UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Faculty of Engineering, Built Environment and Information Technology

EHN 410

E-Business and Network Security

Practical 3 Guide

Original Authors: Mr. G. Naudé and Mr. A. Muller

Last updated by: Miss Elna Fourie (24 April 2025)

Due date: Tuesday, 22 May 2025

Code Assignment: Submissions (AMS and Turnitin) will close at 8h30.

Practical Test: Starting @ 10h30 in the Netlabs.

1 SUBMISSION REQUIREMENTS

1.1 CODE INSTRUCTIONS

A zero mark will be awarded if the submitted code file does not run, produces errors, or does not follow the instructions.

The submission slots on the AMS and Clickup for this *Code Assignment* will be closing two hours before the start of the *Practical Test* on Tuesday (22 May 2025).

No late submissions will be allowed after that point.

Everyone is required to submit code conforming to the requirements listed below:

a. All code must be commented to a point where the implementation of the underlying algorithm can be determined.

b. Your name and student number must be included as comments at the top of the submitted code file.

1. Only insert your modified RC4 code in the RC4.py file.

2. The Receiver and Transmitter classes can not ”know” of each other’s existence. This means only the simulator can call functions of the classes, and you cannot, for example, call the Receiver’s ”decrypt_with_RSA” function from within the Transmitter class.

3. Your name and student number must be included as comments at the top of the submitted code files.

c. Code must be submitted via the EHN 410 ClickUp page *and* AMS. Do not email the code to the lecturer or Assistant Lecturer as the emailed code will be considered not to have been submitted.

d. The ClickUp submission must be a PDF document without a cover page including, just the code.

4. Do not include your modified RC4 code in the ClickUp submission. Include the simulator and backend code in a single PDF file.

5. **The code submission on AMS must be three files, called**

   (a) StudentNumber_Prac_3_RC4.py,

   (b) StudentNumber_Prac_3_Backend.py,

   (c) and StudentNumber_Prac_3_Simulator.py.

   For example, 12345678_Prac_3.py, etc.

e. All print statements *and helper functions* (that were not supplied within this document) used for unit testing, must be removed before submission.

f. All code must be written using Python 3.8 or newer.

g. The code submitted must be the final implementation.

h. No late assignments will be accepted. No excuses for late submission will be accepted.

i. Everyone must do their work. **Academic dishonesty is unacceptable and cases will be reported to the University Legal Office for suspension.**

1.2 SUBMISSION INSTRUCTIONS

The due date is Tuesday, 22 May 2025.

Please note:

a. Code Assignment: Submission slots closing @ 8h30.

   i. Submit your final StudentNumber_Prac_3.py files (all three) on the AMS before 8h30.

   ii. Submit a *.pdf of your code on Turnitin, before 8h30.

   iii. Please upload your Turnitin Receipt onto the AMS submission slot before 8h30.

b. Practical Test: Starts @ 10h30 in the Netlabs.

   i. Written during the Practical session time slot, from 10h30 to 12h30.

   ii. Please keep the 12h30 to 13h30 slot free, in case of any technical difficulties (i.e. load-shedding, etc.).

   iii. Arrive at least 15 minutes before the test starts (10h15).

   iv. It is a coding test, based on the algorithms implemented within the *Code Assignment*.

1.3 ADDITIONAL NOTES

Only the following modules may be imported into your Python implementation file:

a. *Numpy* for data structures such as matrices and arrays.

b. *String* for string manipulation consistency.

You may implement any additional helper functions to complete the functionality described in this practical guide.

2 YOUR TASK

Working alone, you are tasked with implementing a cryptographic authentication system for distributing encryption keys. For this practical, you will make use of an RC4 stream cipher to encrypt/decrypt data between two simulated entities. You may use your RC4 cipher from Practical 2 and modify it where necessary.

To begin, create two class objects and name them "Transmitter" and "Receiver". These objects should act exactly as their names imply, where Transmitter will simulate data transmission and Receiver should receive the data sent by Transmitter.

This system should utilize an RC4 stream cipher to encrypt "digest" – which is a combination of the message to be sent, and the hash value of that message. "message" could either be a string of characters or a .png image. However, the focus of this assignment will not be on the data encryption itself, but rather on more secure key distribution and data authentication. Use an RSA Encryption scheme to encrypt and distribute the necessary RC4 keys between Transmitter and Receiver.

Your simulation system must also have a 1 in 10 chance to randomly flip a single bit in the first 4 bytes of the decrypted plaintext. This is to show how much a single bit changes the hash value and to create the possibility of failed hash checks.

Your system must operate in separate phases to establish a secure data transmission channel.

Phase 1: Key Distribution

1. Receiver sends their RSA public key to Transmitter,
2. Transmitter uses RSA to encrypt a given or generated RC4 key,
3. Transmitter sends the encrypted RC4 key to the Receiver.

Phase 2: Data Signing and Encryption

1. Transmitter processes the message stream through a hash function,
2. Transmitter appends the message with its hash to form the digest,
3. Transmitter encrypts the digest stream using RC4,
4. Transmitter sends the encrypted data to the Receiver.

Phase 3: Data Decryption and Authentication

1. Receiver gets the encrypted ciphertext from Transmitter,
2. Receiver decrypts the data stream using RC4 to get the received digest,
3. Receiver computes the hash for the received message and compares it to the received hash,
4. The message is authenticated when the hashes match.

On the front-end side of the system, you must implement the ability to take user input according to the program flow diagram in Figure 1.

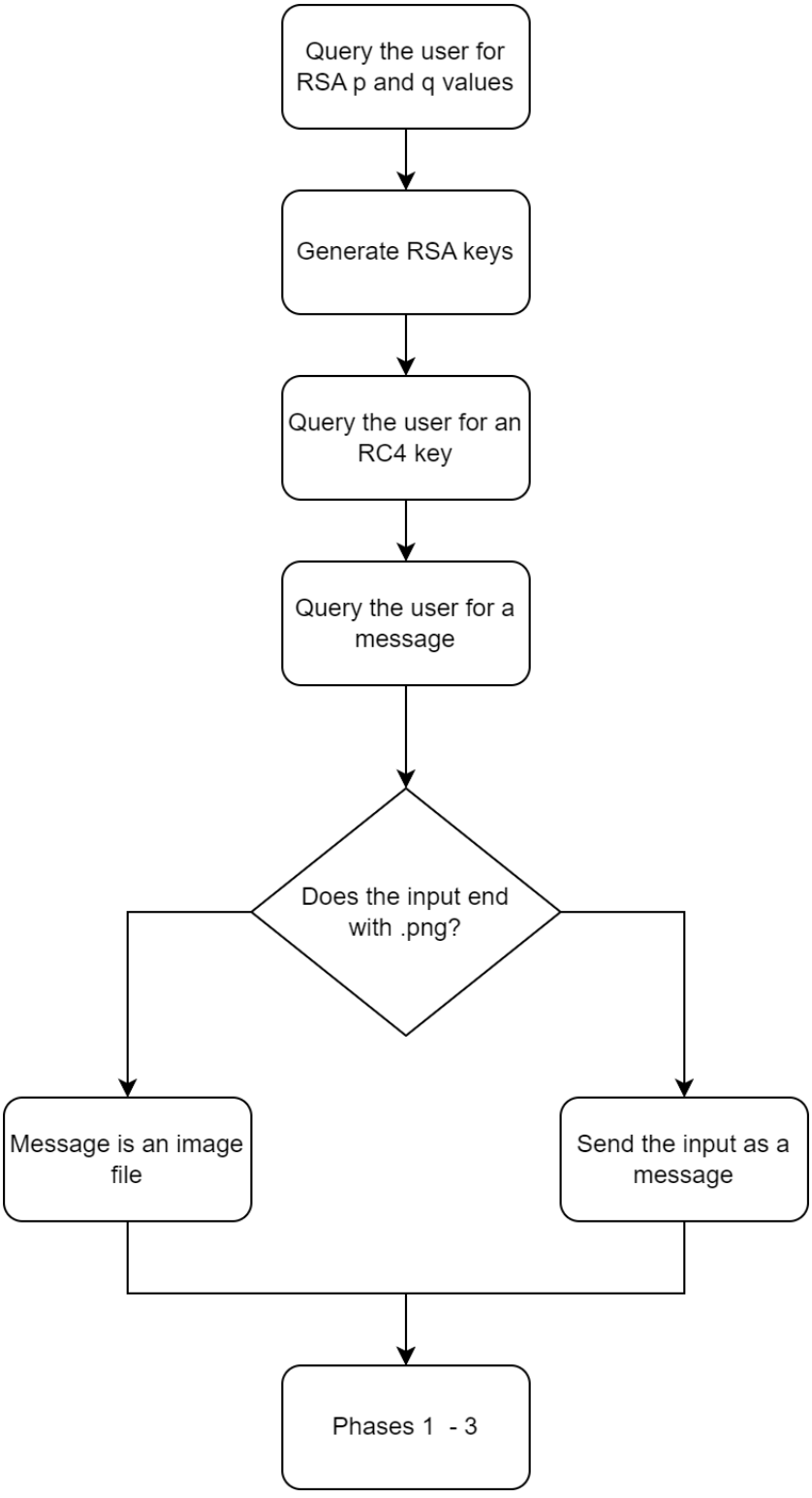

Figure 1: Program flow from a user perspective.

Implement the following functions inside the Backend.py file and use these to implement the Simulator.py file.

2.1 SHA-512

2.1.1 sha_Preprocess_Message(inputHex: str) → str:

- This function pads the input data to a multiple of 1024 bits as required by the SHA-512 algorithm.
- inputHex is the data that needs to be padded. It is a string in hex format, ex. ”49276D205069636B6C65205269636B21”.
- The output is a hex string that has been padded, ex. ”49276D205069636B6C65205269636B21.....80” (Example is shortened, please don’t add ”.” to your output).

2.1.2 sha_Create_Message_Blocks(inputHex: str) → np.ndarray:

- This function returns an ndarray of message blocks (each of which is 1024 bits in length).
- inputHex is the padded message. It is a string in hex format, ex. ”49276D205069636B6C65205269636B21”.
- The output is an ndarray of hex strings (each string’s hex value has 1024 bits), ex. [’49276D205069636B6C6520526...’, ’49276D205069636B6C6520526...’]

2.1.3 sha_Message_Schedule(inputHex: str) → np.ndarray:

- This function returns an ndarray of the 80 words of the message schedule (Figure 11.12).
- inputHex is a message block. It is a string in hex format, ex. ”49276D205069636B6C65205269636B21”.
- The output is an ndarray of hex strings (each string’s hex value has 64 bits), ex. [’49276D205069636B’, ’6C65205269636B21’, ...]

2.1.4 sha_Hash_Round_Function(messageWordHex: str, aHex: str, bHex: str, cHex: str, dHex: str, eHex: str, fHex: str, gHex: str, hHex: str, roundConstantHex: str) → tuple:

- This function performs the round function.
- messageWordHex is one entry of the message schedule W_t. It is a string in hex format, ex. ”49276D205069636B”.
- roundConstantHex is the appropriate K_t value from table 11.4. It is a string in hex format, ex. ”428A2F98D728AE22”.

- aHex to hHex is the current values of a-h (H_{i-1} separated into 8 64-bit hex strings) in Figure 11.11 in a hex string format,
  ex. ”6A09E667F3BCC908”.
- The output is a tuple of the new a-h values as hex strings (H_i separated into 8 64-bit hex strings),
  ex. (’de74d8590d664160’, ’6a09e667f3bcc908’, ’bb67ae8584caa73b’, ’3c6ef372fe94f82b’, ’40900bd4cb1e82fc’, ’510e527fade682d1’, ’9b05688c2b3e6c1f’, ’1f83d9abfb41bd6b’)

2.1.5 sha_F_Function(messageBlock: str, aHex: str, bHex: str, cHex: str, dHex: str, eHex: str, fHex: str, gHex: str, hHex: str) → tuple:

- This function performs the F function on the input block.
- messageBlock is a message block. It is a string in hex format (1024 bits),
  ex. ”49276D205069636B6C65205269636B21...”.
- aHex to hHex is the current values of a-h (H_{i-1} separated into 8 64-bit hex strings) in a hex string format,
  ex. ”6A09E667F3BCC908”.
- The output is a tuple of the new a-h values as hex strings (8 64-bit hex strings),
  ex. (’de74d8590d664160’, ’6a09e667f3bcc908’, ’bb67ae8584caa73b’, ’3c6ef372fe94f82b’, ’40900bd4cb1e82fc’, ’510e527fade682d1’, ’9b05688c2b3e6c1f’, ’1f83d9abfb41bd6b’)

2.1.6 sha_Process_Message_Block(inputHex: str, aHex: str, bHex: str, cHex: str, dHex: str, eHex: str, fHex: str, gHex: str, hHex: str) → tuple:

- This function receives a message block, performs the F function on it, and then adds it with H_{i-1}.
- messageBlock is a message block. It is a string in hex format (1024 bits),
  ex. ”49276D205069636B6C65205269636B21...”.
- aHex to hHex is the current values of a-h (H_{i-1} separated into 8 64-bit hex strings) in a hex string format,
  ex. ”6A09E667F3BCC908”.
- The output is a tuple of the new a-h values as hex strings (H_i separated into 8 64-bit hex strings),
  ex. (’de74d8590d664160’, ’6a09e667f3bcc908’, ’bb67ae8584caa73b’, ’3c6ef372fe94f82b’, ’40900bd4cb1e82fc’, ’510e527fade682d1’, ’9b05688c2b3e6c1f’, ’1f83d9abfb41bd6b’)

2.1.7 sha_Calculat_Hash(inputHex: str) → str:

- This function calculates the hash value of the input.
- inputHex is a hex string of any length,
  ex. ”49276D205069636B6C65205269636B21”
- The function returns a 512-bit hex string,
  ex. ”657A7C7A59CA505015B7149A3743B1D21...”

2.1.8 sha_String_To_Hex(inputStr: str) → str:

- This function receives a string and converts it to a hex string.
- inputStr is a string of any length,
  ex. ”Hi, my name is...”
- The function returns a hex string,
  ex. ”49276D205069636B6C65205269636B21”

2.1.9 sha_Image_To_Hex(inputImg: 3D int ndarray) → str:

- This function receives an image, flattens it, and converts it to a hex string.
- inputImg is an 3D int ndarray.
- The function returns a hex string, ex. ”49276D205069636B6C65205269636B21”

2.1.10 sha_Hex_To_Str(inputHex: str) → str:

- This function receives a hex string and converts it to a character string.
- inputHex is a hex string.
- The function returns a character string,
  ex. ”Hi, my name is...”

2.1.11 sha_He_To_Im(inputHex: str, originalShape: tuple) → 3D int ndarray:

- This function receives a hex string, and converts it to a 3D int ndarray with shape originalShape.
- inputHex is a hex string.
- The function returns a 3D int ndarray.

Inside the Backend.py file, implement the following function for the Transmitter class.

2.2 TRANSMITTER CLASS

2.2.1 encrypt_With_RSA(message: str, RSA Key: tuple) → 1D int ndarray:

- This function receives a hex string, and encrypts it using RSA with a block size of 2 bytes.
- message is a hex string that needs to be encrypted. You may assume that the message will be a length that is divisible by the block size (so no padding is required).
- RSA Key is the RSA key to use during encryption.
- The function returns a 1D int ndarray that consists of the encrypted blocks.

2.2.2 create_Digest(message) → str:

- This function receives a string plaintext or an image plaintext, converts it to a hex string, calculates the hash value, and returns the digest.
- message can be either a string sentence or an image ndarray.
- The function returns a hex string.

2.2.3 encrypt_with_RC4(digest: str, key: str) → 1D int ndarray:

- This function receives a hex string as input and encrypts it using the RC4 cipher.
- Modify your RC4 encrypt functions to receive a hex string as input.
- key can be any string, ex. ”Hi there!”.
- The function returns a 1D int ndarray that contains the encrypted data.

Inside the Backend.py file, implement the following function for the Receiver class.

2.3 RECEIVER CLASS

2.3.1 generate_RSA_Keys(newP: int, newQ: int) → None:

- This function receives the p and q values and generates random private and public keys for RSA.
- The function does not return anything.
- The function populates the following variables of the class:
  - self.p
  - self.q
  - self.n
  - self.phi
  - self.e
  - self.d
  - self.publicKey integer in the form (e, n)
  - self.privateKey integer in the form (d, n)

2.3.2 decrypt_With_RSA(message: 1D int ndarray, RSA Key: tuple) → str:

- This function receives the encrypted RC4 key and decrypts it using RSA.
- message is the encrypted array.
- RSA Key is the RSA key to use during decryption.
- The function returns the decrypted hex string.

2.3.3 decrypt_With_RC4(digest: 1D int ndarray, key: str) → str:

- This function receives the encrypted RC4 array and decrypts it.
- digest is the encrypted digest array.
- key can be any string, ex. ”Hi there!”.
- Modify your RC4 decrypt functions to return a hex string.
- The function returns a hex string.

2.3.4 split_Digest(digest: str) → tuple:

- This function receives the hex string digest and splits it into the message and the hash value.
- This function returns a tuple with the message and the hash,
  ex. (’49276D205069636B6C65205269636B21’, ’657A7C7...’)

2.3.5 authenticate_Message(digest: str) → tuple:

- This function receives a hex string digest and authenticates the message.
- digest is a hex string.
- The function returns a tuple that contains the following:
  - Boolean to show if the message was authenticated successfully,
  - Message as a hex string,
  - Expected hash as a hex string,
  - Calculated has as a hex string,
    ex. (True, '49276D20506...', '657A7C7...', '657A7C7...')

Inside the Simulator.py file, implement code to simulate secure data transmission between the Transmitter and Receiver. Use Figure 2 as an example. The inputs and outputs have been hidden or shortened. Try to stick to the format as much as possible. The code in this file must be executable and will be run manually (the code in Backend.py will be auto-marked).

```
Welcome.
To start a secure transmission chanel, enter RECEIVER's p value:
To start a secure transmission chanel, enter RECEIVER's q value:

PHASE 1

RECEIVER entered p:
RECEIVER entered q:
RECEIVER calculated n:
RECEIVER calculated phi:
RECEIVER calculated public key:  (46591,      )
RECEIVER calculated private key:  (27559,      )

TRANSMITTER Please enter RC4 key:  D4!@x)
TRANSMITTER Given key:  443421407829
TRANSMITTER Encrypted RC4 key: [29131 36576 39765]
RECEIVER Decrypted RC4 key: 443421407829

PHASE 2

TRANSMITTER Please enter a message:
TRANSMITTER Plaintext Hash:  657A7C7A59CA505015B7149A3743B1D21EEB78E0DAD94339A
TRANSMITTER Encrypted Ciphertext ['D9' 'D3' '12' 'EE' 'EC' '53' '53' 'F9' 'E2'
 'A8' '1F' 'EE' '28' '93' 'AC' '39' '9D' 'E3' '6D' 'A2' 'E9' 'D9' '8E'
 'D0' '52' '2' '37' '9B' 'C1' 'D5' '17' 'FD' 'DD' '95' 'E1' 'E9' 'E' '62'
 '33' 'B8' 'F' 'FF' '6F' 'FF' 'B9' '1D' 'AA' '48' '30' '42' '16' 'C2' 'D6'
 '1F' 'B9' '9D' '92' '34' 'F6' '6E' 'F8' '8A' '2' '47' 'B0' '3A' 'E1' 'C9'
 'A6' '63' '82' '5D' '6C' '29' '34']

PHASE 3

RECEIVER Received Ciphertext: 49276D205069636B6C65205269636B21657A7C7A59CA5050

RECEIVER Decrypted Message: I'm Pickle Rick!

RECEIVER Expected Hash:  657A7C7A59CA505015B7149A3743B1D21EEB78E0DAD94339A18EB
RECEIVER Received Hash:  657A7C7A59CA505015B7149A3743B1D21EEB78E0DAD94339A18EB
Message Authenticated.
```

Figure 2: Example output from the simulator