THE GRAINGER COLLEGE OF ENGINEERING

CS 521

Technological Foundations of Blockchain and Cryptocurrency

Grigore Rosu

Topic 4 – Ethereum

ILLINOIS

Bitcoin’s Limitations – Why Something More?

- Lack of Turing-completeness
  - Script language supports no loops; simulating them requires duplicating code
- Value-blindness
  - UTXO is all-or-nothing; no fine-grained control over withdrawal amounts
- Lack of state
  - UTXO are either spent or unspent; no multi-stage contracts or persistent variables
- Blockchain-blindness
  - Scripts cannot access block data (nonce, timestamp, previous hash)

Ethereum

A next-generation smart contract
and decentralized application platform

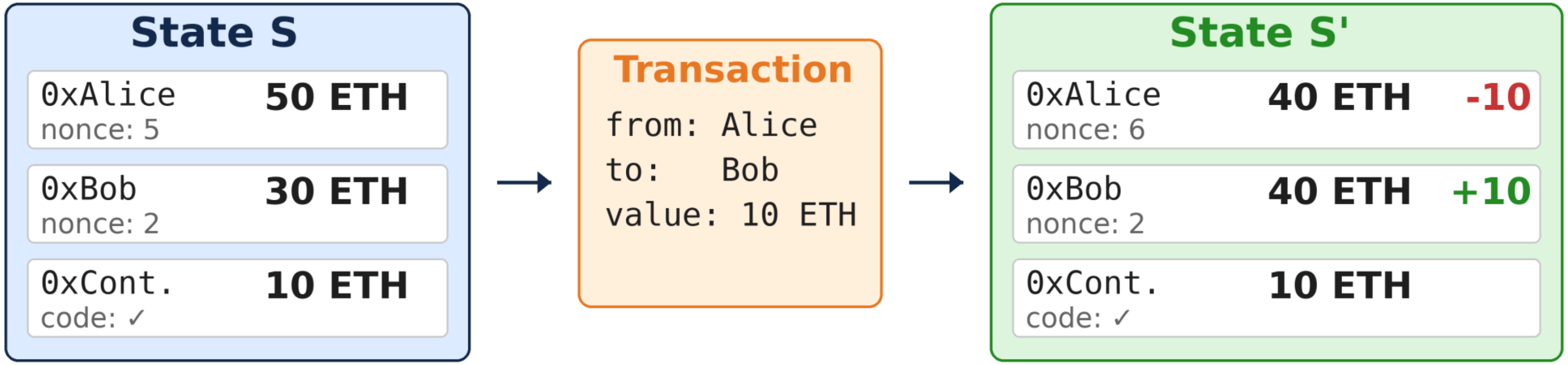
Ethereum: A Programmable Blockchain

- Proposed by Vitalik Buterin in late 2013; whitepaper published 2014
- Goal: a blockchain with a built-in Turing-complete programming language
- Anyone can write smart contracts and decentralized applications (dApps)
  - Users define their own logic in a few lines of code
- Create arbitrary rules for ownership, transaction formats, and state transitions
- Not just digital cash – a general-purpose decentralized world computer
  - Tokens, DeFi, NFTs, DAOs, identity, governance, and more
- Launched July 30, 2015 (Frontier release)

Ethereum as a State Transition System

- State: mapping from addresses (160-bit) to account states
  - Every account has: nonce, balance, storage root, code hash
- State transition function: APPLY(S, TX) → S' or ERROR
  - Validates signature, deducts gas fees, runs code, refunds unused gas
- Transactions morph the state incrementally, block by block
- Key difference from Bitcoin:
  - Bitcoin = UTXO model (stateless); Ethereum = account model (stateful)
- Ethereum blocks contain both the transaction list AND the most recent state (root hash)
- State stored in a Modified Merkle Patricia Trie
  - Allows efficient state lookups and proofs without storing full history

Ethereum State Transition Function



APPLY(S , TX) → S' or ERROR

Yellow Paper: $\sigma(t+1) \equiv Y(\sigma(t), T)$

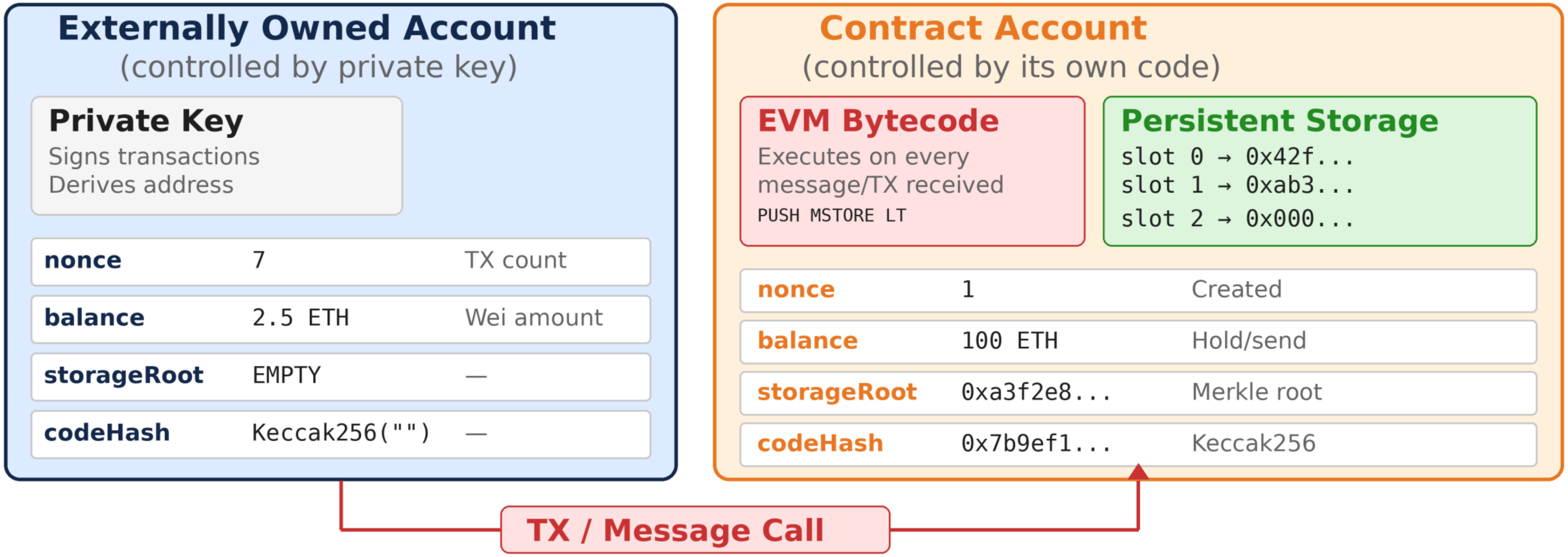
```
APPLY({ Alice: 50, Bob: 30 }, "send 70")
```

→ ERROR (insufficient balance: 50 < 70)

Ethereum Accounts

- Two types of accounts sharing the same 20-byte address space:
  - **Externally Owned Accounts (EOAs)**
    - Controlled by private keys (humans/wallets)
    - Can send transactions by creating and signing them; has no code
  - **Contract Accounts**
    - Controlled by their contract code; activates on every message received
    - Can read/write to internal storage, send messages, create contracts
- Every account has four fields:
  - nonce – transaction counter (prevents replay)
  - balance – amount of Wei (1 ETH = 10^{18} Wei)
  - storageRoot – hash of persistent key/value storage
  - codeHash – hash of EVM bytecode (empty for EOAs)

Ethereum Account Types



Both account types share 20-byte address space (160 bits)

EOA = Keccak-256(publicKey) [rightmost 20 bytes]

Contract = Keccak-256(RLP(sender, nonce)) [rightmost 20 bytes]

EOAs are controlled by private keys; Contract Accounts are controlled by code

Transactions and Messages

- Transaction: a signed data package from an EOA containing:
  - recipient – target address
  - signature – identifies the sender
  - value – amount of Ether to transfer
  - data – optional payload (e.g., function call + arguments)
  - gasLimit – max computational steps allowed
  - gasPrice – fee per computational step
- Messages: virtual objects sent between contracts (internal calls)
  - Like transactions but produced by contracts via CALL opcode
  - Never serialized; exist only in the EVM execution environment
- Gas allocation applies to the entire call chain:
  - If A sends TX with 1000 gas, B uses 600, calls C which uses 300, B has 100 left

The Gas Mechanism

- Problem: Turing-complete code can loop forever – how to prevent DoS?
- Solution: every computational step costs gas
  - An attacker must pay proportionally for computation, bandwidth, and storage
- How it works:
  - Sender pre-pays gasLimit × gasPrice in Ether
  - Each opcode consumes a defined amount of gas
  - Unused gas is refunded after execution
- Gas costs reflect resource consumption:
  - ADD = 3 gas, MUL = 5 gas, SSTORE = 20,000 gas (new slot)
  - Transaction data: 4 gas/zero byte, 16 gas/nonzero byte
- If execution runs out of gas:
  - All state changes revert, but gas is NOT refunded (prevents spam)

The Gas Mechanism

1. User Submits TX

`gasLimit = 100,000`

`gasPrice = 20 Gwei`

Pre-pay:

`100,000 × 20 Gwei`

= 0.002 ETH upfront

2. EVM Executes

```
ADD    → 3 gas
MUL    → 5 gas
SLOAD  → 2,100
SSTORE → 20,000
CALL   → varies
```

Total: 65,000 gas

3a. SUCCESS ✓

State changes COMMIT

Refund unused gas:

`35,000 × 20 Gwei`

= 0.0007 ETH back

3b. OUT OF GAS ✗

ALL state changes REVERT to original

Gas NOT refunded (prevents spam)

Why Gas Matters

- Turing-complete code loops forever
- Gas ensures every computation has cost
- Attacker pays for CPU, BW, storage
- Halting problem solved economically

Selected Gas Costs (EIP-2929+)

TX base:	21,000 gas	**SSTORE:**	22,100 gas	**CREATE:**	32,000 gas	**CALL:**	2,600 gas
ADD:	3 gas	**MUL:**	5 gas	**SLOAD:**	2,100 gas	**BALANCE:**	2,600 gas

Gas prevents infinite loops and DoS attacks by metering every computation

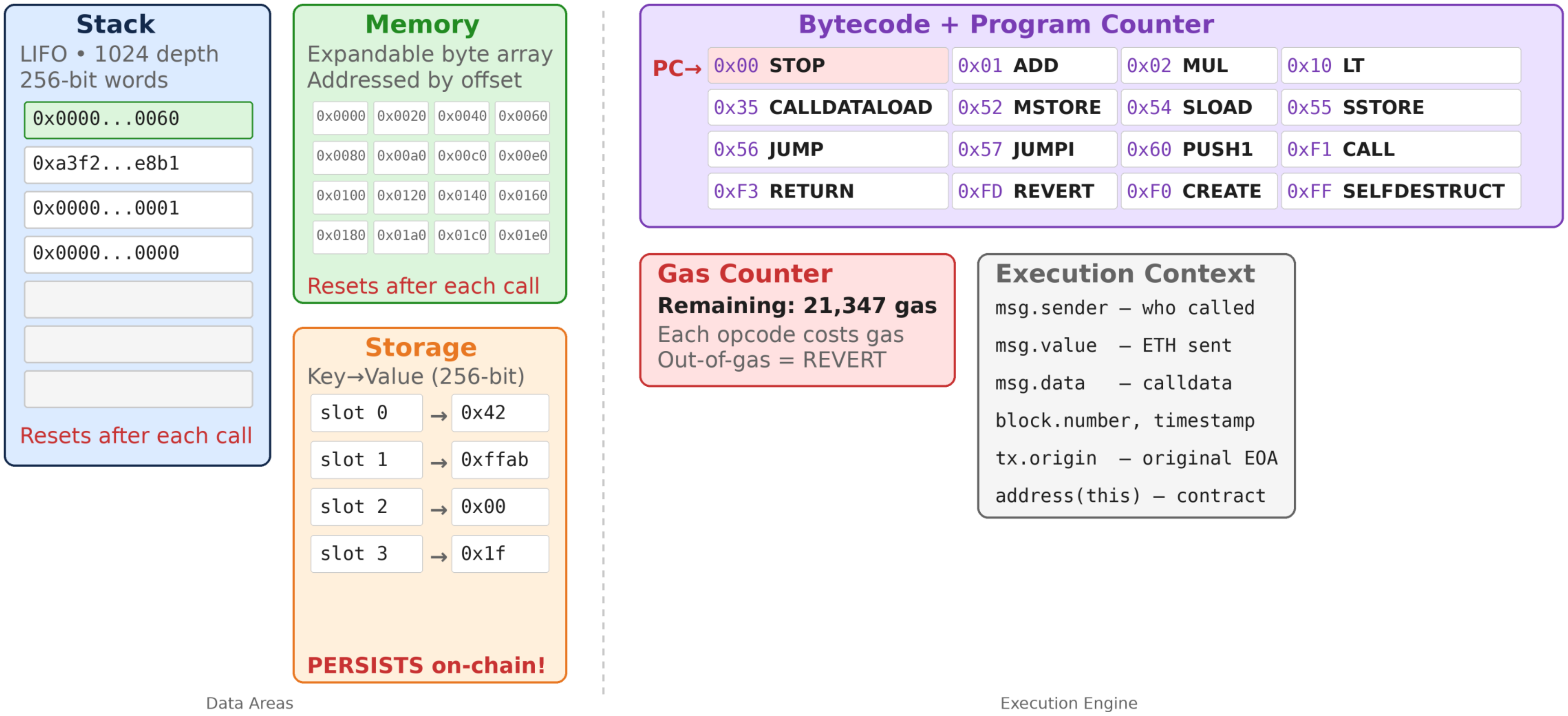
The Ethereum Virtual Machine

A Turing-complete, stack-based execution environment
running identically on every node

EVM Architecture

- Stack-based virtual machine with 1024-item depth, 256-bit words
- Three data storage areas:
  - Stack – LIFO container; push/pop 256-bit values; resets after execution
  - Memory – expandable byte array; resets after execution
  - Storage – persistent key/value store (256-bit → 256-bit); survives between calls
- Execution model:
  - tuple (block_state, transaction, message, code, memory, stack, pc, gas)
- Code = series of bytes; each byte is an opcode (ADD, SUB, SSTORE, CALL, ...)
  - Program counter increments through bytecode; STOP or RETURN halts
- Deterministic: given the same input, every node produces the same output
  - All nodes execute the same code – this IS the world computer

EVM: Stack, Memory, Storage



The EVM is a stack-based machine with three distinct data areas

Smart Contracts and Solidity

- Smart contracts:
  - Reactive agents, executing code when “poked” by a transaction or message
  - Have direct control over their own ETH balance and storage
- Solidity:
  - Primary high-level language for smart contracts; compiles to EVM
  - High-level, statically typed, object-oriented; influenced by C++, JavaScript, Python
- Development workflow:
  - Write Solidity → Compile to bytecode → Deploy → Interact
  - Tools: Remix (web IDE), Hardhat, Foundry
- Key properties:
  - Immutable once deployed (code cannot be changed)
  - Composable – contracts can call other contracts
- Other smart contract languages: Vyper (Python-like), Yul (low-level)

ERC-20: The Token Standard

- A token is just a mapping: address → balance
  - Subtract X from A, give X to B (if A has at least X)
- ERC-20 (2015) standardizes this into 6 functions + 2 events:
  - balanceOf(owner) → uint256
  - transfer(to, amount) — direct send
  - approve(spender, amount) — grant spending rights
  - transferFrom(from, to, amount) — spend on behalf
  - totalSupply() and allowance(owner, spender)
  - Events: Transfer(from, to, value), Approval(owner, spender, value)
- Why it matters:
  - Any ERC-20 token works with every wallet, DEX, and dApp
  - USDC, USDT, UNI, LINK, DAI — all ERC-20
  - Composability: one standard → entire DeFi ecosystem

SimpleToken.sol — Minimal ERC-20 Implementation

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract SimpleToken {
    string public name = "MyToken";
    string public symbol = "MTK";
    uint8  public decimals = 18;
    uint256 public totalSupply;

    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256))
        public allowance;

    event Transfer(address indexed from,
                   address indexed to, uint256 val);
    event Approval(address indexed owner,
                   address indexed spender, uint256);

    constructor(uint256 _supply) {
        totalSupply = _supply * 10**decimals;
        balanceOf[msg.sender] = totalSupply;
    }

    function transfer(address to, uint256 amt)
        public returns (bool) {
        require(balanceOf[msg.sender] >= amt);
        balanceOf[msg.sender] -= amt;
        balanceOf[to] += amt;
        emit Transfer(msg.sender, to, amt);
        return true;
    }
}
```

State variables:
name, symbol, decimals
are token metadata

The core state:
mapping(addr → bal)

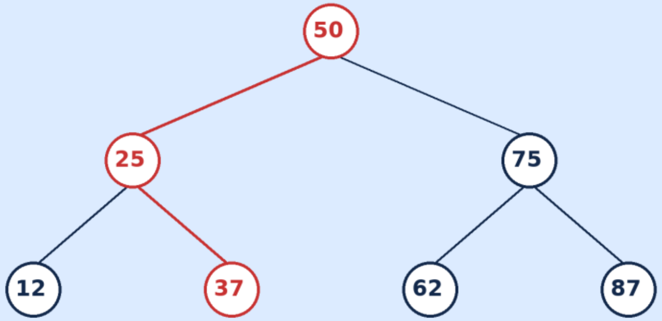
transfer() is the heart of ERC-20:
check → subtract → add
→ emit event

A minimal but complete ERC-20 token — the building block of DeFi

Tree vs. Trie

Binary Search Tree

Compare keys at each node
O(log n) lookup



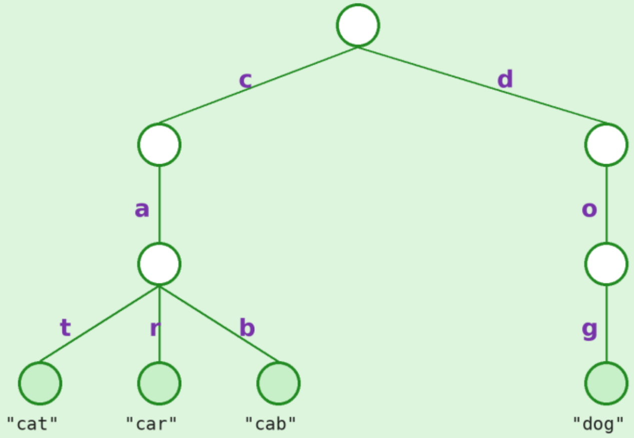
Find 37:
50 → go left
25 → go right
37 → found!

Properties:

- Full key stored in each node
- Compare & branch left/right
- Good for ranges, bad for strings
- O(log n) — depends on # entries
- Non-deterministic (insert order matters)

Trie (Prefix Tree)

Path from root encodes key
O(key length) lookup



Find "car":
root →c→ →a→ →r→ found!
Follow path edges, no comparisons
3 steps = length of key

Properties:

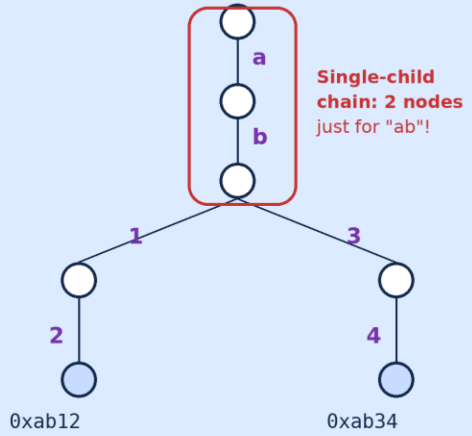
- Key encoded in the path (edges)
- Shared prefixes = shared paths
- Perfect for address lookups
- O(key length) — independent of # entries
- **Deterministic (same keys → same trie)**

Ethereum needs deterministic lookups by address — tries provide exactly this

Patricia Trie: Path Compression

Standard Trie (wasteful)

Single-child chains waste space



7 nodes for just 2 keys

Each nibble = one node, one edge

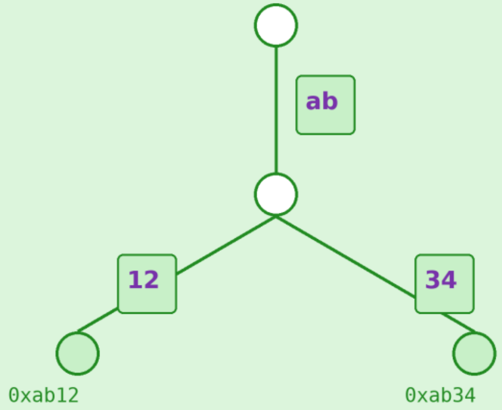
→ Wastes nodes where there is no branching

Why Ethereum needs this:

- Keys = Keccak-256 hashes, 40 nibbles (hex digits)
- Max path depth: 40 edges
- Patricia compresses non-branching segments
- Each internal node stores hash of its subtree
  → Merkle proof: O(log n) hashes to verify
- **Result: Modified Merkle Patricia Trie**
  = trie + path compression + Merkle hashing

Patricia Trie (compressed)

Collapse single-child chains



4 nodes for 2 keys!

"ab" collapsed into single edge

Branch only where keys diverge

Key insight:

Edges store multi-character strings, not single characters. This is what makes it "Patricia" (Practical Algorithm To Retrieve Information Coded In Alphanumeric)

Patricia = Practical Algorithm To Retrieve Information Coded In Alphanumeric

Modified Merkle Patricia Trie (MPT)

- Combines three ideas:
  - 1. Trie – deterministic key→value lookups
  - 2. Patricia – path compression, saves space
  - 3. Merkle – nodes store hash of children
- "Modified" = Ethereum's variant:
  - 16-way branching (one per nibble)
  - Node types: branch, extension, leaf, empty
  - Keys/values serialized before hashing
  - Root hash = Keccak-256 of entire trie
- Why blockchains need this:
  - Root hash in header commits to all data
  - O(log n) proof verifies any key-value pair
  - Deterministic: same data → same root hash

Branch Node

16 children (one per nibble 0–f)
+ optional value field

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f

↑ orange = nibbles with children

Extension Node

Shared prefix
(path compressed)
Points to next node

prefix:"a7b" → next

= Patricia compression!
Collapses single-child chains

Leaf Node

Remaining key suffix + value
End of a key's path in the trie

suffix:"c9"→[nonce,bal]

Merkle Hashing

Every node is hashed: hash = Keccak-256(serialize(node))
Node hash commits to all children's hashes at once
→ Change any leaf → all hashes up to root change

Root Hash

= fingerprint of entire trie
Stored in block header for consensus

Ethereum's Three Tries

- Each block header stores three trie root hashes:
  - Root hash = cryptographic commitment to data set
- State Trie – persistent across blocks, updated not rebuilt
  - Maps account address → account data (nonce, balance, …)
  - Only rehash changed leaf-to-root path (O(log n))
  - Merkle proofs: verify any account without full state
- Transaction Trie – rebuilt fresh each block
  - Maps tx index → transaction data for that block
  - Proves a transaction was included in a block
- Receipt Trie – rebuilt fresh each block
  - Maps tx index → execution result (status, gas, logs)
  - Proves tx outcome (e.g., event was emitted)
  - Separate from tx trie: receipts depend on execution context
- All three roots in every block header

Block Structure and Validation

- Ethereum blocks store more than Bitcoin:
  - Parent hash, timestamp, block number
  - State root, transaction root, receipts root (three MPTs!)
  - Gas limit, gas used, beneficiary (validator address)
- Block validation algorithm (simplified):
  - 1. Verify previous block reference is valid
  - 2. Check timestamp and other header constraints
  - 3. Execute each TX: S[i+1] = APPLY(S[i], TX[i])
  - 4. Check total gas ≤ gas limit
  - 5. Verify final state root matches header
- State root → any node can independently verify

Ethereum Block Structure

Block N

parentHash	0x1a2b...c3d4
number	18,000,000
timestamp	1695801600
beneficiary	0x9c0d...ef12
stateRoot	0x3c..e5f6
txRoot	0x5e..a7b8
receiptsRoot	0x7a..c9d0
gasLimit	30,000,000
gasUsed	15,234,567
baseFee	12 Gwei

Transaction List

TX_0, TX_1, TX_2 ...

State Trie

MPT of all account states.

Key = Keccak-256(address)

Value = [nonce, balance, storageRoot, codeHash]

Persistent; most subtrees shared,
only changed paths rehashed

Transaction Trie

MPT of all TXs in this block.

Key = TX index (0, 1, 2...)

Value = serialized TX

Rebuilt fresh each block.
Proves inclusion via O(log n) Merkle proof

Receipts Trie

Execution results per TX:

status, gas used, logs, bloom filter

Rebuilt fresh each block.
Separate: receipts depend on execution context

Three MPTs provide cryptographic commitments to state, transactions, and receipts

The Merge: From Proof-of-Work to Proof-of-Stake

- Ran Proof-of-Work from 2015 until The Merge (Sept 2022)
- Beacon Chain launched Dec 2020 – parallel PoS chain
  - Validators staked 32 ETH; tested PoS consensus independently
- The Merge: September 15, 2022
  - Execution layer merged with consensus layer (Beacon Chain)
  - Mining permanently ended; PoS validators now produce blocks
  - Triggered by Terminal Total Difficulty (difficulty bomb)
- Impact:
  - ~99.95% reduction in energy consumption
  - Block time stabilized at exactly 12 seconds
  - Issuance fell ~97%: PoS costs negligible → still profitable (~4-5% APY)
- No TX history lost – first PoS block attached to last PoW block

The Merge: PoW to PoS Timeline

Proof-of-Work Era

Mining • GPU/ASIC • ~93.95 TWh/year peak

Miners compete to find valid block hashes

Proof-of-Stake Era

Validators • 32 ETH stake • 99.95% less energy

Economically secured, not computationally

Impact of The Merge

Energy:	**~99.95%**	~93.95 → ~0.01 TWh/yr
Block time:	**Exactly 12 sec**	Steady, no PoW variance
Reward:	**−97%**	TX fee revenue increased
Validators:	**500,000+**	Lido, Coinbase, Rocket Pool
TX history:	**Fully preserved**	No data lost

How The Merge Worked

1. Beacon Chain launched Dec 2020
2. Ran parallel ~21 months testing
3. Validators staked 32 ETH
4. Terminal Total Difficulty reached
5. Execution layer swapped to PoS
6. Mining permanently discontinued

September 15, 2022: Ethereum completes the transition to Proof-of-Stake

Proof-of-Stake Mechanics

- Validators replace miners; must stake 32 ETH to participate
  - Over 500,000 validators active post-merge
- Every 12 seconds (one slot), the protocol:
  - Randomly selects a block proposer from the validator set
  - Assigns a committee of attesters (~1/32 of all validators)
  - Attesters vote on the proposed block's validity
- Slashing: penalties for malicious behavior
  - Double-signing or contradictory attestations → lose portion of staked ETH
  - Inactivity leak: offline validators gradually lose stake
- Finality: Casper FFG (Friendly Finality Gadget)
  - Checkpoints every 32 slots (1 epoch ~6.4 min); finalized after 2 epochs
- Shanghai upgrade (April 2023) finally enabled staking withdrawals

Ethereum Applications: DeFi, NFTs, DAOs

- DeFi (Decentralized Finance)
  - Automated Market Makers (Uniswap), lending (Aave, Compound)
  - Stablecoins (DAI, USDC) – hedging contracts from the whitepaper, realized
  - Over $100B total value locked at peak
- NFTs (Non-Fungible Tokens – ERC-721)
  - Unique digital ownership: art, collectibles, gaming items, real-world assets
  - Each token has a unique ID; ownership tracked on-chain
- DAOs (Decentralized Autonomous Organizations)
  - On-chain governance with token-weighted voting
  - Treasury management without traditional legal structures
  - The original whitepaper vision of "decentralized autonomous organizations"
- Other applications:
  - Identity systems (ENS), prediction markets, decentralized storage, layer-2 rollups

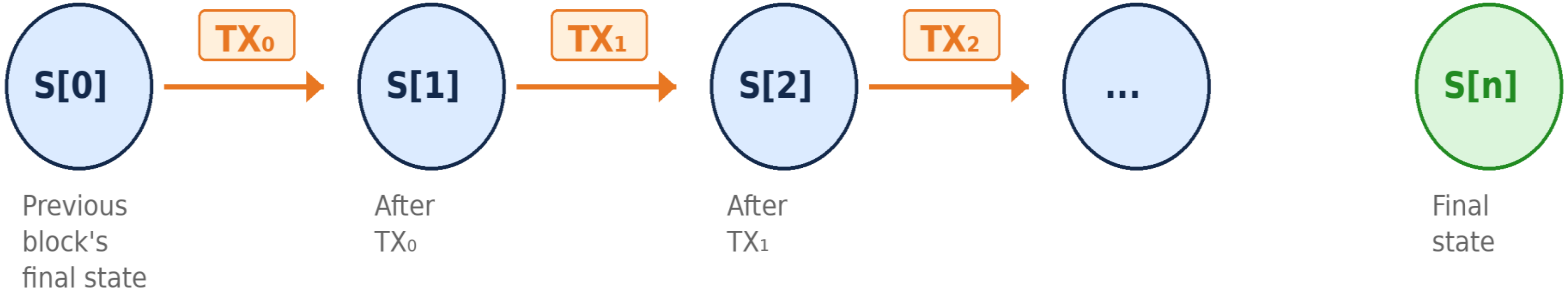
Ethereum Milestones

- Nov 2013 – Vitalik Buterin publishes the Ethereum Whitepaper
- Jul 2014 – Public crowdsale raises ~$18M (31,500 BTC)
- Jul 30, 2015 – Frontier launch (genesis block)
- Jun 2016 – The DAO hack ($60M); hard fork creates ETH/ETC split
- Dec 2017 – CryptoKitties congests the network; scaling debate intensifies
- Jan 2018 – ETH price peaks near $1,400
- Aug 2021 – EIP-1559 (London): base fee burn makes ETH deflationary
- Sep 15, 2022 – The Merge: PoW → PoS
- Apr 2023 – Shanghai: staking withdrawals enabled
- 2024–2025 – Dencun (proto-danksharding) and Pectra upgrades

KEVM: Formal Semantics of EVM (The Jello Paper)

- The Yellow Paper defines the EVM – but in informal mathematics
  - Ambiguities and inconsistencies discovered over the years
- KEVM: a complete, executable formal semantics of the EVM in the K Framework
  - Developed at UIUC and Runtime Verification
  - The "Jello Paper" – jellopaper.org – a readable presentation of the semantics
- What makes KEVM special:
  - Complete – passes the entire official Ethereum test suite
  - Executable – can run EVM programs directly
  - Formal – enables mathematical proofs about smart contract correctness
- Practical impact:
  - Formal verification of smart contracts (e.g., ERC-20 implementations)
  - Found real bugs in the Yellow Paper specification
  - Used by Runtime Verification for auditing high-value DeFi protocols

Block Processing: Sequential State Transitions



S[i+1] = APPLY(S[i] , TX[i])

S[0] = previous block's final state

If ANY returns ERROR → block INVALID

S[n] = final state → Merkle root in header

Block Validation:

1. Previous block exists and is valid
2. Timestamp in valid range
3. Total gas ≤ GASLIMIT
4. **MerkleRoot(S[n]) == stateRoot** ✓

Each transaction is applied sequentially; the final state root must match the header

Ethereum vs. Bitcoin: How It All Fits Together

- State model: UTXO (Bitcoin) vs. Accounts (Ethereum)
  - Stateless vs. stateful; simple transfers vs. complex interactions
- Scripting: limited stack-based (Bitcoin) vs. Turing-complete EVM (Ethereum)
- Consensus: PoW (Bitcoin) vs. PoS (Ethereum, since 2022)
- Block time: ~10 min (Bitcoin) vs. ~12 sec (Ethereum)
- Data in blocks: transactions only (Bitcoin) vs. transactions + state root (Ethereum)
- Supply: capped at 21M BTC vs. no hard cap, but deflationary since EIP-1559
- Purpose: peer-to-peer electronic cash vs. general-purpose world computer
  - Digital gold vs. decentralized application platform