Faculté des sciences d'Agadir
Centre d'Excellence IT – Masters
A.U. :2025/2026
Pr. EL BAZZI Mohamed Salim

TP 5 : Sécurité et intégrité dans la Blockchain

Exercice 1 : Cryptographie asymétrique (RSA)

1. Génération de clés RSA

```
!pip install cryptography > /dev/null
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import serialization, hashes
# Génération d'une paire de clés RSA 2048 bits
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048
)
public_key = private_key.public_key()
print("Clés RSA générées avec succès")
```

TAF :

- Change `key_size` à 4096 et mesure la différence de temps d'exécution.
- Pourquoi une clé plus longue est-elle plus sûre, mais plus lente ?

2. Chiffrement et déchiffrement d'un message

```
message = b"Blockchain et cryptographie asymetrique"
# Chiffrement
ciphertext = public_key.encrypt(
    message,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)
# Déchiffrement
plaintext = private_key.decrypt(
    ciphertext,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)
print("Message initial :", message)
print("Message déchiffré :", plaintext.decode())
```

Question :
Pourquoi ne peut-on pas déchiffrer le message avec la clé publique ?

Exercice 2 : Signature numérique

Garantir que le message **provient du bon expéditeur** et **n'a pas été altéré**.

1. Création d'une signature

```
message = b"Transaction : Alice -> Bob : 3 BTC"
# Signature avec la clé privée
signature = private_key.sign(
    message,
    padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
    ),
    hashes.SHA256()
)
print("Signature (hex) :", signature.hex()[:80], "...")
```

2. Vérification de la signature

```
try:
    public_key.verify(
        signature,
        message,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
    print("Signature valide — le message est authentique.")
except Exception:
    print("Signature invalide.")
```

TAF :

- Modifie le message après signature.
- Vérifie de nouveau : que constatez-vous ?

3. Fonctions réutilisables

```
# TODO : compléter ces deux fonctions
def signer_message(privkey, msg: bytes) -> bytes:
    """Retourne la signature RSA du message donné."""
    return privkey.sign(
        msg,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
def verifier_signature(pubkey, msg: bytes, sig: bytes) -> bool:
    """Retourne True si la signature est valide, False sinon."""
    try:
        pubkey.verify(
            sig,
            msg,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )
        return True
    except:
        return False
# Test
msg = b"Test de signature"
sig = signer_message(private_key, msg)
print("Valide ?", verifier_signature(public_key, msg, sig))
```

Exercice 3 : Arbre de Merkle

Créer un **Merkle Tree** pour garantir l'intégrité d'un ensemble de transactions.

1. Construction du Merkle Tree

```
import hashlib
def sha256_hex(data: bytes) -> str:
    return hashlib.sha256(data).hexdigest()
def hash_pair(left: str, right: str) -> str:
    return sha256_hex(bytes.fromhex(left) + bytes.fromhex(right))
def make_leaf_hashes(transactions):
    return [sha256_hex(tx.encode()) for tx in transactions]
def merkle_root(leaf_hashes):
    if not leaf_hashes:
        return ''
    current = leaf_hashes
    while len(current) > 1:
        if len(current) % 2 == 1:
            current.append(current[-1])
        current = [hash_pair(current[i], current[i+1]) for i in range(0,
len(current), 2)]
    return current[0]
# Exemple
txs = ["Alice->Bob:5", "Bob->Charlie:3", "Dave->Eve:1"]
leaves = make_leaf_hashes(txs)
root = merkle_root(leaves)
print("Merkle Root :", root)
```

2. Vérification d'une transaction (preuve d'inclusion)

```
def merkle_proof(leaf_hashes, index):
    proof = []
    idx = index
    current = leaf_hashes.copy()
    while len(current) > 1:
        if len(current) % 2 == 1:
            current.append(current[-1])
        sibling_idx = idx ^ 1
        sibling_hash = current[sibling_idx]
        is_left = sibling_idx < idx
        proof.append((sibling_hash, is_left))
        idx //= 2
        current = [hash_pair(current[i], current[i+1]) for i in range(0,
len(current), 2)]
    return proof
def verify_proof(leaf_hash, proof, root):
    computed = leaf_hash
    for sibling, is_left in proof:
        if is_left:
            computed = hash_pair(sibling, computed)
        else:
            computed = hash_pair(computed, sibling)
    return computed == root
# Test
leaf_hashes = make_leaf_hashes(txs)
root = merkle_root(leaf_hashes)
idx = 1
proof = merkle_proof(leaf_hashes, idx)
print("Transaction :", txs[idx])
print("Proof valide :", verify_proof(leaf_hashes[idx], proof, root))
```

TAF:

- Change une transaction et vérifie de nouveau la preuve.
- Explique pourquoi le résultat devient `False`.

Exercice 4 : Bloc signé avec Merkle Root

Assembler les briques vues précédemment dans un bloc vérifiable.

```
import time
class Block:
    def __init__(self, index, transactions, prev_hash, miner_private_key):
        self.index = index
        self.timestamp = time.time()
        self.transactions = transactions
        self.prev_hash = prev_hash
        self.merkle_root = merkle_root(make_leaf_hashes(transactions))
        # Hash global du bloc
        header =
f"{self.index}{self.timestamp}{self.merkle_root}{self.prev_hash}".encode()
        self.hash = sha256_hex(header)
        # Signature du mineur
        self.signature = signer_message(miner_private_key,
self.hash.encode())
    def verify_block(self, miner_public_key):
        """Vérifie la signature et la validité interne du bloc."""
        header =
f"{self.index}{self.timestamp}{self.merkle_root}{self.prev_hash}".encode()
        return verifier_signature(miner_public_key,
sha256_hex(header).encode(), self.signature)
# Simulation
block = Block(1, ["Alice->Bob:2", "Bob->Charlie:1"], "000000abc",
private_key)
print("Hash du bloc :", block.hash)
print("Merkle Root :", block.merkle_root)
print("Signature valide :", block.verify_block(public_key))
```

TAF:

1. Crée plusieurs blocs et relie-les en chaîne.
2. Modifie une transaction d'un bloc et revérifie la signature et la racine Merkle.
   vous constaterez que toute altération invalide le bloc.