Chapter 1

General knowledge

1.1 Converting data

Binary to hexadecimal

Use python built-in functions.

```
bin_data = b"data"
hex_data = bin_data.hex()
bin_data = bytes.fromhex(hex_data)
```

Binary to base 64

Use python module **base64**.

```
import base64

bin_data = b"data"
b64_data = base64.b64encode(bin_data)
bin_data = base64.b64decode(b64_data)
```

1.2 PKCS7 Padding

Some block cyphers might require padding. The order of operation must always be:

1. Pad data
2. Encrypt padded data

3. Exchange data
4. Decrypt encrypted data
5. Unpad plaintext

```
from cryptography.hazmat.primitives import padding

padder = padding.PKCS7(block_size).padder()
padded_data = padder.update(data) + padder.finalize()

unpadder = padding.PKCS7(block_size).unpadder()
data = unpadder.update(padded_data) + unpadder.finalize()
```

1.3 Randomization

Random keystream

You can generate a random string of bytes by using an OS function.

```
import os

rand = os.urandom(num_bytes)
```

LFSR keystream

This is included in the **pylfsr** module.

```
from pylfsr import LFSR

seed = [0, 0, 0, 1, 0]
fpoly = [3, 2, 1]  # c3=1, c2=1, c1=1
L = LFSR(fpoly = fpoly, initstate = seed, verbose = True)

seq = L.runKCycle(num_bits)
```

Chapter 2

Symmetric encryption

2.1 AES256

AES256 is a **block cypher algorithm** with a 32B key, and it can use different modes of operation. The general syntax is:

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

block_size = 16  # e.g.
key = os.urandom(32)
iv = os.urandom(block_size)  # if needed, depends on mode
nonce = os.urandom(block_size)  # if needed, depends on mode

message = b"A secret message"  # must be binary

cypher = Cypher(algorithms.AES(key), mode = <mode>)

encryptor = cypher.encryptor()
ct = encryptor.update(message) + encryptor.finalize

decryptor = cypher.decryptor()
pt = decryptor.update(ct) + decryptor.finalize
```

CBC mode

Needs an Initialization Vector.

```
cypher = Cypher(algorithms.AES(key), modes.CBC(iv))
```

ECB mode

Padding is required.

```
cypher = Cypher(algorithms.AES(key), modes.ECB())
```

CTR mode

Requires a nonce (unique and never reused). This mode is not reccomended for block cyphers with a block size of less than 128b.

```
cypher = Cypher(algorithms.AES(key), modes.CTR(nonce))
```

Effects of modifying ciphertexts in different modes

- **CBC and ECB modes:** The entire block of the altered byte is corrupted.
- **CTR mode:** Only the affected byte is corrupted.

2.2 ChaCha20

ChaCha20 is a **stream cypher algorithm**. It requires a 32B key and a 16B nonce.

```
nonce = os.urandom(16)

cipher = Cipher(algorithms.ChaCha20(key, nonce), mode=None)

encryptor = cipher.encryptor()
ct = encryptor.update(message)

decryptor = cipher.decryptor()
pt = decryptor.update(ct)
```

Chapter 3

Assymetric encrytption

3.1 RSA

RSA key generation

```
from cryptography.hazmat.primitives.asymmetric import rsa

key_size = 2048
public_exponent = 65537

priv_key = rsa.generate_private_key(
    public_exponent, key_size
)

pub_key = priv_key.public_key()
```

We can obtaint the numbers from the key pair:

```
u = pub_key.public_numbers()
e = u.e
n = u.n

v = priv_key.private_numbers()
p = v.p
q = v.q
d = v.d
```

PEM serialization

To output the key pair in PEM format:

```
from cryptography.hazmat.primitives.serialization \
import load_pem_private_key, load_pem_public_key
from cryptography.hazmat.primitives import serialization

encoding = serialization.Encoding.PEM

# public key
format = serialization.PublicFormat.SubjectPublicKeyInfo
pem_pub_key = pub_key.public_bytes(encoding, format)

# private key
format = serialization.PrivateFormat.TraditionalOpenSSL

pwd = b”password”  # e.g.
encryption_algorithm = serialization.BestAvailableEncryption(pwd)

pem_priv_key = priv_key.private_bytes(
    encoding, format, encryption_algorithm
)
```

You can deserialize a serialized PEM key with:

```
pub_key = load_pem_public_key(pem_pub_key)
priv_key = load_pem_private_key(pem_priv_key, pwd)
```

RSA encryption/decryption (with padding)

```
padder = padding.PKSC1v15()

ct = public_key.encrypt(message, padder)

pt = public_key.decrypt(message, padder)
```

For OAEP padding, we need to set the padder as:

```
padder = padding.OAEP(
    mgf = padding.MGF1(
        algorithms.hashes.SHA256()
    ),
    algorithm = hashes.SHA256(),
    label = None
)
```

Note that encrypting the same message won't give the same cyphertext when using PKCS1v15 and OAEP padding, as there is a random bytes string appended.

Chapter 4

Hybrid encryption (RSA OAEP + AES256 CTR)

1. Encrypt the symmetric AES key with the public RSA key and add padding.

   ```
   encrypted_sym_key = pub_key.encrypt(key, padder)
   ```

2. Generate a nonce.

   ```
   nonce = os.urandom(block_size)
   ```

3. Encrypt with AES in CTR mode (see AES256 CTR).

To decrypt:

1. Decrypt and unpad the AES key with the private RSA key.

   ```
   key = priv_key.decrypt(encrypted_key, padder)
   ```

2. Decrypt the cyphertext with AES in CTR mode (see AES256 CTR).

Chapter 5

Key exchange

AES key wrapping

Key wrapping is encrypting a symmetric key using another symmetric key in order to transmit it through an untrusted channel.

```
from cryptography.hazmat.primitives.keywrap \
    import aes_key_wrap, aes_key_unwrap

    wrapped_key = aes_key_wrap(wrapping_key, key)
    key = aes_key_unwrap(wrapping_key, wrapped_key)
```