Blockchain Technologies 1

Assignment - 1

Aibek

SE-2419

Contents

Module 1: Introduction to Blockchain Technology

1. Distributed vs. Centralized Ledgers

Explain how a distributed ledger differs from a centralized ledger in terms of trust, confidentiality, fault-tolerance, and attack surface. Provide at least 3 real-world examples for each.

Answer:

The essential discrepancy between distributed ledger (DL), commonly represented by blockchain technology, and centralized ledger (CL), usually a traditional database, is their architecture and the mechanisms they use for data integrity and coordination. A centralized system has clients that are connected to a single central server governed by an administrator, while a distributed ledger network is based on the peer-to-peer model where control as well as data distribution occur across many nodes.

Trust

Centralized Ledger

- **Trust Model**: Centralized systems rely on a trust-based model. Trust is implicitly or explicitly placed in a single central authority, administrator, or intermediary (like a bank) who manages the entire system and controls the data
- **Technical Reasoning**: The integrity and authenticity of the ledger are maintained exclusively by this single entity, meaning there is no technical guarantee against malicious actions by the controller

Distributed Ledger

- **Trust Model**: Distributed ledgers operate on a trustless model (or distributed trust). Trust is established and maintained through cryptographic security and a consensus mechanism rather than relying on a single third party,,. Participants collectively agree on the state of the network.
- **Technical Reasoning:** Consensus protocols (like Proof-of-Work or Byzantine Fault Tolerance variants) ensure that new records are added only if participants collectively agree to do so. Transparency, where all participants possess the same verifiable information, reinforces this trust among participants

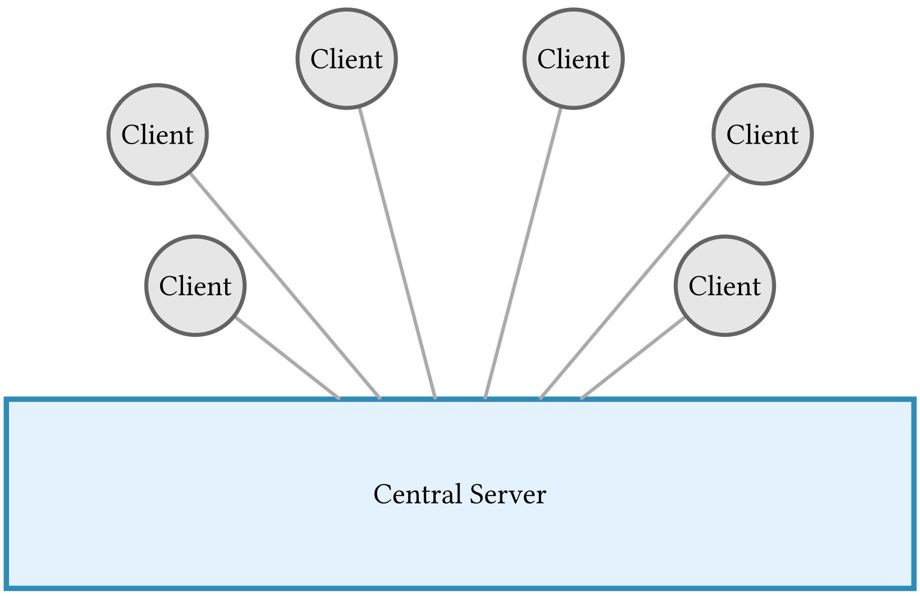


Figure 1: Centralized Ledger (CL): Single Point of Control

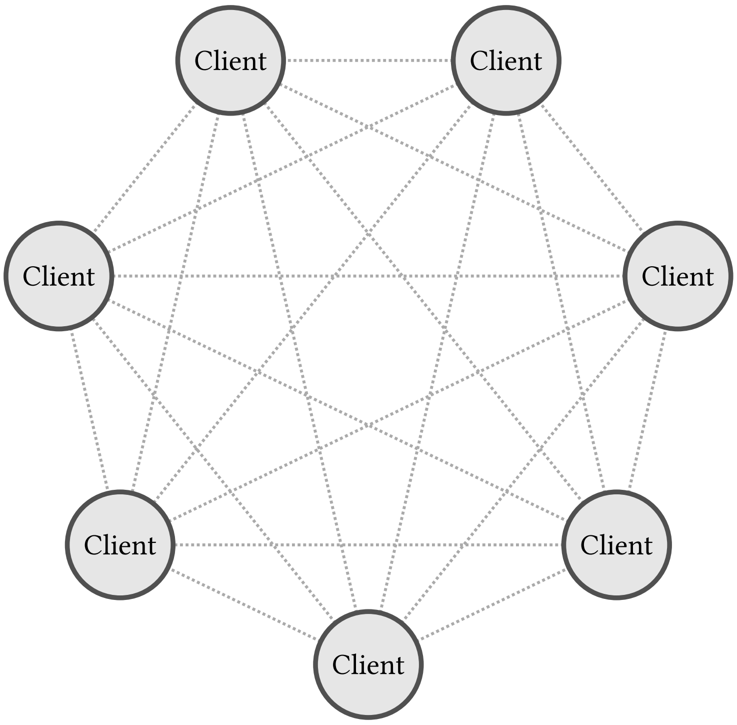


Figure 2: Decentralized Ledger (DL): No Single Point of Control

Confidentiality

Centralized Ledger:

- **Data Access:** The central authority is responsible for maintaining confidentiality and the access control policies established by it are the means through which this is done. The data is so to speak owned by the controlling entity, and access is limited to a few select individuals.
- **Technical Reasoning:** The central server's data security utilizes traditional methods such as authentication, access control, and physical security measures.

Distributed Ledger:

- **Data Access:** Confidentiality is a big issue and very different for each type of DL:

  - Public (Permissionless) DLs has the main concern of transparency, thus all transactions are recorded in a shared, publicly available digital ledger. The user identities are usually anonymous but the transaction data is generally public.
  - Private/Consortium (Permissioned) DLs have the access limited only to the known participants, thus providing both confidentiality and transparency only within that group.
- **Technical Reasoning:** Cryptography is the main source of security for confidentiality and uses methods like encryption to the communication links (in transit) or the data when it is stored (at rest)

Fault-tolerance

Centralized Ledger:

- **Resilience:** Centralized systems have very low fault tolerance as it is based on the one and only central server. This is exactly what a single point of failure means.
- **Technical Reasoning:** When the only central node ceases to function, the whole database becomes user-unreachable. The system is stuck because there is no distributed copy or coordinated failover, which is not the case since the core design is not supporting it.

Distributed Ledger:

- **Resilience:** High fault-tolerance is the very property of distributed ledgers. The reason is that the data being distributed over many nodes, and the system is still available and on-going if one or more nodes go down.
- **Technical Reasoning:** Replication is the way to bring about fault tolerance. Typically, distributed systems that require consensus operate based on strict thresholds to guarantee safety, such as $N \geq 3F+1$ for tolerating Byzantine faults (where N is total nodes and F is faulty nodes). The system achieves liveness as long as a majority or a sufficient quorum of non-faulty nodes remains up and running and continues to process requests

Attack Surface

Centralized Ledger:

- **Vulnerability:** The entire attack surface is concentrated in one single point-the central authority.
- **Technical Reasoning:** In total, an invader aspiring to disrupt, corrupt, or freeze the system will only have to take over the central node (the database, API endpoints, or the governing administrator)together with the system. Once the point of failure is breached, the hacker has full control of the data and system operations,.

Distributed Ledger:

- **Vulnerability:** The network's overall security is based on the fact that all participating nodes are the points of attack. Through cryptographic hashing, the data is secured so that a chain is formed which is almost impossible to alter and which has very high resistance to penetration through tampering.

- **Technical Reasoning:** An adversary who wants to execute a devastating attack must first break the economic barriers and seize the control of a considerable part of the network which usually means a supermajority or more than 50% of the computational power (in Proof of Work networks like Bitcoin) or over one-third of the total validator stake/nodes (in BFT/PoS networks). Changing a record is practically impossible because its hash is linked to consecutive blocks, hence altering one record requires recalculating the hashes of all subsequent blocks, making data tampering-proof.

Feature	Centralized Ledger	Distributed Ledger
Trust	Relies on a single authority	Trust is distributed among nodes
Confidentiality	High (Owner controls access)	Varies (Public vs. Permissioned)
Fault-tolerance	Low (Single point of failure)	High (Redundancy)
Attack Surface	Central server is the target	Consensus mechanism/Nodes

Table 1: CL vs DL (feautres)

Ledger Type	Trust	Confidentiality	Fault-Tolerance
Centralized	Traditional Banking Systems (like SWIFT, single bank ledgers)	Corporate ERP Systems (data managed by the company)	Traditional Web2 Applications (for example, a service running on one cloud server/database)
Distributed	Bitcoin (Public, permissionless network based on Proof-of-Work)	Ethereum (Public DApp platform using smart contracts and consensus)	Consortium Blockchains (like supply chain solutions involving several organizations using permissioned networks)

Table 2: Real-World Examples

2. Definition of Immutability

Provide a rigorous technical definition of immutability. Explain how hash functions contribute to this and describe one scenario where immutability fails.

Answer:

Immutability in a blockchain refers to an unalterable and non-removable transaction after it has been confirmed and included in a block that belongs to the canonical chain. The ledger is only for adding new transactions: newly processed blocks can only be added one after another, and the entire history is kept permanently. Mistakes are not rectified by replacing them in the past, but rather with the addition of new reversing transactions.

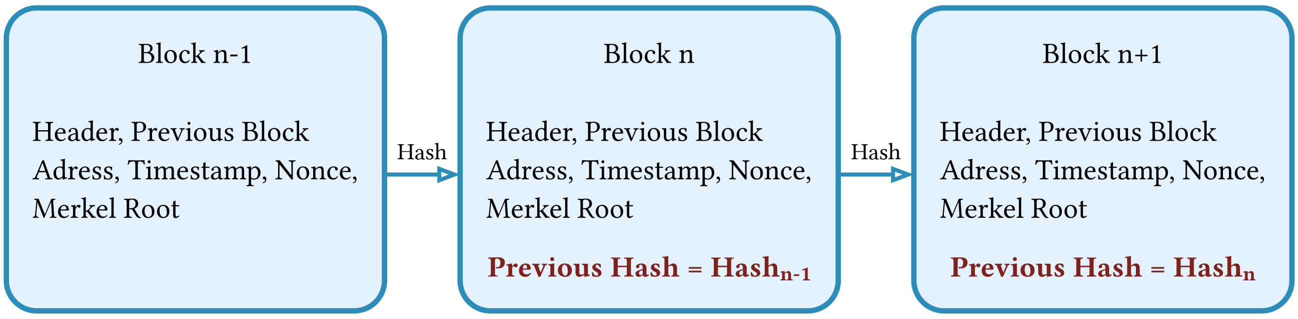
How Hash Chaining Ensures Immutability The link between the blocks is established with the cryptographic hash of the preceding block:

An alteration of even 1 bit in a past block dramatically changes its hash (avalanche effect).

This leads to the "previous hash" reference in the next block being broken.

To cover up the tampering, a hacker needs to perform the same intense computing work that involves re-mining the compromised block plus all the consecutive blocks.

In a Proof-of-Work system, the massive power cost needed makes this practically impossible.



Block n-1
Header, Previous Block Adress, Timestamp, Nonce, Merkel Root
Hash
Block n
Header, Previous Block Adress, Timestamp, Nonce, Merkel Root
Previous Hash = Hash$_{n-1}$
Hash
Block n+1
Header, Previous Block Adress, Timestamp, Nonce, Merkel Root
Previous Hash = Hash$_n$

Blockchain Hash Chaining – Enforcing Immutability

Figure 3: Each block stores the hash of the previous block (in red).
Modifying any block changes its hash, breaking the chain.

Immutability isn't absolute it's economic finality. Suppose an adversary gains control of the hash power of the network that is more than 50%.

They first make the transaction public and then create a hidden chain that does not include that transaction and is longer than the public one. When the hidden chain is revealed, the participants in the network apply the longest-chain rule and accept it. The time of the original transaction is then erased from the canonical history ⟶ successful double-spend.

In this way, the immutability of the blockchain is guaranteed by cryptography but the costs that have to be borne for acquiring majority hash power finally determine the security.

3. Transparency vs. Privacy

Evaluate blockchain transparency vs. privacy.

- Compare Bitcoin vs. Ethereum
- Explain mixers, stealth addresses, and ZK proofs.

Answer:

Blockchain Transparency vs. Privacy

Blockchain technology in its very nature supports transparency. Transparency is considered the primary attribute since the global digital ledger used for recording transactions is accessible and verifiable by all. This common and checkable data creates trust between the parties and ensures that there is always evidence because all the information is capable of being traced. Thus, transparency together with immutability creates an environment that does not need the involvement of third-party intermediaries which results in lower costs and faster processing.

The conflict between transparency and privacy is evident, on one hand, the decentralization of the network renders intermediaries unnecessary, on the other hand, the requirement for digital cash to be decentralized implies that both accountability (the prevention of double-spends) and anonymity (the grant of privacy) have to be dealt with. Regardless, the very nature of blockchain guarantees that each transaction is public and can be verified. In a public blockchain scenario, which is characterized by wide geographical distribution and lack of trust, there might be some bad actors that try to listen in, therefore, in these situations, encryption is the standard method used to ensure confidentiality.

Comparison of Bitcoin and Ethereum

When Bitcoin and Ethereum are fundamentally different in their approach, in terms of both purpose and consensus mechanism, one finds the real point of comparison coming with transaction throughput, speed, and other performance metrics.

	Bitcoin	**Ethereum**
Purpose	A credible alternative to traditional fiat currencies (medium of exchange, potential store of value)	A platform to run programmatic contracts and applications via Ether
Consensus	Proof-of-Work (PoW)	Proof-of-Stake (PoS) (as per forecast, reflecting the transition)
Transaction Model	Unspent Transaction Output (UTXO). Your balance is the sum of all distinct, unspent outputs, which must be spent entirely in a single transaction	Account-based model (Externally Owned Accounts and Contract Accounts),. An account holds a single balance that increases or decreases
Block Time	10 minutes on average	12 seconds on average
TPS	3-7 transactions per second	10-30 transactions per second
Supply	Finite supply, capped at 21 million BTC	No fixed maximum supply, issuance to validators but some ETH is burned, sometimes making supply net-deflationary

Table 3: Bitcoin vs Ethereum

Mixers

Services or protocols that break the on-chain link between source and destination of funds.

Mixers take the coins from a lot of users and mix them up so that it is hard to identify which output is from which input. For Bitcoin, this is usually done through collaborative transactions (like CoinJoin), whereas for Ethereum, mixers mostly rely on smart contracts that take deposits and later let people withdraw to a new address using a secret. On the one hand, mixers enhance privacy, on the other hand, they can create compliance and legal issues if used to hide illegal money.

Stealth address

Stealth addresses generate unique, one-time payment addresses through cryptographic key agreements. This ensures the recipient's privacy by making it impossible to trace the connection between their public identity and on-chain transactions.

Stealth addresses hide the connection between a receiver's public identity and the real address that receives funds. Thus, they protect the privacy of the recipient rather than that of the sender or the amount. The usual scenario is that the receiver discloses some long term public key data, the sender and the receiver produce a one-time payment address using a Diffie–Hellman like key agreement and a derivation scheme, which makes it appear to the blockchain as if the recipient is using a new random address that is not easily associated with them. This provides a level of privacy comparable to using a new address for each payment but with on-chain derivation and optional scanning keys, so only the recipient can recognize and spend from these stealth outputs.

ZK proofs and privacy

Zero-knowledge (ZK) proofs let a party prove that "this transaction or statement is valid under the rules" without revealing underlying data such as which exact notes are spent, which address is used, or what attribute values are. On Ethereum, ZK is used in several ways:

- Mixers like Tornado Cash: prove "I own one of the deposits in this Merkle tree" without revealing which leaf.
- ZK-rollups and private payment/DEX systems: prove correctness of batched transactions while hiding individual details.
- ZK identity: prove properties such as age, membership, or ownership of a credential without disclosing the identifier or full record, enabling Sybil-resistant yet privacy-preserving identity.

Overall, Bitcoin's base layer leans heavily toward transparent auditability with bolt-on privacy tools, while Ethereum's programmable environment exposes more activity but also makes sophisticated privacy primitives like mixers, stealth address schemes, and ZK systems first-class smart contract applications.

4. DApp Architecture

Define DApp architecture in detail.
Describe how the following components interact:
- Smart contract layer
- Off-chain backend
- Frontend
- Wallets (EIP-1193 providers)
- Nodes (RPC, full, light)

Answer:
A Decentralized Application (DApp) is a software program that runs on a blockchain or a peer-to-peer (P2P) network of computers and thus does not depend on centralized servers. DApps rely on smart contracts to carry out their core consumptions, in this way, all operations are decentralized, transparent, and secured by cryptography. One of the key differences between DApps and Web2

applications is the way they handle data and business logic: the former spreads them over various nodes while the latter concentrates them on one or more servers and thus gets rid of bugs, allows for constant improvements, etc., at the cost of centralization.

The DApp architecture is based on three main pillars: the frontend (user interface), the smart contract layer (business logic), and the blockchain (data storage).

Component Interactions in Transaction Flow

The transactions in a DApp come along with a structured flow starting from the user input to the on-chain execution, and this is the point where the seamless interactions among the components happen.

Frontend

The frontend user interface (UI) is the and is usually made using web technologies like HTML, CSS, and JavaScript frameworks (e.g. React). It handles the user actions like transfer and swap that happen with the help of button clicks and creates the transaction payload that calls the specific functions of smart contracts. In this part, there are no changes of states and it acts like a bridge that connects users to the blockchain with the help of libraries such as ethers.js or web3.js.

Wallets (EIP-1193 Providers)

Wallets are usually browser extensions like MetaMask and represent Externally Owned Accounts (EOAs) that are controlled by private keys and follow EIP-1193 standards for provider interfaces. When the frontend makes a request, the wallet asks for user approval, then signs the transaction with cryptographic methods, and finally estimates and pays the gas fee. The wallet then sends back the signed transaction to the frontend so it can be sent out.

Smart Contract Layer

Smart contracts (for example, on Ethereum) are an inherent part of the blockchain that contain immutable bytecode executing the business logic in the Ethereum Virtual Machine (EVM). When a signed transaction is available, the EVM acts on the called function in accordance with the rules laid down beforehand, modifies the global state (e.g. balances) and generates events for logging. The process is predictable and consensus is maintained among nodes.

Nodes (RPC, Full, Light)

Nodes are the major part of the P2P blockchain network maintainers, they store and validate the ledger.

Remote Procedure Call (RPC):

Acts as the API gateway, e.g., JSON-RPC via Infura or Alchemy, facilitating transactions and balance or event querying by the frontends and wallets.-

Full nodes:

Keep the complete chain history, run all transactions via EVM, block validation, and consensus enforcement.-

Light nodes:

Have less resource requirements, keep only block headers and apply Simplified Payment Verification (SPV) to connections with full nodes for querying without entire storage.

Step	Components involved	Action
1. Initiation	Frontend ⟶ Wallet	UI captures input, wallet signs tx fiveable+1
2. Broadcast	Wallet ⟶ RPC Node	Signed tx sent via RPC geeksforgeeks
3. Validation/ Execution	Full/Light Nodes ⟶ Smart Contracts	EVM runs logic, state updates fiveable
4. Confirmation	Nodes ⟶ Frontend/Off-chain	Polling/receipts update UI, events indexed geeksforgeeks

Table 4: Transaction Flow Summary

This architecture ensures trustless, verifiable operations ideal for applications like DeFi and NFTs.

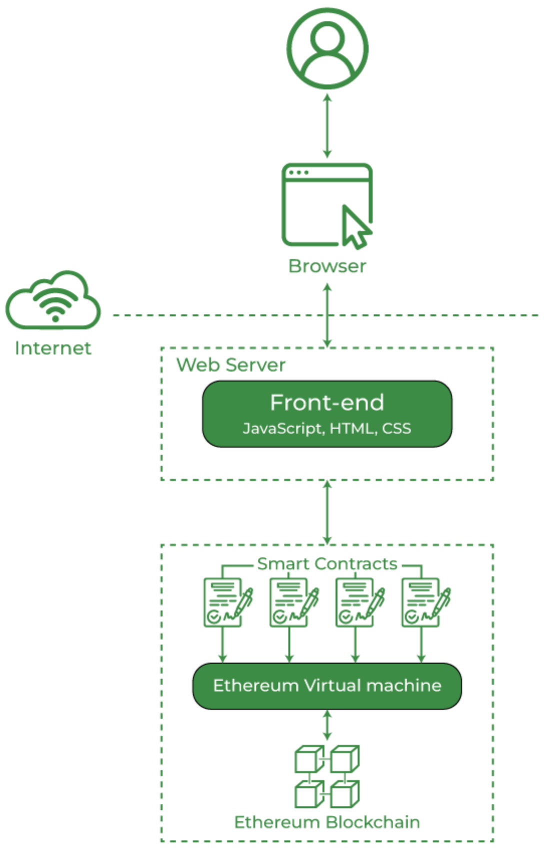


Figure 4: DApp Architecture diagram

The architecture of the Decentralized Application (DApp) is presented in an uncomplicated diagram (Figure 4) consisting of three layers: first, a user actions take through a frontend based on the browser, and then the frontend connects to the smart contracts running in the Ethereum Virtual Machine (EVM) on the Ethereum blockchain.

Moving from the top to the bottom:

- First and foremost, the user through a browser gets the frontend (HTML, CSS, JavaScript) from a web server over the Internet.

- Then the frontend sends requests (such as sending transactions or calling functions) to the smart contracts that have already been deployed.
- Finally, the execution of the smart contracts takes place in the Ethereum Virtual Machine, which consequently updates and reads data from the Ethereum blockchain thus achieving a decentralized storage of contract state and transaction history.

Module 2: Cryptography Fundamentals

1. SHA-256 Computations

Compute SHA-256 hashes using at least two tools

Answer:

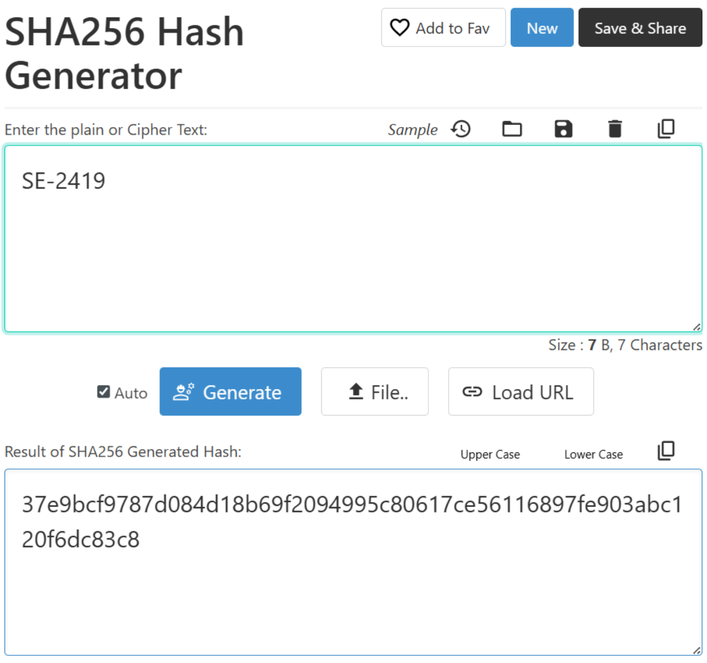
1. Node.js Code:

JavaScript
```javascript
const crypto = require('crypto');

function createSHA256Hash(inputString) {
  const hash = crypto.createHash('sha256');
  hash.update(inputString);
  return hash.digest('hex');
}

const myString = 'SE-2419';
const hash = createSHA256Hash(myString);
console.log(hash);
```

1. Node.js Output:

Shell
```shell
37e9bcf9787d084d18b69f2094995c80617ce56116897fe903abc120f6dc83c8
```

2. Online hashing tool



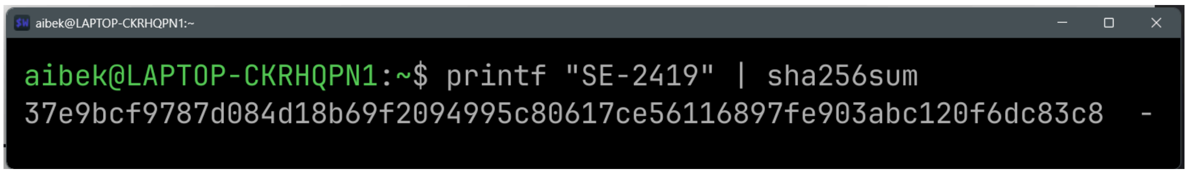
```
SE-2419
```

Size : 7 B, 7 Characters

```
37e9bcf9787d084d18b69f2094995c80617ce56116897fe903abc120f6dc83c8
```

Figure 5: Screenshot of online hashing website

3. Linux Terminal:



```
aibek@LAPTOP-CKRHQPN1:~
aibek@LAPTOP-CKRHQPN1:~$ printf "SE-2419" | sha256sum
37e9bcf9787d084d18b69f2094995c80617ce56116897fe903abc120f6dc83c8  -
```

```Shell
prinitf "SE-2419" | sha256sum
37e9bcf9787d084d18b69f2094995c80617ce56116897fe903abc120f6dc83c8  -
```

Listing 2: Code

2. Comparison

Write a comparison of the outputs, and explain why all must be identical despite different tools.

Answer:
The output of the computed SHA-256 hash of the string "SE-2419" was the same on all three platforms: Node.js, the online hashing tool, and the Linux `sha256sum` utility.

The uniformity of the output is one of the major characteristics of cryptographic hash functions and is determined by the idea of determinism.

Deterministic Algorithm:
SHA-256 is a universally accepted deterministic mathematical algorithm that operates through the application of a standard. Consequently, the hash function for any particular input sequence of bits will always output a fixed-size 256-bit output. It does not matter which tool, programming language, or operating system is employed for the calculation as long as the strict following of the Secure Hash Standard (NIST FIPS 180-4) is ensured.

The Role of Byte Stream: n If the outputs were not the same, the most likely cause of the difference would be a variation in the input byte stream. For instance, one tool might have included a hidden character (like a null terminator or a space) that the other tools did not. But since the input byte stream for "SE-2419" was the same for all methods, the hash output was the same: 37e9bcf9787d084d18b69f2094995c80617ce56116897fe903abc120f6dc83c8

3. Collison resistance

- Attempt to modify one bit of input
- Show how drastically the hash changes
- Explain why finding two identical SHA-256 hashes is computationally infeasible
- Estimate the probability of collision using the birthday paradox formula

Answer:
In a bid to showcase the collision resistance property of the SHA-256 algorithm, we will take the original input "SE-2419" along with its hash and then carry out a minimal alteration on the input by appending a newline character, "n", which is a minor change of only one byte in the sequence adding a single byte of value 0x0A. This can be seen as a "one-bit" or minimal perturbation, although the adding of "n" alters 8 bits, however, it shows the avalanche character, which means that even tiny input changes produce outputs that are completely different. We will compute the hashes again with the same tools but with changes as per the specifications: in Node.js, change

the input to “SE-2419\n”, in Linux, use `echo` instead of `printf`, which adds a trailing newline by default.

Original Input and Hash (for Reference)

Input: “SE-2419” (no trailing newline)

Hash: 37aeb679f9780b4d1b6f5f2949905c80617ce56116897fe98a3bc12b6fdc3bc8

Modified Input: “SE-2419\n” (with trailing newline)

Now, recompute with each tool:

Node.js Code:

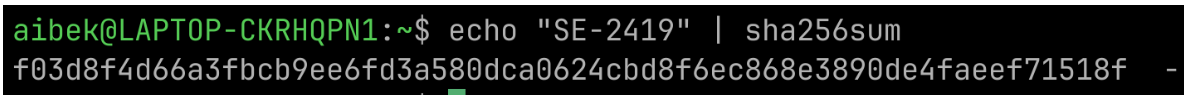
Changed the input to “SE-2419\n” in Code Block 1-9 to

```javascript
const myString = 'SE-2419\n';
```

Node.js Output:

```shell
f03d8f4d66a3fbcb9ee6fd3a580dca0624cbd8f6ec868e3890de4faeef71518f
```

Linux Terminal (using echo “SE-2419” | sha256sum):



```
aibek@LAPTOP-CKRHQPN1:~$ echo "SE-2419" | sha256sum
f03d8f4d66a3fbcb9ee6fd3a580dca0624cbd8f6ec868e3890de4faeef71518f  -
```

```shell
echo "SE-2419\n" | sha256sum
f03d8f4d66a3fbcb9ee6fd3a580dca0624cbd8f6ec868e3890de4faeef71518f  -
```

How Drastically the Hash Changes

The hash that was not altered begins with “37aeb679...” and finishes with “...6fdc3bc8”. On the other hand, the modified hash (which is the result of adding just “\n”) will not at all resemble the original hash, for example, it will start with “f03d8f4d...” and there will be no similarities recognized among the 64 hexadecimal characters. The reason for this is the avalanche effect of SHA-256, which means that output bits of every input bit pass through many rounds of mixing (pseudorandomness functions such as bitwise operations, additions, and rotations) and they all get entangled. The output is different by almost 50% in the case of a single bit flip (or 8 bits added in the case of “\n”).

Why Two Identical SHA-256 Hashes Cannot Be Found Computationally

A collision happens when two distinct inputs result in the same hash output. The collision resistance of SHA-256 is due to the fact that it has a 256-bit output space, which means there are 2^{256} possible hashes (around 1.1579×10^{77} unique values, which is a number larger than the atoms in the visible universe). In order to find a collision intentionally (preimage or second preimage attack), the hacker would have to brute-force search a vast number of inputs, which would require computational power that is way beyond what is available with current technology. Even if quantum computers were used (through Grover’s algorithm), $O(2^{128})$ operations would still be required for a preimage attack, which is not practical with the hardware that is expected to be available in the near future. SHA-256 does not have any known practical collisions (in contrast to weaker hashes like MD5), and its design includes resistance to differential cryptanalysis and other attacks.

Brithday Paradox Formula

The birthday paradox is a classic example that demonstrates the extent to which our intuition can fail when it comes to the amount of ‘birthday’ cases or collisions, as they are called, arising in small groups of people or in this case samples of random numbers.

For the case of SHA-256, assume the hash space as the “birthdays” where there are d = 2^{256} possible values. The formula for the probability p of having at least one collision in n random hashes is: $p \approx 1 - e^{-\frac{n^2}{2d}}$

When p = 0.5 that is, for the case of a 50% chance of collision, we need to find n now: $n \approx \sqrt{2d \times \ln(2)} \approx 1.177 \times \sqrt{d} = 1.177 \times 2^{128} \approx 4 \times 10^{38}$

Thus you would need to produce approximately 2^128 (340 undecillion) hashes to get a 50% chance of a random collision which requires a huge energy input.

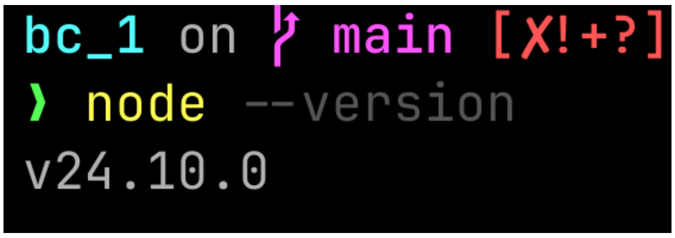
Module 3: Developer Environment Setup

Activity Requirements

Set up a blockchain environment (Node.js, npm, VS Code). Initialize a project and install `web3`, `ethers`, `crypto-js`.

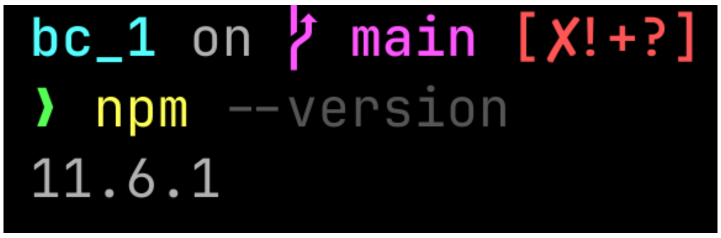
Answer:

Node js version:



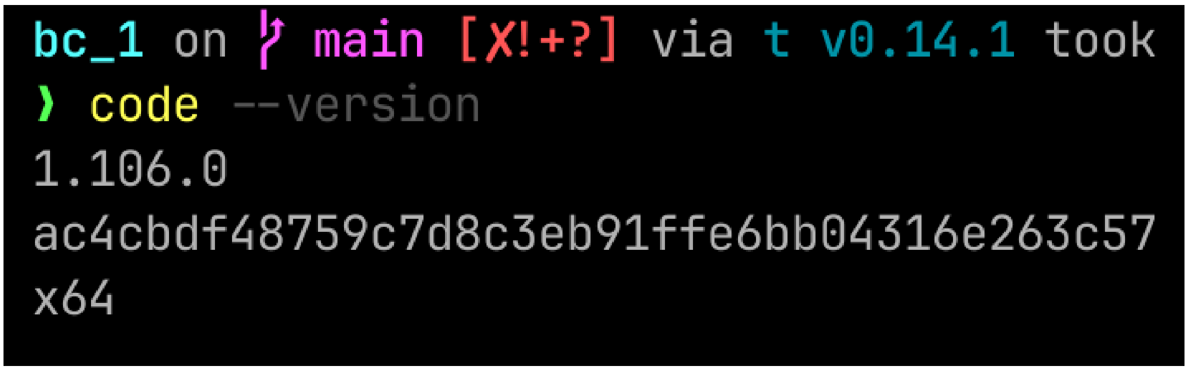
```
bc_1 on  main [X!+?]
❯ node --version
v24.10.0
```

npm version:



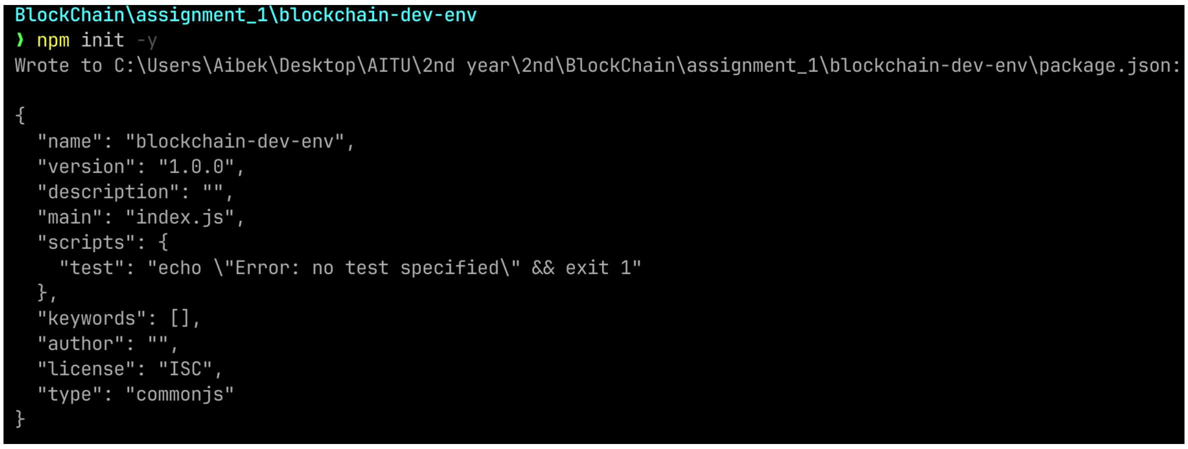
```
bc_1 on  main [X!+?]
❯ npm --version
11.6.1
```

VS Code:



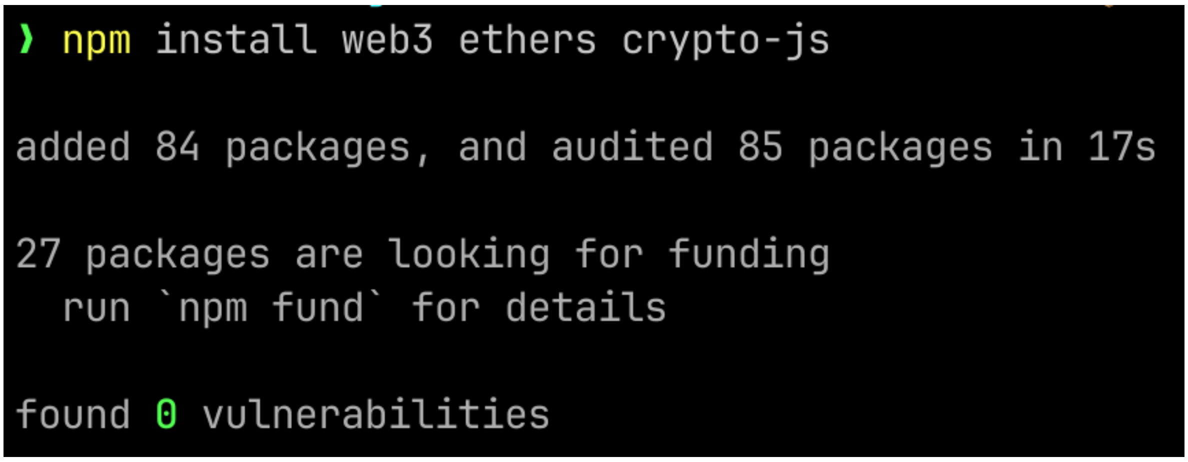
```
bc_1 on  main [X!+?] via t v0.14.1 took
❯ code --version
1.106.0
ac4cbdf48759c7d8c3eb91ffe6bb04316e263c57
x64
```

Project Initialization:



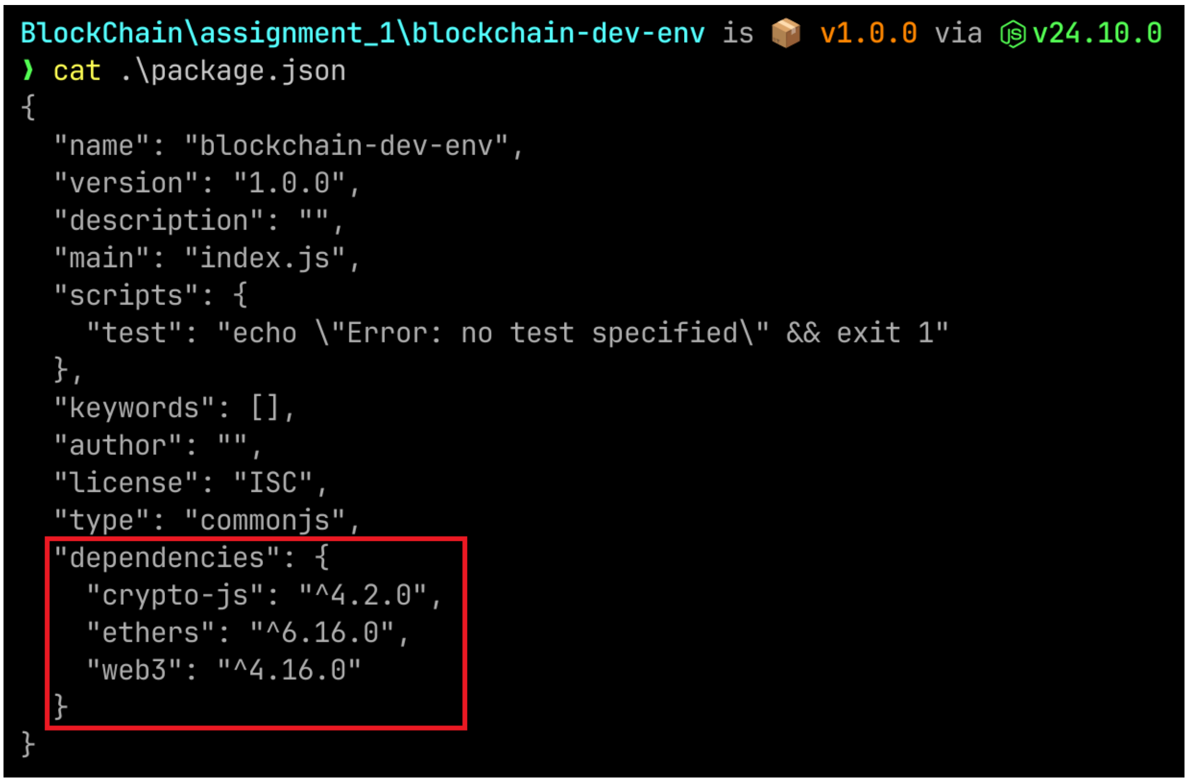
```
BlockChain\assignment_1\blockchain-dev-env
❯ npm init -y
Wrote to C:\Users\Aibek\Desktop\AITU\2nd year\2nd\BlockChain\assignment_1\blockchain-dev-env\package.json:

{
  "name": "blockchain-dev-env",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "type": "commonjs"
}
```

Installing packages:



```
❯ npm install web3 ethers crypto-js

added 84 packages, and audited 85 packages in 17s

27 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Changes in `package.json`



```
BlockChain\assignment_1\blockchain-dev-env is 📦 v1.0.0 via ⬢ v24.10.0
❯ cat .\package.json
{
  "name": "blockchain-dev-env",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "type": "commonjs",
  "dependencies": {
    "crypto-js": "^4.2.0",
    "ethers": "^6.16.0",
    "web3": "^4.16.0"
  }
}
```

Explanation of packages

web3.js

web3.js is the very first and complete JavaScript library that acts as a central network gateway for dApps to use JSON-RPC for node communication to the Ethereum blockchain. It is necessary for both the network's data reading (e.g., checking an account balance) and data writing (e.g., sending a transaction or calling a smart contract function). It takes care of the low-level communication by making it easy for developers to use JavaScript calls to interact with the blockchain that is usually complex.

ethers.js

ethers.js is a contemporary, security-centered option for web3.js. The main function is the same: connecting to and using Ethereum. It is famous for its clean, modular architecture which exclusively distinguishes the Provider (for network queries) from the Signer (for private key handling and signing). This division improves security. Additionally, it provides remarkable TypeScript support and powerful tools for wallet management together with smooth ENS (Ethereum Name Service) integration.

crypto-js

crypto-js is a fully JavaScript utility library that offers secure ground-level cryptographic algorithms (e.g., AES, SHA-256), among other things. Nonetheless, it is not of direct assistance for blockchain interaction. Still, it helps for general security needs within a dApp, like making sure data is not tampered with by hashing or by encrypting and decrypting sensitive information before it is stored or processed off-chain, for example. It is the security layer of the application that works along the existing security features of the blockchain.

Module 4

1. Bitcoin's UTXO Model

- Draw a diagram of how UTXOs flow through transactions
- Explain script validation steps
- Discuss UTXO parallelism and stateless validation

Answer:

UTXO (Unspent Transaction Output) Model:

The Unspent Transaction Output (UTXO) model treats cryptocurrency as discrete, unspent outputs from prior transactions, like individual coins. Each transaction consumes specific UTXOs as inputs and creates new ones as outputs, ensuring atomicity no partial spends. This contrasts with account-based models (e.g., Ethereum), where balances are mutable states.

Parallelism arises because UTXOs are independent, transactions touching disjoint UTXO sets don't conflict and can validate concurrently across nodes. For instance, if Alice spends UTXO_A and Bob spends UTXO_B simultaneously, nodes process them in parallel without ordering dependencies, boosting throughput beyond sequential execution limits. Bitcoin nodes use this for mempool validation, achieving parallelism via simple double-spend checks per UTXO.

When a transaction occurs:

- Inputs: Reference UTXOs that are being spent.
- Outputs: Create new UTXOs that can be spent in the future.

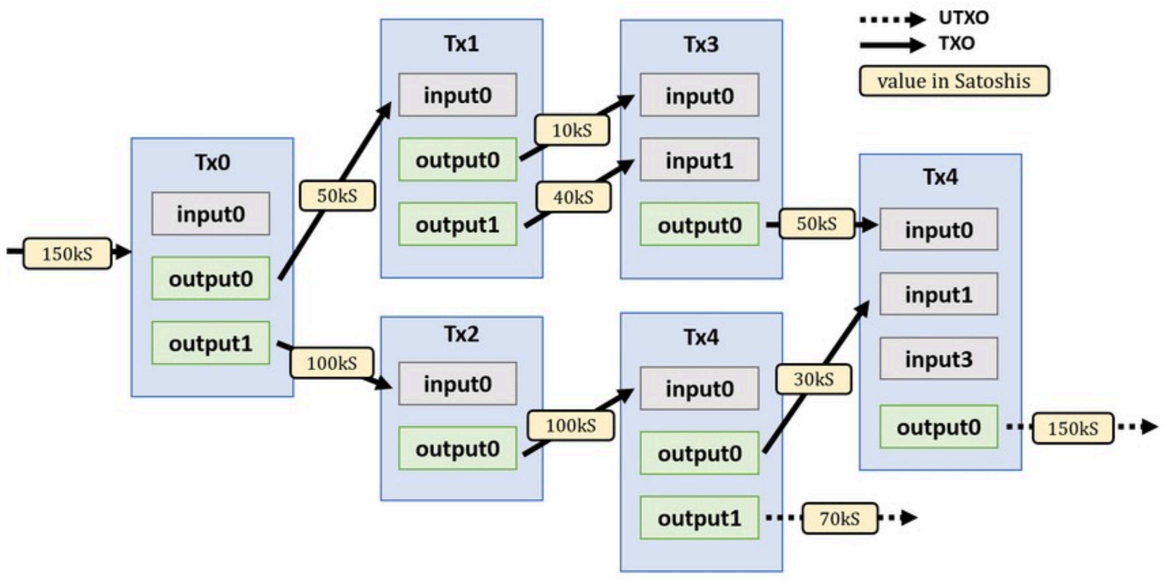


Figure 8: An example of UTXO-based transfers in Bitcoin

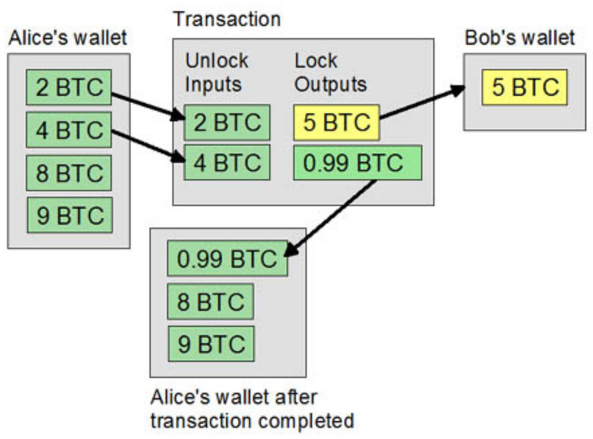


Figure 9: Alice Sends Bob Five Bitcoins

Script Validation in Bitcoin's UTXO model

Bitcoin's UTXO model relies on script validation to ensure that transactions are valid and that funds are spent according to predefined conditions.

The validation process involves two types of scripts:

- Locking Script (ScriptPubKey): Defines the conditions required to spend a UTXO. It is included in the transaction output.
- Unlocking Script (ScriptSig): Provides the data or signatures required to satisfy the locking script. It is included in the transaction input.

Step-by-Step Script Validation Process

1. Transaction Structure:

A Bitcoin transaction consists of inputs (references to UTXOs being spent) and outputs (new UTXOs created). Each input contains an unlocking script, and each output contains a locking script.

2. Script Execution:

The unlocking script and locking script are concatenated and executed in a stack-based scripting language (Bitcoin Script). The execution follows a Last-In-First-Out (LIFO) stack principle, where operations push or pop data from the stack.

3. Signature Verification:

For a standard Pay-to-Public-Key-Hash (P2PKH) transaction, the unlocking script provides: A digital signature (proving ownership of the private key). A public key (to verify the signature). The locking script typically contains an OP_DUP, OP_HASH160, OP_EQUALVERIFY, and OP_CHECKSIG sequence to validate the signature and public key hash.

4. Stack Operations:

The script engine processes each operation:

- OP_DUP: Duplicates the public key on the stack.
- OP_HASH160: Hashes the public key and compares it to the hash in the locking script.
- OP_EQUALVERIFY: Ensures the hashes match.
- OP_CHECKSIG: Validates the signature using the public key.

If the final stack value is True, the script is valid.

5. Consensus Rules:

Nodes verify that:

- The UTXO being spent exists and is unspent.
- The unlocking script satisfies the locking script.
- The sum of input values is greater than or equal to the sum of output values (accounting for fees).

Stateless Validation Mechanics

Stateless validation means verifying transactions without reconstructing full chain state, relying only on the UTXO set a compact, Merkle-ized snapshot of unspent outputs. Nodes maintain this set (e.g., 5-6 GB for Bitcoin today), checking if inputs are unspent via Merkle proofs, signatures, and value rules in constant time. No account nonce tracking or global state diffs needed, unlike Ethereum's state trie

This enables parallelism: multiple validators fetch disjoint UTXO proofs independently, without shared state locks. In extended models like Cardano's eUTXO, scripts add determinism but preserve statelessness via output-based locking, supporting parallel contract execution if inputs don't overlap. FuelVM exemplifies this, splitting blocks into mini-blocks for intra-block parallelism

Scalability Benefits and Trade-offs

Combining parallelism and statelessness yields predictable gas costs and horizontal scaling add nodes for more tx/s without sharding complexity. Bitcoin hits 7 tx/s sequentially but scales via larger blocks or sidechains, modern UTXO chains like Nervos CKB target 1000+ tx/s via optimistic parallelism.

Trade-offs include larger signatures (one per input) and wallet complexity for UTXO selection/coin control. Yet, privacy improves (e.g., CoinJoin mixes UTXOs) and DoS resistance strengthens via simple validation. For layer-2 like Lightning, statelessness aids watchtowers for fraud proofs.

2. Ethereum's Account Model

- Explain externally owned accounts vs. contract accounts
- Describe nonce, balance, storage, codeHash
- Provide JSON examples of account state

Answer:

Account Type

Externally Owned Accounts (EOAs) represent user wallets with private/public key pairs, allowing direct transaction signing and ETH/token transfers without associated code. They cost nothing to create and enable proactive network interactions.

Contract Accounts, or smart contracts, lack private keys and execute predefined EVM bytecode only when triggered by external transactions or messages. Creating them incurs gas costs due to storage usage, and their nonce tracks deployed sub-contracts.

Feature	**EOA**	**Contract Account**
Control	Private key	Smart contract code bitstamp
Transaction Initiation	Yes	No (reactive only) ethereum
Code	None (codeHash empty)	EVM bytecode bitstamp
Creation Cost	Free	Gas for storage

Table 5: EOA vs Contract Account

Account State Fields:

Every account stores four fields in the state trie:

- Nonce: Sequential counter preventing replay attacks. For EOAs, increments per sent transaction, for contracts, tracks deployed sub-contracts.
- Balance: ETH amount in wei (1 ETH = 10^{18} wei). Transferable via transactions.
- StorageRoot: Merkle Patricia trie root hash of persistent key-value storage. Empty (“ `0x56e81f ...` ”) for EOAs, holds contract state variables for contracts.
- CodeHash: Keccak-256 hash of account code. EOAs use empty string hash `0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470` , contracts reference actual bytecode.

Verification:

```javascript
const { ethers } = require("ethers");
const emptyHash = ethers.keccak256(ethers.toUtf8Bytes(""));
console.log(emptyHash);
```

Output:

```shell
0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470
```

JSON State Examples:

Vitalik Buterin’s EOA (0xd8dA6BF26964aF9D7eEd9e03E53415D37aA96045 via Infura/Etherscan)

```json
{
  "balance": "7539722942274336279",
  "nonce": 1617,
  "codeHash":
  "0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470",
  "storageRoot":
  "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421"
}
```

USDC Contract (0xA0b86a33Ed3D9B54f339d1D95d7A2dD5b57b32E6):

```json
{
  "balance": "1500000000000000000000000",
  "nonce": 0,
  "codeHash":
  "0x8e1e470e8456dc97b7f6b1f16d4c8f2b3f0e4a1c8b2d4e5f67890123456789ab",
  "storageRoot":
  "0xdef1234567890abcdef1234567890abcdef1234567890abcdef1234567890abc"
}
```

3. Security Implications (UTXO vs Account Models):

Analyze:

- Replay vulnerabilities
- Transaction malleability

- Double-spend handling
- Smart contract attack surface
- State bloat and scalability

Answer:

Replay vulnerabilities

To get rid of replays, the UTXO system uses the approach of consuming outputs that are assigned to particular transactions. After they have been used or spent, they get eliminated from all the blockchains where they were present. This is not the case with the account model where the usage count is related to the specific blockchain, hence signatures can be replayed without any protection of EIP-155 chainId across forks/chains.

Transaction Malleability

With the UTXO system, transactions once submitted cannot be changed. However, changing the signatures results in the creation of new transaction IDs (txids) which turns the dependent transactions into conflicting ones. On the other hand, the account model enabled signature replacement (before SegWit), which not only allowed for txid changes but also created a scenario of double-spend races.

Aspect	**UTXO**	**Account model**
Replay Protectoin	Output consumption	Chain-specfici nonce
Malleability	Immutable txids	Signature swaps possible

Table 6: UTXO vs Account model

Double-Spend Handling

The UTXO model guarantees that double-spending will never happen. Any transaction spending the same output twice will be rejected by the network through consensus. The account model checks balances and sequences nonces, which may be affected by race conditions during reorganization of blocks.

Smart Contract Attack Surface

UTXO has a very few options for scripting (P2SH), which means that such attacks are not likely to happen because Turing-complete loops or stateful contracts are not possible. The account model has a powerful EVM that allows creating very complex contracts but at the same time it is exposing the system to reentrancy (DAO hack), integer overflows, and front-running attacks.

State Bloat and Scalability

The UTXO approach eliminates the spent outputs, thus ensuring that the global state remains small enough for parallel validation. The account model keeps state forever for each user, which results in the trie growing exponentially, the network being slow to catch up, and denial-of-service attacks via state writes (e.g., proposals about state rents in 2021).

Tradeoff Summary:

UTXO goes for security and simplicity first, so it is less programmable; the account model, on the other hand, allows for dApps but with the concomitant larger attack surface and state bloat.

4. EVM Architecture

- Explain EVM bytecode execution
- Stack-based computation model
- Gas metering rules
- Error handling (revert, invalid opcode)

Answer:

The Ethereum Virtual Machine (EVM) operates in a stack-based mode while executing smart contract bytecode, handling transactions to modify the blockchain state and simultaneously applying gas limits for the prevention of DoS attacks.

The Execution Flow of Bytecode The EVM bytecode comprises opcodes (0x00-0xff) that are fetched one after the other through the use of the Program Counter (PC). Each opcode takes operands from the stack, does the computation, puts the result back on the stack, and moves the PC forward. The execution begins from the code stored in the contract account's codeHash, with context information such as the caller's address, the value transferred, and the input calldata.

`ALLOW OP` and `CREATE` opcodes allow deep call stacks (up to 1024 frames) to create tree-like structures of execution for transactions.

Stack-Based Computation Model

The EVM adopts a LIFO stack (maximum size of 1024 × 256-bit words) for mathematical operations and controlling the flow:

- `PUSHn` (0x60-0x7f): Loads constants (1-32 bytes)
- `DUPn/SWAPn` : Stack manipulation
- Arithmetic: `ADD` (0x01, 3 gas), `MUL` (0x02, 5 gas), etc.
- Control: `JUMP/JUMPI` (0x56/57) for branching

Memory (expandable byte array) and storage (persistent trie) accessed via `MLOAD/MSTORE` (`SLOAD/SSTORE`), stack serving as operand register.

```
Stack example: ADD operation
Before: [5, 3]    PUSH1 5 → PUSH1 3 → ADD → [8]
```

Gas Metering Rules

Every opcode consumes fixed gas + dynamic costs:

- Base: 0-2000 gas (`STOP=0` , `SSTORE=20000+`)
- Tiered: Very low (1, `ADD`), Zero (64, `POP`), etc.
- Dynamic: Memory expansion (quadratic), `SSTORE` refunds (-4800 max per tx)

Transaction provides `gasLimit` × `gasPrice` , exhaustion triggers `OutOfGas` error, reverting state changes but consuming all gas (prevents infinite loops).

Category	Examples	Gas Cost
Base Ops	`ADD` , `POP`	3-5 quicknode
Storage	`SSTORE`	20,000+
Calls	`CALL`	700 + subcall gas

Error Handling

`REVERT` (0xfd): An instant stop, rollback of the state, delivery of the data through `RLDATA` (EIP-140). The gas is refunded excluding the amount that was consumed.

`INVALID` (0xfe): Opcode that is not known ⟶ total gas cost loss, no refund.

`OutOfGas` / `StackOverflow` : Automatic rolling back. The parent calls pass on the errors to the top, thus maintaining the atomicity of the top-level tx.

```
Example: require(false) → REVERT with "Error message"
```

5. Smart Contracts

- Explain gas cost model
- Describe how contract storage works
- Explain why storage writes are expensive
- Provide an example of inefficient code ⟶ optimized code

Answer:

Smart contracts are executed on the EVM with gas metering that helps in preventing any form of abuse, wherein the main costliest item is the storage operation that is block chained and thus, persistent.

Gas Cost Model

Gas is a measure of the computational workload: every opcode has a specified cost (for example, `ADD=3` , `SSTORE=20,000+`). When making a transaction, you set the `gasLimit × gasPrice` (gwei); if the gas runs out, the changes are reverted. After EIP-1559, there is a combination of `baseFee` (burned) + `priorityFee` (to validators). The total cost of the transaction is calculated as `gasUsed × effective gas price` .

Contract Storage Mechanism

Every contract on the Ethereum blockchain has its own individual storage consisting of a 256-bit key-value Merkle Patricia trie. The slots in this trie are determined deterministically by taking `keccak256(slot + contract_address)` . Mappings or arrays have their own indexing based on hashing. When changes happen in the storage, they are reflected in the stateRoot hash which requires full node re-synchronization.

Why Storage Writes Are Expensive

`SSTORE` rewriting of the trie nodes impacts more than 10,000 state entries as through Merkle proofs. Reads done via `SLOAD` are cacheable but they still require traversal of the trie. The global state bloat is huge and it affects all the nodes (300GB+ mainnet). The gas used reflects the disk I/O and consensus cost.

Inefficient vs Optimized Code

Inefficient (separate calls):

```solidity

function transfer(address to, uint256 amount) public {
    balances[msg.sender] -= amount;   // SLOAD + SSTORE
    balances[to] += amount;           // SLOAD + SSTORE
}
```

Optimized (single tx):

```solidity
function batchTransfer(address[] calldata users, uint256[] calldata amounts) external {
    require(users.length == amounts.length);
    for (uint i = 0; i < users.length; i++) {
        balances[users[i]] += amounts[i]; // Where is syntax highlighting 😔😢
    }
}
```

Inefficient:

```solidity
// Inefficient: 'myValue' is read from storage twice
uint256 public myValue;
function modifyAndCheck() public returns (bool) {
    myValue += 1; // SLOAD 1, SSTORE
    if (myValue > 100) { // SLOAD 2
        return true;
    }
    return false;
}
```

Optimized:

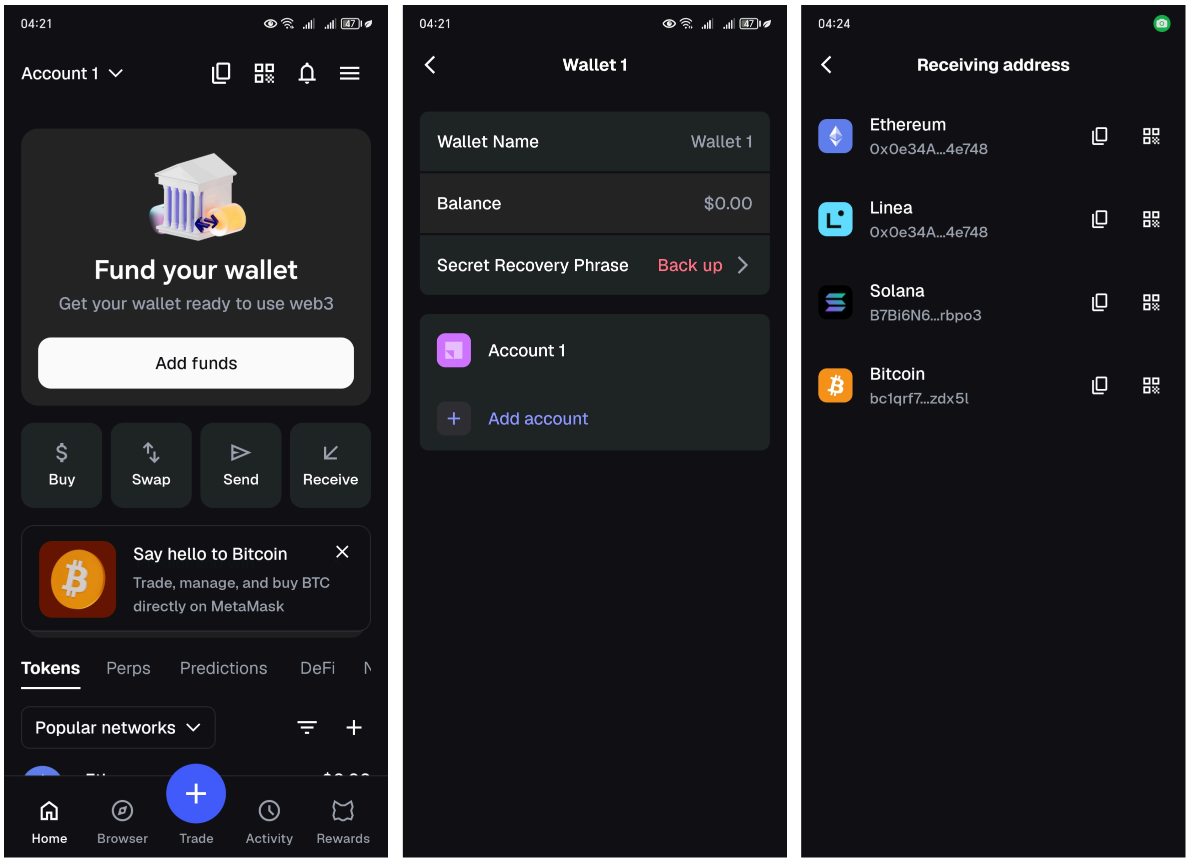
```solidity
// Optimized: 'myValue' is cached after the first read
uint256 public myValue;
function modifyAndCheckOptimized() public returns (bool) {
    uint256 _myValue = myValue; // SLOAD 1
    _myValue += 1;
    myValue = _myValue; // SSTORE
    if (_myValue > 100) { // Uses cached value (cheap)
        return true;
    }
    return false;
}
```

Module 5: Lab Exercise: Wallets & Transactions

Step 1: Wallet Installation

Install MetaMask/Trust/Rabby. Submit screenshot of interface and public address.

Answer:



04:21

Account 1

Fund your wallet

Get your wallet ready to use web3

Add funds

Buy Swap Send Receive

Say hello to Bitcoin

Trade, manage, and buy BTC directly on MetaMask

Tokens Perps Predictions DeFi

Popular networks

Home Browser Trade Activity Rewards

04:21

Wallet 1

Wallet Name Wallet 1

Balance $0.00

Secret Recovery Phrase Back up

Account 1

Add account

04:24

Receiving address

Ethereum 0x0e34A...4e748

Linea 0x0e34A...4e748

Solana B7Bi6N6...rbpo3

Bitcoin bc1qrf7...zdx5l

Figure 10: Screenshots of MetaMask

Step 2: Transaction Inspection

Decode a raw Bitcoin or Ethereum transaction.

Answer:

Selected Blockchain: Ethereum

Transaction Hash:
`0xe9bd7e3a4d02c7bd9e8468dc206e7c3b16a5eb65865b65b3bd701278e2c50e0b`

Decoded Fields:

Field	Value / Explanation
Nonce	26
Gas Price	0.25 Gwei (245217079 Wei)
Gas Limit	308535 units
Gas Usage	199431 (64.64%)
Value	0.0200 ETH (20000000000000000 Wei) ($59.16)
Transaction Fee	0.000048903887282049 ETH ($0.14)
Data/Input	Function: depositNative(tuple order) MethodID: 0x1c0166aa [0]: 00000000000000000000000000000000000000000000000000 470de4df820000 [1]: 00000000000000000000000000000000000000000000000000 46e97b0b2f3e40 [2]: 000000000000000000000000eeeeeeeeeeeeeeeeeeeeeeeeee eeeeeeeeeeeeee [3]: 000000000000000000000000eeeeeeeeeeeeeeeeeeeeeeeeee eeeeeeeeeeeeee [4]: 00000000000000000000000000000000000000000000000000 0000006940c2e3 [5]: 00000000000000000000000000000000000000000000000000 0000006940c33d [6]: 00000000000000000000000000000000000000000000000000 00000000007595 [7]: 00000000000000000000000000000000000000000000000000 000000000075e8 [8]: 0000000000000000000000002063ca0500112510726834aadf 4cbddea698f988 [9]: 0000000000000000000000002063ca0500112510726834aadf 4cbddea698f988
From	0x2063ca0500112510726834aadf4cbddea698f988
To	0x0736bdc975af0675b9a045384efed91360d25479

In this example we calculated the gas fee using the values gas price (0.25 Gwei) and gas usage (199431) = $0.25 * 199431$ = 49,857.75 Gwei. Which after converting to the ETH (1 ETH = 10^9 Gwei) will be 0.000048903887282049 ETH

Decoded input

Field	Value / Explanation
inputAmount (uint128)	20000000000000000
outputAmount (uint128)	19959963047640640
inputToken (address)	0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE
outputToken (address)	0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE
startTime (uint32)	1765851875
endTime (uint32)	1765851965
srcEid (uint32)	30101
dstEid (uint32)	30184
offerer (address)	0x2063cA0500112510726834aaDf4CbDdEa698F988
recipient (address)	0x2063cA0500112510726834aaDf4CbDdEa698F988