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:

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

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 $\Sigma = (\mathrm{KeyGen}, \mathrm{Sign}, \mathrm{Verify})$.

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

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 $D_1, \ldots, D_n$.
- Compute leaf hashes $H_1 = H(D_1), \ldots, H_n = H(D_n)$.
- 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)

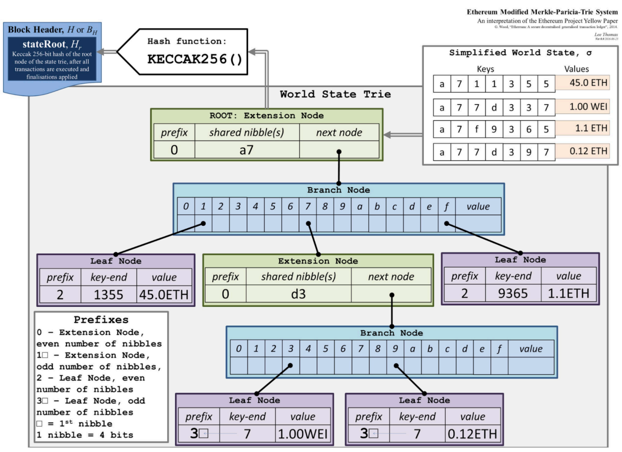
Ethereum combines Merkle hashing with Patricia tries.

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.



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:

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

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: $\text{addr} = H(\text{pubKey})$.
- 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 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 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 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 Solidity-Specific Hazards

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 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 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 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 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 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 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 $t < n/2$.

Setup/Network	Synchrony	Partial Synchrony
No Setup	t < n/3	t < n/3
With Setup	t < n/2	t < n/3

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.

6 Week 6

Permissionless Protocols and Dynamic Availability

Many blockchain protocols, such as Bitcoin, operate in a **permissionless** setting:

- Anyone can join or leave the network without prior authorization.
- Anyone can read the ledger and, after acquiring coins, submit transactions.
- Participation in block production is open to anyone with the relevant resource (e.g. hash power).

The system must tolerate **dynamic availability**:

- Nodes can come online or go offline at any time.
- New nodes need to bootstrap from the existing chain (e.g. follow the longest valid chain).
- There is no prior knowledge of how many parties are online at any moment.

Classic BFT protocols do not handle this type of open, highly dynamic participation. :contentReference[oaicite:0]index=0

Bitcoin's Energy Problem and Motivation for PoS

Bitcoin uses Proof-of-Work (PoW) to solve consensus in a permissionless setting, but at a high cost:

- Massive energy consumption due to hash-based mining.
- Significant electronic waste from rapidly obsolete mining hardware.
- An arms race between honest miners and attackers, where both must constantly burn energy.

Empirical studies show that the climate damages per unit of Bitcoin value can be very high (comparable to some heavy industries), motivating **more energy efficient** consensus mechanisms such as Proof-of-Stake (PoS). :contentReference[oaicite:1]index=1

Nakamoto Design Recap

The Nakamoto-style design has three main components:

- Blocks are linked in a chain via hash pointers.
- In case of forks, nodes adopt the chain with the most accumulated resource (e.g. most work).
- New blocks are produced by a randomized lottery that selects leaders.

In PoW, the probability of winning the lottery is proportional to hash power. In PoS, the idea is to make it proportional to **stake**, that is the amount of cryptocurrency controlled by a party. :contentReference[oaicite:2]index=2

Proof-of-Stake Basics

In PoS systems:

- **Stake** is the amount of tokens or coins controlled by a public key.
- The protocol selects block producers based on stake.
- Security requires that honest parties collectively control a majority of stake.

PoS is resource based but uses a *digital* resource. This makes PoS energy efficient, since no physical work is required for the lottery. :contentReference[oaicite:3]index=3

Time Slots and Leader Election

Many PoS protocols divide time into **slots**:

- The slot length is chosen so that messages can propagate within one slot under the network assumptions.
- Ideally, one block is produced per slot.
- Parties use their local clock to determine the current slot and act accordingly.

Time slotting simplifies analyzing and designing leader election: in each slot, the protocol selects which stakeholders are eligible to create a block. :contentReference[oaicite:4]index=4

From PoW to PoS: Design Attempts

We want to replace the PoW condition

$$H(x, s, ctr) < T$$

with a PoS-based condition that selects leaders in proportion to stake, but without introducing new vulnerabilities.

Attempt 1: Hash with Stake-Dependent Threshold

Require

$$H(x, s, vk_n) < T \cdot \text{stakeFactor}_n$$

- The right-hand side scales with the stake of node n.

Problem: Grinding Attack on x The adversary can vary x (for example the transaction Merkle root) until the inequality holds, gaining unfair advantage.

Attempt 2: Remove the Dependence on Block Content

Require

$$H(s, vk_n) < T \cdot \text{stakeFactor}_n$$

- Now the left-hand side no longer depends on x.

Problem: Content Malleability Since transactions are not included in the hash, an attacker can change the block contents without affecting the PoS condition. This breaks ledger integrity.

Attempt 3: Bind to History with Signatures

Keep the PoS condition similar, but add:

- Use the signing key for vk_n to sign the block content and a hash of the whole history.

This prevents content malleability, because changing the history requires forging signatures.

Problem: Posterior Corruption The adversary can corrupt a leader after the fact and obtain their old signing keys, then re-sign a different history at no cost (costless simulation of the past).

Attempt 4: Key Evolving Signatures (KES)

Introduce **Key Evolving Signatures**:

- Each party periodically updates its signing key and deletes the old one.
- The public key is fixed but internally refers to an evolving sequence of secret keys.
- Old signatures remain verifiable, but old signing keys are supposed to be irretrievably erased.

This blocks posterior corruption, since even if a party is corrupted later, its previous signing keys should no longer exist.

Problem: Adaptive Attacks If the public verification key vk that will be eligible in some future slot is known in advance, an adversary can adaptively corrupt that party before it produces its block.

Attempt 5: Verifiable Random Functions (VRFs)

Use a **VRF** in the PoS condition:

$$\text{VRF}(sk_n, s) < T \cdot \text{stakeFactor}_n$$

- Each party privately evaluates the VRF using its secret key.
- The VRF output is publicly verifiable using vk_n.
- A party only reveals the VRF proof if it wins the slot.

This hides the leader schedule, mitigating adaptive corruptions.

Problem: Stalling Hazard If no party satisfies the inequality in a slot, no block is produced and the same inputs could be reused, potentially stalling progress.

Attempt 6: Adding Time to the VRF Input

To prevent stalling, extend the input with a time dependent value (such as slot or timestamp):

$$\text{VRF}(sk_n, s \parallel ts) < T \cdot \text{stakeFactor}_n$$

- As time advances, the input changes, so the lottery is re-run with fresh randomness.
- Combined with KES and signatures, this forms the basis of modern PoS protocols such as Ouroboros.

This sequence of attempts highlights subtle attack surfaces (grinding, content malleability, posterior corruption, adaptive corruption, stalling) and how PoS designs address them.

Dynamic Stake and Key Grinding

In realistic systems, stake changes over time:

- Tokens are transferred between accounts.
- New stakeholders appear and old ones may leave.

The stake distribution becomes dependent on the particular chain branch, since different branches can contain different transactions.

Key Grinding Attack:

- Keys are generated locally and are cheap to create.
- An adversary can try many potential key pairs and keep those whose VRF outputs give favorable leadership schedules.

Mitigations include committing to keys before using them and deriving VRF inputs from shared randomness that is itself derived from past protocol outputs.

Long-Range Attacks

A **long-range attack** exploits the fact that in PoS, creating blocks is cheap once the stake is in adversarial hands:

- Starting from an old block, the adversary constructs a private fork consisting only of adversarial blocks.
- On that fork, the adversary accumulates all block rewards, so their stake gradually dominates.
- Since producing blocks has no physical cost, the adversary can simulate very long chains offline.
- A newly joining node that sees both the honest chain and the adversarial long fork may be fooled into adopting the wrong one.

Two main countermeasures:

- **Checkpointing**: honest participants rely on external trusted checkpoints (for example, from software updates or social consensus).
- **Chain density**: use local density statistics to distinguish honest chains from long-range forks (for example as in Ouroboros Genesis, where the honest chain is denser shortly after forks).

Permissioned Ledgers and PKI

In a **permissioned** ledger:

- Only authorized participants can produce blocks or submit transactions.
- The set of participants is often static and known from the genesis block.

A Public-Key Infrastructure (PKI) is used:

- Certification authorities issue X.509 certificates binding public keys to identities.
- Nodes authenticate each other using certificates and establish secure channels (for example via TLS).
- Certificates can be revoked if keys are compromised or algorithms become insecure.

Centralized vs Distributed Permissioned Ledgers

Centralized permissioned ledger (PoA):

- A single server maintains the log of transactions.
- Clients authenticate to the server to read or write.
- Correctness and liveness rely entirely on the honesty and availability of the server.

Distributed permissioned ledger:

- Multiple servers maintain replicated copies of the log.
- All servers share a genesis block containing the authorized participants.

- Writers send their inputs to all servers.
- Servers run a consensus protocol to decide which inputs to append.

Access control for readers and writers can be enforced via certificates attached to the genesis block and subsequent updates. :contentReference[oaicite:12]index=12

Classical BFT Consensus vs Nakamoto Consensus

Classical BFT consensus:

- Typically permissioned and runs among a fixed set of nodes.
- Does not rely on resource assumptions such as majority hash power or stake.
- Communication complexity is high: all parties participate in each consensus instance.

Nakamoto-style consensus:

- Typically permissionless.
- Relies on an assumption that honest nodes control a majority of some resource (hash power or stake).
- Communication complexity is low: only the current leader needs to broadcast its block.

Graded Broadcast and BFT-Ledger Construction

A key tool in BFT protocols is **graded broadcast** (graded consensus):

- There is one sender and several receivers.
- Each receiver outputs a pair (M_i, G_i) where M_i is a message and $G_i \in \{0, 1, 2\}$ is a grade.

Properties:

- If the sender is honest, all honest receivers output the same message with grade 2.
- If some honest receiver outputs $(M, 2)$, then all honest receivers output M with grade in $\{1, 2\}$.

Graded broadcast can be implemented in three communication rounds using threshold counts of messages (at least $2n/3$ or at least $n/3$), assuming fewer than $n/3$ Byzantine faults.

By combining graded broadcast with a **binary consensus** protocol, one can build a BFT ledger where each phase appends a new block when all parties agree on the same message. :contentReference[oaicite:13]index=13

Byzantine Binary Consensus and EIG

In **Byzantine binary consensus**, each party starts with a bit $v_i \in \{0, 1\}$ and must output a bit u_i such that:

- Termination: all honest parties eventually output.
- Agreement: all honest parties output the same bit.
- Validity: if all honest parties start with the same bit v, then they must output v.

One generic solution uses the **Exponential Information Gathering** (EIG) algorithm:

- In round 1, each party sends its input to all others.
- In each subsequent round, parties forward what they heard in the previous round.
- Each party builds a labeled tree of possible message chains and then uses majority rules bottom up to decide an output.

In the synchronous setting, EIG terminates in $t + 1$ rounds with up to t Byzantine faults, and achieves agreement and validity under appropriate bounds on n and t. :contentReference[oaicite:14]index=14

Impossibility Results and Thresholds

There are several classical impossibility results:

- For synchronous systems, consensus is impossible if $n < 3t + 1$.
- For asynchronous systems, deterministic consensus is impossible with even a single fault (FLP result).
- With cryptographic setup and signatures, some bounds can be relaxed, but there are still lower bounds on the number of rounds and required number of parties.

These results justify why Nakamoto-style protocols assume resource majority instead of attempting fully asynchronous BFT consensus.

BFT-style PoS Protocols

BFT-style PoS protocols (such as Algorand) combine PoS leader selection with BFT agreement:

- In each slot, a committee of stakeholders is selected using a PoS lottery (often via VRFs).
- The committee runs a BFT protocol to agree on the next block.
- Blocks are finalized once the committee reaches agreement.

Security questions include:

- How to derive secure randomness for committee selection.
- How to prevent grinding attacks on randomness.
- How to handle adaptive corruptions and long-range attacks.

Open Questions in Ledger Protocols

The lecture closes with several open research questions:

- How to achieve permissionless clock synchronization and ensure that parties agree on time slots.
- How to incentivize key erasure in KES based systems, and whether rational parties might keep old keys.
- Whether systems can self-heal after temporary periods in which the adversary controls a majority of the resource (stake or hash power).

These questions are central to the ongoing development of secure and robust PoS and ledger protocols. :contentReference[oaicite:16]index=16

7 Week 7

7.1 Cryptocurrency Economics and Incentives

This lecture introduces the economic foundations of blockchain protocols. Traditional analyses assume participants are either *honest* or *malicious*. Instead, real systems involve rational agents who maximize their own utility.

Key questions:

- Why should participants follow the protocol?
- Are consensus properties (safety, liveness) emergent from incentives?
- How do rewards, fees, and protocol rules shape rational behavior?

7.2 Mining Incentives in Bitcoin

A miner's revenue consists of:

- **Transaction fees**:

$$\text{fee} = \sum \text{inputs} - \sum \text{outputs}.$$

- **Block reward**: newly minted coins created in the coinbase transaction.

Coinbase transaction:

- First transaction in every block.
- No inputs; miner freely sets `scriptSig`.
- Output value must not exceed blockreward + total fees.

7.2.1 Bitcoin monetary policy

- Initial reward: 50 BTC.
- Reward halves every 210,000 blocks.
- Supply approaches a cap of ≈ 21 million BTC.

Early eras create a disproportionate fraction of all BTC, leading to highly concentrated wealth distribution.

7.3 Mining Profitability and Hardware

Bitcoin may be mined using:

- CPUs
- GPUs
- ASICs (specialized hardware)

Due to difficulty and energy cost, most hardware today mines at a net loss. Mining profitability depends on:

- electricity price,
- hardware efficiency,
- block reward and fee market,
- network difficulty.

This motivates mining pools.

7.4 Nash Equilibrium in Blockchain Protocols

Each participant has a **utility function**:

$$f_i(x_1, \ldots, x_n),$$

mapping all parties' strategies to a real-valued reward.

Nash equilibrium: no party can increase utility by unilaterally deviating.

Two mining reward models:

- **Absolute rewards**: total BTC earned.
- **Relative rewards**: fraction of total BTC earned.

Bitcoin under absolute rewards: honest mining is a Nash equilibrium. **Bitcoin under relative rewards**: not a Nash equilibrium — selfish mining yields higher expected utility.

7.5 Selfish Mining Attack

Selfish mining is a deviation strategy allowing a coalition of miners to increase their *relative* share of block rewards.

Key steps:

1. Miner finds a block but withholds it (keeps it private).
2. If honest miners find competing blocks, attacker strategically releases withheld blocks.
3. Honest miners adopt attacker blocks first if attacker controls message propagation.

Effects:

- Public chain grows slower.
- Attacker obtains $> \alpha$ fraction of rewards even with only α fraction of hash power.
- With difficulty adjustment, absolute rewards may also increase.

7.6 Block Reward Zero Attack

When block rewards become negligible, miners rely solely on transaction fees.

A deviation becomes profitable:

- If two blocks compete at the same height,
- a rational miner may choose the block leaving *more unclaimed fees* for the next block,
- thus incentivizing mining on low-fee forks.

This threatens long-term stability when block rewards fade.

7.7 Bribery Attacks

An attacker creates a fork and includes a special transaction τ_0 that offers a bribe to miners who extend the attacker's block.

- τ_0 double-spends its input on the public chain.
- If attack fails, attacker loses nothing (the bribe is invalidated).
- If miners adopt the fork, attacker may succeed at rewriting history.

Shows that economic incentives can be manipulated to undermine security.

7.8 Mining Pools and Pool Attacks

Miners join pools to reduce variance in income.

7.8.1 Pool mechanics

- Pool sets an internal (higher) target T_{pool}.
- Miners submit "shares" proving partial work.
- If a real block is found, rewards are split proportional to contributed shares.

7.8.2 Block withholding attack

Pool A infiltrates Pool B with hash power α':

- Infiltrators submit shares but withhold valid blocks.
- Pool B's effective block rate decreases.
- Pool A gains a larger *relative* share of total network rewards.

This attack demonstrates competitive pressure among pools.

7.9 Utility in Real-World Markets

Real utility differs from in-protocol cryptocurrency utility.

Factors:

- Fiat-denominated costs (hardware, electricity).
- Exchange rate volatility BTC/USD.
- Transaction fees and liquidity.
- Market reaction to attacks.

Surprisingly:

- Markets often fail to penalize attacks.
- E.g. Ethereum Classic suffered multiple 2020 double-spend attacks without severe price decline.

Thus "attack decreases price, so attackers won't attack" is not a reliable security assumption.

7.10 Will Miners Attack Their Own System?

Game-theoretic conclusion:

- If an attack is profitable in expected value *inside the protocol*,
- and the external market does not sufficiently reduce price,
- rational miners may attack even if they hold stake in the ecosystem.

This highlights the need for:

- careful incentive engineering,
- protocol design robust against rational deviations,
- mechanisms that align long-term network health with miner profit.

7.11 Quick Summary

- Bitcoin rewards (fees + subsidy) define miner incentives.
- Under absolute rewards, honest mining is an equilibrium; under relative rewards, selfish mining breaks equilibrium.
- Bribery, fee-driven attacks, and pool attacks reveal deeper incentive vulnerabilities.
- Markets often fail to punish protocol-level attacks.
- Economic analysis is necessary to understand and secure blockchain protocols.

8 Week 8

8.1 Anonymity, Pseudonymity and Privacy

8.1.1 Identity models

Eponymous systems:

- Actions are directly linked to real-world identities.
- Examples: Facebook (real-name policy), verified accounts with KYC, parliamentary voting records.

Pseudonymous systems:

- Identities are represented by arbitrary tags or usernames.
- A single user may control many pseudonyms; one pseudonym may be shared.
- Examples: Reddit / X handles, email addresses, graffiti tags such as "Banksy".
- **Sybil attack**: a single operator creates many pseudonyms

to appear as many distinct participants.

Anonymous systems:

- Actions cannot be linked to any particular participant.
- Only the size of the *anonymity set* (the group of indistinguishable users) is known.
- Examples: secret ballots in elections, Tor usage as seen by a website.

8.1.2 Anonymity vs fungibility

- **Fungibility**: any unit of a currency is interchangeable with any other.
- In Bitcoin every satoshi has a visible history on chain, so coins can be tainted or blacklisted.
- Privacy weaknesses therefore also impact fungibility.

8.2 Bitcoin Privacy and Transaction Graph Analysis

8.2.1 Address reuse and pseudonymity

- Users can create arbitrary many new addresses without on-chain cost.
- In principle this gives pseudonymity, but transaction graph analysis can cluster addresses.

8.2.2 Transaction graph analysis

- Nodes: addresses or transactions; edges: flows of value.
- Common patterns:
  - **Peeling chain**: large UTXO is repeatedly split, sending small amounts and a "peel" change output each time.
  - **Star**: one address sends to many others in a hub-and-spoke pattern.
- These patterns help chain-analytic companies link addresses to the same user.

8.2.3 Indistinguishability examples

- If Alice receives 50 BTC twice, the on-chain view may or may not reveal whether payments go to one person or two different ones.
- In many realistic spending patterns, observers can assign higher probability to one "world" (e.g. Alice-only) than to another (Alice plus Charlie), breaking privacy.

8.3 Centralized Mixing and CoinJoin

8.3.1 Centralized mix

- Users send coins to a *trusted party* that returns coins to recipients in shuffled order.
- The anonymity set has size n (the number of participants), but:
  - the mixer can steal funds or log all mappings,
  - it becomes a single point of failure and a target for regulation or seizure.

8.3.2 CoinJoin idea

- Multiple users jointly create a single transaction that mixes their inputs and outputs.
- Example: Alice and Charlie each spend inputs to standardized outputs (e.g. 1 BTC each).
- After mixing, an external observer cannot tell which input funded which standardized output.

8.3.3 Multiple-input CoinJoin protocol

- Parties: n participants and one temporary *leader*.
- Setup:
  - Each participant sends to the leader:
    * their recipient address b_i,
    * their change address c_i,
    * the corresponding amounts.
  - Leader constructs a combined transaction from all inputs and outputs.
- Signing:
  - Leader broadcasts the unsigned multi-input transaction to all participants.
  - Each participant checks it and sends their signature back.
  - Once all signatures are collected, the leader broadcasts the final transaction.
- Abort issues and questions:
  - Any participant can abort and prevent publication (DoS).
  - A malicious leader can correlate inputs and outputs and violate privacy.
  - It is non-trivial to restart without the offending party while preserving privacy.

8.3.4 Passive vs active adversaries

- **Passive** adversary: only observes on-chain transactions, gains anonymity set of size n per CoinJoin.
- **Active** adversary: participates in protocol (e.g. as leader) and sees mappings between participants and addresses, breaking privacy.

8.4 Mix-nets and CoinJoin

8.4.1 Mix-nets

- A mix-net is a sequence of "mix" servers that shuffle and re-encrypt messages.
- As long as at least one mix in the chain is honest, it is impossible to link input messages to outputs.
- Two main variants:
  - **Decryption mix-nets**: messages are layered with multiple public-key encryptions, each mix strips one layer and shuffles.
  - **Re-encryption mix-nets**: mixes probabilistically re-encrypt and shuffle.

8.4.2 Using mix-nets for CoinJoin

- All participants first publish their public keys; mappings from public keys to accounts are public.
- They feed their addresses and amounts through a decryption mix-net.
- The final mix (leader) receives a shuffled list of outputs and can assemble the CoinJoin transaction but cannot tell which participant contributed which.
- Participants then sign the transaction as usual.

8.4.3 Limitations and Mimblewimble

- Even with CoinJoin, output balances remain visible on chain.
- Mimblewimble-style systems use **Pedersen commitments**:
  - Commitments hide balances but are additively homomorphic.
  - A transaction proves that sum of inputs minus outputs equals zero without revealing amounts.

8.5 Blind Signatures, E-cash and Fair Swaps

8.5.1 Blind signatures

- A user blinds a message m, sends it to a signer who produces a signature on the blinded value.
- After unblinding, the user obtains a valid signature on m that cannot be linked back to the signing interaction.
- Properties:

- Signature verifies against m and public key vk.
- Signer never sees the cleartext message.
- Signer cannot link later published (m, σ) pairs to specific users.

8.5.2 Chaum's e-cash

- Bank issues e-coins by blind-signing structured coin tokens.
- User withdraws an e-coin, then later presents it to a shop.
- Shop checks with the bank that the coin has not been spent before and that the signature is valid.
- Gives strong payer anonymity, but the bank is a trusted central party.

8.5.3 Applying e-cash to Bitcoin payments

- A trustee exchanges on-chain BTC for off-chain e-coins via blind signatures.
- Users spend e-coins to shops, which redeem them for BTC from the trustee.
- Anonymity holds only if the trustee behaves correctly and does not log linkages or disappear with funds.

8.5.4 Fair swaps

- Goal: two parties exchange secrets or assets so that either both receive their output or none do.
- Purely network-based fair exchange is impossible under standard assumptions.
- Workarounds:
  - **Optimistic fair exchange** with a trusted third party used only for dispute resolution.
  - **Resource-based fair exchange**: parties exchange partial information, so aborting gives the other party an advantage.
  - **Fair swaps with penalties**: use a smart contract that escrows deposits and penalizes aborting parties financially.

8.6 Group Signatures, Ring Signatures and Monero

8.6.1 Group signatures

- System has a group manager and an opening authority.
- Any registered member can sign on behalf of the group:
  - Verifier is convinced that "some group member" signed.
  - Opening authority can later reveal which one, if necessary.
- Useful for accountable anonymity.

8.6.2 Traceable signatures

- Allow tracing across multiple signatures from the same user.
- Provide linkability or revocation for misbehaving members while preserving anonymity in normal cases.

8.6.3 Ring signatures

- Signer chooses an arbitrary set of public keys (ring).
- Produces a signature that proves "one of these keys signed" but not which.
- No group manager or central authority is required.
- Verifier only knows that the signer is a member of the chosen anonymity set.

8.6.4 Monero / Cryptonote

- Uses **linkable ring signatures**:
  - Each coin output is one member of a ring.
  - A key image ensures that spending the same output twice is detectable and linkable.
- **Stealth addresses**:
  - Sender creates one-time addresses for the recipient, unlinkable on chain.
  - Recipient scans chain with their private view key to detect payments.
- Provides stronger default privacy than Bitcoin, but anonymity still depends on real-world threat models and how rings are constructed.

8.6.5 Anonymity set relevance

- A large anonymity set is only useful if the adversary cannot shrink it using side information.
- Example: a student using Tor on a campus network might be the only Tor user at a given time, making deanonymization easier despite Tor's design.

8.7 Commitments, Zerocash and ZK-SNARKs

8.7.1 Commitments over the UTXO set

- For each coin, create a commitment $\psi = \text{Commit}(\rho, (v, sn))$:
  - ρ: randomness,
  - v: value,
  - sn: serial number.
- ψ is recorded in the ledger; opening reveals v and sn while hiding them from observers.
- Spending a coin reveals sn and proves that some ledger commitment contains (v, sn) without showing which.

8.7.2 Merkle tree over commitments

- All commitments are stored in a Merkle tree.
- Spending proof shows:
  - existence of a leaf ψ_i equal to $\text{Commit}(\rho, (v, sn))$,
  - membership via a Merkle inclusion proof.
- Statement and witness sizes grow only logarithmically in the number of coins.

8.7.3 ZK-SNARK definition

- A SNARK is a **succinct, non-interactive argument of knowledge** for statements of the form:

$$\exists w : R(x, w) = 1$$

  where R is a polynomial-time relation.
- Properties:
  - **Soundness**: cheating provers cannot convince verifier of false statements, except with negligible probability.
  - **Zero-knowledge**: proof reveals no information about witness w beyond truth of the statement.
  - **Succinctness**: proof size and verification time are independent of witness size and running time of R.

8.7.4 Zerocash coins and pour operations

- Accounts have a key pair (vk, sk).
- Coin structure includes:
  - vk: public key of owner,
  - v: value,
  - random seeds ρ, s, s',
  - derived serial number $sn = \text{PRF}_{sk}(\rho)$,
  - commitments $k = \text{Commit}(s, vk \| \rho)$ and $\psi = \text{Commit}(s', v \| k)$.
- **Pour** operation:
  - Consume one coin and create two new coins of values v_1, v_2 with $v_1 + v_2 = v$.
  - Reveal serial number sn and prove in zero-knowledge that:

    * some Merkle leaf corresponds to a valid coin with that *sn*,
    * the value is conserved,
    * new coins are well formed.

8.7.5 Common Reference String (CRS)

- Many SNARK schemes require a shared setup string.
- Generated via trusted setup or secure multi-party computation.
- **Updateable reference strings** and alternative proof systems (e.g. Bulletproofs) reduce trust but often at higher computational cost.

8.8 Network Security and the Bitcoin P2P Layer

8.8.1 Overlay networks and requirements

- Blockchain protocols run on overlay networks built on top of the Internet.
- Desired properties:
  - Timely (synchronous) message delivery within known bounds.
  - Reliable point-to-point channels.
  - Reliable broadcast for blocks and transactions.

8.8.2 Bitcoin's P2P design

- Peers are identified by IP addresses and communicate over TCP.
- Nodes maintain a limited number of outbound and inbound connections.
- Public vs private networks:
  - Public IP hosts reachable directly.
  - Private IP hosts behind NAT use a router to reach the Internet.

8.8.3 Peer discovery and address tables

- New nodes contact DNS seeders to get initial lists of peer IPs.
- Nodes gossip addresses via `ADDR` messages.
- Each node maintains:
  - a `new` table: addresses learned but not yet tried,
  - a `tried` table: peers that have successfully connected.
- Connection selection uses a bias based on the ratio ρ between tried and new addresses, and recency of timestamps.

8.8.4 Eclipse attacks

- Goal: isolate a victim node so all its connections are controlled by the attacker.
- Steps:
  - Attacker fills victim's `tried` table with adversarial IPs by repeatedly connecting.
  - Attacker floods `new` table with unreachable "trash" IPs via unsolicited `ADDR` messages.
  - When victim restarts, it reconnects mostly to adversarial addresses.
  - Attacker also saturates incoming connection slots so honest nodes cannot connect.
- Consequences:
  - Victim's view of the blockchain is fully controlled and can be censored.
  - Victim's blocks and transactions may never reach the honest network.

8.8.5 Mitigations

- Restrict unsolicited `ADDR` messages.
- Diversify peers across IP ranges and autonomous systems.
- Probe peers before evicting long-lived entries from the `tried` table.

8.8.6 Information propagation and MITM

- Bitcoin uses an inventory-based protocol:
  - Node announces object hashes via `inv`.
  - Peers request full objects via `getdata`.
- Connections have a timeout window (for example, 20 minutes) before being dropped.
- A man-in-the-middle that delays or modifies messages can significantly slow block propagation and harm security.

8.9 Wallets, SPV and Key Management

8.9.1 Full node wallets

- Store the entire blockchain plus UTXO set.
- Verify all transactions and blocks independently.
- Provide strongest security and privacy but require large storage and bandwidth.

8.9.2 Merkle trees and SPV

- Blocks store only the Merkle root of transactions in their header.
- This allows lightweight **Simple Payment Verification (SPV)** clients:
  - Download only block headers, not full blocks.
  - Rely on full nodes to provide relevant transactions plus Merkle proofs.
  - Verify PoW chain and inclusion of their own transactions.

8.9.3 SPV protocol and security

- SPV wallet:
  - connects to several full nodes,
  - sends a Bloom filter encoding its addresses so servers can filter relevant transactions,
  - verifies signatures, basic transaction validity and inclusion under the correct Merkle root.
- Assumes an honest majority of hash power.
- Does not hold full UTXO set and does not see unrelated transactions.
- A malicious server can only mislead the wallet temporarily by building an invalid fork or hiding transactions.

8.9.4 HD wallets and seeds

- Hierarchical Deterministic (HD) wallets derive many key pairs from one master private key (BIP-32).
- A short mnemonic seed encodes the master key and can recover the whole wallet.
- Seed should be backed up offline, usually on paper and optionally encrypted with a passphrase.

8.9.5 Hot vs cold wallets

- **Hot wallet**:
  - Private keys on an Internet-connected device.
  - Convenient for everyday spending.
  - Higher risk of theft via malware or compromises.
- **Cold wallet**:
  - Private keys stored offline.

  - Safer against remote attacks, less convenient for frequent payments.
  - Balance can still be monitored via public keys on an online device.

8.9.6 Paper, brain and hardware wallets

- **Paper wallet**:
  - Private key printed on paper, possibly encrypted with a passphrase.
  - Secure against online attacks, but vulnerable to physical loss or damage.
- **Brain wallet**:
  - Private key derived from a human-memorizable passphrase.
  - Extremely unsafe in practice due to low-entropy passwords; many have been brute-forced.
- **Hardware wallet**:
  - Dedicated device (e.g. Trezor, Ledger) storing keys in secure hardware.
  - Keys never leave the device; device signs transactions and returns them to the host computer.
  - Provides strong security even if the host computer is compromised.
  - Protected by PIN and backed up via a seed.

8.10 Overall Takeaways

- Blockchain privacy ranges from pseudonymous to anonymous, and depends heavily on protocol design and user behavior.
- Techniques such as CoinJoin, mix-nets, blind signatures, ring signatures and ZK-SNARK-based systems aim to enlarge and protect anonymity sets.
- Network-layer security, especially resistance to eclipse and MITM attacks, is critical for maintaining consensus and censorship resistance.
- Wallet design (full node vs SPV, hot vs cold, HD vs hardware) reflects trade-offs between usability, security and privacy.:contentReference[oaicite:0]index=0

9 Week 9

9.1 Security-Critical Computations

Many computations in adversarial environments must guarantee correctness, privacy, and robustness even when participants misbehave.

Three categories:

- **Deterministic with public inputs**: correctness is publicly checkable (e.g. recomputation).
- **Probabilistic with public inputs**: requires coordination to avoid bias (e.g. coin flipping).
- **Private inputs**: requires Secure Multiparty Computation (MPC).

9.2 Passive vs Active Adversaries

Passive adversary (semi-honest):

- Follows protocol steps exactly.
- Tries to learn extra information from received messages.

Active adversary (malicious):

- May deviate arbitrarily.
- Sends malformed or inconsistent messages.
- May send different data to different parties.
- Can refuse to participate at critical moments (selective abort).

Most MPC protocols secure against passive adversaries *completely fail* in the active setting.

9.3 Coin Flipping and Selective Abort

Goal: generate a shared random bit between two distrustful parties.

Passive-secure protocol:

1. Alice commits to x.
2. Bob sends y.
3. Alice opens x.
4. Output: $x \oplus y$.

Selective abort attack: Alice learns the final bit $x \oplus y$ before Bob does. If she dislikes the outcome, she refuses to open the commitment.

Thus:

- Alice learns the output.
- Bob learns nothing.

This violates **fairness**, showing:

Two-party fair coin flipping is impossible without extra assump

9.4 Secret Sharing

9.4.1 Additive sharing

A bit s is shared as:

$$s = a \oplus b,$$

where a is uniform and $b = s \oplus a$.

Each share reveals nothing individually.

9.4.2 Shamir secret sharing

Let $p(x)$ be a degree-t polynomial with $p(0) = s$.

- Share to party i: $p(i)$.
- Any $t+1$ points reconstruct $p(0)$.
- Fewer than $t+1$ points reveal no information.

9.5 Verifiable Secret Sharing (VSS)

Passive schemes break under malicious behavior. **VSS ensures shares are consistent even if the dealer is malicious**.

Key tools:

- **Commitments**: dealer commits to polynomial coefficients (binding and hiding).
- **Encrypted shares**: each $p(i)$ is encrypted under pk_i.
- **NIZK proofs**: public proofs that encrypted shares form a valid Shamir sharing.

During reconstruction:

- Parties broadcast shares.
- Reject inconsistent or invalid shares.
- Honest majority reconstructs the correct secret despite malicious participants.

VSS is the foundation for **active-secure MPC**.

9.6 Secure Multiparty Computation (MPC)

Goal:

$$y = f(x_1, \ldots, x_n)$$

such that correctness and privacy hold even against malicious parties.

9.6.1 Computation on shared values

Addition is local:

$$[a] + [b] = [a + b].$$

Multiplication uses Beaver triples: Shared values (u, v, w) with $w = uv$ are preprocessed. Enables multiplication with only one communication round.

9.6.2 Output reconstruction and fairness

Reconstruction requires all parties to reveal shares.

Fairness problem: A malicious party can learn the output first and then abort, preventing others from learning it.

Thus *MPC cannot guarantee fairness without additional assumptions.*

9.7 Fairness and Its Impossibility

A protocol is fair if:

- either all honest parties receive output,
- or none do.

Impossibility: In two-party settings, fairness cannot be achieved without:

- trusted third parties,
- financial penalties,
- time-locked mechanisms,
- or blockchain enforcement.

9.8 Fair Swaps via Blockchains

Blockchains provide:

- immutable logs,
- time-locks,
- enforceable penalties.

Two transactions:

- **Publish TX**: reveals data or share.
- **Refund TX**: time-locked; refunds deposit if publish does not occur.

Properties:

- Rational players cannot gain by aborting.
- Aborting results in monetary loss.
- Fairness becomes enforceable without secrecy on chain.

9.9 Fair MPC via Fair Swap

1. MPC produces additive or Shamir sharings of output y: each party holds y_i.
2. Parties exchange shares via blockchain-enforced fair swaps.

If a party aborts:

- honest parties receive compensation,
- no adversary gains advantage.

9.10 N-Party Ladder Fairness

Generalization of fair swaps to n parties.

Structure:

- Party P_i reveals a prefix $(w_1, \dots, w_i)$.
- Deposits scale with position: iB for P_i.
- Time-locks are staggered to prevent profitable aborts.

If P_k aborts:

- parties $1, \dots, k-1$ recover compensation,
- harm is limited to a predictable range.

Ensures honest behavior is the dominant strategy.

9.11 Publicly Verifiable Secret Sharing (PVSS)

PVSS strengthens VSS:

- Anyone (not only participants) can verify correctness of shares.
- Enables public randomness beacons and distributed key generation.

Mechanisms:

- Encryption of shares.
- Public zero-knowledge proofs of correct sharing.

9.12 Post-Quantum Security

Quantum computers threaten classical cryptography.

9.12.1 Shor's algorithm

Breaks:

- RSA,
- Diffie-Hellman,
- Elliptic-curve signatures (ECDSA, Schnorr).

9.12.2 Grover's algorithm

Quadratic speedup for brute force. Security level of n-bit symmetric keys drops to $n/2$ bits.

Mitigation:

- AES-256,
- doubling hash output lengths.

9.13 Post-Quantum Cryptography (PQC)

NIST-selected schemes:

- **CRYSTALS-Kyber** (encryption/key exchange),
- **CRYSTALS-Dilithium** (signatures),
- **Falcon** (compact signatures),
- **SPHINCS+** (hash-based signatures).

Most rely on lattice problems:

- Learning With Errors (LWE),
- Module-LWE,
- Shortest Vector Problem (SVP).

9.14 Impact on Blockchains

Quantum threats:

- ECDSA signatures become forgeable.
- Public keys exposed on chain may be harvested today and broken later.
- Grover's algorithm affects PoW difficulty.

Mitigations:

- Move to PQ signatures.
- Use hybrid classical–PQC schemes.
- Increase PoW hash output size.

9.15 Quick Summary

- Passive-secure MPC fails under malicious behavior.
- Active adversaries require VSS and PVSS for correctness.
- Fairness is impossible without additional assumptions.
- Blockchains allow enforcing fairness via deposits and time-locks.
- Ladder fairness extends fair exchange to arbitrary n.
- Quantum computers break classical public-key crypto.
- PQC offers quantum-safe alternatives for future blockchains.

10 Week 10

10.1 Finance and Financial Assets

Finance concerns the creation, management, and investment of money and financial assets.

Financial assets are non-physical assets whose value derives from contractual claims, such as:

- Bank deposits
- Stocks (equity securities)
- Bonds (debt securities)
- Loans

Financial services include lending, borrowing, issuing securities, and fund management. Financial markets are marketplaces where such assets are traded.

10.2 Decentralized Finance (DeFi)

Decentralized Finance (DeFi) refers to financial products and services built on decentralized blockchain infrastructure.

Key characteristics:

- No centralized intermediaries (banks, brokers, exchanges).
- Users retain direct control over their assets (non-custodial).

- Security depends on the underlying blockchain.

However, DeFi is exposed to new hazards due to the public and permissionless nature of blockchains.

10.3 Securities and Regulation

A security is a fungible, negotiable financial instrument with value, including:

- Equity securities (stocks)
- Debt securities (bonds)

Under the U.S. Howey Test, a security involves:

- Investment of money in a common enterprise.
- Expectation of profits derived from the efforts of others.

This distinction is critical when evaluating DeFi tokens, ICOs, and crowdfunding schemes.

10.4 Exchanges and Decentralized Exchanges (DEXs)

Exchanges facilitate trading between assets. Centralized exchanges maintain custody and order books off-chain.

Decentralized Exchanges (DEXs):

- Fully on-chain execution.
- Trades between native currency (e.g. ETH) and tokens (e.g. ERC20).
- Censorship-resistant but slower and more expensive due to blockchain fees.
- Typically lack KYC/AML.

10.5 Automated Market Makers (AMMs)

AMMs replace order books with liquidity pools.

For a trading pair A-B:

- Reserves: x units of A and y units of B.
- Invariant: $x \cdot y = k$.

Swaps change the ratio of reserves, implicitly determining price. Large trades cause slippage.

Liquidity providers are incentivized to keep pools large.

10.6 DEX Attacks and MEV

DEXs are vulnerable to transaction ordering attacks:

- Front-running: attackers pay higher gas fees to execute before victims.
- Sandwich (insertion) attacks: attacker trades before and after a victim to extract profit.

Miners and validators can exploit transaction ordering (MEV). Such attacks arise naturally from transparent mempools.

10.7 Loans and Decentralized Lending

Traditional loans involve risk assessment, interest, and potential default.

Decentralized loans rely on:

- Price oracles for asset valuation.
- Over-collateralization.
- Automatic liquidation when collateral value drops.

Lenders deposit capital into smart-contract vaults; borrowers lock collateral.

10.8 Flash Loans

Flash loans are loans issued and repaid within a single atomic transaction.

Properties:

- No default risk (transaction reverts if unpaid).
- Borrower can use large capital briefly.

They enable:

- Arbitrage between DEXs.
- Oracle manipulation attacks.
- Wash trading and market manipulation.

10.9 Market Capitalization

Market capitalization is defined as:

$$\text{Market Cap} = \text{circulating supply} \times \text{price}.$$

Issues:

- Price is determined by marginal trades on centralized exchanges.
- Market cap can be artificially inflated without new real-world money.

Key question:

- How much real-world value actually backs crypto market capitalization?

10.10 Stablecoins

Stablecoins aim to maintain stable value (typically \$1).

10.10.1 Fiat-backed Stablecoins

Characteristics:

- Centralized issuer.
- 1:1 promise with fiat held in escrow.

Risks:

- Fractional reserves.
- Opaque auditing.
- Regulatory intervention.

Example: Tether (USDT), which poses systemic risk due to scale and opacity.

10.10.2 Crypto-backed Stablecoins

Mechanism:

- Over-collateralization using crypto assets.
- Reliance on price oracles.

Problems:

- Leverage amplifies price bubbles.
- Liquidation cascades cause death spirals.

Example: MakerDAO required centralized intervention during market crashes.

10.10.3 Algorithmic Stablecoins

Design:

- No collateral.
- Supply adjusted algorithmically using bonds.

Failure reasons:

- Depend on market confidence.
- Collapse when demand evaporates.

No major algorithmic stablecoin has survived long-term.

10.11 Layer 2 Solutions

Layer 1 blockchains scale poorly due to:

- Limited throughput.
- High latency.

Layer 2 solutions:

- Move most activity off-chain.
- Rely on Layer 1 for dispute resolution and final settlement.

Trade-off:

- Improved scalability at the cost of modified trust assumptions.

10.12 Digital Economy on a Blockchain

Blockchains enable:

- Recording monetary transactions transparently.

- Algorithmic creation of money.

Issues:

- Tokens are often treated as speculative assets rather than currency.
- Valuation of on-chain economies is unclear.

10.13 Blockchain Applications

Across applications, blockchains excel at managing digital state but struggle with real-world integration.

Applications include:

- Name registration.
- Land ownership.
- Local economies.
- Supply chain tracking.
- Philanthropy.
- Crowdfunding.
- Prediction markets.
- IoT micropayments.
- Gaming and NFTs.

Common limitation:

- Oracles and off-chain trust reintroduce centralization.

10.14 Key Takeaways

- DeFi removes intermediaries but introduces new attack surfaces.
- Market metrics such as capitalization can be misleading.
- Stablecoins pose systemic financial risks.
- Layer 2 improves scalability but weakens guarantees.
- Blockchain applications succeed best when assets are purely digital.