#2 Ethereum Basics

Lecture Notes for CS190N: Blockchain Technologies and Security — October 1, 2025

In this lecture, we focus on how Ethereum extends the original idea of Bitcoin from digital money into a programmable world computer. We begin by revisiting Bitcoin's design to understand why its choices made it secure but limited. From there, we introduce Ethereum's account model and global state, explaining how these enable smart contracts and decentralized applications. We then study transactions in detail, looking at how they are structured, the different types that exist, and how gas ensures fair use of computational resources. By the end, you will see not only the technical mechanics behind Ethereum, but also how these mechanics make it possible to build applications that go beyond payments and shape today's blockchain ecosystem.

1 FROM DIGITAL CURRENCY TO A WORLD COMPUTER

Bitcoin was the first major breakthrough in decentralized money, and Ethereum expanded the vision of what a blockchain can achieve. In this section we look at how Bitcoin was designed, where its limitations lie, and how those limitations motivated the creation of Ethereum as a general purpose computing platform.

1.1 Bitcoin's Design Philosophy

Bitcoin, launched in 2009, showed that digital money could exist without central banks. Its goal was straightforward: enable peer to peer electronic cash. To achieve this, Bitcoin prioritized security, predictability, and resilience.

The system relied on three pillars. The UTXO model represented value as discrete coins. A restricted scripting language allowed basic transaction logic without enabling arbitrary programs. Proof of Work secured the network against attacks. These choices produced a robust currency system, but they also made Bitcoin narrow in scope.

1.2 The Limits of Programmability

The same features that make Bitcoin reliable also impose boundaries.

The UTXO model is efficient for value transfer but inherently stateless. Each transaction consumes specific outputs and creates new ones. While this supports parallel processing, it makes global rules difficult. For instance, setting a spending cap across multiple transactions is cumbersome because the system does not track balances directly.

The scripting language is intentionally minimal. It enables conditions such as multisignatures or time locks but excludes loops and persistent storage. This makes execution predictable but prevents the creation of complex applications.

In short, Bitcoin excels as money but cannot easily serve as a platform for broader computation.

1.3 Ethereum's Vision

Ethereum was created to go beyond these boundaries. In 2013 Vitalik Buterin proposed a blockchain that could host any program developers wanted, not just money transfers.

The key idea was a world computer: a shared environment where anyone could write and deploy programs that execute autonomously. Ethereum introduced the Ethereum Virtual Machine, a Turing complete engine that can run arbitrary code.

This shift enabled smart contracts, which are pieces of code that live on the blockchain and enforce their own rules. Smart contracts form the basis for decentralized applications ranging from finance and collectibles to governance. Ether, Ethereum's currency, plays a dual role as both money and gas, the resource required to pay for computation.

Ethereum reframed the blockchain from a specialized tool into a general purpose computing platform.

2 THE ETHEREUM WORLD STATE

To understand Ethereum in practice, we focus on its state. The state is the shared record of all accounts and contracts, and it evolves whenever transactions are executed. This section introduces the world state, compares Ethereum's account model with Bitcoin's UTXO system, and explains the structure of accounts.

2.1 State as a Global Machine

Ethereum can be viewed as a global state machine. At any moment the system maintains a snapshot of all accounts and their data. When a transaction is processed, the machine applies a state transition and produces a new snapshot. This model highlights that Ethereum is more than a ledger of payments. It is a programmable system whose central object is state.

2.2 Accounts vs. UTXO

Ethereum's designers replaced Bitcoin's UTXO model with an account model. The difference is much like coins in your pocket compared to a bank ledger. In Bitcoin your balance is the sum of many outputs. In Ethereum your balance is stored directly in your account record.

Table 1 summarizes the key differences and trade offs between these two models.

Feature	**UTXO (Bitcoin)**	**Accounts (Ethereum)**
State management	Stateless, validate outputs	Stateful, balances stored directly
Balance model	Derived from outputs	Explicit balance field
Privacy	Higher, frequent new addresses	Lower, address reuse is common
Programmability	Hard to express shared state	Natural fit for smart contracts
Scalability	Parallel verification possible	Conflicts require sequential handling

Table 1. Comparison of the UTXO model and the account model.

Ethereum chose the account model because it makes smart contracts practical, even if it reduces privacy and complicates scalability in some scenarios.

2.3 Two Types of Accounts

Within the account model Ethereum distinguishes between two roles. Externally Owned Accounts are controlled by users with private keys and are the only accounts that can initiate transactions. Contract Accounts are controlled by code and respond when triggered but cannot initiate actions themselves.

Table 2 highlights the main contrasts between these two account types.

EOAs provide initiative while contracts provide automation. Together they define who acts and who enforces logic in the Ethereum network.

2.4 Account Anatomy

Each account in Ethereum is defined by four fields:

- `nonce`: Counts transactions for EOAs or contract creations for contracts. Prevents replay attacks.
- `balance`: Ether holdings in Wei, where one Ether equals 10^{18} Wei.

Property	EOA	Contract Account
Control	Private key	Smart contract code
Who initiates	Can start transactions	Reacts only when called
Code	None	Immutable bytecode
Storage	None	Persistent storage
Creation	New keypair	Deployment transaction

Table 2. Comparison of Externally Owned Accounts and Contract Accounts.

- `codeHash`: Empty for EOAs, hash of code for contracts.
- `storageRoot`: Null for EOAs, root of persistent storage for contracts.

These fields are the foundation of Ethereum's state. Every observable change on chain corresponds to updates in one or more of them.

3 TRANSACTIONS

With the state model in place, the next question is how this state changes over time. The answer is transactions. Transactions are the gateway to the Ethereum world. Every balance transfer, contract deployment, and contract interaction begins with one. This section explains what transactions do, how they are structured, and the different types that exist.

3.1 What a Transaction Does

A transaction is a signed instruction from an externally owned account. It can transfer Ether, deploy a contract, or call an existing contract. During execution a transaction may trigger internal calls between contracts, but only the outer transaction is recorded on the blockchain. The distinction is important because it highlights that only user controlled accounts can initiate activity on the network.

3.2 Structure and Fields

Understanding transaction fields makes it easier to interpret block explorers and debug code.

A transaction includes:

- `nonce`: Ensures correct ordering and prevents replay.
- `to`: Recipient address, or empty for contract creation.
- `value`: Amount of Ether to transfer.
- `data`: Payload for contract calls or bytecode for deployment.
- `gasLimit`: Maximum gas that can be consumed.
- `maxFeePerGas` and `maxPriorityFeePerGas`: Define fee payments.
- `v, r, s`: Signature proving authorization.

Each field contributes to making transactions safe and predictable for the network.

3.3 Types of Transactions

Ethereum supports three types of transactions, and each modifies the state in a distinct way. Table 3 provides a structured overview, and the text below explains when to use each type and what to watch for.

Simple transfer. This transaction moves Ether between accounts. It reduces the sender's balance and increases the recipient's. Transfers to contracts may succeed or fail depending on whether the contract can accept Ether. This is the default way to send value without invoking code.

Contract call. Here the recipient is a contract account, and the transaction includes input data that specifies which function to run. The EVM executes the function, which may update storage or trigger further calls. Most application interactions use contract calls. Developers should size the gas limit with storage writes and external calls in mind.

Contract deployment. If the recipient field is empty, the data field carries the compiled bytecode of a new contract. When the transaction executes, a new contract account is created on chain. This is how new protocols and applications are launched. The derived contract address follows deterministic rules, so deployment planning may rely on predictable addresses.

Transaction type	**Effect on state**
Simple transfer	Moves Ether between accounts
Contract call	Executes contract code, may update storage and trigger calls
Contract deployment	Creates a new contract account with code and storage

Table 3. Three transaction types and their effects on Ethereum state.

Together these three types cover the lifecycle of on chain activity: moving funds, interacting with logic, and creating new logic.

4 GAS: PRICING COMPUTATION

Ethereum's power comes from running arbitrary programs, which raises the question of how to control resource usage. Without limits, programs could run forever and stall the network. The solution is gas. This section explains why gas exists, how it works, and how Ethereum's fee system supports sustainability.

4.1 Why Gas Exists

In computer science the halting problem shows that it is impossible to predict in general whether a program will finish. In a centralized system an administrator can stop runaway code, but in a decentralized system no such authority exists. Ethereum avoids this issue by attaching a cost to every operation. Computation continues only as long as the sender has purchased enough gas.

4.2 How Gas Works

Each EVM instruction consumes a fixed amount of gas. Reading memory is cheap while writing to storage is expensive. The sender specifies a `gasLimit` to cap how much they are willing to spend.

If execution finishes before the limit, unused gas is refunded. If the program runs out of gas, execution halts and all state changes are reverted, though the consumed gas is still paid. This mechanism ensures validators are compensated regardless of outcome and prevents denial of service attacks at no cost to the attacker.

4.3 The EIP 1559 Fee Market

Ethereum introduced a new fee system with EIP 1559. The total fee for a transaction is

$$\text{Fee} = \text{Gas Used} \times (\text{Base Fee} + \text{Priority Fee})\,.$$

The base fee is adjusted by the protocol based on congestion and is burned, which permanently reduces Ether supply. The priority fee is a tip given directly to validators to incentivize inclusion.

Table 4 summarizes the main components of this mechanism.

This system makes transaction fees more predictable and introduces mild deflationary pressure to Ether.

Component	Role in fee mechanism
Base Fee	Adjusts with congestion, burned to reduce supply
Priority Fee	Tip paid to validators for faster inclusion
Total Fee	Gas used multiplied by the sum of base and priority fees

Table 4. The components of the EIP 1559 fee mechanism.

KEY TAKEAWAYS

- Bitcoin proved that secure decentralized money is possible but left programmability limited.
- Ethereum reframed blockchains as general purpose computing platforms.
- The account model, smart contracts, and transaction types define how state evolves.
- Gas ensures open ended computation remains safe and economically sustainable.

REFERENCES