

# Blockchain - Web3

Mathieu Bour

Mewo Informatique



# Le (sublime) formateur (c'est moi !)

- CTO de FANTium
- Ingénieur spécialisé en Blockchain/Web3
- Investisseur dans les cryptomonnaies depuis 2013
- Auditeur de smart-contracts depuis 2021
- (accessoirement) diplômé des Mines de Saint-Étienne en 2020
- 2023-2024 : Riverflow, puis Mobula spécialisé dans la data blockchain
- 2022-2023 : Pooky, un jeu de prédiction de match de foot
- 2021-2022 : \$3M avec DeepSquare sur la blockchain Avalanche



Mathieu Bour

Tél : 06.95.39.72.53

Mail (Mewo) : [mathieu.bour@mewo-campus.fr](mailto:mathieu.bour@mewo-campus.fr)

Mail (pro) : [mat@bour.io](mailto:mat@bour.io)



- DYOR = « Do Your Own Research » : bien que ce cours soit à jour en novembre 2023, je peux m'être trompé. La blockchain n'autorisant pas l'erreur, prenez le temps de faire vos propres recherches.
- La blockchain peut permettre de gagner beaucoup d'argent, mais la très grande majorité des investisseurs perdent leur mise. DYOR.
- En tant que pro-décentralisation, certaines slides peuvent ne pas être objectives, voire tomber dans la poilitique. DYOR.
- Bien que nous évoquerons l'ensemble des types de blockchains, nous utiliserons uniquement les blockchains publiques et décentralisées dans ce cours.

# Sommaire général

- 1 Cryptographie
- 2 Introduction à la blockchain
- 3 Blockchain Ethereum et smart contracts
- 4 Projets

# Cryptographie

# Sommaire

## 1 Cryptographie

- Hachage
- Chiffrement symétrique
- Chiffrement asymétrique
- Signature numérique

## 2 Introduction à la blockchain

## 3 Blockchain Ethereum et smart contracts

## 4 Projets

# Objectifs de ce module

Comprendre les opérations de base de la cryptographie

- ① Hachage
- ② Chiffrement symétrique/asymétrique
- ③ Signature digitale
- ④ Certificats et PKI
- ⑤ Protocoles cryptographiques modernes

# Notations

*Je suis désolé, il faut faire un tout petit peu de maths...*

Dans la suite, je vais noter :

- $\mathbb{B} = \{0, 1\}^{\mathbb{N}}$  l'ensemble des mots binaires
- $\mathbb{B}_{n \in \mathbb{N}} = \{0, 1\}^n$  l'ensemble des mots binaires de taille  $n$

Exemples :

- $B_3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$
- 010010 est un mot binaire de 6 bits, il est donc membre de  $\mathbb{B}_6$



# Histoire de la cryptographie

- Antiquité : premiers chiffrements par substitution
- Moyen-Âge : développement des techniques de cryptanalyse
- Renaissance : chiffrements polyalphabétiques (Vigenère)
- 20ème siècle : machines de chiffrement (Enigma)
- Ère moderne : cryptographie numérique et quantique


# Qu'est-ce que la cryptographie ?

TL ;DR = utiliser les mathématiques au service de la sécurité de l'information

Exemples historiques :

- Chiffrement de César : décalage de lettre de 1 à 25
- Chiffrement de Vigenère : substitutions de lettres à partir d'une clé secrète
- Chiffrement affine : substitution de lettre à l'aide d'une équation affine
- Enigma (seconde guerre mondiale) : machine de chiffrement allemande



Figure – Machine Enigma 

# Objectifs de la cryptographie

- Confidentialité : protection contre la lecture non autorisée
- Intégrité : détection de modifications non autorisées
- Authentification : vérification de l'identité
- Non-répudiation : impossibilité de nier une action

# Somme de contrôle : définition

## Définition : somme de contrôle

Une somme de contrôle est une petite quantité de données additionnelle qui est calculée à partir d'un ensemble plus large de données. Elle est utilisée pour vérifier l'intégrité des données et détecter les erreurs ou les altérations éventuelles.

Exemples :

- Numéro de sécurité sociale
- Numéro de carte bancaire
- IBAN
- Mémoire ECC (Error Correcting Code)

# Somme de contrôle : exemple du numéro de sécurité sociale

Les deux derniers chiffres du numéro de sécurité sociale ne contiennent aucune information mais ils sont utilisés comme somme contrôle, pour limiter les risques de faute d'erreur.

La formule permettant de calculer la clé est la suivante :

$$\text{clé} = 97 - \text{NIR} \bmod 97$$

$$\underbrace{2690549588157}_{\text{numéro NIR}} \underbrace{80}_{\text{clé}}$$

```
1 >>> 97 - 2690549588157 % 97
2 80
```

# Somme de contrôle : exemple de l'ISBN

L'ISBN-13 (International Standard Book Number) utilise une somme de contrôle pour détecter les erreurs de saisie :

- Chaque chiffre est multiplié alternativement par 1 et 3
- La somme est calculée
- Le dernier chiffre est choisi pour que la somme soit divisible par 10

Exemple avec ISBN : 978-2-1234-5680-3

$$(9 \times 1 + 7 \times 3 + 8 \times 1 + 2 \times 3 + 1 \times 1 + 2 \times 3 + 3 \times 1 + 4 \times 3 + 5 \times 1 + 6 \times 3 + 8 \times 1 + 0 \times 3 + \mathbf{3} \times 1) \bmod 10 = 0$$

# Somme de contrôle : codes-barres UPC

Le code UPC (Universal Product Code) à 12 chiffres utilise une somme de contrôle :

- ➊ Additionner les chiffres en position impaire  $\times 3$
- ➋ Additionner les chiffres en position paire
- ➌ La somme de contrôle complète à 10



Exemple : 042100005264

# Fonction de hachage : définition

Comment appliquer cette logique à de l'information binaire? On cherche une somme de contrôle universelle capable de fonctionner sur tout  $\mathbb{B}$ .

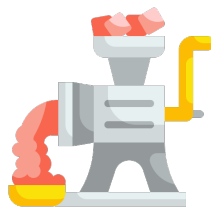
⇒ on les appelle fonction de hachage

## Définition : fonction de hachage

Une fonction de hachage permet de générer un « hash » de n'importe quel mot binaire.

## Définition : hash

Un hash est un mot binaire de taille fixe, dont la taille est spécifique à la fonction de hachage utilisé.





# Fonction de hachage : exemples

```
1 >>> import hashlib
2 >>> hashlib.md5(b"Mathieu").hexdigest()
3 '206299fa740a4327a61b67b6be5c8373'
```

Figure – Calcul d'un MD5 en ligne de commande en Python

```
1 >>> import hashlib
2 >>> hashlib.sha256(b"Mathieu").hexdigest()
3 'f5e088d29801ebb822251d7751bc4b8ff28c50132d8b0a95614b5f048a1d01b6'
```

Figure – Calcul d'un SHA-256 en ligne de commande en Python

# Caractéristiques d'une fonction de hachage

## Uniformité

Une bonne fonction de hachage doit répartir uniformément les valeurs d'entrée sur l'ensemble des valeurs de hachage possibles. Cela signifie que des entrées différentes doivent avoir une probabilité égale de générer des valeurs de hachage différentes.

## Déterminisme

Pour une même valeur d'entrée, la fonction de hachage doit toujours générer la même valeur de hachage. Cela permet d'obtenir des résultats cohérents et reproductibles.

# Caractéristiques d'une fonction de hachage

## Résistance aux collisions

Une fonction de hachage  $h$  de taille  $n$  entraîne obligatoirement des collisions car la taille de  $\mathbb{B}$  est infinie alors que  $\mathbb{B}_n$  n'est « que » de  $2^n$ . Une collision existe quand deux mots binaires  $a$  et  $b$  engendrent le même hash, c'est-à-dire :

$$h(a) = h(b)$$

⇒ une « bonne » fonction de hachage ne possède pas de hash connu.

*Note : Les algorithmes md4, md5 et sha1 ne sont à jour plus considérés comme sûrs.*

# Caractéristiques d'une fonction de hachage

## Sensibilité aux changements

Une petite modification dans l'entrée doit entraîner un changement significatif dans la valeur de hachage. Cela garantit que des entrées similaires génèrent des valeurs de hachage différentes.

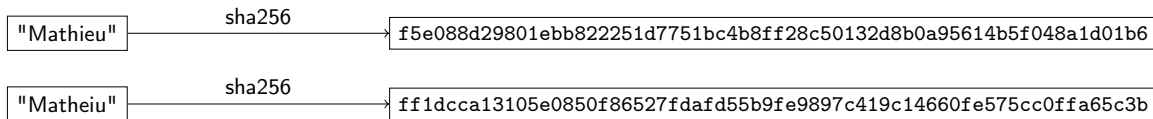


Figure – Sensibilité aux changements de la fonction SHA-256

# Caractéristiques d'une fonction de hachage

## Efficacité

Une bonne fonction de hachage doit être rapide à calculer pour des données de toutes tailles. Les performances de la fonction de hachage sont essentielles, car elle est souvent utilisée dans des applications nécessitant des opérations rapides sur de grandes quantités de données.

Ces performances sont tellement cruciales que certains algorithmes de hachage sont directement implémentés en tant qu'instructions dans les processeurs. Par exemple, les algorithmes SHA-1 et SHA-256 sont **matériellement** présents dans tous les processeurs Intel depuis 2013.

# Caractéristiques d'une fonction de hachage

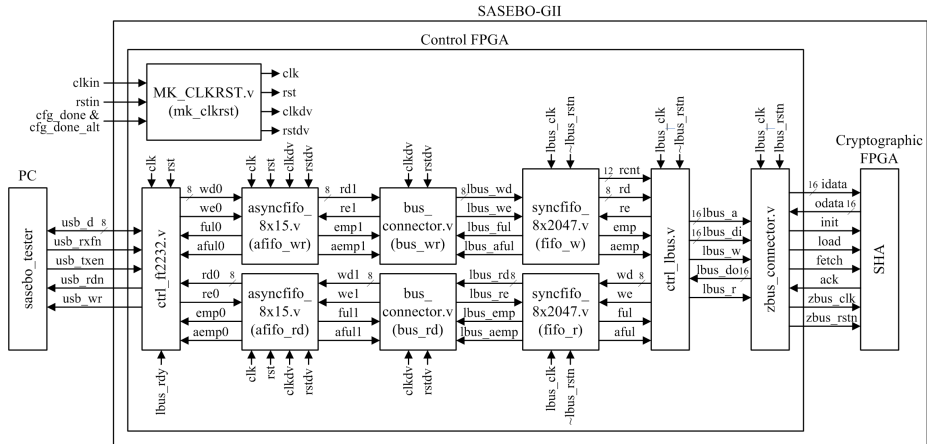


Figure – Implémentation simplifié du SHA-3 sur un FPGA

# Caractéristiques d'une fonction de hachage

## Le plus important : résistance aux attaques

Une fonction de hachage sécurisée doit être résistante à différentes attaques, telles que :

- les attaques de collisions
- les attaques de préimage
- les attaques par force brute

Elle doit être suffisamment robuste pour empêcher un adversaire de trouver des **collisions intentionnelles** ou de retrouver l'**entrée d'origine à partir de la valeur de hachage**.

# Fonction de hachage : résumé

- Une fonction de hachage prend n'importe quel mot binaire entrée et retourne un mot binaire de taille fixe
- Une bonne fonction de hachage possède les caractéristiques suivantes :
  - Uniformité
  - Déterminisme (= pas de hasard)
  - Résistance aux collisions
  - Sensibilité aux changements
  - Efficacité
  - **Résistance aux attaque bruteforce / reverse**



# Fonction de hachage : exercices

## Exercice 1 : se familiariser avec les fonctions de hachage

En CLI, calculez le hash de votre prénom / animal / n'importe quoi. Vous pouvez utiliser des bibliothèques ou des outils en ligne pour effectuer ce calcul. Comparez ensuite la valeur de hachage obtenue avec celle d'autres chaînes de caractères. Observez comment une légère modification de la chaîne d'entrée entraîne un changement significatif dans la valeur de hachage.

## Exercice 2 : calcul du hash d'un fichier

Calculez le hash du whitepaper du bitcoin, accessible sur <https://bitcoin.org/bitcoin.pdf>. Comparez sa valeur avec les autres.

# Applications pratiques du hachage

- Stockage sécurisé des mots de passe
  - Jamais en clair dans la base de données
  - Utilisation de sel (salt) pour renforcer la sécurité
  - Algorithmes spécialisés : bcrypt, Argon2
- Intégrité des fichiers
  - Vérification des téléchargements
  - Détection de corruption
  - Git : identification des commits
- Blockchain
  - Chaînage des blocs
  - Preuve de travail (PoW)
  - Merkle Trees

# Exemple pratique : stockage des mots de passe

```
1  const bcrypt = require('bcrypt');
2  const saltRounds = 10;
3
4  // Hashage d'un mot de passe
5  async function hashPassword(password) {
6    const salt = await bcrypt.genSalt(saltRounds);
7    const hash = await bcrypt.hash(password, salt);
8
9    // Vérification
10   const isValid = await bcrypt.compare('MonMotDePasse123', hash);
11   console.log(isValid ? 'Mot de passe correct' : 'Incorrect');
12 }
13
14 hashPassword('MonMotDePasse123');
```

- Le sel est unique pour chaque utilisateur
- bcrypt inclut automatiquement le sel dans le hash
- Résistant aux attaques par table arc-en-ciel

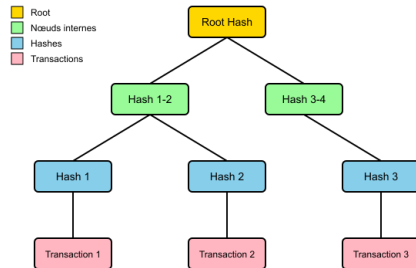
# Merkle Trees dans la blockchain

Structure de données basée sur le hachage :

- Arbre binaire de hashes
- Chaque feuille = hash d'une transaction
- Nœuds internes = hash des enfants
- Racine = hash unique du bloc

Avantages :

- Vérification efficace des transactions
- Preuve d'inclusion simple
- Économie d'espace



# Git et le hachage

```
1  # Voir le hash du dernier commit
2  $ git rev-parse HEAD
3  a1b2c3d4e5f6...
4
5  # Contenu d'un commit
6  $ git cat-file -p a1b2c3d4e5f6
7  tree 7a8b9c0d1e2f...
8  parent 3e4f5g6h7i8j...
9  author John Doe <john@example.com>
10 committer John Doe <john@example.com>
11
12 Initial commit
```

- Chaque objet Git a un hash unique
- Le hash dépend du contenu et des métadonnées
- Permet de détecter toute modification

# Chiffrement : définition

## Définition : chiffrement

Le chiffrement est une technique permettant à deux parties d'échanger de manière sécurisée. Il existe deux grands types de chiffrements, dits symétrique et asymétrique.

Les principales caractéristiques du chiffrement sont :

- Confidentialité : protéger contre l'accès non autorisé, seules les personnes ayant la clé puissent déchiffrer et lire le message.
- Intégrité : détecter toute modification ou altération des données chiffrées. Si les données chiffrées sont altérées, le déchiffrement donnera un résultat incorrect ou une erreur.
- Efficacité : traiter rapidement les données, en particulier lorsqu'il s'agit de volumes importants.
- Sécurité : être résistants aux attaques cryptographiques, telles que les attaques par force brute, les attaques de collision, différentielles. . .

# Modèle de canal de communication

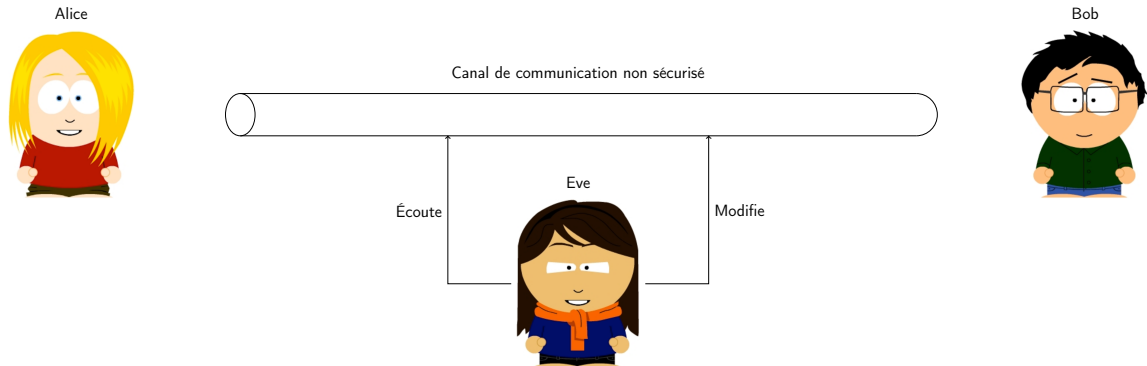


Figure – Modèle de canal de communication non sécurisé

# Chiffrement symétrique

## Définition : chiffrement symétrique

Le chiffrement symétrique est une technique de chiffrement où **une seule et même clé** est utilisée à la fois pour le **chiffrement** et le **déchiffrement** des données. Cela signifie que l'émetteur et le destinataire du message doivent partager la même clé secrète pour pouvoir communiquer de manière sécurisée.

Exemples d'algorithmes de chiffrement symétriques :

- AES
- DES (obsolète) et triple DES



# Chiffrement symétrique : modèle

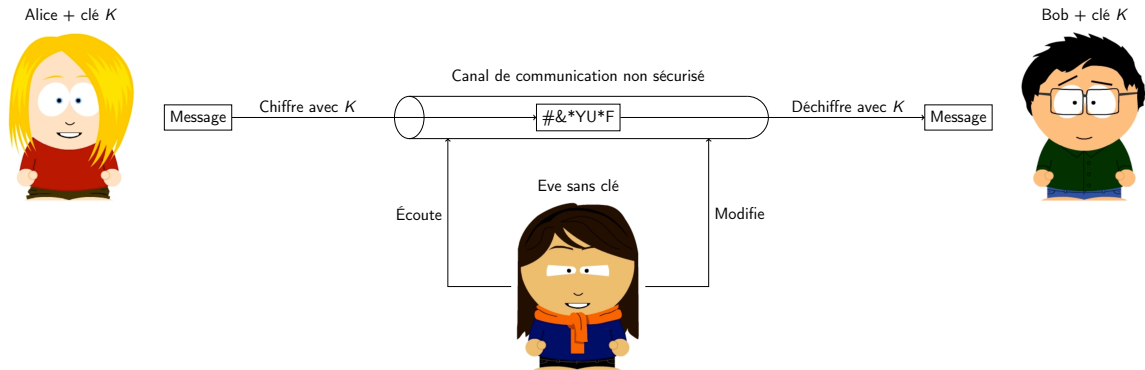


Figure – Chiffrement symétrique avec une clé  $K$

# Chiffrement symétrique : exercice

## Exercice : échange d'information sur canal public

- ➊ Aller sur <https://www.devglan.com/online-tools/aes-encryption-decryption>
- ➋ Choisir une clé de 32 caractères hexadécimaux (par ex : 770A8A65DA156D24EE2A093277530142).
- ➌ Partager la clé avec un ami.
- ➍ Chiffrer un message et le publier sur un canal public.
- ➎ Vérifier que l'ami est capable de déchiffrer le message et personne d'autre.

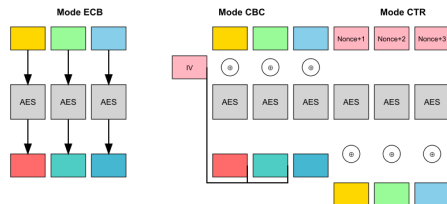
# Chiffrement symétrique : résumé

- Le chiffrement symétrique permet d'échanger des messages secrets.
- Une seule clé pour chiffrer et déchiffrer  $\Rightarrow$  l'émetteur et le destinataire doivent connaître la clé.
- AES est l'algorithme le plus utilisé.

Problème : comment partager la clé de manière sécurisée ?

# Modes de chiffrement

- ECB (Electronic CodeBook)
  - Le plus simple mais le moins sécurisé
  - Motifs visibles dans les données chiffrées
- CBC (Cipher Block Chaining)
  - Utilise un vecteur d'initialisation (IV)
  - Chaque bloc dépend du précédent
- CTR (Counter)
  - Transforme le chiffrement par bloc en flux
  - Parallélisable
- GCM (Galois/Counter Mode)
  - Authentification intégrée
  - Recommandé pour TLS



# Exemple pratique avec Node.js

```
1  const crypto = require('crypto');
2
3  // Génération de la clé et du vecteur d'initialisation
4  const key = crypto.randomBytes(32); // 256 bits pour AES-256
5  const iv = crypto.randomBytes(16); // 128 bits pour AES
6
7  // Chiffrement
8  function encrypt(text) {
9    const cipher = crypto.createCipheriv(
10      'aes-256-cbc', key, iv
11    );
12    let encrypted = cipher.update(text, 'utf8', 'hex');
13    encrypted += cipher.final('hex');
14    return encrypted;
15  }
16
17  // Déchiffrement
18  function decrypt(encrypted) {
19    const decipher = crypto.createDecipheriv(
20      'aes-256-cbc', key, iv
21    );
22    let decrypted = decipher.update(encrypted, 'hex', 'utf8');
```

# Applications courantes

- Chiffrement de disque
  - BitLocker (Windows)
  - FileVault (macOS)
  - LUKS (Linux)
- Communications sécurisées
  - WiFi (WPA3)
  - VPN
  - Session TLS
- Stockage cloud
  - Chiffrement côté client
  - Chiffrement au repos

# Performance et sécurité

Tailles de clé courantes :

- AES-128 : 128 bits
- AES-192 : 192 bits
- AES-256 : 256 bits

Performances :

- Instructions AES-NI
- ~1 Go/s sur CPU moderne
- Adapté au chiffrement temps réel

Bonnes pratiques :

- Rotation régulière des clés
- Stockage sécurisé des clés
- Utilisation de sel (IV)
- Mode authentifié (GCM)

# Chiffrement asymétrique

## Définition : chiffrement asymétrique

Le chiffrement asymétrique, également connu sous le nom de cryptographie à clé publique, est une technique de chiffrement qui utilise une paire de clés distinctes pour le processus de chiffrement et de déchiffrement des données.

Cette paire de clés est composée d'une clé publique et d'une clé privée. Elles sont liées mathématiquement. On peut alors chiffrer un message avec la clé publique et déchiffrer le message avec la clé privée.

Cela permet de recevoir des messages de manière sécurisée sans échange de clé au préalable.

- 1 Alice publie sa clé publique (par exemple sur son profil GitHub).
- 2 Bob va la contacter et lui envoie un message chiffré avec la clé publique d'Alice
- 3 Alice déchiffre le message avec sa clé privée.



# Chiffrement asymétrique : exemple

On considère un algorithme de chiffrement asymétrique générique :

```
1  def cipher(data: str, key: str) -> str:
2  def decipher(data: str, key: str) -> str:
3
4  message = "Hello world"
5  private_key, public_key = generateKeys()
6
7  result = cipher(message, public_key) # "G*#G*#QQ#"
8  clear = decipher(result, private_key) # "Hello world"
```

# Chiffrement asymétrique : communication sécurisée

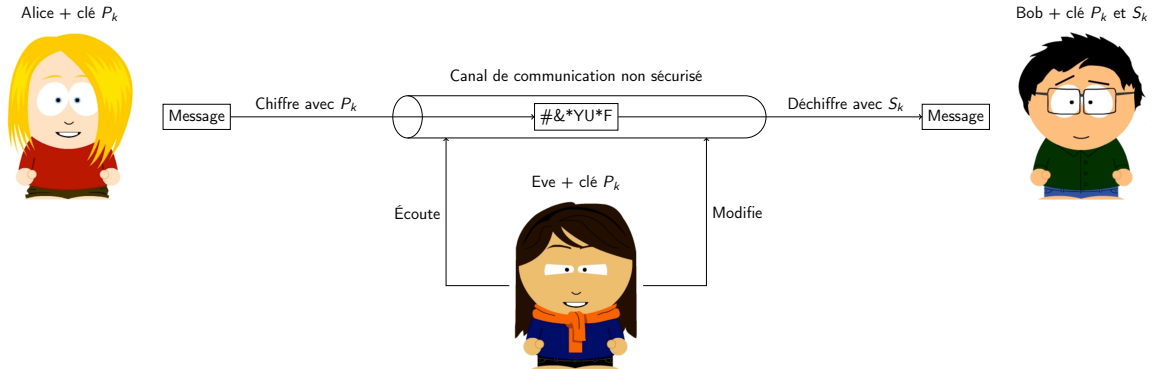


Figure – Chiffrement asymétrique avec une clé ( $P_k, S_k$ )

# Combinaison asymétrique/symétrique

L'utilisation des deux techniques de chiffrement permet de communiquer de manière sécuriser sans avoir à faire un échange de clé au préalable.

- 1 Bob génère son couple de clés  $(P_k, S_k)$  et publie sa clé publique  $P_k$
- 2 Alice veut communiquer avec Bob. Elle génère une clé de chiffrement symétrique  $K$ .
- 3 Alice chiffre  $K$  avec la clé publique de Bob  $P_k$  et envoie le message chiffré à Bob.
- 4 Bob déchiffre le message d'Alice avec sa clé privée  $S_k$  et se retrouve donc en possession de la clé  $K$ .
- 5 Par la suite, Alice et Bob utilise la clé  $K$  et un algorithme de chiffrement symétrique pour communiquer.

# Algorithmes asymétriques courants

- RSA
  - Basé sur la factorisation de grands nombres
  - Utilisé pour le chiffrement et la signature
  - Tailles de clé : 2048-4096 bits
- ECC (Elliptic Curve Cryptography)
  - Basé sur les courbes elliptiques
  - Plus efficace que RSA (clés plus courtes)
  - Courbes populaires : secp256k1 (Bitcoin), Curve25519
- ElGamal
  - Basé sur le problème du logarithme discret
  - Utilisé dans PGP/GPG

# Exemple pratique avec RSA

```
1  const crypto = require('crypto');
2
3  // Génération des clés
4  const { privateKey, publicKey } =
5    crypto.generateKeyPairSync('rsa', {
6      modulusLength: 2048,
7      publicKeyEncoding: {
8        type: 'spki',
9        format: 'pem'
10     },
11     privateKeyEncoding: {
12       type: 'pkcs8',
13       format: 'pem'
14     }
15   });
```

```
1  // Chiffrement
2  const message = 'Message secret';
3  const encrypted = crypto.publicEncrypt(
4    publicKey,
5    Buffer.from(message)
6  );
7
8  // Déchiffrement
9  const decrypted = crypto.privateDecrypt(
10    privateKey,
11    encrypted
12  );
13
14  console.log(decrypted.toString());
15  // Affiche: Message secret
```

# Applications pratiques

## HTTPS/TLS :

- Authentification du serveur
- Échange de clé de session
- Certificats X.509

## SSH :

- Authentification par clé
- Connexion sans mot de passe
- Forward secrecy

## Blockchain :

- Adresses de portefeuille
- Signature de transactions
- Smart contracts

## Email sécurisé :

- PGP/GPG
- S/MIME
- ProtonMail

# Limites et considérations

- Performance
  - 1000-10000× plus lent que le symétrique
  - Limité à de petites quantités de données
  - D'où l'utilisation hybride avec le symétrique
- Sécurité
  - Vulnérable aux attaques quantiques
  - Importance de la génération aléatoire
  - Protection des clés privées critique
- Gestion des clés
  - Distribution des clés publiques
  - Révocation de certificats
  - Durée de vie des clés

# Comparaison Symétrique vs Asymétrique

Critère	Symétrique	Asymétrique
Performance	Rapide	Lent
Taille des clés	128-256 bits	2048-4096 bits
Échange de clés	Difficile	Facile
Usage principal	Données volumineuses	Échange de clés
Algorithmes	AES, ChaCha20	RSA, ECC

- Les deux types sont complémentaires
- Utilisés ensemble dans la plupart des protocoles
- Chacun a ses forces et faiblesses



# Signature numérique : définition

## Définition : signature numérique

Une signature numérique est un mécanisme cryptographique utilisé pour authentifier l'intégrité et l'origine d'un message, d'un document électronique ou d'un ensemble de données.

Elle sert à garantir qu'un document n'a pas été altéré depuis sa signature et qu'il provient bien de l'expéditeur prétendu.

La signature numérique repose sur des algorithmes de cryptographie asymétrique, qui utilisent une paire de clés : une clé privée et une clé publique.

L'expéditeur utilise sa clé privée pour générer une signature numérique unique basée sur le contenu du document. Cette signature est ensuite jointe au document, qui peut être transmis à d'autres parties.

# Signature numérique : exemple

- ① Alice génère sa paire de clé publique/privée  $(P_k, S_k)$
  - ② Alice publie sa clé publique  $P_k$
  - ③ Alice veut signer le message « Alice donne 1 BTC à Bob »
    - ① Alice calcule le hash de « Alice donne 1 BTC à Bob »
    - ② Alice chiffre le hash avec sa clé privée et diffuse le message+le hash signé
  - ④ Tout individu peut maintenant vérifier que le message qu'Alice a chiffré est bien le hash du message qu'elle a publié
- ⇒ Alice a **signé** le message « Alice donne 1 BTC à Bob »

# Algorithmes de signature numérique

- RSA-PSS
  - Version signature de RSA
  - Padding probabiliste
  - Largement utilisé
- ECDSA
  - Basé sur les courbes elliptiques
  - Utilisé dans Bitcoin
  - Signatures plus courtes que RSA
- EdDSA
  - Variante d'ECDSA avec Curve25519
  - Plus rapide et plus sûr
  - Utilisé dans SSH et Signal

# Exemple pratique avec Node.js

```
1  const crypto = require('crypto');
2
3  // Génération des clés
4  const { privateKey, publicKey } =
5    crypto.generateKeyPairSync('ec', {
6      namedCurve: 'secp256k1',
7      publicKeyEncoding: {
8        type: 'spki',
9        format: 'pem'
10     },
11     privateKeyEncoding: {
12       type: 'pkcs8',
13       format: 'pem'
14     }
15   });
```

```
1  // Message à signer
2  const message = "Transaction: Alice -> Bob
   ↳ 1 BTC";
3
4  // Signature
5  const signer =
6    ↳ crypto.createSign('SHA256');
7  signer.update(message);
8  const signature = signer.sign(privateKey);
9
10 // Vérification
11 const verifier =
12   ↳ crypto.createVerify('SHA256');
13 verifier.update(message);
14 const isValid = verifier.verify(
15   publicKey,
16   signature
17 );
18 console.log(isValid ? "Valide!" :
19   ↳ "Invalide!");
```

# Applications des signatures numériques

## Documents électroniques :

- Contrats numériques
- Factures électroniques
- Documents administratifs

## Code et logiciels :

- Signature de code
- Mises à jour système
- Paquets logiciels

## Blockchain :

- Transactions
- Smart contracts
- Gouvernance DAO

## Communication :

- Email (S/MIME, PGP)
- Messages instantanés
- Certificats TLS

# Aspects juridiques

## Cadre légal

En Europe, le règlement eIDAS définit trois niveaux de signature électronique :

- Simple : basique, peu de valeur légale
- Avancée : cryptographiquement sûre
- Qualifiée : maximum de valeur légale

Exigences pour une signature qualifiée :

- Certificat qualifié
- Dispositif sécurisé de création
- Horodatage qualifié
- Conservation à long terme

# Bonnes pratiques

- Sécurité des clés privées
  - Stockage sécurisé (HSM)
  - Sauvegarde et récupération
  - Rotation périodique
- Vérification
  - Validation du certificat
  - Vérification de révocation
  - Horodatage
- Format et standards
  - PAdES pour PDF
  - XAdES pour XML
  - CAdES pour données binaires

# Protocoles de sécurité modernes

- TLS/SSL
  - Sécurisation des communications web (HTTPS)
  - Négociation de clés
  - Authentification des serveurs
- SSH
  - Connexions sécurisées aux serveurs
  - Tunneling sécurisé
- PGP/GPG
  - Chiffrement des emails
  - Signature de documents



# Cryptographie post-quantique

## Défi

Les ordinateurs quantiques pourront casser facilement RSA et d'autres algorithmes basés sur la factorisation et le logarithme discret.

Solutions en développement :

- Réseaux euclidiens
- Codes correcteurs d'erreurs
- Systèmes multivariés
- Cryptographie basée sur les isogénies

# Zero-Knowledge Proofs

## Définition

Protocole permettant de prouver la connaissance d'une information sans la révéler.

Applications :

- Authentification anonyme
- Transactions confidentielles
- Vérification de calculs
- Smart contracts privés

Exemple : zk-SNARKs utilisés dans Zcash pour les transactions privées

# Homomorphic Encryption

## Définition

Système de chiffrement permettant d'effectuer des calculs sur des données chiffrées sans les déchiffrer.

Types :

- Partiellement homomorphe (PHE)
- Quelque peu homomorphe (SHE)
- Complètement homomorphe (FHE)

Applications :

- Cloud computing sécurisé
- Vote électronique
- Analyse de données privées

# Introduction à la blockchain

# Sommaire

- 1 Cryptographie
- 2 Introduction à la blockchain
  - Définitions générales
  - Exemple du Bitcoin
- 3 Blockchain Ethereum et smart contracts
- 4 Projets

# Objectifs de ce module

- ① Comprendre la définition de blockchain en tant que système
- ② Découvrir Bitcoin et le vocabulaire de la Blockchain
- ③ Comprendre le problème du consensus et comment Bitcoin le traite

# Définitions générales

# Contexte historique : origines de la blockchain

- 2008 : Satoshi Nakamoto publie « Bitcoin: A Peer-to-Peer Electronic Cash System »
- Dans ces neuf pages, Nakamoto décrit un système financier et introduit les bases de la blockchain
  - Structure en blocs
  - Cryptographie (hachage, asymétrique, arbres de Merkle...)
  - Transactions
- Fun fact : Satoshi Nakamoto est toujours resté anonyme

## Bitcoin: A Peer-to-Peer Electronic Cash System

Satoshi Nakamoto  
satoshin@gmx.com  
www.bitcoin.org

**Abstract.** A purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution. Digital signatures provide part of the solution, but the main benefits are lost if a trusted third party is still required to prevent double-spending. We propose a solution to the double-spending problem using a peer-to-peer network. The network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work. The longest chain not only serves as proof of the sequence of events witnessed, but proof that it came from the largest pool of CPU power. As long as a majority of CPU power is controlled by nodes that are not cooperating to attack the network, they'll generate the longest chain and outpace attackers. The network itself requires minimal structure. Messages are broadcast on a best effort basis, and nodes can leave and rejoin the network at will, accepting the longest proof-of-work chain as proof of what happened while they were gone.

Figure – Bitcoin Whitepaper by S. Nakamoto  
mewo  
INFORMATIQUE



# Définition générale

Blockchain se traduit par « chaîne de blocs ». Il s'agit donc d'un système permettant de stocker et de partager de l'information au travers d'un **structure de données bien choisie construite à partir de plusieurs blocs** (et c'est tout).

La majorité des systèmes de blockchain possèdent des caractéristiques supplémentaires qui sont utilisées par abus de langage :

- ❶ Présence d'une cryptomonnaie liée à la blockchain (il existe des blockchains SANS cryptomonnaies)
- ❷ Décentralisation
- ❸ Autonome/sans administration centrale
- ❹ Anonymat/pseudonymat des utilisateurs

# Définitions tierces

economie.gouv.fr

*Développée à partir de 2008, c'est, en premier lieu, une technologie de stockage et de transmission d'informations. Cette technologie offre de hauts standards de transparence et de sécurité car elle fonctionne sans organe central de contrôle.*

*Plus concrètement, la chaîne de blocs permet à ses utilisateurs - connectés en réseau - de partager des données sans intermédiaire.*

Wikipédia

*Une blockchain, ou chaîne de blocs, est une technologie de stockage et de transmission d'informations sans autorité centrale. Techniquement, il s'agit d'une base de données distribuée dont les informations envoyées par les utilisateurs et les liens internes à la base sont vérifiés et groupés à intervalles de temps réguliers en blocs, formant ainsi une chaîne.*

# Centralisation

Exemples :

- ❶ L'Euro : la banque centrale européenne est souveraine et peut émettre des euros
- ❷ La force nucléaire en France : contrôlée par l'armée
- ❸ Twitter : la direction peut décider de retirer des privilèges sans l'approbation des utilisateurs (arrivée d'Elon Musk...)

- ⇒ La centralisation place un privilège/pouvoir entre les mains d'un petit groupe
- ⇒ Inversement, les utilisateurs sont tributaire du bon vouloir/bon fonctionnement des systèmes
- ⇒ Une relation de **confiance** est nécessaire

# Exemple du Bitcoin

# Bitcoin : décentralisation

- La blockchain Bitcoin est un réseau peer-to-peer décentralisé
- Le réseau Bitcoin **toujours en ligne** (tant qu'il y a des noeuds)
- Pas d'administration centrale (donc pas de Bitcoin Corp. Limited)
- Tout individu peut y participer en créant un « nœud » = démarrer un logiciel en CLI

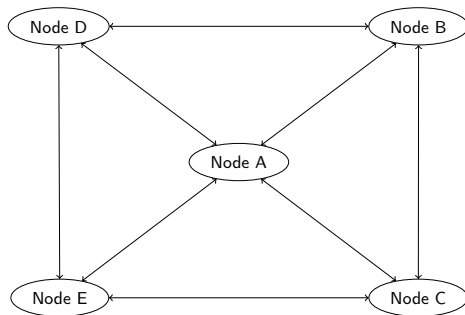


Figure – Réseau peer-to-peer

# Bitcoin : livre de comptes

La blockchain Bitcoin est un système décentralisé permettant aux utilisateurs d'échanger une monnaie numérique, le Bitcoin.

- Les Bitcoin (BTC) sont stockés dans des comptes, identifiés par une adresse
- L'ensemble des soldes des comptes, le « livre de comptes » est stocké à de multiples endroits
- Tout individu peut obtenir un compte gratuitement (on en parle plus tard)
- Envoyer des  $x$  BTC d'une adresse  $a$  à une adresse  $b$  revient à faire

```
1  solde_a -= x; solde_b += x
```

Compte	Solde
0001	12
0002	3.42
0003	4.4
0004	3.6
0005	5
⋮	
1232	30.45
1233	0.34
1234	113.3
1235	4.97

# Bitcoin : opérer un node

- Opérer un node = participer à la blockchain = augmenter la décentralisation
- « Relativement léger » : 2 Go de RAM, 7 Go de disque, connexion 400 kilobits/sec
- Attention, certains pays interdisent d'opérer un node : Afghanistan, Algérie, Bangladesh, Bolivie, Chine, Égypte, Kosovo, Maroc, Népal

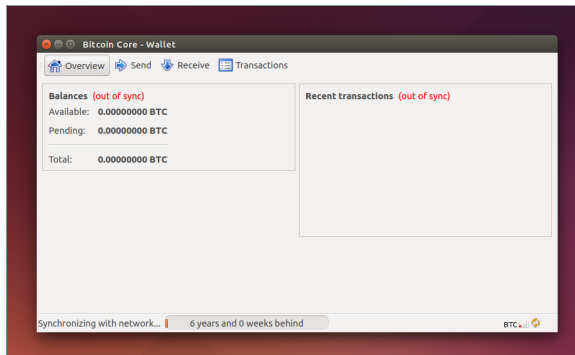


Figure – Bitcoin Core GUI

# Bitcoin : créer un compte

- La blockchain Bitcoin, comme la majorité des blockchain, fonction avec un couple de clé publique/privée.
- Chaque clé privée donne accès à un compte (=adresse) de la blockchain.
- Attention, si la clé privée fuite, le compte est définitivement compromis.

Exemple avec <https://iancoleman.io/bip39/> :

- Clé privée Kwp24SX6Uo1xNUDihM3oeSmg9MHrPm9ZEp7v26jrmxMmC5js4i5f
- Clé publique  
0256fa0a8c520a0a501000845ebaf112295b3c5c29b4b0f7b2d01b933451d9ebf8
- Adresse 1CLHPJK9Z1NFzvQSnP5CozanMo5L35Caq1



# Bitcoin : effectuer une transaction

## Définition : transaction

Sur la blockchain Bitcoin, une transaction est un message signé (avec la clé privée) par un utilisateur de blockchain. Elle contient les informations suivantes :

- L'adresse de l'émetteur.
- L'adresse du destinataire.
- La valeur (=montant) de la transaction.

# Bitcoin : effectuer une transaction

Exemple : Alice, qui possède 10 BTC, peut en envoyer 1 BTC à Bob en signant le message suivant : « Moi, Alice, envoie 1 BTC à Bob »

Une fois la transaction signée, il faut l'envoyer au réseau Bitcoin. Pour cela, Alice doit envoyer la transaction à un node Bitcoin.

Ce dernier va alors vérifier la transaction (si la signature est valide) et si c'est le cas, l'envoyer à d'autres noeuds du réseau, qui vont eux-mêmes vérifier la transaction, etc. Par effet boule de neige, la transaction fini par atteindre rapidement tous les noeuds Bitcoin.

# Bitcoin : effectuer une transaction

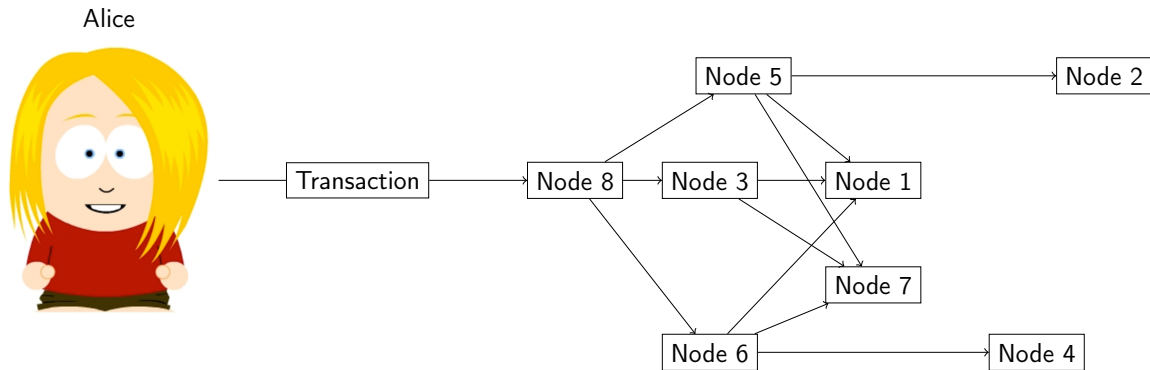


Figure – Diffusion ou « broadcast » d'une transaction

# Bitcoin : problème du consensus

- Dans le schéma précédent, Alice a initialement envoyé sa transaction au Node 8, ce qui implique que celui-ci a eu temporairement une liste de transactions différente des autres nodes.
  - En parallèle, d'autres individus envoient des transactions à d'autres nodes.
  - En pratique, aucun node n'a exactement la même liste de transactions.
- ⇒ C'est le problème du consensus = dans un système décentralisé, comment faire en sorte de mettre tout le monde d'accord ?

# Bitcoin : structure en blocs

## Définition : bloc

Dans une blockchain, un bloc est un ensemble de transactions.

- Dans la blockchain Bitcoin, la taille d'un bloc est limité à 1 MB ce qui correspond à environ 2000 transactions.

```
1  Block #4598794
2
3  Alice sends 10 BTC to Bob
4  John sends 5 BTC to Alfred
5  Juliette sends 3 BTC to Mathieu
6  Bob sends 3 BTC to Mathieu
7  Mathieu sends 4 BNTC to John
8
9  Nonce 90348953a8df7fe9
```

Figure – Exemple textuel d'un bloc

# Bitcoin : consensus par preuve de travail « proof-of-work »

- Afin d'obtenir le consensus, Satoshi Nakamoto a choisi d'utiliser la preuve de travail ou « proof-of-work ».
- Afin de proposer son bloc, le node doit résoudre un problème cryptographique difficile : il doit réussir à concevoir un bloc dont le hash avec l'algorithme SHA-256 commence par un certain nombre de zéro.
- Pour cela, le node fait varier le **nonce** du bloc, qui est un text arbitraire ajouté en fin de bloc.

```
1  Block #4598794
2
3  Alice sends 10 BTC to Bob
4  John sends 5 BTC to Alfred
5  Juliette sends 3 BTC to Mathieu
6  Bob sends 3 BTC to Mathieu
7  Mathieu sends 4 BNTC to John
8
9  Nonce 90348953a8df7fe9
```

Figure – Exemple textuel d'un bloc

# Bitcoin : consensus par preuve de travail « proof-of-work »

- Si le node trouve la solution, il doit la diffuser au reste du réseau qui vérifieront le nonce trouvé.
- En récompense, le node reçoit 6.25 BTC soit environ 150000€ au 14 mai 2023.
- La seule manière de trouver la solution est la force brute (cf. sécurité des fonctions de hachage).
- On appelle également les nodes « mineurs ».
- Un bloc est miné toutes les 10 minutes.

```
1  Block #4598794
2
3  Alice sends 10 BTC to Bob
4  John sends 5 BTC to Alfred
5  Juliette sends 3 BTC to Mathieu
6  Bob sends 3 BTC to Mathieu
7  Mathieu sends 4 BNTC to John
8
9  Nonce 90348953a8df7fe9
```

Figure – Exemple textuel d'un bloc

# Bitcoin : aspect économiques et halving

- Sachant que chaque bloc miné introduit des nouveaux bitcoin dans le marché, la valeur du bitcoin baisse nécessairement sur la durée (plus de bitcoin=moins de valeur pour 1 bitcoin).
- Ce phénomène est comparable à l'inflation des monnaies traditionnelles.

## Halving (de l'anglais *half*, moitié)

Dans la blockchain Bitcoin, les récompenses de bloc sont divisées par deux tous les 210000 blocs, soit environ 4 ans. En 2009, un bloc valait 50 BTC, en 2013 25 BTC, en 2016 12 BTC et en 2020 6.25 BTC.

## Maximum supply

Dû au mécanisme de halving, il ne pourra jamais y avoir plus de 21 millions de BTC en circulation. Aucun État, organisation, armée ou puissance ne pourra en décider autrement.



# Bitcoin : minage

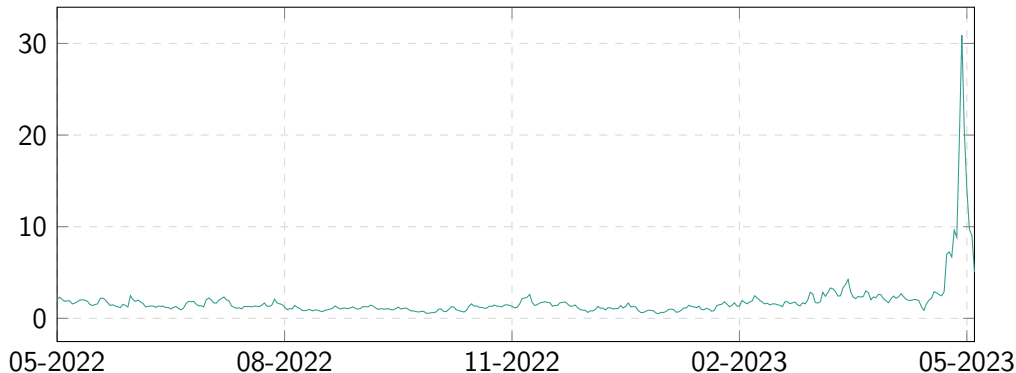
- Vu les récompenses de blocs, le minage s'est rapidement professionnalisé.
- Des entreprises ont créé spécifiquement désigné pour le minage du Bitcoin, 20 fois plus puissants que les meilleurs processeurs AMD/Intel.
- La consommation électrique du minage de Bitcoin atteint 140 TWh par en mars 2023 (Univ. Cambridge), soit l'énergie consommé par l'Égypte ou la moitié du parc nucléaire français.



Figure – Un mineur Antminer L7

# Frais de transaction

Émettre une transaction sur une blockchain n'est jamais gratuit et se paye en BTC.



# Blockchain Ethereum et smart contracts

# Sommaire

- 1 Cryptographie
- 2 Introduction à la blockchain
- 3 Blockchain Ethereum et smart contracts
  - Créer un wallet Ethereum
  - Développer des contrats avec Foundry
  - Le langage Solidity
  - Standards de tokens
  - Tester avec Foundry

# Objectifs de ce module

Comprendre les opérations de base de la cryptographie

- 1 Interagir avec une blockchain de test (Ethereum Sepolia)
- 2 Interagir avec une blockchain de production (Polygon)

# Créer un wallet avec RabbyWallet

- Ceux qui ont déjà un wallet : bonne sieste
- Les autres, installez l'extension RabbyWallet sur votre navigateur (Chrome / Firefox)
- Effectuez l'onboarding et notez bien la phrase de récupération



# Seed phrase

Lors de la création d'un wallet avec RabbyWallet, une phrase de douze mots est générée.

Il faut absolument la sauvegarder en lieu sûr.  
Quiconque l'obtient peut accéder à tout votre  
wallet, vos cryptos et vos NFTs !

# Foundry



# Foundry

Dans ce cours, nous allons utiliser Foundry, un utilitaire permettant de développer des smart-contracts simplement.

# Installation de Foundry : Windows

Il faut d'abord installer Windows Subsystem Linux avec :

```
1 $ wsl --install
```

WSL demandera de redémarrer, puis au redémarrage de choisir un nom d'utilisateur et un mot de passe. Pour confirmer la bonne installation de WSL, exécuter :

```
1 $ wsl --list
```

# Installation de Foundry

Configurer git avec votre nom / email :

```
1 $ git config --global user.email "you@example.com"
2 $ git config --global user.name "Prénom Nom"
```

Installer Foundryup (installateur de Foundry)

```
1 $ curl -L https://foundry.paradigm.xyz | bash
```

Redémarrer le terminal, lancer Foundryup

```
1 $ foundryup
```

Redémarrer le terminal, lancer Forge

```
1 $ forge
2 forge 0.2.0 (31fcf5a 2023-05-19T00:10:33.861185000Z)
```

# Architecture d'un projet Foundry

/	
├── out.....	Fichiers compilés
├── lib.....	Libraries installées
├── src.....	Code source des contrats
├── test.....	Test des contrats
├── .gitmodules	
└── foundry.toml.....	Configuration de Foundry

# Solidity

# Qu'est-ce qu'un smart contract

## Définition : smart contract

Sur la blockchain Ethereum, un smart contract est un bytecode (=code hexadécimal) associé à une adresse.

⇒ Les adresses des smart contracts sont indiscernables des adresses des comptes utilisateurs.

## Définition : Externally Owned Account

Les adresses contrôlés par des utilisateurs sont appelées « Externally Owned Account » (EOA).

Voir le glossaire d'Ethereum : <https://ethereum.org/en/glossary>

# Solidity ?

## Définition : Solidity

Solidity est un **langage de programmation** utilisé pour écrire des smart contracts sur la plateforme Ethereum.

Solidity permet aux développeurs de définir des règles et des logiques spécifiques à un smart contract. Il permet d'écrire des lignes de code qui définissent comment un smart contract doit fonctionner, quelles actions il doit effectuer et comment il doit réagir dans différentes situations. En utilisant Solidity, les développeurs peuvent créer des smart contracts pour diverses applications décentralisées (dApps)

# Solidity : syntaxe en POO

```
1  // SPDX-License-Identifier: UNLICENSED
2  pragma solidity ^0.8.13;
3
4  contract Counter {
5      uint256 public number;
6
7      function setNumber(uint256 newNumber) public {
8          number = newNumber;
9      }
10 }
```



# Solidity : header

Le header d'un smart contract s'écrit en deux lignes :

```
1  // SPDX-License-Identifier: UNLICENSED  
2  pragma solidity ^0.8.13;
```

La ligne 1 définit la licence du fichier :

- UNLICENSED signifie que le code est complètement privé
- `pragma solidity ^0.8.13;` version de Solidity compatible avec le fichier

# Semantic versionning

## Semantic versionning « SemVer »

Le versionnage sémantique est une méthode de numérotation des versions logicielles basée sur des règles spécifiques. Elle se compose de trois nombres séparés par des points : MAJEUR.MINEUR.PATCH. Le numéro MAJEUR est augmenté lorsque des changements incompatibles sont apportés, le numéro MINEUR est augmenté lorsque des fonctionnalités sont ajoutées de manière rétrocompatible, et le numéro PATCH est augmenté pour les corrections de bugs rétrocompatibles.

### Opérateurs

- $=1.2.3$  strictement égal à 1.2.3
- $\wedge 1.2.3 \Rightarrow 1.2.3 < v < 2.0.0$
- $\sim 1.2.3 \Rightarrow 1.2.3 < v < 1.3.0$

### Opérateurs (MAJEUR=0)

- $=0.1.2$  strictement égal à 0.1.2
- $\wedge 0.1.2 \Rightarrow 0.1.2 < v < 0.2.0$

# Solidity : types primitifs

- `uint` un entier non signé sur 256 bits
- `uint8` un entier non signé sur 8 bits
- `uint32` un entier non signé sur 32 bits
- `uint256` un entier non signé sur 256 bits
- `int` un entier signé sur 256 bits
- `int32` un entier signé sur 32 bits
- `address` une adresse Ethereum
- `string` une chaîne de caractères
- `struct` structure, au sens langage C du terme
- `mapping` une association clé-valeur

# Solidity : le type address

Le type `address` est spécial et possède des propriétés :

- `<address>.balance` le montant d'ether détenu par l'adresse
- `<address>.code` le code à l'adresse (vide pour les EOA)
- Dans un contrat, `address(this).balance` donne la balance du contrat

# Solidity : variables spéciales

## `msg` (= message)

Un message représente un appel d'une fonction d'un smart contract.

- `msg.sender` expéditeur du message
- `msg.value` nombre d'Ether envoyés avec le message

## `block`

Metadata du bloc actuel.

- `block.number` numéro du bloc actuel
- `block.timestamp` timestamp UNIX en secondes

# Solidity : contrôle de flow

## if / else / else if

```
1  if (cond) {  
2    // if path  
3  } else if (cond2) {  
4    // else if path  
5  } else {  
6    // else path  
7  }
```

Note : pas de switch / case en Solidity.

## for / while

```
1  for (uint256 i = 0; i < n; i++) {  
2    // do stuff  
3  }  
4  
5  while(cond) {  
6    // do other stuff  
7  }
```

Note : pas de **do while** en Solidity.

# Solidity : visibilité

## Public

```
1  uint public myVariable;  
2  function myFunction() public {  
3      // Function logic  
4  }
```

## Private

```
1  uint private myVariable;  
2  function myFunction() private {  
3      // Function logic  
4  }
```

## Internal

```
1  uint internal myVariable;  
2  function myFunction() internal {  
3      // Function logic  
4  }
```

## External

```
1  // external variables not possible  
2  function myFunction() external {  
3      // Function logic  
4  }
```

# Solidity : modifier

## Définition : modifier

En Solidity, un « modifier » est une fonction spéciale qui permet de modifier le comportement d'autres fonctions dans un contrat intelligent. Les modifiers fournissent un moyen pratique de réutiliser du code et d'ajouter des conditions supplémentaires ou des vérifications avant l'exécution d'une fonction.

## Syntaxe : modifier

```
1  modifier exampleModifier() {  
2      _; // Continue function execution  
3  }  
4  
5  function foobar() public exampleModifier {}
```



# Solidity : interfaces

Comme beaucoup d'autres langages, Solidity dispose d'interfaces<sup>1</sup> qui servent à intégrer des notions de polymorphisme.

- Elles ne peuvent pas hériter d'autres contrats, mais elles peuvent hériter d'autres interfaces.
- Toutes les fonctions déclarées doivent être externes.
- Elles ne peuvent pas déclarer de constructor, de variables d'état ou de modifier.

## Syntaxe : interface

```
1 interface IToken {  
2     function transfer(address recipient, uint amount) external;  
3 }
```

---

1. Voir la documentation Solidity des interfaces

# Solidity : events

- La blockchain est « isolée » du monde extérieur : impossible de contacter le monde extérieur (pas de requête HTTP, notifications, etc.).
- Solidity permet de définir des events qui peuvent être écoutés à l'extérieur de la blockchain.

```
1  // Event declaration
2  event Minted(address indexed to, uint256 amount);
3
4  function transfer(address to, uint256 amount) {
5      balanceOf[to] += amount;
6      emit Minted(to, value); // Event emission
7  }
```

```
1  client.watchEvent({
2      address: '0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48',
3      event: parseAbiItem('event Minted(address indexed to, uint256 amount)'),
4      onLogs: logs => console.log(logs)
5  })
```

# Solidity : events indexed

Les events peuvent avoir certains de leurs arguments marqués comme `indexed`. Cela permet de filtrer sur les valeurs ces arguments.

Exemple : `event Transfer(address indexed from, address indexed to, uint256 value)`.

- Si on considère deux adresses `x` et `y` :
- Il est possible d'écouter les transactions émises par `x`
- Il est possible d'écouter les transactions reçues par `x`
- Il est possible d'écouter les transactions émises par `x` vers `y`
- Il n'est pas possible d'écouter les transactions de 1 ETH ou moins car `uint256 value` n'est pas `indexed`.

# Solidity : erreurs

Souvent, il faut arrêter l'exécution d'un smart contract et renvoyer une erreur (exécution non autorisée, opération impossible, etc.).

Fonction `revert(bool assertion, string message)`  
Revert si `assertion` est évalué à `false`.

Custom errors : permet de déclarer des erreurs avec des paramètres.

```
1  contract Foo {  
2      error Custom(uint256 arg1);  
3  
4      function willRevert() public {  
5          revert Custom(1);  
6      }  
7  }
```

# Exemple : cryptomonnaie avec mint initial

```
1  contract Mewo {
2      uint256 constant MAX_SUPPLY = 10000000000; // 1 billion
3      mapping(address => uint256) public balances;
4
5      constructor() {
6          balances[msg.sender] += MAX_SUPPLY; // Initial mint
7      }
8
9      function transfer(address to, uint256 amount) public {
10         require(balances[msg.sender] >= amount, "Insufficient balance");
11         balances[msg.sender] -= amount;
12         balances[to] += amount;
13     }
14 }
```

# Exemple : cryptomonnaie

```
1  contract Mewo {
2      mapping(address => uint256) public balances;
3
4      function transfer(address to, uint256 amount) public {
5          require(balances[msg.sender] >= amount, "Insufficient balance");
6          balances[msg.sender] -= amount;
7          balances[to] += amount;
8      }
9  }
```

# Exemple : cryptomonnaie avec mint

```
1  contract Mewo {
2      mapping(address => uint256) public balances;
3
4      function mint(uint256 amount) public {
5          balances[msg.sender] += amount
6      }
7
8      function transfer(address to, uint256 amount) public {
9          require(balances[msg.sender] >= amount, "Insufficient balance");
10         balances[msg.sender] -= amount;
11         balances[to] += amount;
12     }
13 }
```

# Exemple : cryptomonnaie avec mint protégé

```
1  contract Mewo {
2      address owner;
3      mapping(address => uint256) public balances;
4
5      constructor() {
6          owner = msg.sender;
7      }
8
9      modifier onlyOwner() {
10         require(msg.sender == owner, "Only owner");
11         _;
12     }
13
14     function mint(uint256 amount) public onlyOwner {
15         balances[msg.sender] += amount;
16     }
17 }
```



# Notion de gas

- Les frais Ethereum ne paie pas à la transaction mais à la **complexité du calcul**
- Transférer de l'Ether entre deux comptes est beaucoup moins coûteux que faire participer à une enchère de NFTs
- La complexité des transactions en Ethereum se mesure en « gas »
- Le prix d'un « gas » se mesure en Gwei avec  $1 \text{ ETH} = 1000000000 \text{ Gwei}$ .
- L'utilisateur peut choisir le prix du « gas » affecté à sa transaction.

## Exemple : Mewo.mint

- Gas utilisé pour la fonction mint : 24634
- Prix du gas : 37 gwei/gas
- Prix d'un ETH = \$1,817.85
- Total =  $24634 \times 37 = 911458 \text{ gwei} = 0.000911458 \text{ ETH} = \$1.657$

# Standards de tokens

# Qu'est-ce qu'un token ?

## Définition : token

Dans le contexte de la blockchain Ethereum, un "token" fait référence à une unité de valeur numérique qui est émise et utilisée sur le réseau Ethereum.

- La création d'un token sur Ethereum est réalisée en mettant en place un smart contract qui définit les règles et les fonctionnalités spécifiques du token.
- Ce smart contract est ensuite déployé sur la blockchain Ethereum, ce qui lui permet d'interagir avec d'autres contrats et d'être utilisé par les utilisateurs du réseau.
- Les tokens sur Ethereum peuvent être transférés entre différentes adresses Ethereum, ce qui permet des transactions peer-to-peer.

# ERC-20 : besoin de standardisation

- Le standard ERC-20 a été créé dans le but de faciliter l'émission et l'interopérabilité des tokens sur la blockchain Ethereum.
- Avant l'introduction du standard ERC-20, chaque token avait son propre smart contract avec des règles et des interfaces spécifiques, ce qui rendait difficile l'interaction entre les tokens et leur intégration dans des applications tierces.
- Le standard ERC-20 définit **une interface** que doivent suivre les contrats qui l'implémentent.
- Grâce à cette norme, les tokens ERC-20 peuvent être facilement créés, gérés, échangés et utilisés par les utilisateurs, les portefeuilles et les plateformes d'échange.

# ERC-20 : tokens fongibles

Le standard ERC-20 définit les tokens « fongibles ».

## Définition : token fongible

Un token fongible est un type de token dans lequel chaque unité est interchangeable avec une autre unité du même type. Cela signifie que chaque token fongible est considéré comme équivalent et peut être remplacé par un autre token identique, sans qu'il y ait de distinction ou de différence entre eux.

- C'est la perception « naturelle » de la monnaie : un billet de 5 euros a exactement la même valeur qu'un autre billet de 5 euros.
- Chaque unité d'un token fongible ERC-20 est identique en termes de valeur, de fonctionnalité et de propriétés.

# ERC-20 : interface

```
1 interface IERC20 {
2     function name() public view returns (string)
3     function symbol() public view returns (string)
4     function decimals() public view returns (uint8)
5     function totalSupply() public view returns (uint256)
6     function balanceOf(address _owner) public view returns (uint256 balance)
7     function transfer(address _to, uint256 _value) public returns (bool success)
8     function transferFrom(address _from, address _to, uint256 _value) public returns
   ↪ (bool success)
9     function approve(address _spender, uint256 _value) public returns (bool success)
10    function allowance(address _owner, address _spender) public view returns (uint256
   ↪ remaining)
11
12    event Transfer(address indexed _from, address indexed _to, uint256 _value)
13    event Approval(address indexed _owner, address indexed _spender, uint256 _value)
14 }
```

# ERC-20 : metadata

Le nom et le symbole d'un token ERC-20 sont encodés dans le contrat avec les fonctions `name` et `symbol`.

```
1  contract ERC20 {
2      function name() public view returns (string) {
3          return "MyToken";
4      }
5
6      function symbol() public view returns (string) {
7          return "MYT";
8      }
9  }
```

# ERC-20 : decimals

Solidity ne supporte pas les nombres flottants donc il n'est a priori pas possible d'échanger 0.5 MYT. La technique consiste à considérer que :

$$1\text{MYT} = 1 \underbrace{000000000000000000}_{\text{decimals} = 18}$$

```
1  contract ERC20 {  
2      function decimals() public view returns (uint8) {  
3          return 18;  
4      }  
5  }
```



# ERC-20 : transfer de tokens

- Le nombre de tokens détenus par une adresse s'obtient avec la fonction `balanceOf`.
- Une adresse peut envoyer des tokens à une autre adresse avec la fonction `function transfer(address to, uint256 amount)`. Contraintes :
  - `to` ne peut pas être l'adresse zéro
  - `msg.sender` doit posséder au moins `amount` tokens

## ERC-20 : transferFrom

Intégrer les tokens ERC-20 avec d'autres smart contracts nécessite de pouvoir transférer au nom de l'utilisateur.

Cela permet notamment d'intégrer des fonctionnalités du type « envoyez des tokens et pourrez faire la chose X ».

Exemple : envoyez 1 USDC et recevez 5 MYT.

```
1  contract Sale {
2      MyToken myt;
3      ERC20 usdc;
4
5      function buy(uint256 amount) public {
6          usdc.transferFrom(msg.sender, address(this), amount);
7          myt.mint(address(this), amount);
8      }
9  }
```

# ERC-20 : allowance

Problème : il faut protéger la fonction `function transferFrom`

## Définition : allowance

Dans le contexte du standard ERC-20, **allowance** fait référence à une fonctionnalité qui permet à une adresse Ethereum spécifique d'accéder à un certain nombre de tokens détenus par une autre adresse Ethereum et de les transférer ultérieurement.

L'allowance est souvent utilisée dans le cadre des transferts de tokens ERC-20 à des tiers de confiance, tels que des contrats ou des dApps (applications décentralisées). Elle permet à un propriétaire de tokens de définir une autorisation spécifique pour un dépensier donné. Cette autorisation est enregistrée dans le smart contract du token ERC-20.

## ERC-20 : allowance

La fonction `approve` permet de définir l'allowance d'une adresse Ethereum.

Reprenons le contrat `Sale` avec les contrats :

- Bob, notre utilisateur
- USDC, un stablecoin
- MyToken, notre token ERC20
- `Sale`, qui vend des MYT contre des USDC

Nous allons analyser l'utilisation de l'allowance par Bob.

Situation initiale (`msg.sender = address(Bob)`) :

- `USDC.balanceOf(address(Bob)) = 10000`
- `USDC.allowance(address(Sale)) = 0`
- `MyToken.balanceOf(address(Bob)) = 0`

Bob envoie `USDC.approve(address(Sale), 2000)`

- `USDC.allowance(address(Sale)) = 2000`

Bob envoie `Sale.buy(1000)`

- `USDC.balanceOf(address(Bob)) = 9000`
- `USDC.allowance(address(Sale)) = 1000`
- `MyToken.balanceOf(address(Bob)) = 5000`

## ERC-20 : `increaseAllowance` / `decreaseAllowance`

En plus de la fonction `approve` certaines implémentations exposent des fonction non-standard permettant de contrôler plus finement l'`allowance`.,

- `increaseAllowance` qui correspond à l'opération `allowance[spender] += amount`
- `decreaseAllowance` qui correspond à l'opération `allowance[spender] -= amount`

Ces deux fonctions sont notamment présentent dans l'implémentation d'OpenZeppelin.

## Mars 2025

# ERC-721

Le standard ERC-721 définit les tokens « non-fongibles », appelées « Non-Fungible Tokens ».

## Définition : token non fongible (NFT)

Un token non fongible (NFT) est un type de token unique et indivisible sur une blockchain. Chaque NFT est distinct et possède des caractéristiques qui le rendent unique.

- Les NFTs tirent leur valeur de leur unicité, de leur rareté et de leur authenticité, ce qui les rend particulièrement adaptés à la représentation d'actifs numériques uniques et à leur propriété vérifiable sur la blockchain.
- Chaque NFT possède un identifiant unique qui le distingue des autres tokens, et ses propriétés et sa provenance sont enregistrées dans le registre immuable de la blockchain.

Exemple hors blockchain : certaines anciennes pièces de monnaies valent plus que leur valeur faciale (numismatique).

# ERC-721 interface

```
1 interface ERC721 {
2     function balanceOf(address _owner) external view returns (uint256);
3     function ownerOf(uint256 _tokenId) external view returns (address);
4     function safeTransferFrom(address _from, address _to, uint256 _tokenId, bytes data)
5         ↪ external payable;
6     function safeTransferFrom(address _from, address _to, uint256 _tokenId) external
7         ↪ payable;
8     function transferFrom(address _from, address _to, uint256 _tokenId) external
9         ↪ payable;
10    function approve(address _approved, uint256 _tokenId) external payable;
11    function setApprovalForAll(address _operator, bool _approved) external;
12    function getApproved(uint256 _tokenId) external view returns (address);
13    function isApprovedForAll(address _owner, address _operator) external view returns
14        ↪ (bool);
15 }
```



# ERC-721 : comparaison avec l'ERC-20

- `balanceOf` renvoie le nombre de NFT possédés par une adresse
- pas de fonction `transfer`, uniquement `transferFrom` (un transfer « classique » se fait en appelant `transferFrom` avec sa propre adresse en premier argument)
- la fonction `safeTransferFrom` permet de s'assurer que le destinataire est un EOA ou un contract compatible
- `safeTransferFrom` prend en paramètre l'identifiant du NFT, pas le nombre de tokens à transférer

# ERC-721 : approve / getApproved

- `approve(address operator, uint256 tokenId)` permet d'autoriser le l'adresse operator à appeler la fonction `transferFrom`.
- Il ne peut y avoir qu'un seul operator approuvé, c'est-à-dire que seule la dernière adresse utilisée avec la fonction `approved` possède les privilèges.
- Le dernier opérateur approuvé est obtainable en appelant la fonction `getApproved(uint256 tokenId)`.

## ERC-721 : setApprovalForAll / isApprovedForAll

- Il est possible d'autoriser une adresse pour toute une collection (très pratique sur les marketplaces par exemple) avec la fonction `setApprovalForAll(address operator, bool approved)`.
- Le getter associé est `isApprovedForAll`.

Attention, `setApprovalForAll` peut être dangereux :

- L'autorisation s'applique à toute la collection, y compris sur les tokens reçu dans le futur.
- Si l'adresse de l'opérateur est compromise, la collection peut être entièrement volée.

# Autres standards

- Le standard ERC-1155 est un hybride entre les standard ERC-20 et ERC-721. Il permet de créer de multiples tokens, fongibles ou non fongibles, dans un seul contrat.
- Le standard ERC-777 fut une tentative d'améliorer le standard ERC-20 mais il a récemment été déprécié par de nombreuses bibliothèques dû à son manque d'adoption.

# Tester avec Foundry

# Tester un contrat ?

## Pourquoi ?

- Les contrats sont des programmes exposés aux utilisateurs finaux  $\Rightarrow$  un bug peut engendrer **une perte financière** pour l'utilisateur
- Les contrats sont immutables = une fois déployés, il ne peuvent être modifiés
- En gros, les contrats doivent être **parfaits du premier coup**.

## Comment ?

- ➊ Imaginer et créer un scénario (cela revient à définir les variables d'état d'un contrat)
- ➋ Exécuter la ou les transactions à tester
- ➌ Vérifier que le nouvel état correspond à celui attendu.

# Exemple : Ownable

```
1  contract Ownable {
2      address public owner;
3
4      modifier onlyOwner() {
5          require(msg.sender == owner);
6      }
7
8      function transferOwnership(address
9          ↪ _owner) onlyOwner {
10         owner = _owner;
11     }
```

Tester la fonction `transferOwnership`, c'est :

- 1 `transferOwnership` renvoie une erreur si `msg.sender` n'est pas l'owner
- 2 `transferOwnership` ne renvoie pas d'erreur si `msg.sender` est l'owner et que le nouvel owner est `msg.sender`

# Foundry : créer un test

- Dans Foundry, les tests s'écrivent en Solidity.
- Par convention, on teste le fichier `Contract.sol` dans `Contract.t.sol`
- Deux méthodes possibles pour placer les tests :
  - juste à côté du fichier source
  - dans le dossier test

```
1  contract OwnableTest {
2      address user1 = makeAddr("user1");
3      Ownable ownable;
4
5      function setUp() public {
6          Ownable ownable = new Ownable();
7          ownable.transferOwnership(user1);
8      }
9
10     function test_owner() public {
11         assertEq(ownable.owner(), user1);
12     }
13 }
```



# Foundry : cheat codes

Les cheatcodes permettent d'effectuer des actions normalement impossibles dans la blockchain. Voici les principaux :

- `hoax(address who)` le prochain call sera fait en tant que `who`
- `hoax(address who, uint256 amount)` le prochain call sera fait en tant que `who` et sa balance en ether sera de `amount`
- `deal(address who, uint256 amount)` change la balance de `who` en `amount`
- `startHoax(address who)` et `stopHoax()` permettent de garder l'impersonification sur plusieurs calls

# Foundry : assertions

Les assertions permettent de valider l'exactitude des données / variables. Foundry propose notamment les assertions suivantes :

- `assertEq(x, y)` vérifie que x et y sont égaux
- `assertApproxEqAbs(x, y, maxDelta)` vérifie que x et y sont égaux +/- maxDelta
- `assertTrue(bool cond)` vérifie que cond est égal à `true`
- `assertFalse(bool cond)` vérifie que cond est égal à `false`

# Projets

# Consignes générales

Ça y est, l'heure de l'évaluation est arrivée ! Voici quelques consignes pour votre projet de fin de module.

- Je vais présenter des sujets différents, choisissez-en **un seul**.
- Aucun problème si un sujet n'est pas pourvu/choisi par beaucoup d'étudiants.
- Les instructions donnent des pistes, ne sont pas exhaustives.
- Développez une application qui a **du sens pour vous**.
- Tous les projets contiennent du Solidity et vous devrez tester vos smart contracts.
- Envoyez votre code source et vos tests sur GitHub.
- Sécurité avant tout : je serais le hacker qui tentera d'exploiter votre protocole.
- La pression, on la boit :)

# Projet 1 : Casino avec nombres aléatoires issus de Chainlink VRF

Vous souhaitez construire un casino dans la blockchain. Pour cela, vous devrez imaginer un jeu de hasard permettant de parier de l'Ether.

Le hasard n'existant pas dans la blockchain Ethereum, vous utiliserez le service Chainlink VRF pour générer vos nombres aléatoires.

Règles du jeu :

- Chaque joueur envoie la quantité d'Ether de son choix
- Quand tous le dernier joueur a envoyé son part, un nombre aléatoire est tiré
- À partir du nombre, le jeu choisi un gagnant
- Le gagnant rafle tout !

## Projet 2 : Faucet React (front-end)

Le but de ce projet est de créer une application web React qui aura pour but de permettre à un utilisateur de demander à recevoir des tokens ERC-20. Pour cette application, il faudra :

- créer un token ERC-20 de votre choix
- utiliser React (je suggère Next.js / Vite) et la bibliothèque Wagmi pour les smart contracts
- l'utilisateur appellera lui-même la fonction de mint
- bonus : inclure une limite de requêtes/montant par jour

The mockup shows a web application titled "SEPOLIA FAUCET" with the tagline "Fast and reliable. 0.5 Sepolia ETH/day." Below the title is a text input field labeled "Enter Your Wallet Address (0x...) or ETH Mainnet ENS Domain". A prominent "Send Me ETH" button is positioned below the input field. Underneath the button, a link reads "Please [signup or login](#) with Alchemy to request ETH. It's free!". A reCAPTCHA verification box follows, containing the text "I'm not a robot" and the reCAPTCHA logo with links for "Privacy" and "Terms". At the bottom, a section titled "Your Transactions" is shown with a single entry represented by a hyphen "-".

## Projet 3 : Marketplace NFT

Implémentez un marketplace NFT permettant de mettre en vente des NFT au standard ERC-721. Sur cet échange, il y aura deux acteurs : les vendeurs et les acheteurs.

Les vendeurs peuvent :

- mettre en vente un NFT à un certain prix en Ether ou ERC-20
- changer le prix d'une de leurs offres
- annuler une de leurs offres

Les acheteurs peuvent :

- accepter une offre d'un vendeur s'il possède les tokens nécessaire

Le contrat doit exposer des fonctions permettant de lister les offres par collection (adresse du token ERC-721).

## Projet 4 : Système de vote par token ERC-20

Les organisations autonomes décentralisées (DAO) sont des acteurs majeurs des blockchains. Elles permettent de gouverner des projets grâce à des systèmes de vote décentralisés.

Vous devrez créer un token ERC-20 qui permet d'accéder à un contrat de vote. Ce contrat permettrait au détenteurs du tokens de voter des propositions **au prorata** de leur balance.

Les fonctionnalités attendues sont :

- tout détenteur de plus de X tokens peut faire une proposition (à vous de choisir la structure de donnée d'une proposition...)
- tout vote pour toute proposition possède une date de fermeture
- tout détenteur de tokens peut voter pour une proposition (oui/non)
- **attention : assurez-vous que les tokens d'un utilisateur ne puissent pas être utilisés plusieurs fois**
- bonus : implémentez un quorum minimum à atteindre pour qu'une proposition soit acceptée



## Projet 5 : Plateforme de Staking

Le staking consiste à détenir et à bloquer des tokens pour soutenir les opérations du réseau blockchain et en tirer des récompenses. Il permet aux détenteurs de participer à la sécurisation du réseau et de gagner des intérêts sur leurs avoirs.

Développez un système permettant à des détenteurs d'un token ERC=20 de bloquer leurs tokens et de gagner des récompenses sur le temps. Les fonctionnalités attendues sont :

- tout détenteur peut bloquer tout ou une partie de ses tokens pour une durée déterminée
- durant cette période, l'utilisateur va obtenir des récompenses en fonction de la quantité de tokens bloqués et de la durée depuis le début du staking
- à la fin de la période de staking, l'utilisateur peut :
  - retirer ses tokens et ses récompenses
  - re-staker ses tokens et ses récompenses

## Projet 6 : Jeu de prédiction

Les cryptomonnaies sont connues pour leur grande volatilité. Le but de ce projet est de proposer aux utilisateurs de parier sur la montée ou la descente d'actifs.

Vous utiliserez les Chainlink Data Feeds comme source de vérité pour les prix.

Les fonctionnalités attendues sont les suivantes :

- tout utilisateur peut placer une prédiction sur une cryptomonnaie et sur la montée / descente (à vous de définir les règles exactes)
- les gains des gagnants sont payés par les mises des perdants
- le contrat doit conserver une commission d'un certain pourcentage sur les gain

## Projet 7 : Collection de NFT avec des traits

Les collections de NFTs possèdent bien souvent des traits, qui des caractéristiques off-chains. Les traits des NFT sont définis dans un fichier JSON dont l'URL est renvoyée par la fonction :

```
1 interface IERC721Metadata {  
2     function tokenURI(uint256 tokenId) returns (string memory);  
3 }
```

Créez une collection de NFT mintable via un contrat de vente supportant les traits :

- exposer les manifestes JSON des tokens via une petite application back-end (langage au choix)
- stockez les caractéristiques des tokens dans une base de données
- générez les traits au hasard lorsqu'un token est minté :
  - soit en utilisant un client Ethereum et en écoutant les events de Mint
  - soit en utilisant le produit Alchemy Custom Webhooks