Recap

World state: set of accounts identified by 32-byte address.

Two types of accounts:

1. owned accounts (EOA): address = H(PK)
2. contracts: address = H(CreatorAddr, CreatorNonce)
Recap: Transactions

- **To:** 32-byte address  ($0 \rightarrow$ create new account)
- **From:** 32-byte address
- **Value:** # Wei being sent with Tx  ($1 \text{ Wei} = 10^{-18} \text{ ETH}, \quad 1 \text{ GWei} = 10^{-9} \text{ ETH}$)
- **Tx fees (EIP 1559):** gasLimit, maxFee, maxPriorityFee
- **data:** what contract function to call & arguments (calldata)
  - if To = 0: create new contract  \textbf{code} = (init, body)
- **[signature]:** if Tx initiated by an owned account (EOA)
Recap: Blocks

Validators collect Tx from users:

⇒ run Tx **sequentially** on current world state

⇒ new block contains **updated world state**, Tx list, log msgs
The Ethereum blockchain: abstractly

prev hash

updated world state

accts.

prev hash

updated world state

accts.

Tx

log messages

log messages

Tx
Write code in Solidity (or another front-end language)

⇒ compile to EVM bytecode

   (other projects use WASM or BPF bytecode)

⇒ validators use the EVM to execute contract bytecode in response to a Tx
The EVM
The EVM

see https://www.evm.codes

Stack machine (like Bitcoin) but with JUMP
• contract can create or call another contract ⇒ composability

Two types of zero initialized memory:
• **Persistent storage** (on blockchain): SLOAD, SSTORE (expensive)
• **Volatile memory** (for single Tx): MLOAD, MSTORE (cheap)
• LOG0(data): write data to log tree (not readable by EVM)
• Tx Calldata (16 gas/byte): readable by EVM in current Tx
  (near future: support for cheap 128KB blobs)
Every instruction costs gas

Why charge gas?

• Tx fees (gas) prevents submitting Tx that runs for many steps.
• During high load: block proposer chooses Tx from mempool that maximize its income.

\[
\text{if } \text{gasUsed} \geq \text{gasLimit}: \text{ block proposer keeps gas fees (from Tx originator)}
\]

\[
\text{calculated by EVM} \quad \text{specified in Tx}
\]
Gas prices spike during congestion

GasPrice in Gwei:

20 Gwei = 20 × 10^{-9} ETH

Average Tx fee in USD

popular project launch
Gas calculation: EIP1559

Every block has a “baseFee”: the minimum gasPrice for Tx in the block

**baseFee** is computed from total gas in earlier blocks:

- earlier blocks at gas limit (30M gas) $\implies$ base fee goes up 12.5%
- earlier blocks empty $\implies$ base fee decreases by 12.5%

If earlier blocks at “target size” (15M gas) $\implies$ baseFee does not change
Gas calculation

A transaction specifies three parameters:

- **gasLimit**: max total gas allowed for Tx
- **maxFee**: maximum allowed gas price
- **maxPriorityFee**: additional “tip” to be paid to block proposer

Computed **gasPrice** bid (in Wei = $10^{-18}$ ETH):

\[
gasPrice \leftarrow \min(\text{maxFee}, \text{baseFee} + \text{maxPriorityFee})
\]

Max Tx fee: \( \text{gasLimit} \times \text{gasPrice} \)
Gas calculation (informal)

\( \text{gasUsed} \leftarrow \text{gas used by Tx} \)

\[
\text{Send } \text{gasUsed} \times (\text{gasPrice} - \text{baseFee}) \text{ to block proposer}
\]

\[
\text{BURN } \text{gasUsed} \times \text{baseFee}
\]

\[\Rightarrow \text{ total supply of ETH can decrease}\]
Gas calculation

(1) if \( \text{gasPrice} < \text{baseFee} \): abort

(2) If \( \text{gasLimit} \times \text{gasPrice} > \text{msg.sender.balance} \): abort

(3) deduct \( \text{gasLimit} \times \text{gasPrice} \) from \( \text{msg.sender.balance} \)

(4) set \( \text{Gas} \leftarrow \text{gasLimit} \)

(5) execute Tx: deduct gas from \( \text{Gas} \) for each instruction
   if at end (\( \text{Gas} < 0 \)): abort, Tx is invalid (proposer keeps \( \text{gasLimit} \times \text{gasPrice} \))

(6) Refund \( \text{Gas} \times \text{gasPrice} \) to \( \text{msg.sender.balance} \) (leftover change)

(7) \( \text{gasUsed} \leftarrow \text{gasLimit} - \text{Gas} \)
   (7a) BURN \( \text{gasUsed} \times \text{baseFee} \)
   (7b) Send \( \text{gasUsed} \times (\text{gasPrice} - \text{baseFee}) \) to block producer
# Example baseFee and effect of burn

<table>
<thead>
<tr>
<th>block #</th>
<th>gasUsed</th>
<th>baseFee (Gwei)</th>
<th>ETH burned</th>
</tr>
</thead>
<tbody>
<tr>
<td>15763570</td>
<td>21,486,058</td>
<td>16.92</td>
<td>0.363</td>
</tr>
<tr>
<td>15763569</td>
<td><strong>14,609,185</strong> (&lt;15M)</td>
<td>16.97</td>
<td>0.248</td>
</tr>
<tr>
<td>15763568</td>
<td>25,239,720</td>
<td>15.64</td>
<td>0.394</td>
</tr>
<tr>
<td>15763567</td>
<td>29,976,215  (&gt;15M)</td>
<td>13.90</td>
<td>0.416</td>
</tr>
<tr>
<td>15763566</td>
<td><strong>14,926,172</strong> (&lt;15M)</td>
<td>13.91</td>
<td>0.207</td>
</tr>
<tr>
<td>15763565</td>
<td><strong>1,985,580</strong> (&lt;15M)</td>
<td>15.60</td>
<td>0.031</td>
</tr>
</tbody>
</table>

\[ \approx \text{gasUsed} \times \text{baseFee} \]

new issuance > burn \implies \text{ETH inflates}

new issuance < burn \implies \text{ETH deflates}
Eth total supply (since merge)
Why burn ETH ???

EIP1559 goals (informal):
• users incentivized to bid their true utility for posting Tx,
• block proposer incentivized to not create fake Tx, and
• disincentivize off chain agreements.

Suppose no burn (i.e., baseFee given to block producer):
⇒ in periods of low Tx volume proposer would try to increase volume by offering to refund the baseFee *off chain* to users.
Let’s look at the Ethereum blockchain etherscan.io:

<table>
<thead>
<tr>
<th>Latest Blocks</th>
<th>From/to address</th>
<th>Tx value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bk 15778674</td>
<td>Fee Recipient</td>
<td>0x39feb77c9f90fae6196... → 0x52de8d3f6db3a66d3c... 0.088265 Ether</td>
</tr>
<tr>
<td>7 secs ago</td>
<td>Lido: Execution Layer Rec...</td>
<td>◇ areyougay.eth → 0x404f5a67f72787a6d6b... 0.2 Ether</td>
</tr>
<tr>
<td>Bk 15778673</td>
<td>Fee Recipient</td>
<td>Optimism: State Root Pr... → Optimism: State Commit... 0 Ether</td>
</tr>
<tr>
<td>19 secs ago</td>
<td>Lido: Execution Layer Rec...</td>
<td>0xb3336d324ed828dbc8... → Uniswap V3: Router 2 0 Ether</td>
</tr>
<tr>
<td>Bk 15778672</td>
<td>Fee Recipient</td>
<td>0x1deaf9880c1180b02... → Uniswap V3: Router 2 0.14 Ether</td>
</tr>
<tr>
<td>31 secs ago</td>
<td>Flashbots: Builder</td>
<td>0x10c5a61426b506dcba... → Uniswap V2: Router 2 0 Ether</td>
</tr>
<tr>
<td>Bk 15778671</td>
<td>Fee Recipient</td>
<td>◇ defiantplatform.eth → 0x617dee16b86534a5d7... 0 Ether</td>
</tr>
<tr>
<td>43 secs ago</td>
<td>Lido: Execution Layer Rec...</td>
<td>34 txns in 12 secs</td>
</tr>
</tbody>
</table>
Let’s look at a transaction ...

Transaction ID: 0x14b1a03534ce3c460b022185b4 ...

From: 0x1deaf9880c1180b02307e940c1e8ef936e504b6a

To: Contract 0x68b3465833fb72a70ecdf485e0e4c7bd8665fc45 (Uniswap V3: Router 2)

Value: 0.14 Ether ($182)

Data: Function: multicall() [calls multiple methods in a single call]

Contract generated a call to Contract 0xC02aaA39b22 ... (value:0.14)
Let’s look at the To contract ...

Contract 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2
(Wrapped ETH: called from Uniswap V3: Router 2)

Balance: 4,133,236 Ether
Code: 81 lines of solidity

```solidity
function withdraw(uint wad) public {
    require(balanceOf[msg.sender] >= wad);
    balanceOf[msg.sender] -= wad;
    msg.sender.transfer(wad);
    Withdrawal(msg.sender, wad); // emit log event
}
```

anyone can read

code snippet
Remember: contracts cannot keep secrets!

Contract 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2

(Wrapped ETH)

Anyone can read contract state in storage array

⇒ never store secrets in contract!

Solidity variables stored in S[] array

etherscan.io
Solidity

docs:  https://docs.soliditylang.org/en/latest/

Several IDE’s available
interface IERC20 {
    function transfer(address _to, uint256 _value) external returns (bool);
    function totalSupply() external view returns (uint256);
    ...
}

contract ERC20 is IERC20 { // inheritance
    address owner;
    constructor() public { owner = msg.sender; }
    function transfer(address _to, uint256 _value) external returns (bool) {
        ...
        implementation ...
    }
}
Value types

- uint256
- address (bytes32)
  - `_address.balance`, `_address.send(value)`, `_address.transfer(value)`
  - call: send Tx to another contract
    ```
    bool success = _address.call{value: msg.value/2, gas: 1000}(args);
    ```
  - delegatecall: load code from another contract into current context
- bytes32
- bool
Reference types

- structs
- arrays
- bytes
- strings
- mappings:
  - Declaration: `mapping (address => unit256) balances;`
  - Assignment: `balances[addr] = value;`

```solidity
struct Person {
    uint128 age;
    uint128 balance;
    address addr;
}
Person[10] public people;
```
Globally available variables

- **block**: .blockhash, .coinbase, .gaslimit, .number, .timestamp
- **gasLeft()**
- **msg**: .data, .sender, .sig, .value
- **tx**: .gasprice, .origin
- **abi**: encode, encodePacked, encodeWithSelector, encodeWithSignature
- **Keccak256(), sha256(), sha3()**
- **require, assert**  e.g.: require(msg.value > 100, “insufficient funds sent”)
Function visibilities

- **external**: function can only be called from outside contract.
  
  Arguments read from calldata

- **public**: function can be called externally and internally.
  
  if called externally: arguments copied from calldata to memory

- **private**: only visible inside contract

- **internal**: only visible in this contract and contracts deriving from it

- **view**: only read storage (no writes to storage)

- **pure**: does not touch storage

```solidity
function f(uint a) private pure returns (uint b) { return a + 1; }
```
Inheritance

- Inheritance

  contract Destructable is **owned** {
    function destroy() public **onlyOwner** { selfdestruct(owner); }
  }

  code of contract “owned” is compiled into contract Destructable

- Libraries: library code is executed in the context of calling contract
  - library Search { function **IndexOf**(); }
  - contract A { function B { Search.IndexOf(); } }

```
contract owned {
  address owner;
  constructor() { owner = msg.sender; }
  modifier onlyOwner {
    require( msg.sender == owner); _; }
}
```
ERC20 tokens


- A standard API for fungible tokens that provides basic functionality to transfer tokens or allow the tokens to be spent by a third party.

- An ERC20 token is itself a smart contract that maintains all user balances:
  
  ```
  mapping(address => uint256) internal balances;
  ```

- A standard interface allows other contracts to interact with every ERC20 token. No need for special logic for each token.
ERC20 token interface

- function `transfer(address _to, uint256 _value) external returns (bool);`
- function `transferFrom(address _from, address _to, uint256 _value) external returns (bool);`
- function `approve(address _spender, uint256 _value) external returns (bool);`
- function `totalSupply() external view returns (uint256);`
- function `balanceOf(address _owner) external view returns (uint256);`
- function `allowance(address _owner, address _spender) external view returns (uint256);`
How are ERC20 tokens transferred?

contract ERC20 is IERC20 {
    mapping (address => uint256) internal balances;

    function transfer(address _to, uint256 _value) external returns (bool) {
        require(balances[msg.sender] >= _value, "ERC20_INSUFFICIENT_BALANCE");
        require(balances[_to] + _value >= balances[_to], "UINT256_OVERFLOW");
        balances[msg.sender] -= _value;
        balances[_to] += _value;
        emit Transfer(msg.sender, _to, _value); // write log message
        return true;
    }
}

Tokens can be minted by a special function mint(address _to, uint256 _value)
ABI encoding and decoding

- Every function has a 4 byte selector that is calculated as the first 4 bytes of the hash of the function signature.
  - For `transfer`, this looks like `bytes4(keccak256("transfer(address,uint256)"));`
- The function arguments are then ABI encoded into a single byte array and concatenated with the function selector.
  - This data is then sent to the address of the contract, which is able to decode the arguments and execute the code.
- Functions can also be implemented within the fallback function
Calling other contracts

- Addresses can be cast to contract types.
  
  ```solidity
  address _token;
  IERC20Token tokenContract = IERC20Token(_token);
  ERC20Token tokenContract = ERC20Token(_token);
  ```

- When calling a function on an external contract, Solidity will automatically handle ABI encoding, copying to memory, and copying return values.
  
  ```solidity
  tokenContract.transfer(_to, _value);
  ```
Stack variables

- Stack variables generally cost the least gas
  - can be used for any simple types (anything that is \( \leq 32 \) bytes).
    - `uint256 a = 123;`
  - All simple types are represented as `bytes32` at the EVM level.
- Only 16 stack variables can exist within a single scope.
Calldata

- Calldata is a read-only byte array.

- Every byte of a transaction’s calldata costs gas

  (16 gas per non-zero byte, 4 gas per zero byte).

- It is cheaper to load variables directly from calldata, rather than copying them to memory.
  - This can be accomplished by marking a function as `external`.
Memory (compiled to MSTORE, MLOAD)

- Memory is a byte array.
- Complex types (anything > 32 bytes such as structs, arrays, and strings) must be stored in memory or in storage.
  ```
  string memory name = “Alice”;
  ```
- Memory is cheap, but the cost of memory grows quadratically.
Storage array  (compiled to SSTORE, SLOAD)

- Using storage is very expensive and should be used sparingly.
- Writing to storage is most expensive.
  
  Reading from storage is cheaper, but still relatively expensive.

- mappings and state variables are always in storage.

- Some gas is refunded when storage is deleted or set to 0

- Trick for saving has: variables < 32 bytes can be packed into 32 byte slots.
Event logs

- Event logs are a cheap way of storing data that does not need to be accessed by any contracts.

- Events are stored in transaction receipts, rather than in storage.
Security considerations

- Are we checking math calculations for overflows and underflows?
  - done by the compiler since Solidity 0.8.
- What assertions should be made about function inputs, return values, and contract state?
- Who is allowed to call each function?
- Are we making any assumptions about the functionality of external contracts that are being called?
Re-entrency bugs
contract Bank{

    mapping(address=>uint) userBalances;

    function getUserBalance(address user) constant public returns(uint) {
        return userBalances[user];
    }

    function addToBalance() public payable {
        userBalances[msg.sender] = userBalances[msg.sender] + msg.value;
    }

    // user withdraws funds
    function withdrawBalance() public {
        uint amountToWithdraw = userBalances[msg.sender];

        // send funds to caller ... vulnerable!
        if (msg.sender.call{value:amountToWithdraw}() == false) { throw; }
        userBalances[msg.sender] = 0;
    }
}
contract Attacker {
    uint numIterations;
    Bank bank;

    function Attacker(address _bankAddress) {   // constructor
        bank = Bank(_bankAddress);
        numIterations = 10;
        if (bank{value:75}.addToBalance() == false) { throw; }   // Deposit 75 Wei
        if (bank.withdrawBalance() == false) { throw; }           // Trigger attack
    }
}

function () {   // the fallback function
    if (numIterations > 0) {
        numIterations --; // make sure Tx does not run out of gas
        if (bank.withdrawBalance() == false) { throw; }
    }
}  }  }  }
Why is this an attack?

(1) Attacker $\rightarrow$ Bank.addToBalance(75)

(2) Attacker $\rightarrow$ Bank.withdrawBalance $\rightarrow$

   Attacker.fallback $\rightarrow$ Bank.withdrawBalance $\rightarrow$

   Attacker.fallback $\rightarrow$ Bank.withdrawBalance $\rightarrow$ ...

withdraw 75 Wei at each recursive step
How to fix?

```solidity
function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];

    userBalances[msg.sender] = 0;
    if (msg.sender.call{value:amountToWithdraw}() == false) {
        userBalances[msg.sender] = amountToWithdraw;
        throw;
    }
}
```
Next lecture:  DeFi contracts