

Bitcoin Protocol Deep Dive

BSc Blockchain, Crypto Economy & NFTs

Course Instructor

Module A: Blockchain Foundations

By the end of this lesson, you will be able to:

- Explain the UTXO (Unspent Transaction Output) model
- Describe the structure of a Bitcoin transaction
- Understand transaction inputs and outputs
- Recognize different Bitcoin Script types
- Trace the lifecycle of a transaction from creation to confirmation
- Distinguish between legacy and SegWit transaction formats

The UTXO Model: Core Concept

What is a UTXO?

- Unspent Transaction Output = a chunk of bitcoin that can be spent
- Bitcoin does not track account balances (unlike Ethereum)
- Instead, tracks individual “coins” (UTXOs)
- Your wallet balance = sum of all UTXOs you can spend

Analogy: Physical Cash

- UTXOs are like bills in your wallet
- You do not have “100 EUR balance” – you have five 20 EUR bills
- To pay 30 EUR: give one 20 EUR bill + one 10 EUR bill
- To pay 25 EUR with a 50 EUR bill: receive 25 EUR change

Key Principle:

- Each UTXO can only be spent once
- Spending a UTXO creates new UTXOs
- Blockchain tracks which UTXOs are unspent

UTXO Model vs. Account Model

UTXO Model (Bitcoin)

- No account balances
- Each transaction consumes old UTXOs, creates new ones
- Stateless verification
- Better privacy (new address per transaction)
- Parallel transaction processing

Example:

- Alice has UTXOs: 3 BTC, 2 BTC
- Sends 4 BTC to Bob
- Consumes both UTXOs (5 BTC total)
- Creates: 4 BTC to Bob, 1 BTC change to Alice

Why Bitcoin Uses UTXO:

- Easier to verify transaction validity (check UTXO existence)
- No global state required for validation
- Natural double-spend prevention

Account Model (Ethereum)

- Global account balances
- Transactions modify balances
- Stateful verification (need current balance)
- Simpler mental model
- Sequential nonces prevent replay

Example:

- Alice account: 5 ETH
- Sends 4 ETH to Bob
- Deduct from Alice: $5 - 4 = 1$ ETH
- Add to Bob: $0 + 4 = 4$ ETH

Bitcoin Transaction Structure

High-Level Components:

- ① **Version:** Protocol version number (currently 1 or 2)
- ② **Inputs:** List of UTXOs being spent
- ③ **Outputs:** List of new UTXOs being created
- ④ **Locktime:** Earliest block height or timestamp when transaction is valid (usually 0)

Transaction Hash (txid):

- Double SHA-256 of entire transaction
- Unique identifier (e.g., a1b2c3d4...)
- Used to reference transaction in inputs

Size and Fees:

- Transaction size measured in bytes (not kilobytes)
- Fee = size × fee rate (satoshis per byte)
- Miners prioritize higher fee-rate transactions
- Typical transaction: 200-400 bytes

Transaction Inputs

Each Input Contains:

- ① **Previous Transaction Hash:** txid of transaction containing UTXO
- ② **Output Index:** which output from previous transaction (0, 1, 2, ...)
- ③ **ScriptSig (Unlocking Script):** provides signature and public key
- ④ **Sequence Number:** originally for transaction replacement (now mostly unused)

Example Input:

- Previous tx: 5a3c7b... (Alice received 3 BTC)
- Output index: 0 (first output of that transaction)
- ScriptSig: signature proving Alice owns the UTXO

Multiple Inputs:

- Transaction can have many inputs
- Allows combining multiple UTXOs
- Each input must be signed separately
- Example: combine 1 BTC + 2 BTC + 1.5 BTC = 4.5 BTC total

Transaction Outputs

Each Output Contains:

- ① **Value:** Amount of satoshis (1 BTC = 100,000,000 satoshis)
- ② **ScriptPubKey (Locking Script):** conditions to spend this output

Example Output:

- Value: 4,000,000,000 satoshis (40 BTC)
- ScriptPubKey: "Pay to Bob's public key hash"

Change Outputs:

- When input value > payment amount, create change output
- Change goes back to sender (usually new address for privacy)
- Example: Spend 5 BTC UTXO to send 3 BTC -> create 3 BTC to recipient + 1.999 BTC change (0.001 BTC fee)

Transaction Fee:

- Fee = Sum of inputs - Sum of outputs
- Not explicitly stated in transaction
- Miner collects the difference

Bitcoin Script: A Stack-Based Language

What is Bitcoin Script?

- Simple, stack-based programming language
- Not Turing-complete (no loops, limited expressiveness)
- Executed during transaction validation
- Determines whether transaction is valid

How It Works:

- ① Combine ScriptSig (from input) + ScriptPubKey (from previous output)
- ② Execute script operations left to right
- ③ Use a stack (LIFO data structure)
- ④ Transaction valid if final stack value is TRUE

Basic Operations:

- OP_DUP: duplicate top stack item
- OP_HASH160: hash top stack item with SHA-256 then RIPEMD-160
- OP_EQUALVERIFY: check if top two items equal, fail if not
- OP_CHECKSIG: verify signature against public key

P2PKH: Pay-to-Public-Key-Hash

Most Common Transaction Type:

ScriptPubKey (Locking Script):

```
OP_DUP OP_HASH160 <PubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

ScriptSig (Unlocking Script):

```
<Signature> <PubKey>
```

Execution Steps:

- ① Push signature and public key onto stack
- ② OP_DUP: duplicate public key
- ③ OP_HASH160: hash one copy of public key
- ④ Compare hash with <PubKeyHash> from ScriptPubKey
- ⑤ OP_EQUALVERIFY: verify they match
- ⑥ OP_CHECKSIG: verify signature with remaining public key
- ⑦ Stack contains TRUE -*i* transaction valid

Why Use Public Key Hash Instead of Public Key Directly?

- Shorter addresses (20 bytes vs 33 bytes)
- Extra layer of security (quantum resistance)

P2SH: Pay-to-Script-Hash

Purpose:

- Allows complex spending conditions (multi-signature, time-locks, etc.)
- Hides complexity until spending time
- Sender only needs recipient's P2SH address

ScriptPubKey:

OP_HASH160 <ScriptHash> OP_EQUAL

ScriptSig:

<Signature1> <Signature2> ... <RedeemScript>

Verification Process:

- ① Hash the redeem script
- ② Verify hash matches ScriptHash in ScriptPubKey
- ③ Execute redeem script with provided signatures
- ④ Transaction valid if redeem script evaluates to TRUE

Example: 2-of-3 Multi-Signature:

- Redeem script requires 2 signatures out of 3 possible keys
- Spender provides 2 signatures + redeem script
- Network verifies both signatures are valid

Problem with Legacy Transactions:

- Signature data (witness) included in transaction hash
- Enables transaction malleability: signature can be modified without invalidating transaction
- Prevents secure second-layer solutions (Lightning Network)
- Wastes block space (signatures are large)

SegWit Solution (BIP 141, activated 2017):

- Separate signature data from transaction data
- Signature moved to separate “witness” field
- Transaction hash excludes witness (fixes malleability)
- Witness data discounted in block size calculation (enables more transactions per block)

Address Formats:

- P2WPKH (native SegWit): starts with “bc1q” (Bech32 encoding)
- P2SH-wrapped SegWit: starts with “3” (backward compatible)
- Example: bc1qw508d6qejaxtdg4y5r3zarvary0c5xw7kv8f3t4

SegWit Benefits

Block Capacity Increase:

- Legacy: 1 MB block size limit
- SegWit: measured in “weight units” (max 4 million)
- Witness data: 1 byte = 1 weight unit
- Non-witness data: 1 byte = 4 weight units
- Effective capacity: 2-2.7 MB per block (depending on transaction types)

Lower Transaction Fees:

- Witness data discounted by 75%
- Same transaction costs less with SegWit
- Incentivizes SegWit adoption

Enables Lightning Network:

- Fixes transaction malleability
- Allows secure off-chain payment channels
- Instant, low-fee micropayments

Script Versioning:

- Enables future upgrades (e.g., Taproot/SegWit v1)
- Soft fork compatibility

Key Improvements:

- **Schnorr Signatures:** replace ECDSA
 - More efficient (smaller signatures)
 - Enable signature aggregation
 - Better privacy (multi-sig looks like single-sig)
- **MAST (Merklized Abstract Syntax Trees):**
 - Complex scripts hidden until execution
 - Only reveal executed branch
 - Smaller transaction size for complex scripts
- **Privacy Enhancements:**
 - All transactions look similar on-chain
 - Multi-sig indistinguishable from single-sig
 - Complex smart contracts look like simple payments

Address Format:

- Starts with “bc1p” (Bech32m encoding)
- Example: bc1p5d7rjq7g6rdk2yhzks9smlaqtedr4dekq08ge8ztwac72sfr9rusxg3297

Transaction Lifecycle: Step by Step

1. Transaction Creation:

- Wallet selects UTXOs to spend
- Constructs inputs and outputs
- Calculates appropriate fee
- Creates change output if necessary

2. Transaction Signing:

- Wallet signs each input with corresponding private key
- Signature proves ownership of UTXOs
- Transaction now ready for broadcast

3. Broadcast to Network:

- Wallet sends transaction to connected Bitcoin nodes
- Nodes validate transaction (syntax, signatures, UTXO existence)
- Valid transaction added to mempool (memory pool)
- Nodes relay transaction to peers (propagation)

Transaction Lifecycle: Continued

4. Mempool:

- Holding area for unconfirmed transactions
- Each node maintains its own mempool
- Transactions sorted by fee rate
- Miners select transactions for next block from mempool

5. Mining and Confirmation:

- Miner includes transaction in candidate block
- Miner solves proof-of-work puzzle
- Block broadcast to network
- Nodes validate block and add to blockchain
- Transaction receives first confirmation

6. Additional Confirmations:

- Each new block adds one confirmation
- 6 confirmations typically considered final (1 hour)
- Transaction becomes increasingly irreversible
- UTXOs consumed by transaction are now spent
- New UTXOs created can now be spent by recipients

Transaction Validation Rules

Syntax Validation:

- Transaction size within limits
- Output values non-negative
- Output values do not exceed input values
- No duplicate inputs (double-spend within transaction)

Semantic Validation:

- All referenced UTXOs exist and are unspent
- Signatures valid for all inputs
- Script execution succeeds for all inputs
- Transaction fee is non-negative
- Locktime constraints satisfied

Contextual Validation:

- UTXOs not already spent in blockchain
- No conflicting transaction in mempool
- Sufficient fee for mempool acceptance

Rejection Reasons:

- Invalid signature - \downarrow likely fraud attempt
- Double-spend - \downarrow UTXO already spent
- Dust output - \downarrow output value too small (spam prevention)

Replace-by-Fee (RBF)

Problem:

- Transaction stuck in mempool with low fee
- Need to increase fee to speed up confirmation

RBF Solution (BIP 125):

- Create replacement transaction with same inputs
- Increase fee by at least 1 satoshi per byte
- Signal RBF by setting sequence number $\mid 0xffffffff$
- Nodes replace old transaction with new one in mempool

Use Cases:

- Fee bump: increase fee when network congested
- Output modification: change recipient or amount (before confirmation)
- Cancel transaction: send funds back to yourself with higher fee

Limitations:

- Only works for unconfirmed transactions
- Recipient should wait for confirmation before accepting payment
- Not all wallets support RBF

Child-Pays-for-Parent (CPFP)

Alternative Fee Bumping Method:

Scenario:

- Alice sends low-fee transaction to Bob
- Transaction stuck in mempool
- Bob wants faster confirmation

CPFP Mechanism:

- Bob creates new transaction spending Alice's unconfirmed output
- Bob's transaction has high fee
- Miners must include Alice's transaction to mine Bob's
- Combined fee rate makes both transactions attractive
- Miner includes both in same block

Comparison with RBF:

- RBF: sender bumps fee
- CPFP: receiver bumps fee
- RBF requires original transaction to signal support
- CPFP works for any transaction

Coinbase Transactions

Special Transaction Type:

Unique Properties:

- First transaction in every block
- No inputs (creates new bitcoins)
- Miner collects block reward + transaction fees
- Must wait 100 confirmations before spending (maturity rule)

Structure:

- Input: special coinbase input (previous tx hash = all zeros)
- ScriptSig: arbitrary data (often miner identification)
- Output: block reward + fees to miner address

Block Reward Schedule:

- Genesis block (2009): 50 BTC
- Halving every 210,000 blocks (4 years)
- Current (2024): 6.25 BTC
- Next halving (2024): 3.125 BTC
- Final halving (2140): block reward becomes zero
- Total supply: 21 million BTC

Transaction Fees: Economics

Fee Market Dynamics:

- Block space is scarce (limited to 4 MB weight per 10 minutes)
- Users compete for inclusion via fees
- Miners prioritize highest fee-rate transactions
- Fee rates fluctuate based on demand

Fee Estimation:

- Wallets estimate fee based on mempool state
- Target confirmation time: 1 block (high fee), 6 blocks (medium), 24 blocks (low)
- Fee estimation services: mempool.space, bitcoinfees.earn.com

Historical Fee Trends:

- Low congestion: \approx 1 sat/vbyte (\approx 0.10 USD per transaction)
- Moderate congestion: 10-50 sat/vbyte (1-5 USD)
- High congestion (bull market): 100-500 sat/vbyte (10-50 USD)
- Extreme congestion (2017, 2021): \gtrsim 1000 sat/vbyte (\gtrsim 50 USD)

Future: Layer 2 Solutions

- Lightning Network for micropayments
- Liquid Network for faster settlements
- Reduce on-chain transaction pressure

- Bitcoin uses the UTXO model: transactions consume old outputs and create new ones
- Each transaction has inputs (UTXOs being spent) and outputs (new UTXOs)
- Bitcoin Script enables flexible spending conditions without Turing completeness
- P2PKH (legacy), P2SH (multi-sig), SegWit, and Taproot offer increasing efficiency and privacy
- Transaction lifecycle: creation -> signing -> broadcast -> mempool -> mining -> confirmation
- Fees determined by market competition for block space
- SegWit and Taproot improve scalability and privacy

Design Philosophy:

Bitcoin prioritizes security and decentralization over transaction throughput. The UTXO model, simple scripting language, and conservative upgrade approach ensure long-term stability and auditability.

Discussion Questions

- ① Why does Bitcoin use the UTXO model instead of the account model like Ethereum?
- ② How does the fee market incentivize miners to include transactions in blocks?
- ③ What are the trade-offs between using legacy addresses, SegWit, and Taproot?
- ④ How does transaction malleability affect second-layer solutions like Lightning?
- ⑤ Why is the coinbase maturity rule (100 confirmations) necessary?
- ⑥ How could you design a transaction that can only be spent after a certain date?

Topics to be covered:

- Mining mechanics and the proof-of-work algorithm
- Nonce searching and difficulty adjustment
- Block header structure
- Mining difficulty and hash rate
- Block rewards and the halving schedule
- 51% attacks and mining centralization risks
- Energy consumption and environmental concerns

Preparation:

- Review hash function properties (pre-image resistance)
- Explore Bitcoin mining pools and hash rate distribution
- Consider the economics of mining profitability