else:
    p = 0

if q_input:
    q = int(q_input)
else:
```

```
    q = 0

if (not isPrime(p)) or (not isPrime(q)) or ((p * q) < int(2 ** 16 - 1)) or (p == q):
    print(f"{FAIL}A condition has been violated, setting p and q to random prime numbers.{ENDC}")
    p = int(np.random.choice(smallPrimes))
    q = int(np.random.choice(bigPrimes))

receiver.generate_RSA_Keys(p, q)

print(f"{OKCYAN}\n{line * 3}\n{line}   PHASE 1   {line}\n{line * 3}\n{ENDC}")

print(receiver.printRec())
publicKey = receiver.publicKey

RC4_K = input(f"{OKGREEN}TRANSMITTER{ENDC} Enter the RC4 Key: ")
if not (RC4_K):
    print("Nothing entered, setting key to a random string.")
    RC4_K = ''.join([np.random.choice(list(stringPrintable)) for _ in range(0, 2 * np.random.randint(3, 7))])
if len(RC4_K)<3:
    print(f"{WARNING}Your RC4 Key was not secure, we will replace it with a secure key.")
    RC4_K = ''.join([np.random.choice(list(stringPrintable)) for _ in range(0, 2 * np.random.randint(3, 7))])
if len(RC4_K) % 2 == 1:
    print("To encrypt the key, it must have an even number of bytes, adding a pad to the key.")
    RC4_K = "0" + RC4_K
print(f"RC4 key: {RC4_K}")

RC4_Khex = sha_String_To_Hex(RC4_K)
RC4_K_enc = transmitter.encrypt_With_RSA(RC4_Khex, publicKey)
RC4_K_dec = receiver.decrypt_With_RSA(RC4_K_enc, receiver.privateKey)

transmitter_RC4Key = RC4_K
receiver_RC4Key = sha_Hex_To_Str(RC4_K_dec)

print(f"{OKGREEN}TRANSMITTER{ENDC} RC4 Key in hex: {RC4_Khex}")
print(f"{OKGREEN}TRANSMITTER{ENDC} RC4 Key (encrypted): {RC4_K_enc}")
print(f"{OKBLUE}RECEIVER{ENDC} RC4 Key (decrypted): {RC4_K_dec}")

print(f"{OKCYAN}\n{line * 3}\n{line}   PHASE 2   {line}\n{line * 3}\n{ENDC}")

M = input(f"{OKGREEN}TRANSMITTER{ENDC} Enter a message: ")

if (not M):
    print(f"{FAIL}\nYou should have entered a valid message!\n{ENDC}")
    M = "In cryptography, encryption is the process of encoding " \
        "information. This process converts the original representation of the information, known as plaintext, " \
        "into an alternative form known as ciphertext. Ideally, only authorized parties can decipher a ciphertext " \
        "back to plaintext and access the original information. Encryption
```

```
    does not itself prevent interference but " \
154         "denies the intelligible content to a would-be interceptor. For
    technical reasons, an encryption scheme " \
155         "usually uses a pseudo-random encryption key generated by an
    algorithm. It is possible to decrypt the message " \
156         "without possessing the key but, for a well-designed encryption
    scheme, considerable computational resources " \
157         "and skills are required. An authorized recipient can easily
    decrypt the message with the key provided by the " \
158         "originator to recipients but not to unauthorized users.
    Historically, various forms of encryption have been " \
159         "used to aid in cryptography. Early encryption techniques were
    often used in military messaging. Since then, " \
160         "new techniques have emerged and become commonplace in all areas
    of modern computing.Modern encryption " \
161         "schemes use the concepts of public-key and symmetric-key. Modern
    encryption techniques ensure security " \
162         "because modern computers are inefficient at cracking the
    encryption. This text was taken from wikipedia."
163
164 if M[-4:] == ".png":
165     isImage = True
166
167     temp = input("Do you want to see the image? Enter 'yes' or 'no': ")
168     if not temp:
169         temp = 'no'
170     showImage = False
171     if temp == 'yes':
172         showImage = True
173     temp = input("Do you want to see the image hex string? Enter 'yes' or
    'no': ")
174     if not temp:
175         temp = 'no'
176     printImage = 0
177     if temp == 'yes':
178         printImage = 1
179
180     img = Image.open(M)
181     img = np.array(img)
182     if showImage:
183         temp = Image.fromarray(img)
184         temp.show(title="Plaintext image")
185     originalSize = img.shape
186     PM_hex = sha_Image_To_Hex(img)
187 else:
188     PM_hex = sha_String_To_Hex(M)
189
190 print(f"{OKGREEN}TRANSMITTER: {ENDC}Message/image is \n{M}")
191 PM_hash = sha_Calculate_Hash(PM_hex)
192 P_Digest = PM_hex + PM_hash
193 C_digest = transmitter.encrypt_with_RC4(P_Digest, transmitter_RC4Key)
194 if showImage:
195     # C_image = np.array(C_digest[0:np.prod(originalSize)])
196     C_image = np.array(C_digest[0:-64]) # 64 here, last 512 bits is digest
    = 128 hits = 128 nibble = 64 byte
197     C_image = np.reshape(C_image, originalSize)
198     C_image = C_image.astype(np.uint8)
```

```
    temp = Image.fromarray(C_image)
    temp.show(title="Ciphertext image")

if isImage:
    print(f"{OKGREEN}TRANSMITTER: {ENDC}Plaintext message: \n{PM_hex if printImage==1 else 'Not shown'}")
    print(f"{OKGREEN}TRANSMITTER: {ENDC}Plaintext hash: \n{PM_hash}")
    print(f"{OKGREEN}TRANSMITTER: {ENDC}Plaintext digest: \n{P_Digest  if printImage==1 else 'Not shown'}")
    print(f"{OKGREEN}TRANSMITTER: {ENDC}Ciphertext digest: \n{sha_Image_To_Hex(C_digest) if printImage==1 else 'Not shown'}")
else:
    print(f"{OKGREEN}TRANSMITTER: {ENDC}Plaintext message: \n{PM_hex}")
    print(f"{OKGREEN}TRANSMITTER: {ENDC}Plaintext hash: \n{PM_hash}")
    print(f"{OKGREEN}TRANSMITTER: {ENDC}Plaintext digest: \n{P_Digest}")
    print(f"{OKGREEN}TRANSMITTER: {ENDC}Ciphertext digest: \n{sha_Image_To_Hex(C_digest)}")

print(f"{OKCYAN}\n{line * 3}\n{line}   PHASE 3   {line}\n{line * 3}\n{ENDC}")

digest_dec = receiver.decrypt_With_RC4(C_digest, transmitter_RC4Key)

M_dec, H_dec = receiver.split_Digest(digest_dec)

changeBit = np.random.choice([True, False], p=[0.1, 0.9])
if changeBit:
    M_dec_str = sha_Hex_To_Str(M_dec)
    print(f"{WARNING}Transmission error occurred!{ENDC}")
    byteChange = np.random.randint(0, 4)
    bitChange = np.random.randint(0, 8)

    newByte = ord(M_dec_str[byteChange])
    temp = int(2 ** bitChange)

    newByte = chr(newByte ^ temp)
    M_dec = M_dec_str[0:byteChange] + newByte + M_dec_str[byteChange + 1:]

    M_dec = sha_String_To_Hex(M_dec)

H_calculated = sha_Calculate_Hash(M_dec)

print(f"Expected hash:\n{H_dec}")
print(f"Calculated hash:\n{H_calculated}")

if isImage:
    P_dec = sha_Hex_To_Im(M_dec, originalSize)
else:
    P_dec = sha_Hex_To_Str(M_dec)

if H_calculated != H_dec:
    print(f"{FAIL}Message not authenticated. The authorities have been alerted!{ENDC}")
    if isImage:
        print(f"The erroneous image was:\n{sha_Image_To_Hex(P_dec) if printImage==1 else 'Not shown'}")
        if showImage:
```

```
            img = np.array(P_dec)
            img = np.reshape(img, originalSize)
            img = img.astype(np.uint8)
            img = Image.fromarray(img, "RGB")
            img.show(title="Erroneous image")
    else:
        print(f"The erroneous message was:\n{P_dec}")
else:
    print(f"{OKGREEN}Message authenticated{ENDC}")
    if isImage:
        print(f"The image sent was:\n{sha_Image_To_Hex(P_dec) if
   printImage==1 else 'Not shown'}")
        if showImage:
            img = np.array(P_dec)
            img = np.reshape(img, originalSize)
            img = img.astype(np.uint8)
            img = Image.fromarray(img)
            img.show(title="Image Received and authenticated")
    else:
        print(f"The message sent was:\n{P_dec}")
```

A.3 RC4 Code

```
# David Budnitsky
# 20453508

import numpy as np


# region RC4
def rc4_Init_S_T(key: str) -> np.ndarray:
    """
    Generates initial S and T arrays. Returns a 2D array holding S and T in elements 0 and 1 respectively
    :param key: The encryption key
    :return: [S, T]
    """
    S = [i for i in range(0, 256)]

    K = [ord(k) for k in list(key)]

    T = np.array([])
    while len(T) < 256:
        T = np.concatenate((T, K))

    T = T[0:256]

    S = np.array(S).round(0).astype(int)
    T = np.array(T).round(0).astype(int)

    ans = np.array([S, T]).round(0).astype(int)
    return ans


def rc4_Init_Permute_S(sArray: np.ndarray, tArray: np.ndarray) -> np.ndarray:
    """
    Performs initial permutation on the S array
    :param sArray: S array
    :param tArray: T array
    :return: The permuted S array
    """

    j = 0
    for i in range(0, 256):
        j = (j + sArray[i] + tArray[i]) % 256
        temp = sArray[i]
        sArray[i] = sArray[j]
        sArray[j] = temp

    # sArray = np.array(sArray).round(0).astype(int)
    return np.asarray(sArray)


# returns (i, j, sArray, k)
def rc4_Generate_Stream_Iteration(i: int, j: int, sArray: np.ndarray) -> tuple:
    """
```

```
    Generates a random byte stream byte
    :param i: Value used in stream generation
    :param j: Value used in stream generation
    :param sArray: last Modified S array
    :return: tuple containing (i,j,sArray, k)
    """
    i = (i + 1) % 256
    j = (j + sArray[i]) % 256
    temp = sArray[i]
    sArray[i] = sArray[j]
    sArray[j] = temp

    t = (sArray[i] + sArray[j]) % 256
    k = sArray[t]

    return tuple((i, j, sArray, k))


def rc4_Process_Byte(byteToProcess: int, k: int) -> int:
    """
    :param byteToProcess: byte to be processed
    :param k: k value
    :return: biwise XOR of k and byteToProcess
    """
    return np.bitwise_xor(byteToProcess, k)


def rc4_Encrypt_String(plaintext: str, key: str) -> np.ndarray:
    """
    Example usage:
    P = "hello world"
    Phex = sha_String_To_Hex(P)
    C = rc4.rc4_Encrypt_String(Phex, "qwerty")
    Pdec = rc4.rc4_Decrypt_String(C, "qwerty")
    Pstr = sha_Hex_To_Str(Pdec)
    print(Pstr)
    :param plaintext: The plaintext to encrypt. Input is a hex string and
   encryption will be done byte by byte.
    :param key: The key to initalise S and T with
    :return: Encrypted text as an int np.ndarray
    """
    if len(plaintext) % 2 == 1:
        plaintext = '0' + plaintext

    P = [int(plaintext[i:i + 2], 16) for i in range(0, len(plaintext), 2)]

    S, T = rc4_Init_S_T(key)
    S = rc4_Init_Permute_S(S, T)

    C = []
    i = 0
    j = 0
    for byte in P:
        (i, j, S, k) = rc4_Generate_Stream_Iteration(i, j, S)
        c = rc4_Process_Byte(byte, k)
        C.append(c)
    C = np.array(C).round(0).astype(int)
```

```
    return C


def rc4_Decrypt_String(ciphertext: np.ndarray, key: str) -> str:
    """
    Decrypts ciphertext using key provided.

    Example usage:
    P = "hello world"
    Phex = sha_String_To_Hex(P)
    C = rc4.rc4_Encrypt_String(Phex, "qwerty")
    Pdec = rc4.rc4_Decrypt_String(C, "qwerty")
    Pstr = sha_Hex_To_Str(Pdec)
    print(Pstr)

    :param ciphertext: Ciphertext to be decrypted, int np array
    :param key: Key to decrypt with
    :return: Hex-string plaintext.
    """
    S, T = rc4_Init_S_T(key)
    S = rc4_Init_Permute_S(S, T)

    i = 0
    j = 0
    P = ""
    for byte in ciphertext:
        (i, j, S, k) = rc4_Generate_Stream_Iteration(i, j, S)
        p = hex(rc4_Process_Byte(byte, k))[2:].upper().zfill(2)
        P = P + p
    return P


def rc4_Encrypt_Image(plaintext: np.ndarray, key: str) -> np.ndarray:
    """
    :param plaintext: 3D image array to encrypt
    :param key: Key to encrypt with
    :return: 1D array of ciphertext image
    """
    P = plaintext.flatten()
    S, T = rc4_Init_S_T(key)
    S = rc4_Init_Permute_S(S, T)

    C = []
    i = 0
    j = 0
    for char in P:
        (i, j, S, k) = rc4_Generate_Stream_Iteration(i, j, S)
        c = rc4_Process_Byte(char, k)
        C.append(c)
    C = np.array(C).round(0).astype(int)
    return C


def rc4_Decrypt_Image(ciphertext: np.ndarray, key: str) -> np.ndarray:
    """
    :param ciphertext: Ciphertext to decrypt as a 1D int array
    :param key: Key to use for decryption
```

```
166         :return:
167         """
168         S, T = rc4_Init_S_T(key)
169         S = rc4_Init_Permute_S(S, T)
170 
171         P = []
172         i = 0
173         j = 0
174         for char in ciphertext:
175             (i, j, S, k) = rc4_Generate_Stream_Iteration(i, j, S)
176             c = rc4_Process_Byte(char, k)
177             P.append(c)
178         P = np.array(P).round(0).astype(int)
179         return P
180 
181 # endregion RC4
```