1 Week 1

1.1 Why Study Blockchains

High level idea: Blockchains are a new paradigm for distributed secure systems.

They combine:

- Cryptography (hashes, signatures, PoW).
- Distributed systems (consensus, P2P networks).
- Economics and game theory (incentives, equilibria).

Why they matter:

- Help understand modern security mechanisms:
  - Key management and PKI.
  - Software integrity and update mechanisms.
  - Privacy enhancing technologies.
- Enable new organizational forms:
  - Cryptocurrency and DeFi.
  - DAOs, on chain governance.
  - Token based coordination and funding.
- Bitcoin is a concrete proof that a large scale open system can run for years without a central operator.

1.2 Blockchains and Distributed Ledgers

1.2.1 Concepts

Blockchain: a distributed, append only data structure that maintains a consistent log of transactions across many nodes.

Distributed ledger: a general term for systems that maintain shared state among multiple parties without a single trusted authority.

Desired properties:

- **Safety:** all honest nodes agree on the same history (no conflicting logs).
- **Liveness:** valid transactions are eventually included and confirmed.

Bitcoin is the first widely deployed blockchain protocol that achieves these properties in a permissionless setting.

1.3 Endless Ledger Parable

1.3.1 Book and Scribes

Intuition: model Bitcoin as an endlessly growing ledger maintained by many *scribes*.

- There is a shared ledger (a book) with many numbered pages.
- Anyone can become a scribe and propose a new page.
- Each page records a batch of transactions.
- The ledger never stops growing: new pages are added over time.

Constraint: adding a page requires expensive work.

- To write page i, the scribe must solve a hard puzzle, like throwing many dice until a rare pattern appears.
- This represents Proof of Work (PoW).

1.3.2 Forks and Longest Chain Rule

Multiple copies of the ledger may exist:

- Different scribes work in parallel and may produce conflicting next pages.
- Question: which ledger is the "correct" one?

Rule: everyone follows the ledger with the largest number of valid pages.

- If several ledgers have the same maximum length, pick the first one you received and keep writing on top of it.
- Pages not on the longest ledger become *orphan* pages.

This is the **longest chain rule**: choose the chain with the greatest cumulative work.

1.3.3 Randomness and Symmetry Breaking

Each page is produced by a random process (dice throwing, PoW search):

- With many scribes, the chances that two of them keep finding pages in perfect lockstep are tiny.
- Eventually one scribe gets ahead, creating a longer ledger.
- Other scribes then switch to this longer ledger.

Randomness breaks symmetry and lets the system converge on a single chain.

1.3.4 Incentives for Scribes

To motivate scribes to do the costly work:

- The rules allow the scribe who creates a valid new page to insert a special record awarding them a reward.
- In Bitcoin this is the block reward (newly minted coins) plus transaction fees.

Key points:

- Anyone with computing resources can become a miner.
- More computing power implies higher probability of winning the next block.

1.4 Scalable Service Provision Problem

General IT question:

How can we scale an online service to the whole world when participants do not trust each other and there is no central authority?

Traditional answers:

- **Federation:** multiple providers cooperate (e.g. email, XMPP).
- **Centralization:** one dominant provider (e.g. large social networks, cloud services).

Blockchains show a third option: decentralized provision by *resource owners* instead of fixed organizations.

1.4.1 Software Only Launch (SOL)

Goal: deploy a system purely by publishing software.

- Release an open source program.
- Announce a start time.
- Anyone can download the program and run it.
- When enough nodes run the software, the system "self boots" and becomes operational.

Bitcoin is a successful example of such a software only launch.

1.5 Hash Functions

1.5.1 Definition and Basic Properties

A hash function H maps inputs of arbitrary length to fixed length outputs.

Requirements:

- Efficient to compute.
- Output looks random and is well spread over the output space.

Cryptographic security properties:

- **Pre image resistance:** given y, it is hard to find any x with $H(x) = y$.
- **Second pre image resistance:** given x, it is hard to find $x' \neq x$ with $H(x') = H(x)$.
- **Collision resistance:** it is hard to find any pair $x \neq x'$ such that $H(x) = H(x')$.

1.5.2 Birthday Paradox

If there are n possible hash outputs, collisions appear surprisingly early.

- Approximate number of random samples needed for a collision with probability $\approx 50\%$: $k \approx 1.177\sqrt{n}$.
- For hash outputs of t bits, $n = 2^t$, so attacks based on collisions cost about $2^{t/2}$ operations.

1.5.3 Examples

- Broken: MD5, SHA 1 (known collisions).
- Current families: SHA 2 and SHA 3 with 224, 256, 384, 512 bit outputs.
- Bitcoin uses SHA 256 (from SHA 2 family).

1.6 Digital Signatures

1.6.1 API

A signature scheme $\Sigma = (\text{KeyGen}, \text{Sign}, \text{Verify})$:

- **KeyGen**:
  $$(sk, vk) \leftarrow \text{KeyGen}(1^\lambda)$$
  where sk is the secret signing key, vk is the public verification key.
- **Sign**:
  $$\sigma \leftarrow \text{Sign}(sk, m)$$
  where m is the message.
- **Verify**:
  $$b \leftarrow \text{Verify}(vk, m, \sigma) \in \{0, 1\}$$
  output 1 (accept) or 0 (reject).

1.6.2 Security Intuition

Existential unforgeability under chosen message attack (EU CMA):

- Attacker can obtain signatures on messages of their choice from a signing oracle.
- Even with this advantage, attacker should not be able to produce a valid signature on a new message that was never signed before.

1.6.3 Constructions

- Based on RSA (integer factorization hard).
- Based on discrete logarithms (DSA).
- Elliptic curve variants (ECDSA, Schnorr).

Bitcoin uses ECDSA originally and later also supports Schnorr style signatures.

1.7 Proof of Work (PoW)

1.7.1 Definition

A PoW scheme allows a prover to demonstrate that a certain amount of computational work has been done.

Typical hash based PoW:

$$\text{Find } w \text{ such that } H(\text{data}\|w) \le T$$

where T is a difficulty target.

1.7.2 Simple Algorithm

```
ctr = 0
while H(data || ctr) > T:
    ctr = ctr + 1
return ctr
```

Properties:

- **Fast verification:** given w, one hash evaluation checks whether the condition holds.
- **No shortcuts:** for a well designed hash function, there is no significantly faster way than brute forcing different w.

1.7.3 Variants

- Standard Hashcash style PoW (used in Bitcoin).
- Memory hard PoW (e.g. scrypt, Equihash) to force large RAM usage.
- ASIC resistant PoW to reduce the advantage of specialized hardware.

1.8 Resource Based Systems

1.8.1 Resource Types

In resource based systems participation is tied to control of some scarce resource:

- **Proof of Work (PoW):** computational power.
- **Proof of Stake (PoS):** ownership of currency or tokens.
- **Proof of Space / Capacity:** available storage.
- **Proof of Time / Identity:** trusted hardware or other timing assumptions.

The system is not pinned to a fixed set of identities. Instead, any entity that can show a valid proof of resource can participate in maintaining the ledger.

1.8.2 PoW vs PoS

PoW:

- Pros: simple design, well studied, direct link between cost and security.
- Cons: high energy usage, hardware centralization (ASIC farms), environmental concerns.

PoS:

- Pros: lower energy usage, security based on economic value at stake.
- Cons: subtle security issues (nothing at stake, long range attacks), more complex protocol design.

1.9 Tokenomics

1.9.1 Basic Idea

Tokenomics studies how to use tokens and rewards to align incentives of participants.

Typical cycle:

- Users pay fees to use the service (transactions, smart contracts).
- The protocol distributes rewards to resource providers (miners, validators).
- Providers sell some of their tokens to cover costs and profit.

Goal: set parameters so that:

- Providing honest service is economically attractive.
- Attacking or misbehaving is economically disfavored.

1.10 Decentralized Service Provision

To run a decentralized service in an open network, the protocol must handle:

- **DoS resistance:** prevent abuse by spamming transactions or connections.
- **Consistency:** all honest participants eventually agree on the same state.
- **Liveness and censorship resistance:** valid transactions should not be permanently excluded.
- **Fairness of rewards:** contributions of resource providers should be measured and rewarded in a predictable way.

1.10.1 Reward Sharing

Rewards can be distributed:

- Per block or per action (e.g. each mined block gets a fixed reward).
- Per epoch (e.g. aggregate rewards over a time window then share according to contribution).

Design challenge:

- Ensure that rational, self interested participants collectively form a robust and secure system.
- Avoid centralization and cartel behavior where possible.

1.11 Quick Summary

- Blockchains provide a new way to build global services without central operators.
- The "endless ledger" parable explains how longest chain and randomness yield consensus.
- Hash functions and digital signatures are the basic cryptographic tools for integrity and authenticity.
- Proof of Work ties block production to computational effort and is easy to verify.
- Resource based systems use PoW, PoS, or other proofs to select and reward maintainers.
- Tokenomics and reward sharing mechanisms align incentives so that honest behavior is profitable.

2 Week2

2.1 Authenticated File Storage

Goal: Store a file on an untrusted server but keep only a short local state so that later you can check whether the server returned the correct data.

Client has identifier F and data D. It sends (F, D) to the server, and wants to delete D locally while still being able to verify any future response from the server.

2.1.1 Naive solution (does not help)

Client keeps a full local copy of D and checks equality with any D' returned by the server. This gives integrity but saves no storage.

2.2 Basic Cryptographic Tools

2.2.1 Hash-based authentication

Hash function H is collision resistant.

- Upload: client sends (F, D) to server.
- Commit: client stores only $h = H(D)$, deletes D.
- Retrieval: server returns D'.
- Verify: client accepts if $H(D') = h$, rejects otherwise.

Properties:

- Client keeps a short fixed-size value.
- Integrity relies on collision resistance of H.

2.2.2 Digital signatures

Signature scheme $\Sigma = (\text{KeyGen}, \text{Sign}, \text{Verify})$.

- Key generation: client runs KeyGen to get (sk, vk).
- Upload: compute $\sigma = \text{Sign}(sk, \langle F, D\rangle)$, send (F, D, σ) to server.
- Client keeps only vk.
- Retrieval: server returns (D', σ').
- Verify: accept if $\text{Verify}(vk, \langle F, D'\rangle, \sigma') = 1$.

Difference from pure hashing:

- Signatures are publicly verifiable (third parties can check).
- Useful for transferable proofs of origin and integrity.

2.3 Merkle Trees

2.3.1 Structure

Goal: authenticate large data split into blocks, and allow efficient verification of individual blocks.

- Split data D into blocks $D_1, \dots, D_n$.
- Compute leaf hashes $H_1 = H(D_1), \dots, H_n = H(D_n)$.
- Build a binary tree:

  $$H_{i,j} = H(H_i \| H_j)$$

  up to a single *Merkle root* MTR.

Client stores only MTR as the commitment to the entire file.

2.3.2 Merkle-based storage protocol

- Upload: client sends full D to server, builds Merkle tree locally and computes root MTR, then keeps only MTR.
- Retrieval of a block D_x: server returns D_x and a *proof of inclusion* π.
- Verification: client uses D_x, π, and H to recompute a root and checks that it equals stored MTR.

2.3.3 Proof of inclusion

For a block D_x:

- Proof consists of all sibling hashes along the path from the leaf $H(D_x)$ to the root.
- Verifier:
  1. Starts from $H(D_x)$.
  2. Iteratively combines with sibling hashes and hashes upward.
  3. Checks whether the final value equals MTR.

Tree height is $O(\log n)$ for n leaves, so proof size and verification time are $O(\log n)$.

2.3.4 Applications

- BitTorrent: verify file chunks during download.
- Bitcoin: Merkle tree of transactions inside each block.
- Ethereum: variants of Merkle trees for state and transactions.

2.4 Merkle Trees for Sets

Goal: store a set S on a server and later prove membership or non-membership of any element x.

Construction:

- Sort the elements of S.
- Build a Merkle tree where leaves are sorted elements.

2.4.1 Membership proof

If $x \in S$, the server provides a normal proof of inclusion for the leaf corresponding to x.

2.4.2 Non-membership proof

If $x \notin S$:

- Find neighbors $H_<$ and $H_>$ in the sorted order such that $H_< < x < H_>$.
- Provide inclusion proofs for $H_<$ and $H_>$.
- Show they are adjacent in the sorted set representation.
- Conclude that x is not present.

2.5 Tries and Patricia Tries

2.5.1 Trie (prefix tree)

Data structure for a set of key-value pairs $\{(key, value)\}$ where keys are strings.

- Each edge is labeled with a character.
- A path from the root spells out a key.
- Nodes may store values for keys ending at that node.

Operations:

- **add(key,value)**: follow or create edges for each character, then store the value at the final node.
- **query(key)**: follow edges by characters, and check whether the final node has a value.

2.5.2 Patricia Trie

Compressed version of a Trie:

- Any chain of nodes where each node has a single child and no value can be merged into a single edge labeled with a substring.
- Saves space and reduces tree height.

Patricia tries are widely used in blockchain systems for efficient key-value storage.

2.6 Merkle Patricia Trie (MPT)

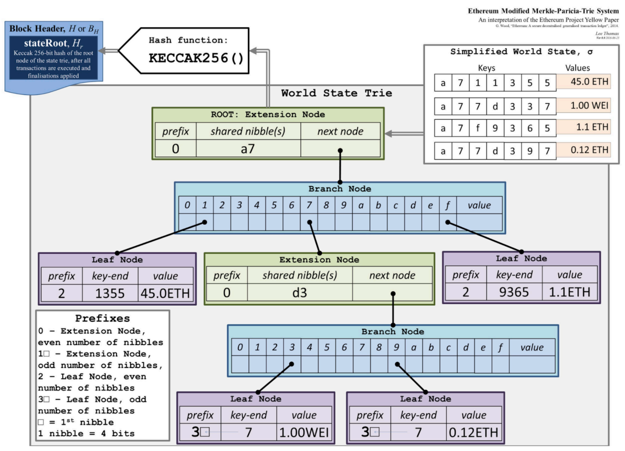
Ethereum combines Merkle hashing with Patricia tries.

2.6.1 Node types

Keys are encoded in hexadecimal nibbles. There are three logical node types:

- **Leaf node**: stores remaining key fragment and the value.
- **Extension node**: stores a shared key prefix and a pointer to another node.
- **Branch node**: has up to 16 child pointers (for hex digits) plus an optional value.

Each logical node is serialized and hashed. Child pointers store the hash of the child node, so the root hash commits to the entire key-value map.



Ethereum Modified Merkle-Paricia-Trie System
An interpretation of the Ethereum Project Yellow Paper
G. Wood, "Ethereum: A secure decentralised generalised transaction ledger", 2014.
Lee Thomas
Block Header, H or B_H
stateRoot, H_r
Keccak 256-bit hash of the root node of the state trie, after all transactions are executed and finalisations applied
Hash function:
KECCAK256()
Simplified World State, σ
Keys
Values
a 7 1 1 3 5 5 45.0 ETH
a 7 7 d 3 3 7 1.00 WEI
a 7 f 9 3 6 5 1.1 ETH
a 7 7 d 3 9 7 0.12 ETH
World State Trie
ROOT: Extension Node
prefix | shared nibble(s) | next node
0 | a7
Branch Node
0 1 2 3 4 5 6 7 8 9 a b c d e f value
Leaf Node
prefix | key-end | value
2 | 1355 | 45.0ETH
Extension Node
prefix | shared nibble(s) | next node
0 | d3
Leaf Node
prefix | key-end | value
2 | 9365 | 1.1ETH
Prefixes
0 – Extension Node, even number of nibbles
1□ – Extension Node, odd number of nibbles,
2 – Leaf Node, even number of nibbles
3□ – Leaf Node, odd number of nibbles
□ = 1st nibble
1 nibble = 4 bits
Branch Node
0 1 2 3 4 5 6 7 8 9 a b c d e f value
Leaf Node
prefix | key-end | value
3□ | 7 | 1.00WEI
Leaf Node
prefix | key-end | value
3□ | 7 | 0.12ETH

2.6.2 Properties

- Root hash acts as a commitment to the whole dictionary.
- Inclusion proofs: show the path from the root down to a leaf and the content of intermediate nodes.
- Non-inclusion proofs: show that the search path terminates at a node that proves no matching key exists.
- Ethereum uses an MPT for the global world state, and the state root hash is stored in the block header.

2.7 Blockchain Data Structures

2.7.1 Block structure

A block consists of:

- Random nonce ctr.
- Data x (e.g. transactions, root hashes).
- Pointer s to the previous block (usually a hash of the previous header).

The header typically includes (ctr, x, s). The pointer field s creates a hash chain of blocks back to the genesis block.

2.7.2 Proof of Work (PoW)

In PoW systems a block header must satisfy:

$$H(\mathrm{ctr} \| x \| s) \le T$$

where T is a global difficulty target.

- Miners fix x and s, iterate over ctr, and search for a header whose hash is below T.
- The same hash is also used as the block identifier.

2.8 Bitcoin Overview

2.8.1 High-level protocol

1. New transactions are broadcast to the network.
2. Nodes collect transactions into candidate blocks.
3. Nodes perform PoW to find valid blocks.
4. A node that finds a valid block broadcasts it.
5. Other nodes validate all transactions and the PoW before accepting.
6. Nodes start mining on top of the longest valid chain.

2.8.2 UTXO model

Bitcoin represents ownership via unspent transaction outputs (UTXOs).

- A transaction has multiple inputs and outputs.
- Each **input** references a previous output (by transaction hash and index) and provides a script that authorizes spending.
- Each **output** specifies a value and a script defining how it may be spent in the future.

2.8.3 Scripts

Typical pay-to-public-key-hash (P2PKH) transaction:

- Output script (`scriptPubKey`):

$$\text{OP\_DUP OP\_HASH160 } \langle \text{pubKeyHash} \rangle \\ \text{OP\_EQUALVERIFY OP\_CHECKSIG}$$

- Input script (`scriptSig`) when spent:

$$\langle sig \rangle \ \langle pubKey \rangle$$

During validation, the combined script is executed to check that the spender owns the corresponding private key.

2.8.4 Merkle tree of transactions in a block

- All transactions in a block are organized as a Merkle tree.
- The Merkle root is stored in the block header.
- Simplified payment verification (SPV) clients download only block headers and proofs of inclusion for specific transactions.

2.9 Bitcoin Network

2.9.1 P2P topology

- All nodes run the same open-source protocol.
- Each node maintains connections to a set of peers.
- The network is permissionless: nodes may join or leave at any time.

Bootstrapping:

- Peer-to-peer nodes come "pre-installed" with some peers by IP / host.
- A user can also manually configure known peers.

2.9.2 Gossip protocol

- When a node learns about a new transaction or block, it forwards it to its peers.
- Each peer that sees a new item forwards it to its own peers.
- Nodes ignore items they have already seen.
- This peer-to-peer diffusion eventually spreads data to most honest nodes.

2.9.3 Eclipse attacks and connectivity assumption

Connectivity assumption: every honest node can reach every other honest node through some path in the network.

Eclipse attack:

- Attacker surrounds a victim with malicious peers.
- Victim only connects to attacker-controlled nodes and is cut off from the honest network.
- This can delay or hide blocks and transactions, enabling double spending or inconsistent views.

Maintaining good connectivity and diverse peer sets is critical for blockchain security.

2.10 Quick Summary

- Hashes and signatures support basic authenticated storage.
- Merkle trees give efficient proofs of inclusion with size $O(\log n)$.
- Merkle trees extend to sets and enable non-membership proofs.
- Tries store key-value maps by sharing prefixes; Patricia tries compress long paths.
- Merkle Patricia tries combine hashing and compressed tries; Ethereum uses them for state.
- Blockchain data structures link blocks using hash pointers and PoW.
- Bitcoin uses UTXOs, scripts, Merkle trees of transactions, and a P2P gossip network.

3 Week 3

3.1 Smart Contracts: Concept

Smart contract: a computer program that runs on the blockchain.

- Code is stored on chain and executed by all full nodes.
- Execution is deterministic: all honest nodes get the same outcome.
- Code can read:
  - Its own internal storage.
  - Transaction context (sender, value, data).
  - Recent block data (in Ethereum).
- Code of a deployed contract cannot change (immutability).

Legal caution: from a legal point of view, "smart contracts" are usually neither legally smart nor contracts; they are programs enforcing some rules on chain.

3.2 Bitcoin Transactions and Script

3.2.1 Transaction structure

A Bitcoin transaction consists of:

- **Inputs**:
  - Reference to a previous transaction output (tx hash + index).
  - `scriptSig`: unlocking script that proves right to spend.
- **Outputs**:
  - `value`: amount of BTC.
  - `scriptPubKey`: locking script specifying spending conditions.

Validation rule: for each input, the node executes

$$\texttt{scriptSig} \| \texttt{scriptPubKey}$$

on a stack machine and checks that it finishes with value `TRUE` on top of the stack.

3.2.2 Bitcoin Script basics

- Stack based, not Turing complete.
- Data (e.g. `<sig>`, `<pubKey>`) is pushed to the stack.
- **Opcodes**:
  - Arithmetic: `OP_ADD`, `OP_ABS`, …
  - Stack operations: `OP_DROP`, `OP_SWAP`.
  - Comparisons: `OP_EQUAL`, `OP_EQUALVERIFY`.
  - Crypto: `OP_HASH160`, `OP_SHA256`.
  - Signatures: `OP_CHECKSIG`, `OP_CHECKMULTISIG`.
  - Timelocks: `OP_CHECKLOCKTIMEVERIFY`, `OP_CHECKSEQUENCEVERIFY`.

3.2.3 P2PKH example

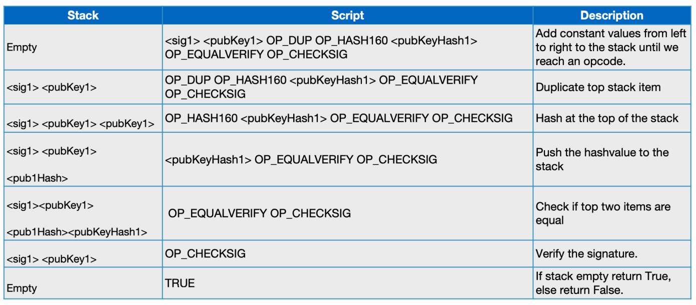
Output script (`scriptPubKey`):

```
OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

Input script (`scriptSig`) when spending:

```
<sig> <pubKey>
```

Execution steps:



Stack	Script	Description
Empty	<sig1> <pubKey1> OP_DUP OP_HASH160 <pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG	Add constant values from left to right to the stack until we reach an opcode.
<sig1> <pubKey1>	OP_DUP OP_HASH160 <pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG	Duplicate top stack item
<sig1> <pubKey1> <pubKey1>	OP_HASH160 <pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG	Hash at the top of the stack
<sig1> <pubKey1> <pub1Hash>	<pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG	Push the hashvalue to the stack
<sig1><pubKey1> <pub1Hash><pubKeyHash1>	OP_EQUALVERIFY OP_CHECKSIG	Check if top two items are equal
<sig1> <pubKey1>	OP_CHECKSIG	Verify the signature.
Empty	TRUE	If stack empty return True, else return False.

3.3 Limitations of Bitcoin Script

- **No loops** and restricted control flow: not Turing complete.
- **No internal state**: script cannot store persistent variables.
- **Value blind**: script cannot inspect the exact amount being sent, it just validates spending conditions.
- **Blockchain blind**: cannot access block header fields (e.g. nonce, previous hash) except for basic locktime operations.

These limitations make Bitcoin simple and secure but restrict expressiveness.

3.4 Extending Bitcoin Functionality

Two main options:

3.4.1 Build on top of Bitcoin

- Use Bitcoin as a base layer and encode extra logic at a higher protocol layer.
- Pros: reuses existing security and mining power; cheaper to deploy.
- Cons: limited flexibility because higher layer must respect Bitcoin's script and transaction model.

3.4.2 Build an independent blockchain

- Design a new protocol from scratch (e.g. Ethereum).
- Pros: can add new opcodes and richer state; more expressive smart contracts.
- Cons: must bootstrap own validators or miners; higher development and maintenance cost.

3.5 Ethereum Overview

Ethereum keeps the blockchain idea but turns it into a **universal replicated state machine**.

- One global state shared by all nodes.
- Transactions are state transition requests.
- A virtual machine (EVM) applies transactions to the state.
- Turing complete bytecode language for smart contracts.
- Decentralized applications (DApps) can be deployed and executed on chain.

Consensus and Sybil resistance:

- Originally Proof of Work (Ethash), now Proof of Stake (validators, staking, Gasper).

3.6 Ethereum Accounts

3.6.1 Global state

Ethereum's global state is a mapping from 20-byte addresses to account objects.
Each account has:

- `address`: 160-bit identifier.
- `balance`: amount of Ether (in wei).
- `nonce`: number of sent transactions.
- (for contract accounts) `code` and `storage`.

3.6.2 Two types of accounts

Externally Owned Account (EOA):

- Address is derived from a public key: addr $= H(\text{pubKey})$.
- Controlled by a private key held by a user.
- No code or storage.

Contract account:

- Address is derived from creator address and nonce.
- Contains immutable contract code and persistent storage.
- Cannot initiate transactions on its own; reacts to incoming transactions or messages.

3.6.3 UTxO vs account model

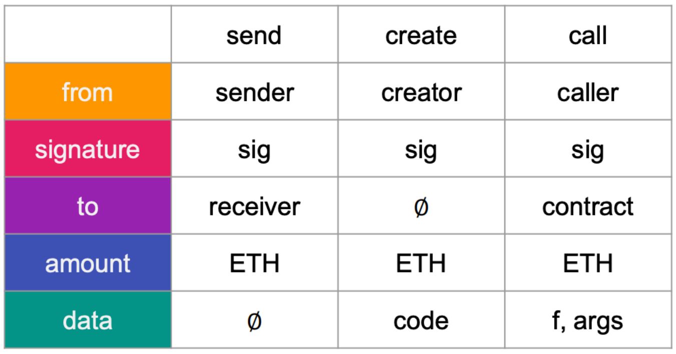
- UTxO:
  - Better for privacy and parallelism.
  - Outputs are spent or unspent.
- Account model (Ethereum):
  - Conceptually simpler.
  - More compact representation of balances and state.

3.7 Ethereum Transactions

A transaction includes:

- `from`: recovered from the signature (sender's address).
- `to`: recipient address (EOA or contract). If empty, this is a contract creation tx.
- `value`: amount of Ether in wei.
- `data`: payload. For contracts this encodes which function to call and its arguments. Empty for simple ETH transfers.
- `nonce`: counts how many transactions the sender has already sent, prevents replay.
- `gasLimit` (`startgas`): maximum gas units the sender is willing to use.
- `gasPrice`: price per gas unit (or fee parameters in newer fee model).
- `signature`: proves authorization by the sender.

3.7.1 Types of transactions



	send	create	call
from	sender	creator	caller
signature	sig	sig	sig
to	receiver	∅	contract
amount	ETH	ETH	ETH
data	∅	code	f, args

3.8 Ethereum Block Structure and Production

3.8.1 Block header format

Each Ethereum block contains the list of transactions and a commitment to the most recent global state.
The block header stores:

- **prev**: hash pointer to the previous block.
- **hash**: block hash (identifier).
- **time**: block timestamp.
- **gasLimit**: maximum total gas allowed in the block.
- **gasUsed**: total gas consumed by included transactions.
- **nonce**: used in the old PoW design (no longer relevant under PoS).
- **difficulty**: PoW difficulty (historical field).
- **miner**: address of the block proposer (validator under PoS).
- **extra**: optional metadata.
- **state root**: Merkle Patricia Trie root of the global account state.
- **transaction root**: Merkle root of all transactions in this block.
- **receipt root**: Merkle root of all transaction receipts (status, logs, gas usage).

The **state root** commits to all accounts, where each account record contains:

- **address**
- **code** (empty for EOAs)
- **storage** (persistent key/value data)
- **balance** (in wei)
- **nonce** (number of sent transactions)

3.8.2 Block production and rewards

- Blocks contain: the ordered transaction list and the most recent state (via the state root).
- Typical block time: about **12 seconds**.
- Since 2022, Ethereum uses **Proof-of-Stake (Gasper)** for Sybil resistance and consensus.

- Previously, Ethereum used **Proof-of-Work** with Ethash (memory-hard PoW).

Rewards and fees:

- **Before the Merge (PoW):** block miner received a fixed block reward (e.g. 2 ETH) plus all transaction fees.
- **After the Merge (PoS):**
  - The block proposer (validator) receives a **base block reward** (newly issued ETH), depending on total ETH staked.
  - Each transaction has a **base fee** which is *burned*, reducing total ETH supply.
  - Users can add **priority fees (tips)** that are paid directly to the block proposer.

3.9 Messages between contracts

Contracts cannot create real transactions, but can send *messages*:

- Messages are internal calls generated by running contract code.
- They exist only inside EVM execution and are not part of the P2P network.
- A message can transfer ETH and call functions of other contracts.

Execution chain:

- External transaction triggers code of a contract.
- That contract can send messages to other contracts.
- Each recipient runs its code in turn.

3.10 Ethereum Virtual Machine (EVM)

- Stack based architecture with 1024-element stack of 256-bit words.
- Bytecode instruction set; each opcode has a defined gas cost.
- Three data areas:
  - **Stack**: last-in first-out, used for intermediate values.
  - **Memory**: transient byte array, cleared between transactions.
  - **Storage**: persistent key-value store, written to and read by contract code.
- Crypto primitives: hash functions, signature verification.
- Can read execution context: `msg.sender`, `msg.value`, `block.timestamp`, etc.

3.11 Gas and Transaction Fees

3.11.1 Why gas is needed

- Every node must execute all transactions and keep the full state.
- Without limits, a malicious contract could run forever (halting problem).
- Gas sets a hard bound on how much computation and storage a transaction may consume.

3.11.2 Gas fields in a transaction

- **Gas limit** (`startgas`): maximum gas units the sender allows for this transaction.
- **Gas price**: price per gas unit in wei (or equivalent fee parameters).

Maximum fee the sender is willing to pay is:

$$\text{maxFee} = \text{gasLimit} \times \text{gasPrice}.$$

3.11.3 Execution with gas

Simplified algorithm for a transaction:

1. Check that gasLimit $\times$ gasPrice $\leq$ balance. If not, reject.
2. Deduct gasLimit $\times$ gasPrice from sender balance.
3. Set `gas` $\leftarrow$ gasLimit.
4. Execute bytecode, decreasing `gas` by the cost of each operation.
5. If execution finishes normally, refund unused gas:

$$\text{refund} = \text{gasRemaining} \times \text{gasPrice}.$$

6. If gas reaches zero before finishing (out of gas), revert state changes. The prepaid fee is not refunded.

Note: blocks also have a *block gas limit*, so the sum of gas used by all transactions in a block cannot exceed this bound.

3.12 Introduction to Solidity

Solidity is a high level language that compiles to EVM bytecode.

- Syntax resembles JavaScript.
- Statically typed: every variable must have an explicit type.
- Supports contracts, state variables, functions, events, modifiers, inheritance, and interfaces.

3.12.1 Basic contract example

A minimal contract usually:

- Starts with a `pragma` line specifying the compiler version range.
- Declares a `contract` with a name.
- Contains one or more functions. For example, a simple contract `HelloWorld` may have a `print` function marked `public` and `pure` that returns the string `"Hello World!"`.

3.13 Solidity: Variables and Types

3.13.1 State vs local variables

- **State variables**:
  - Declared at contract level.
  - Stored in contract storage (persistent and expensive to change).
- **Local variables**:
  - Declared inside functions.
  - Live only during function execution.
  - Value types are kept on the stack; reference types require an explicit data location.

3.13.2 Value types

- **bool**: `true` or `false`.
- **int** and **uint**: signed and unsigned integers of 8 to 256 bits (e.g. `uint256`, `int8`).
- **address**: 20-byte address; `address payable` can receive Ether.
- **bytes1** to **bytes32**: fixed size byte arrays.
- **enum**: user defined type with a finite set of named values.

Variables without explicit initialization receive the default zero value for their type.

3.13.3 Reference types

- Dynamic arrays: `uint[]` or `bytes` or `string`.
- Static arrays: for example, `uint[5]`.
- **mapping(KeyType => ValueType)**: key-value dictionary, non-iterable.
- **struct**: groups multiple fields into one type, e.g. a `Voter` struct with fields for weight, address, and whether they have voted.

3.14 Visibility and Function Types

3.14.1 Visibility of functions and variables

- **public**:

  - Functions callable from outside and inside contracts.
  - For public state variables, the compiler generates a getter automatically.
- **external**:
  - Callable only from outside the contract.
  - Cannot be used for state variables.
- **internal**:
  - Callable only inside the contract or from derived contracts.
- **private**:
  - Callable only inside the contract that defines them (not visible in children).

3.14.2 Function modifiers (state mutability)

- **view**: function promises not to modify state, but may read it.
- **pure**: function promises not to read or modify state (depends only on its arguments).
- **payable**: function can receive Ether along with the call.

Remember: on-chain data is publicly visible regardless of visibility keywords; these only restrict who can *invoke* a function.

3.15 Solidity Inheritance and Interfaces

- Solidity supports multiple inheritance between contracts.
- The keyword `is` is used to derive one contract from another.
- Derived contracts can access non-`private` state and internal functions of parents.
- Interfaces are abstract contracts that only declare function signatures (no implementation). Implementing contracts must provide the body of each function.

Typical pattern:

- Define an interface `Regulator` that specifies functions like `checkValue` and `loan`.
- Implement a concrete `Bank` contract `is Regulator` that maintains an internal balance, implements deposit/withdraw functions, and provides concrete definitions of `checkValue` and `loan`.

3.16 Data Location: storage, memory, calldata

- **storage**:
  - Persistent key-value store for state variables.
  - Expensive to read and write; changes are stored on chain.
- **memory**:
  - Temporary area for reference types inside functions.
  - Cleared after the function ends.
- **calldata**:
  - Read-only location for function arguments of external functions.
  - Cheaper than memory for dynamic types.

Assignment behaviour:

- `storage` ↔ `memory`: data is copied.
- `memory` ↔ `memory`: references are passed.
- Local variables of type `storage` act as references (aliases) to existing state variables.

3.17 Events and Modifiers

3.17.1 Events

- Provide a logging mechanism inside the EVM.
- Event arguments are stored in the transaction log, not in contract storage.
- Off-chain clients (e.g. in JavaScript or Python) can subscribe to events and react to them.
- A typical pattern is an event such as `Deposit(from, id, value)` emitted whenever Ether is deposited into the contract.

3.17.2 Modifiers

- Modifiers are reusable preconditions or wrappers around functions.
- The body of the modifier is injected at the point where `_;` appears.
- Common example: an `onlyOwner` modifier that checks `msg.sender == owner` before executing the function body.
- A contract `Owned` may set the `owner` to the deployer in the constructor and define `onlyOwner`; a derived contract `Mortal` can then protect a `close` function with this modifier.

3.18 Global Variables and Units

3.18.1 Ether units

- `1 ether == 10^18 wei`.
- Other common units: `wei`, `gwei`, `szabo`, `finney`.

3.18.2 Time units

- Suffixes: `seconds`, `minutes`, `hours`, `days`, `weeks`.
- Example: `1 hours == 60 minutes`.

3.18.3 Common global variables

- Block properties: `block.timestamp`, `block.number`, `block.coinbase`.
- Transaction and message context: `msg.sender`, `msg.value`, `msg.data`, `tx.origin`.
- Address helpers: `addr.balance`, `addr.transfer(...)`, `addr.call(...)`.

3.19 Fallback and Receive Functions

- `receive()`:
  - Declared as `receive() external payable`.
  - Executed when the contract receives Ether with empty data.
- `fallback()`:
  - Declared as `fallback() external` (optionally `payable`).
  - Executed when no other function matches the call data, or when data is non-empty and `receive` does not exist.
- Both functions should be simple and use little gas to avoid unexpected failures.

3.20 Sending Ether: transfer, send, call

- `transfer`:
  - Forwards 2300 gas to the recipient.
  - Reverts on failure.
  - Historically considered safe against re-entrancy because the gas stipend is small.
- `send`:
  - Also forwards 2300 gas.
  - Returns a boolean success flag instead of reverting; caller must check it.
- Low-level `call` with value:
  - Can send Ether and forward an arbitrary amount of gas.
  - Returns a success flag and returned data.
  - Flexible but vulnerable to re-entrancy if state updates and external calls are not ordered carefully.

Best practice: use `call` together with the checks–effects–interactions pattern and, if needed, explicit re-entrancy guards.

3.21 Interacting with Other Contracts

- Contracts can create new contracts and call existing ones.
- A typical pattern: a “factory” contract that deploys new instances of another contract (e.g. `Universe` creating many `Planet` contracts), stores their addresses in an array, and emits an event each time a new instance is created.

- Interaction is done by using the other contract's type and calling its functions as methods.

3.22 Quick Summary

- Bitcoin uses a stack based, non Turing complete scripting system to authorise spending of UTxOs.
- Ethereum generalises blockchains into a universal replicated state machine with accounts, contracts, and global state.
- The EVM is a stack machine with storage, memory, and stack; gas ensures that computation is bounded and paid for.
- Solidity is a high level language for writing contracts, with explicit types, visibility, data locations, and support for inheritance.
- Events, modifiers, and global variables are key tools for building practical contracts.
- Ether transfers use `transfer`, `send`, or `call`; understanding their differences is important for security.

4 Week 4

4.1 1. Lecture Overview

This lecture focuses on identifying security hazards in smart contracts and designing safer contract architectures. We examine four main attack vectors:

- Denial-of-Service (DoS)
- Griefing attacks
- Reentrancy attacks
- Front-running

Key defensive patterns include Pull-over-Push, Checks–Effects–Interactions, safe fallback design, avoiding `tx.origin`, safe randomness, and overflow/underflow protection.

4.2 2. Denial-of-Service and Griefing

Unbounded loops can make functions impossible to execute once arrays grow large. Example insecure pattern:

```
for (uint i=0; i<investors.length; i++) {
    investors[i].addr.send(investors[i].dividendAmount);
}
```

As the array size increases, gas requirements exceed block limits, leading to DoS. Griefing attacks intentionally exploit this by causing certain `send()` calls to fail, blocking all refunds.

Solution: Pull-over-Push. Instead of transferring funds inside a loop, store refund balances and allow users to withdraw individually.

```
// Pull model
refunds[user] += amount;
function withdrawRefund() external {
    uint r = refunds[msg.sender];
    refunds[msg.sender] = 0;
    msg.sender.transfer(r);
}
```

4.3 3. Reentrancy Attacks

A reentrancy attack occurs when an external call triggers a fallback function that re-enters the vulnerable contract before state updates occur.

Vulnerable pattern:

```
uint amount = balances[msg.sender];
require(msg.sender.call.value(amount)());
balances[msg.sender] = 0;
```

Attackers repeatedly drain funds through recursive fallback calls. The DAO attack (2016) exploited this weakness, resulting in a $50M loss.

Mitigation: Checks–Effects–Interactions.

```
uint amount = balances[msg.sender];
balances[msg.sender] = 0;
msg.sender.transfer(amount);
```

Additional protections: mutex locks, Pull-over-Push pattern.

4.4 4. Solidity-Specific Hazards

4.1 Forcibly sending Ether. Ether can be sent to a contract without triggering fallback functions, e.g.:

- `selfdestruct(target)`
- precomputed contract addresses
- block reward redirection

Thus, do not rely on strict balance equality.

4.2 Delegatecall hazards. `delegatecall` uses the caller's storage and `msg.sender`. Malicious libraries can overwrite critical variables like `owner`.

4.3 Misuse of `tx.origin`. Using `tx.origin` for authorization enables phishing attacks. Always use `msg.sender`.

4.4 Fallback complexity. Fallback functions should contain minimal logic to avoid vulnerabilities such as reentrancy.

4.5 Default values. Uninitialized mapping entries return default values (e.g., 0). Incorrect handling led to the Nomad Bridge hack (2022).

4.5 5. Merkle Tree Vulnerabilities

Sparse Merkle Trees assign empty values to uninitialized leaves, enabling forged proofs. Binance Bridge was exploited through a manipulated AVL Merkle proof. Avoid custom cryptographic implementations unless formally verified.

4.6 6. Front-running Attacks

Miners reorder transactions by gas price.

Example:

```
registerName("alice")
```

An attacker sends:

```
registerName("alice") // higher gas price
```

Commit–Reveal Scheme prevents this:

- Commit: `hash(value, nonce)`
- Reveal later
- Verify hash

4.7 7. Randomness Hazards

Sources like `block.timestamp`, `block.number`, `blockhash`, `msg.sender` are predictable or miner-controlled.

Future blockhash is also insecure because miners can withhold or reorder blocks.

Secure randomness: Commit–Reveal. Both parties commit to random values, reveal later, then combine (e.g., XOR). If either party is honest, randomness is secure.

4.8 8. Integer Overflow and Underflow

Prior to Solidity 0.8, arithmetic did not include overflow checks. Example:

```
balance[msg.sender] -= value;
```

Mitigation:

- Use Solidity 0.8+ (automatic checks)
- Use SafeMath for older versions

4.9 9. Gas Fairness

Depending on contract design:

- Last contributor may pay all gas
- Beneficiary may pay
- All parties may share costs

Fairness must be considered in crowdfunding or payout logic.

4.10 10. Example of an Insecure Contract

The Rock–Paper–Scissors example demonstrates:

- Commit has no nonce, so the commitment `sha256(hand)` can be brute-forced (only 3 possibilities), letting an attacker discover the opponent's move before choosing their own.
- Anyone can call `open()`, meaning any account can reveal moves for players or manipulate when the game resolves.
- No deposit validation, so players can join without paying, and the contract incorrectly assumes it always holds exactly 1 ETH to pay the winner.
- `selfdestruct` sends all funds to the caller, allowing any user who triggers `open()` to steal the entire contract balance.

5 Week 5

5.1 The Byzantine Generals Problem

The Byzantine Generals Problem describes the difficulty of reaching agreement in a distributed system where some participants may behave arbitrarily or maliciously.

Key ideas:

- Nodes may send conflicting or incorrect information.
- Honest parties cannot tell which messages are trustworthy.
- Reliable agreement requires a protocol that tolerates Byzantine faults.

This motivates the study of consensus protocols in adversarial environments.

5.2 The Consensus Problem

Consensus formalizes what it means for multiple parties to "agree" in a distributed setting.

A protocol must satisfy three properties:

- **Termination**: every honest party eventually outputs a value.
- **Agreement**: all honest parties output the same value.
- **Validity**: if all honest parties start with the same input v, the output must be v.

Strong validity ensures that the output must originate from an honest party's input.

5.3 Honest Majority Requirement

There are fundamental impossibility results:

- An adversary controlling too many parties can force violations of agreement or validity.
- In systems without trusted setup, consensus requires $t < n/3$ Byzantine faults.
- With cryptographic setup (e.g., PKI), consensus under synchrony is possible with $t < n/2$.

Intuition: honest parties cannot distinguish among scenarios where different subsets of participants are corrupted; thus, too many adversarial parties break correctness.

5.4 Classical vs Ledger Consensus

Traditional consensus:

- "One-shot" – decides a single value.
- Uses authenticated channels and fixed participants.
- Ensures termination, agreement, validity.

Ledger (blockchain) consensus:

- Runs indefinitely.
- Must incorporate new transactions continuously.
- Must tolerate dynamic participation and an unauthenticated, peer-to-peer network.

Ledger consensus replaces classical conditions with:

- **Common Prefix** (consistency)
- **Chain Growth** (liveness)
- **Chain Quality** (adversarial influence bounded)

These properties define when a blockchain behaves like a consistent, append-only log.

5.5 The Bitcoin Backbone Model

Bitcoin is modeled as a protocol executed in synchronous rounds by many parties.

Core components:

- **Chain validation predicate**: checks PoW correctness and structural validity.
- **Chain selection rule**: adopt the valid chain with the most cumulative work ("max-valid" or longest chain rule).
- **Proof of Work function**: probabilistic mechanism enabling random leader election.

A block contains:

- ctr: nonce for PoW.
- x: data such as transaction roots.
- s: hash pointer to previous block.

Finding a valid block requires hashing until:

$$H(\text{ctr}\|x\|s) < T$$

where T is the global difficulty target.

5.6 Key Security Properties of Blockchains

5.6.1 Common Prefix (Consistency)

Honest parties' blockchains differ only in the most recent k blocks. This ensures finalized blocks do not get reverted.

Attack example:

- **Racing attack**: adversary tries to build a secret chain to overtake the honest chain.

5.6.2 Chain Growth

In any sufficiently long period, the honest chain grows by at least a linear number of blocks.

Attack example:

- **Abstention attack**: adversary withholds blocks to slow the chain.

5.6.3 Chain Quality

In any window of blocks, the fraction of adversarial blocks is bounded. This prevents adversaries from dominating the chain even if they temporarily get lucky.

Attack example:

- **Block withholding attack**: adversary tries to replace honest blocks whenever they find their own.

5.7 From Blockchain Properties to Ledger Consensus

- **Consistency** of the ledger derives from the Common Prefix property.
- **Liveness** derives from the combination of Chain Growth and Chain Quality.
- Honest blocks eventually appear and become irreversible after k confirmations.

Thus, the blockchain implements a robust form of asynchronous, permissionless consensus.

5.8 Proof of Work and Mining

PoW mining is essentially a randomized leader election mechanism.

Properties:

- Parallelizable: miners increase success probability by adding more hash power.
- Easy verification: given the nonce, checking PoW is one hash computation.
- Hard to shortcut: success probability is proportional to computational power.

5.9 Mining Pools

Because PoW rewards are probabilistic, miners cooperate in pools:

- Miners submit “shares” to prove contributed work.
- Pool distributes rewards proportionally to contributed hash power.

5.10 Dynamic Availability and Difficulty Adjustment

Bitcoin must operate despite fluctuating numbers of miners.

- If total hash power increases, blocks appear too quickly.
- If hash power decreases, block production slows.

Bitcoin adjusts difficulty every 2016 blocks to keep block intervals stable.

Let f be the probability that at least one honest miner finds a block in a round:

- If f is too small: chain grows too slowly $\rightarrow$ liveness suffers.
- If f is too large: many forks occur $\rightarrow$ consistency suffers.

Difficulty adjustment keeps f within a safe range.

5.11 Difficulty Raising Attack

If the difficulty adjustment mechanism were poorly designed, an adversary with minority hash power could:

- create a private chain with artificially high difficulty,
- exploit increased variance to occasionally overtake the honest chain.

Bitcoin avoids this by bounding difficulty adjustment using the epoch threshold.

5.12 Summary

- Consensus in adversarial networks requires dealing with Byzantine faults.
- Classical consensus properties map to blockchain properties: Common Prefix, Chain Growth, Chain Quality.
- Bitcoin’s PoW, longest chain rule, and difficulty adjustment together implement a secure ledger.
- Attacks such as racing, withholding, and difficulty raising highlight why the assumptions and parameter choices matter.