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

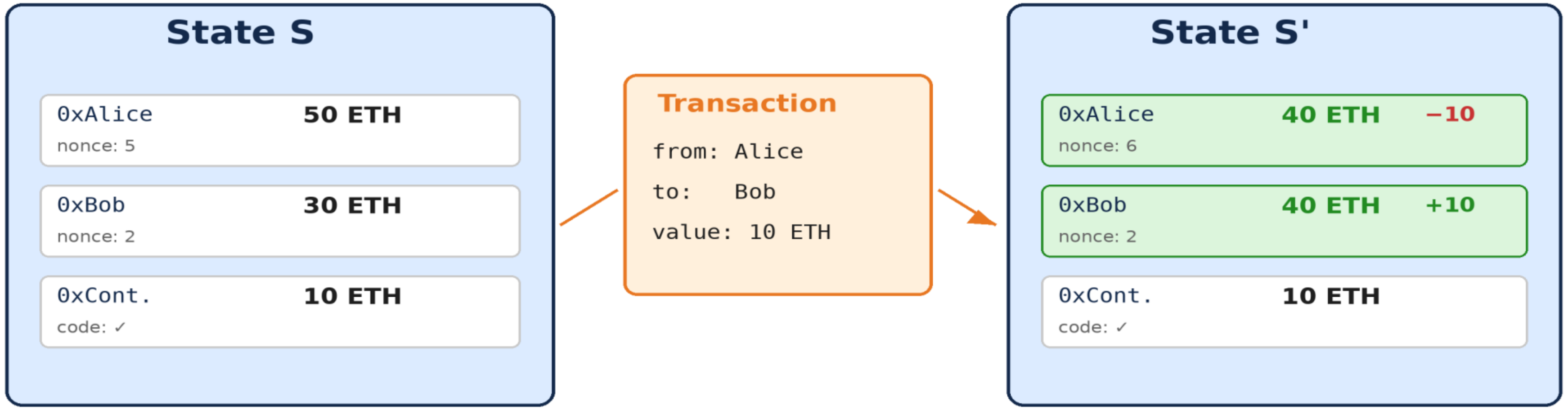
Ethereum: A Programmable Blockchain

- 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")
```

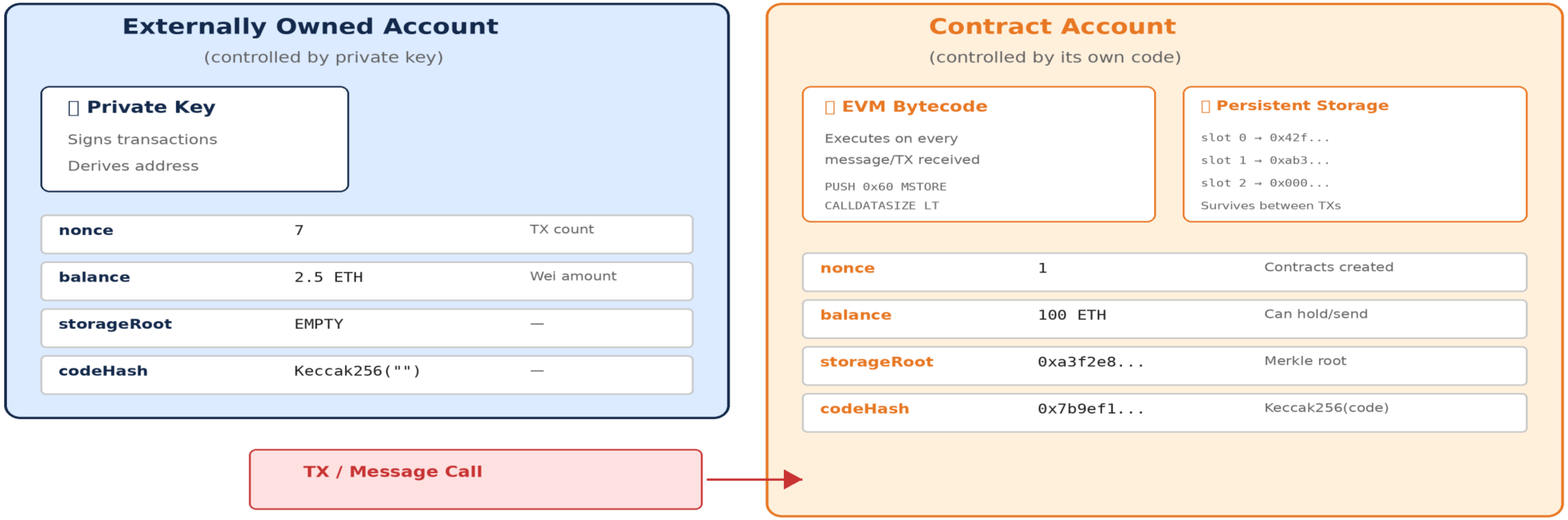
→ ERROR (insufficient balance: 50 < 70)

From the Ethereum Whitepaper: APPLY(S, TX) produces a new state S'

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



Externally Owned Account

(controlled by private key)

Private Key

Signs transactions

Derives address

nonce	7	TX count
balance	2.5 ETH	Wei amount
storageRoot	EMPTY	—
codeHash	Keccak256("")	—

TX / Message Call

Contract Account

(controlled by its own code)

EVM Bytecode

Executes on every message/TX received

```
PUSH 0x60 MSTORE
CALLDATASIZE LT
```

Persistent Storage

```
slot 0 → 0x42f...
slot 1 → 0xab3...
slot 2 → 0x000...
```

Survives between TXs

nonce	1	Contracts created
balance	100 ETH	Can hold/send
storageRoot	0xa3f2e8...	Merkle root
codeHash	0x7b9ef1...	Keccak256(code)

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

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

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

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

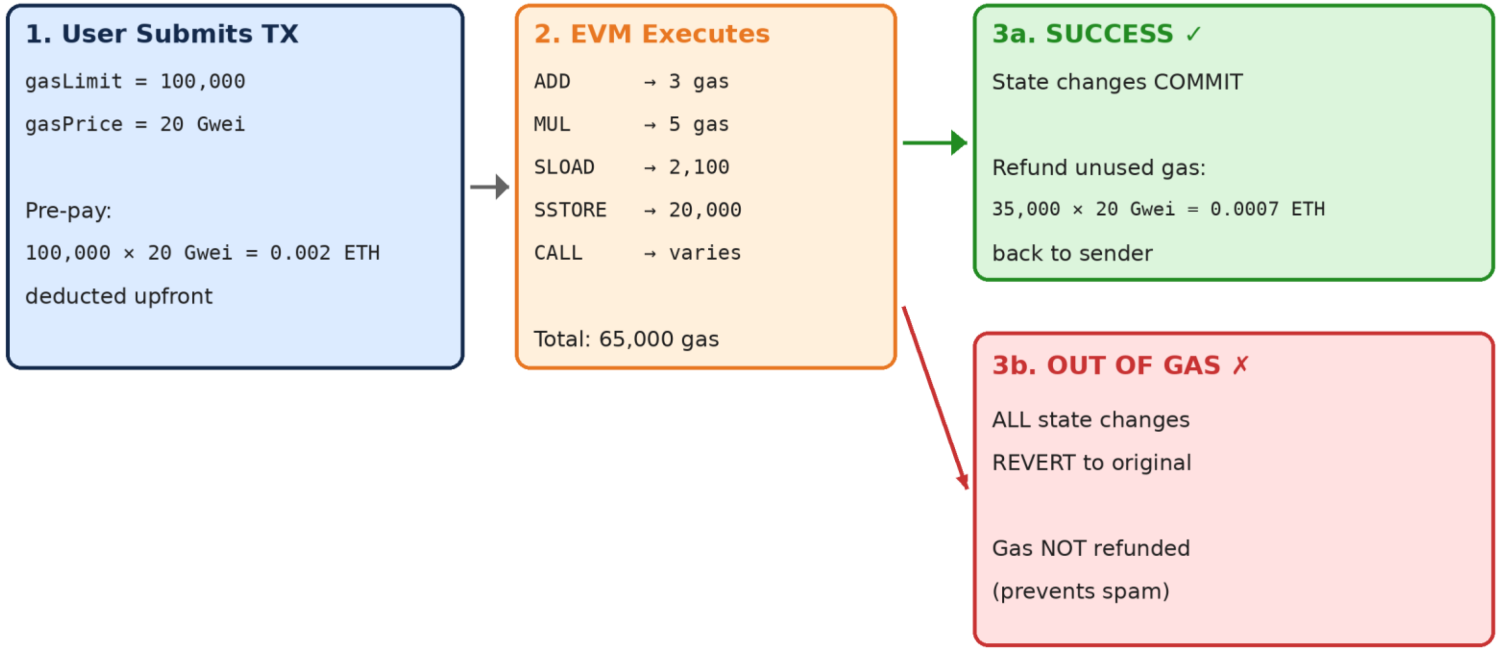
Transactions and Messages

- 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`

deducted 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 to sender

3b. OUT OF GAS ✗

ALL state changes REVERT to original

Gas NOT refunded (prevents spam)

Why Gas Matters

- Turing-complete code can loop forever
- Gas ensures every computation has a cost
- Attacker pays for CPU, bandwidth, storage
- Halting problem solved economically
- Validators compensated for execution work

Selected Gas Costs (EIP-2929+)

TX base fee:	21,000 gas	**SSTORE (new):**	22,100 gas	**CREATE:**	32,000 gas	**CALL (cold):**	2,600 gas
ADD / SUB:	3 gas	**MUL / DIV:**	5 gas	**SLOAD (cold):**	2,100 gas	**SLOAD (warm):**	100 gas
BALANCE:	2,600 gas	**LOG0:**	375 gas	**CALLDATACOPY:**	3+3/word gas	**EXTCODESIZE:**	2,600 gas

Gas prevents infinite loops and DoS attacks by metering every computation

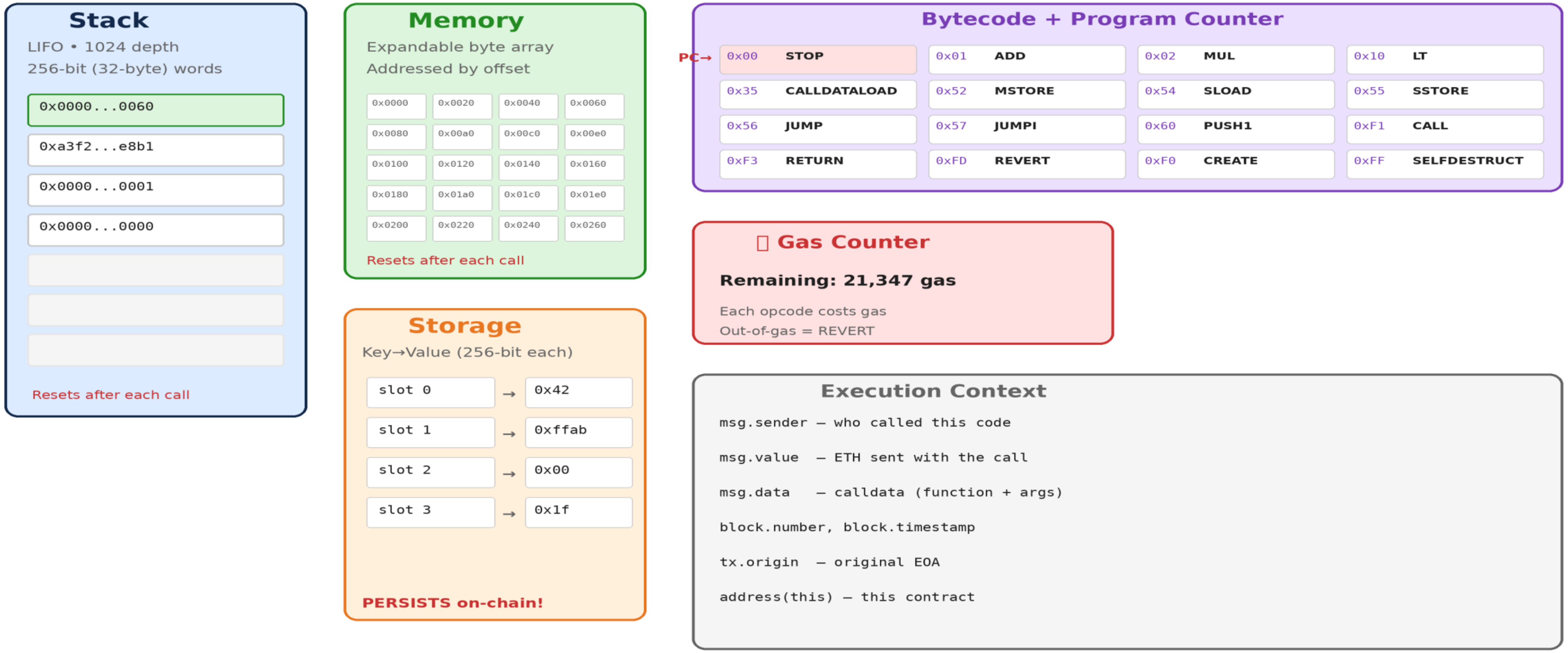
The Ethereum Virtual Machine

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



Stack

LIFO • 1024 depth
256-bit (32-byte) words

```
0x0000...0060
0xa3f2...e8b1
0x0000...0001
0x0000...0000
```

Resets after each call

Memory

Expandable byte array
Addressed by offset

0x0000	0x0020	0x0040	0x0060
0x0080	0x00a0	0x00c0	0x00e0
0x0100	0x0120	0x0140	0x0160
0x0180	0x01a0	0x01c0	0x01e0
0x0200	0x0220	0x0240	0x0260

Resets after each call

Storage

Key→Value (256-bit each)

slot 0	→	0x42
slot 1	→	0xffab
slot 2	→	0x00
slot 3	→	0x1f

PERSISTS on-chain!

Bytecode + Program Counter

PC→

0x00 **STOP**	0x01 **ADD**	0x02 **MUL**	0x10 **LT**
0x35 **CALLDATALOAD**	0x52 **MSTORE**	0x54 **SLOAD**	0x55 **SSTORE**
0x56 **JUMP**	0x57 **JUMPI**	0x60 **PUSH1**	0xF1 **CALL**
0xF3 **RETURN**	0xFD **REVERT**	0xF0 **CREATE**	0xFF **SELFDESTRUCT**

Gas Counter

Remaining: 21,347 gas

Each opcode costs gas
Out-of-gas = REVERT

Execution Context

```
msg.sender — who called this code
msg.value  — ETH sent with the call
msg.data   — calldata (function + args)
block.number, block.timestamp
tx.origin  — original EOA
address(this) — this contract
```

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

Smart Contracts and Solidity

- Smart contracts: autonomous agents living in the Ethereum execution environment
  - Execute code when “poked” by a transaction or message
  - Have direct control over their own Ether balance and storage
- Solidity: the primary high-level language for writing smart contracts
  - High-level, statically typed, object-oriented; influenced by C++, JavaScript, Python
  - Compiles to EVM bytecode
- Development workflow:
  - Write contract in Solidity → Compile to bytecode → Deploy via transaction → 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)

A Concrete Smart Contract: Token System

- All a token system fundamentally is: a database with one operation
  - Subtract X units from A and give X units to B, if A has at least X
- The basic logic (from Ethereum Whitepaper):

```
– def send(to, value):
–     if self.storage[msg.sender] >= value:
–         self.storage[msg.sender] -= value
–         self.storage[to] += value
```

- ERC-20 standard formalizes the interface:
  - totalSupply(), balanceOf(address), transfer(to, value)
  - approve(spender, value), transferFrom(from, to, value)
  - Any token following ERC-20 works with every wallet, DEX, and dApp
- This elegance is why Ethereum became the platform for thousands of tokens

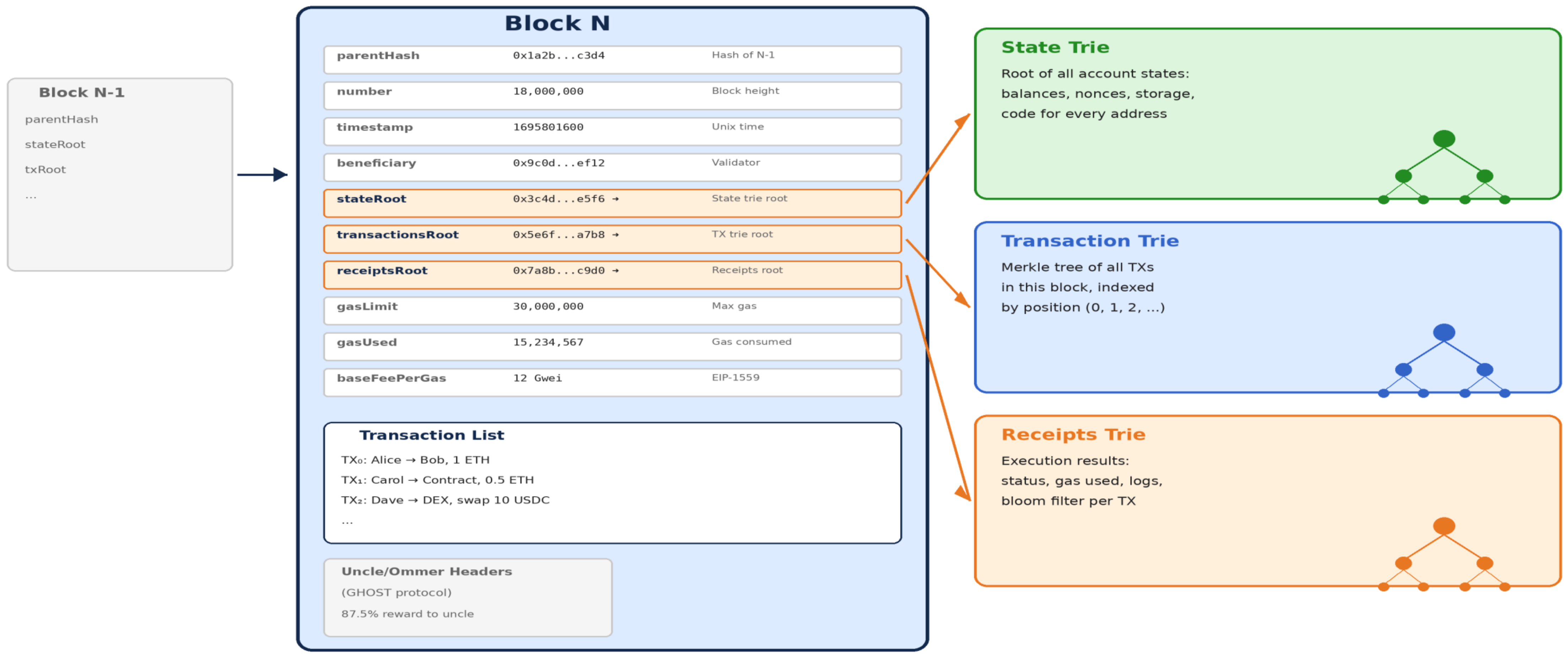
State Storage: Modified Merkle Patricia Trie

- Ethereum state is stored in a Patricia Trie (not just a Merkle tree)
  - Root hash in block header = cryptographic commitment to entire world state
- Why not a simple Merkle tree?
  - Need to insert and delete accounts, not just verify membership
  - Need efficient key-value lookups, not just ordered data
- Patricia Trie combines:
  - Radix trie (compact prefix tree) for efficient lookups by address
  - Merkle hashing for tamper-evidence and light client proofs
- Three tries in every block header:
  - State trie – all account states
  - Storage trie – each contract's persistent data
  - Transaction trie – all transactions in the block
- Efficiency: between adjacent blocks, most of the trie is identical

Block Structure and Validation

- Ethereum blocks store more than Bitcoin blocks:
  - Parent hash, timestamp, block number, difficulty
  - State root, transaction root, receipts root (three tries!)
  - Gas limit, gas used, beneficiary (validator/miner address)
- Block validation algorithm (simplified):
  - 1. Verify previous block exists and is valid
  - 2. Check timestamp constraints
  - 3. For each TX: S[i+1] = APPLY(S[i], TX[i]); check total gas ≤ GASLIMIT
  - 4. Verify final state root matches header
- GHOST Protocol (Greedy Heaviest Observed Subtree):
  - Includes “uncle” (stale) blocks in chain weight calculation
  - Uncle blocks receive 87.5% of base reward; nephew gets 12.5%
  - Reduces centralization bias from network latency

Ethereum Block Structure



parentHash	0x1a2b...c3d4	Hash of N-1
number	18,000,000	Block height
timestamp	1695801600	Unix time
beneficiary	0x9c0d...ef12	Validator
stateRoot	0x3c4d...e5f6 →	State trie root
transactionsRoot	0x5e6f...a7b8 →	TX trie root
receiptsRoot	0x7a8b...c9d0 →	Receipts root
gasLimit	30,000,000	Max gas
gasUsed	15,234,567	Gas consumed
baseFeePerGas	12 Gwei	EIP-1559

Transaction List

TX$_0$: Alice → Bob, 1 ETH
TX$_1$: Carol → Contract, 0.5 ETH
TX$_2$: Dave → DEX, swap 10 USDC
…

Uncle/Ommer Headers
(GHOST protocol)
87.5% reward to uncle

State Trie
Root of all account states: balances, nonces, storage, code for every address

Transaction Trie
Merkle tree of all TXs in this block, indexed by position (0, 1, 2, …)

Receipts Trie
Execution results: status, gas used, logs, bloom filter per TX

Three Merkle tries provide cryptographic commitments to state, transactions, and receipts

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

- Ethereum ran Proof-of-Work from launch (July 2015) until September 2022
- Beacon Chain launched December 1, 2020 – a parallel PoS chain
  - Ran in parallel; validators staked 32 ETH; tested PoS consensus independently
- The Merge: September 15, 2022
  - Execution layer (transactions) merged with consensus layer (Beacon Chain)
  - Mining permanently deprecated; PoS validators now produce blocks
  - Triggered by reaching Terminal Total Difficulty (difficulty bomb)
- Impact:
  - ~99.95% reduction in energy consumption
  - Block time stabilized at exactly 12 seconds
  - Total block reward income (USD) fell by ~97%
- No transaction history was lost – first PoS block attached to last PoW block

The Merge: PoW to PoS Timeline

Proof-of-Work Era

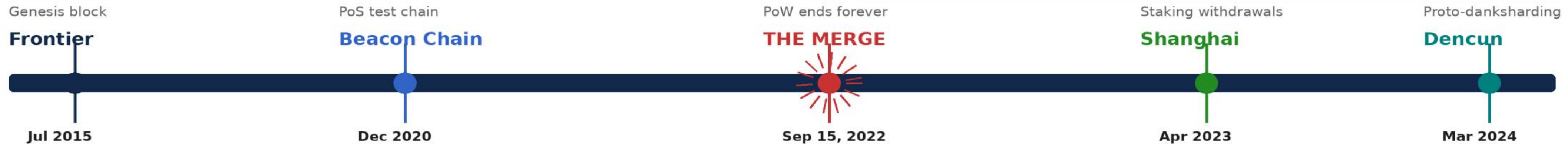
Mining • GPU/ASIC • ~93.95 TWh/year peak

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 reduction:	**~99.95%**	~93.95 → ~0.01 TWh/yr
Block time:	**Exactly 12 sec**	Steady, no PoW variance
Block reward:	**−97%**	TX fee revenue increased
Active validators:	**500,000+**	Lido, Coinbase, Rocket Pool
Herfindahl index:	**1009**	19% less concentrated
TX history:	**Fully preserved**	No data lost

How The Merge Worked

1. Beacon Chain launched Dec 2020 as separate PoS consensus chain
2. Ran in parallel with PoW mainnet for ~21 months of testing
3. Validators staked 32 ETH to join (~$50K at the time)
4. Terminal Total Difficulty reached: 58,750,000,000,000,000,000,000
5. Execution layer swapped consensus from PoW to Beacon Chain 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

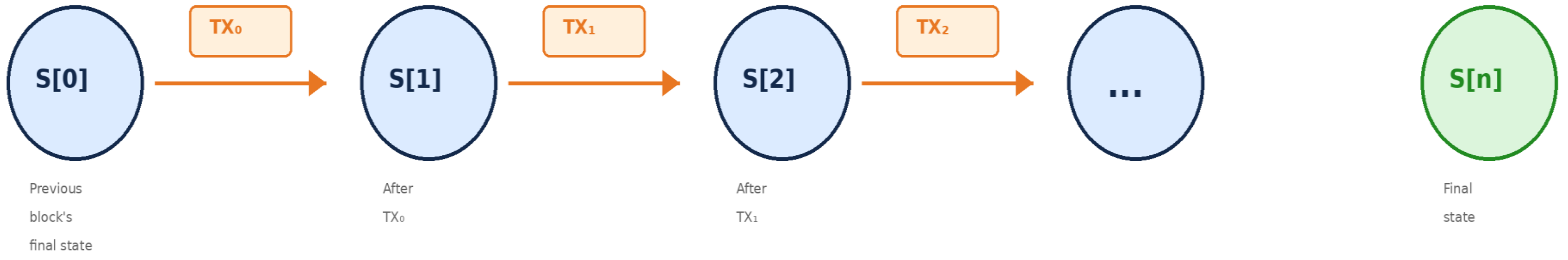
Ethereum Milestones

- 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
- KEVM paper published at CSF 2018 (IEEE Computer Security Foundations)

Block Processing: Sequential State Transitions



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

S[0] = state at end of previous block

If ANY returns ERROR → entire block is INVALID

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

Block Validation:

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

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
- Both are decentralized, permissionless, and secured by cryptographic primitives