EVM (part 2)

Slides from Dan Boneh

Recap: Ethereum consensus

consensus layer (beacon chain)

Today: EVM

compute layer (execution chain): The EVM

consensus layer (beacon chain)

Ethereum compute layer: the EVM

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

Two types of accounts:

(1) externally owned accounts (EOA):

controlled by ECDSA signing key pair (pk,sk).

sk: signing key known only to account owner

(2) contracts: controlled by code.

code set at account creation time, does not change

Data associated with an account

Account data	Owned (EOA)	Contracts	
address (computed):	H(pk)	H(CreatorAddr, CreatorNonce)	
code:	$\perp$	CodeHash	
storage root (state):	$\perp$	StorageRoot	
balance (in Wei):	balance	balance	(1 Wei = 10^{-18} ETH)
nonce:	nonce	nonce	

(#Tx sent) + (#accounts created): anti-replay mechanism

State transitions: Tx and messages

Transactions: signed data by initiator

- **To:** 32-byte address of target (0 → create new account)
- **From**, **[Signature]**: initiator address and signature on Tx (if owned)
- **Value**: # Wei being sent with Tx (1 Wei = 10^{-18} ETH)
- Tx fees (EIP 1559): **gasLimit, maxFee, maxPriorityFee** (later)
- if To = 0: create new contract **code = (init, body)**
- if To ≠ 0: **data** (what function to call & arguments)
- **nonce**: must match current nonce of sender (prevents Tx replay)
- chain_id: ensures Tx can only be submitted to the intended chain

State transitions: Tx and messages

Transaction types:

owned → owned: transfer ETH between users

owned → contract: call contract with ETH & data

Transaction Hash	Method	Block	Age	From	To	Amount	Txn Fee
0x427c31b0a2...	Transfer	22334348	15 secs ago	beaverbuild	0x123B9747...C68A0A806	0.010695384 ETH	0.00000883
0xe208fcc8c90...	Transfer	22334348	15 secs ago	0x9Bd017bc...3f27E661e	0x571AD708...73fa8766C	0.00010801 ETH	0.00000885
0x1def593585c...	Transfer	22334348	15 secs ago	0x906457C3...D2f990e1E	0xeEF5599e...28121eD24	0.04994585 ETH	0.00000885
0x2e25b793a1...	Transfer	22334348	15 secs ago	0x016606Ac...44B4049Ee	0xC1BEd8E8...835EF8B09	0.182092191 ETH	0.00000886
0x838bd63745...	Pre Collaterali...	22334348	15 secs ago	0x57Dd832f...D8818bF4d	0x3A4F41da...71bF4Cc59	0 ETH	0.00003483
0xf60eb42b563...	Transfer	22334348	15 secs ago	0x68B0F72D...61E84f7f5	0x3CAcB76c...C6f71cFB1	0.009995342 ETH	0.00000898
0x1cea36b658...	Transfer	22334348	15 secs ago	0xf87BDC2b...0Dde8AF6b	0x3CAcB76c...C6f71cFB1	0.009988976 ETH	0.00000898
0xc9c9b3b7aa...	Transfer	22334348	15 secs ago	0x1242f107...39E37822f	0x3CAcB76c...C6f71cFB1	0.052833105 ETH	0.00000898
0x8b1a4a52d8...	Swap	22334348	15 secs ago	0xa84B9E91...b4187Cc08	Aggregation Router V6	0.03 ETH	0.00008007
0x20ba850ab8...	Transfer	22334348	15 secs ago	0xEDCa7976...A8C8503c1	0xf066CEd2...A3C9b977F	0.024854505 ETH	0.00000903
0x3cece221e2...	Transfer	22334348	15 secs ago	0x3CAcB76c...C6f71cFB1	0xDdEc41b0...0a08fE03c	0.01 ETH	0.00000904
0x8eaa3695daf...	Transfer	22334348	15 secs ago	Cobo: Custody	0x0D65fEE7...F4822b078	0.4495 ETH	0.00000905
0xbd1d8e3b8df...	Transfer	22334348	15 secs ago	0x97Ca787E...0Ec2DF5b9	Tether: USDT Stablecoin	0 ETH	0.00001767
0x256605a075...	Transfer	22334348	15 secs ago	0x681fF4BF...8C9017Bd9	Tether: USDT Stablecoin	0 ETH	0.00001775
0xf3b21d58ad7...	Transfer	22334348	15 secs ago	0x9320f4d9...B153B82a5	Circle: USDC Token	0 ETH	0.00001945
0x7cc7874007...	0x59805e69	22334348	15 secs ago	0xE0b7DEab...41D058C79	0xE54E03fC...0D8CB2dAf	0 ETH	0.00002283

0x1242f107bf3412a893b5f47b9b1a7b539e37822f

Messages: virtual Tx initiated by a contract

Same as Tx, but no signature (contract has no signing key)

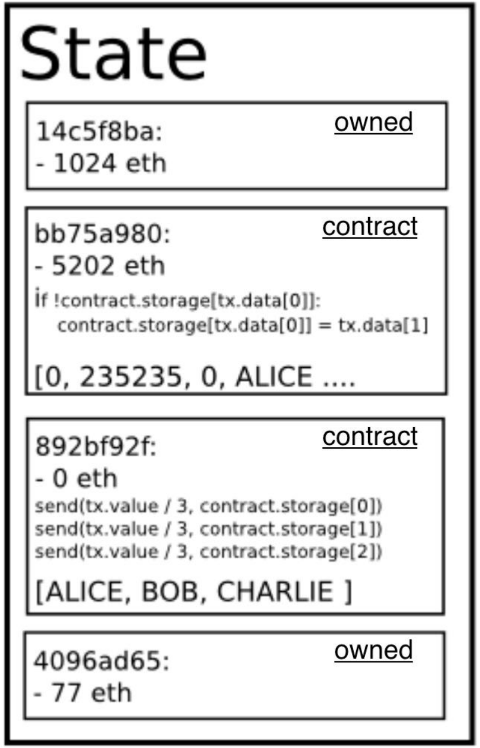
contract → EOA: contract sends funds to user

contract → contract: one program calls another (and sends funds)

One Tx from user: can lead to many Tx processed. Composability!

Tx from owned addr → contract → another contract
Tx from owned addr → contract → another contract → EOA

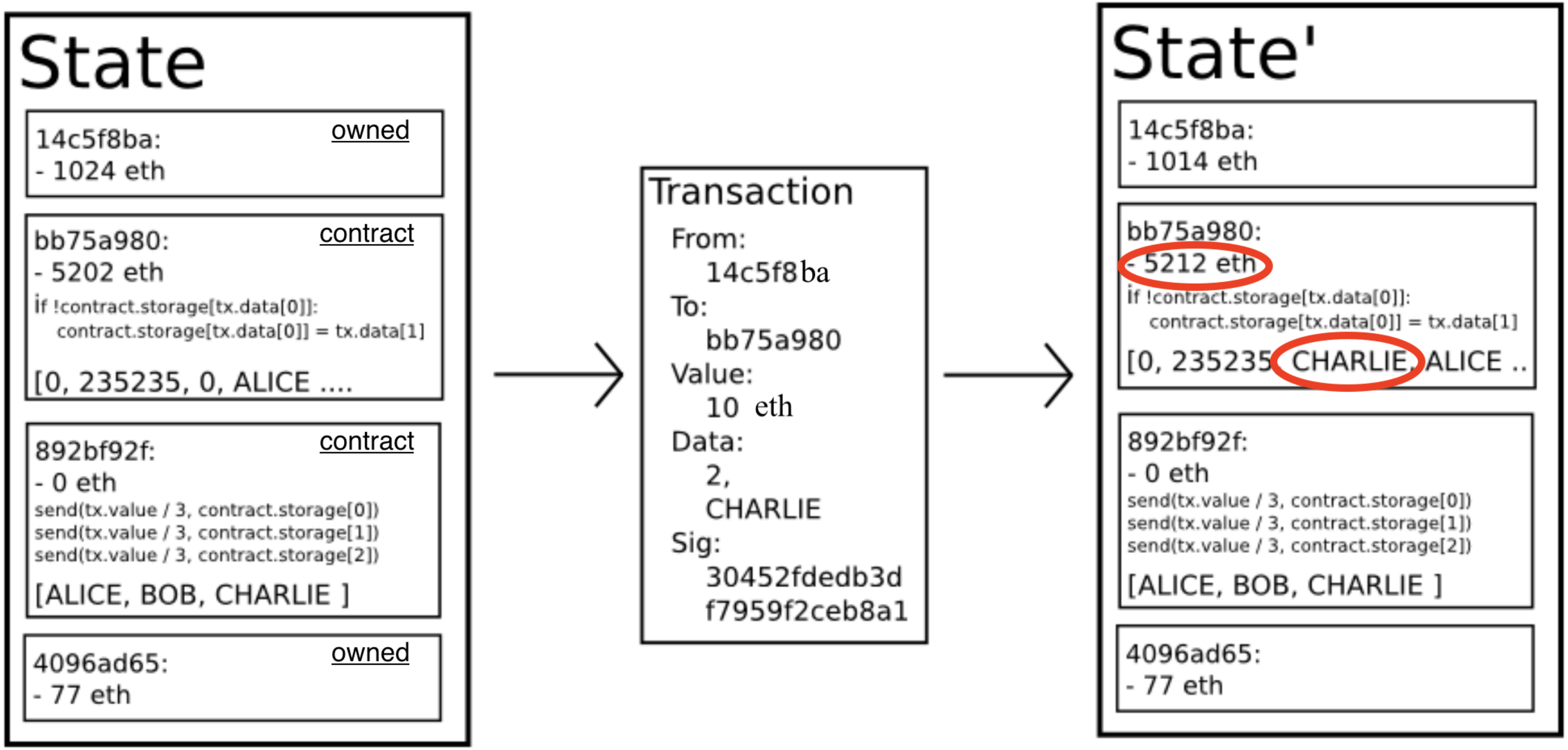
Example Tx



world state (four accounts)

updated world state

Example Tx



world state (four accounts) updated world state

An Ethereum Block

Block proposer creates a block of n Tx: (from Txs submitted by users)

- To produce a block do:
  - for i=1,…,n: execute state change of Tx_i sequentially (can change state of >n accounts)
  - record updated world state in block

Other validators re-execute all Tx to verify block $\Rightarrow$ sign block if valid $\Rightarrow$ enough sigs, epoch is finalized.

Block header data (simplified)

(1) consensus data: proposer ID, parent hash, votes, etc.

(2) address of gas beneficiary: where Tx fees will go

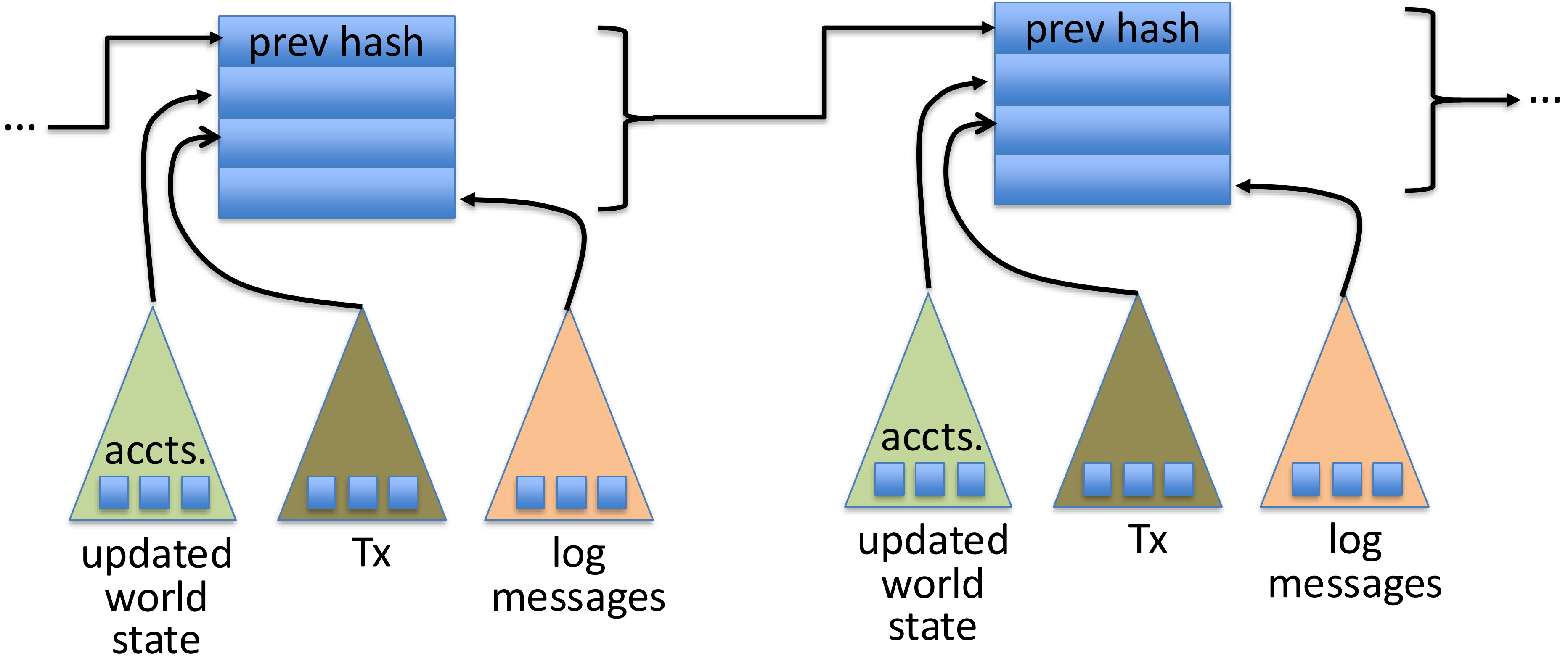
(3) world state root: Merkle hash of <u>all</u> accounts in the system

(4) **Tx root**: Merkle hash of all Tx processed in block

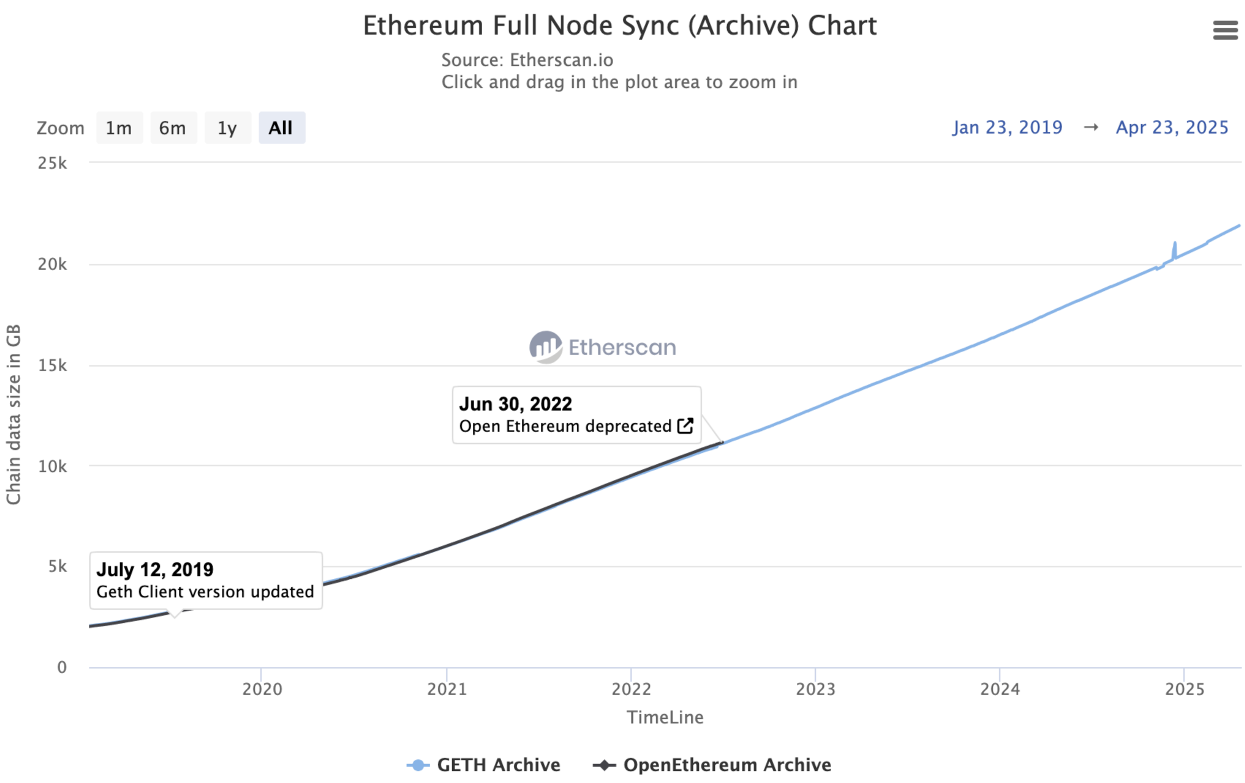
(5) **Tx receipt root**: Merkle hash of log messages generated in block

(5) Gas used: used to adjust gas price (target 15M gas per block)

The Ethereum blockchain: abstractly



Amount of memory to run a node



An example contract: NameCoin

```
contract nameCoin {

    struct nameEntry {
        address owner; // address of domain owner
        bytes32 value; // IP address
    }

    // array of all registered domains
    mapping (bytes32 => nameEntry) data;

   // event emitted when registering domain
   event Register(address indexed owner, bytes32 name);
```

An example contract: NameCoin

```
function nameNew(bytes32 name) {

    // registration costs is 100 Wei

    if (data[name] == 0   &&   msg.value >= 100) {
        data[name].owner = msg.sender    // record domain owner
        emit Register(msg.sender, name)   // log event
    }
}
```

Code ensures that no one can take over a registered name

Serious bug in this code! Front running. Solved using commitments.

An example contract: NameCoin

```
function nameUpdate(bytes32 name, bytes32 newValue, address newOwner) {

    // check if message is from domain owner,
    //                and update cost of 10 Wei is paid

    if (data[name].owner == msg.sender   &&   msg.value >= 10) {

          data[name].value = newValue;     // record new value

          data[name].owner = newOwner;     // record new owner
    }
}
```

An example contract: NameCoin

```
    function nameLookup(bytes32 name) {
        return data[name];
    }

}  // end of contract
```

Used by other contracts

Humans do not need this
(use etherscan.io)

EVM mechanics: execution environment

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

⇒ compile to EVM bytecode

(some projects use WASM or BPF bytecode)

⇒ validators use the EVM to execute contract bytecode

in response to a Tx

The EVM

Stack machine (like Bitcoin) but with JUMP

- max stack depth = 1024
- program aborts if stack size exceeded; block proposer keeps gas
- contract can <u>create</u> or <u>call</u> another contract

In addition: 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

see https://www.evm.codes

Every instruction costs gas, examples:

SSTORE addr (32 bytes), **value** (32 bytes)

- zero → non-zero: 20,000 gas
- non-zero → non-zero: 5,000 gas (for a cold slot)
- non-zero → zero: 15,000 gas refund (example)

Refund is given for reducing size of blockchain state

CREATE : 32,000 + 200 × (code size) gas;

CALL **gas**, addr, **value**, args

SELFDESTRUCT addr: kill current contract (5000 gas)

Gas calculation

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.

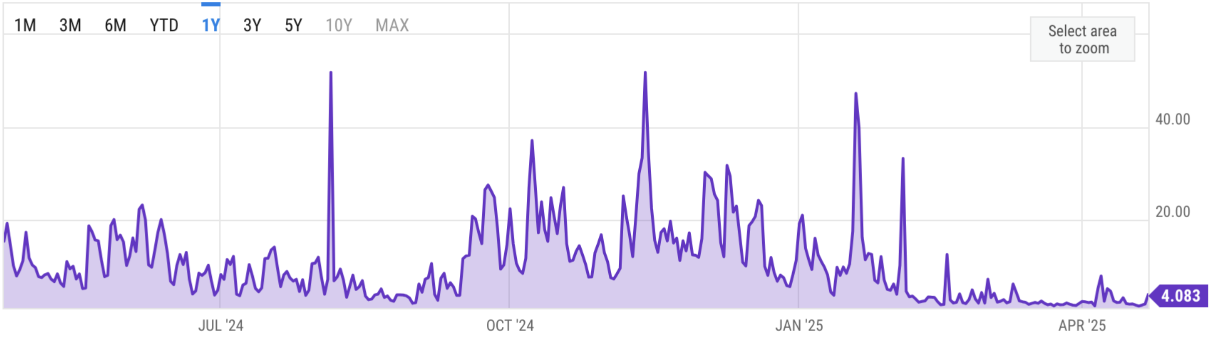
Old EVM: (prior to EIP1559)

- Every Tx contains a gasPrice ``bid'' (gas $\rightarrow$ Wei conversion price)
- Producer chooses Tx with highest gasPrice (max sum(gasPrice $\times$ gasLimit))

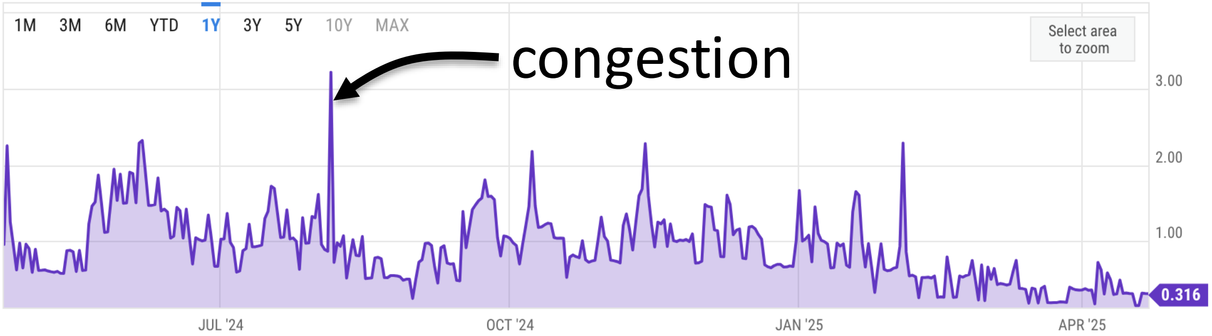
$\Longrightarrow$ not an efficient auction mechanism (first price auction)

Gas prices spike during congestion

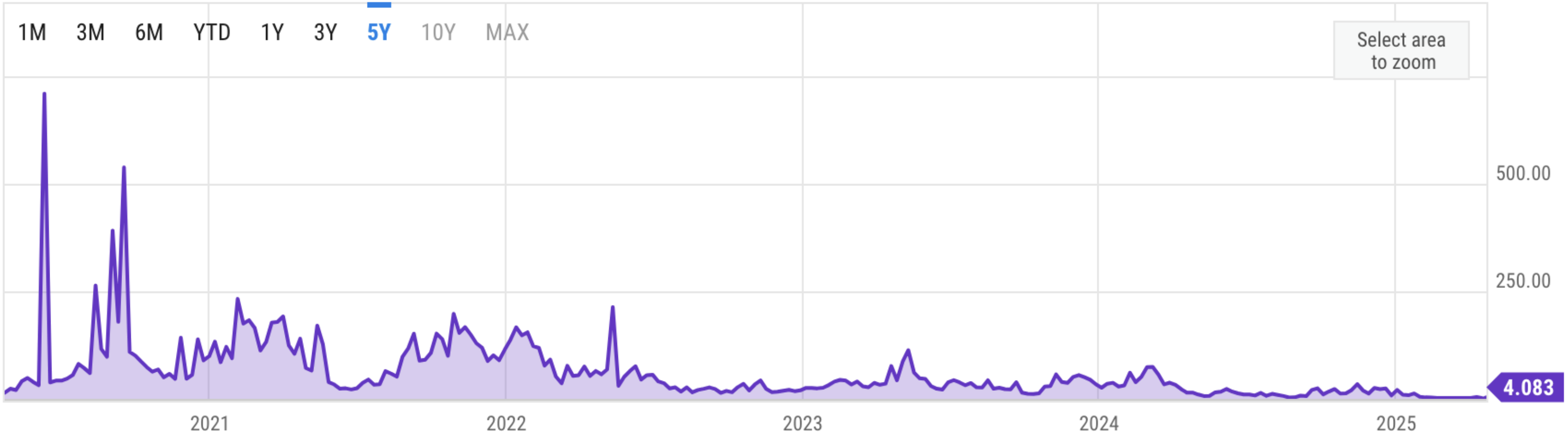
GasPrice in Gwei:



Average Tx fee in USD



Gas prices spike during congestion



Gas calculation: EIP1559

<u>EIP1559 goals</u> (informal):

- users incentivized to bid their true utility for posting Tx,
- block proposer incentivized to not create fake Tx, and
- disincentivize off chain agreements

Gas calculation: EIP1559

Every block has a "baseFee":

the **minimum** gasPrice for all Tx in the block

baseFee is computed from total gas in earlier blocks:

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

interpolate in between

If earlier blocks at "target size" (15M gas) $\Longrightarrow$ base fee does not change

Gas calculation

EIP1559 Tx specifies three parameters:

- **gasLimit**: max total gas allowed for Tx
- **maxFeePerGas:** maximum allowed gas price
- **maxPriorityFeePerGas**: additional "tip" to be paid to block proposer

Computed **gasPrice** bid:

gasPrice = min(**maxFeePerGas**, **baseFeePerGas** + **maxPriorityFeePerGas**)

Max Tx fee: **gasLimit** × **gasPrice**

Gas calculation (informal)

gasUsed ⇽ gas used by Tx

Send **gasUsed** $\times$ (**gasPricePerGas – baseFeePerGas**) to proposer

BURN **gasUsed** $\times$ **baseFeePerGas**

$\Rightarrow$ total supply of ETH can decrease

Gas calculation

(1) if **gasPrice** < **baseFeePerGas**: abort

(2) If **gasLimit × gasPrice** < msg.sender.balance: abort

(3) deduct **gasLimit × gasPrice** from msg.sender.balance

---

(4) set **Gas** ⇽ **gasLimit**

(5) execute Tx: deduct gas from **Gas** for each instruction

if at end (**Gas** < 0): abort, Tx is invalid (proposer keeps **gasLimit × gasPrice)**

(6) Refund **Gas** × **gasPrice** to msg.sender.balance

---

(7) **gasUsed** ⇽ **gasLimit – Gas**

(7a) BURN **gasUsed × baseFeePerGas**

(7b) Send **gasUsed ×** (**gasPricePerGas – baseFeePerGas**) to proposer

Base fee
0.7 GWei

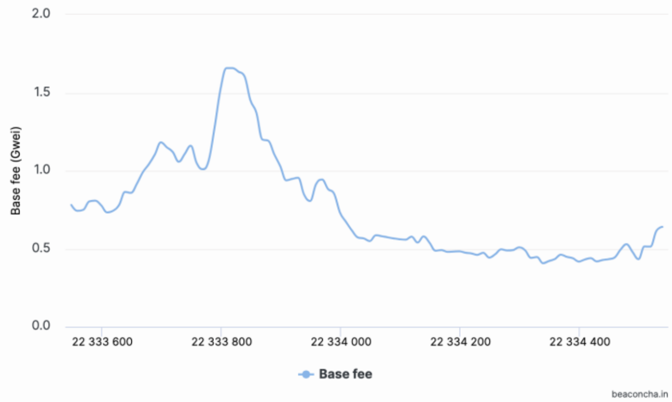
Total burned
5243421.3 ETH
$9,423,330,940.03

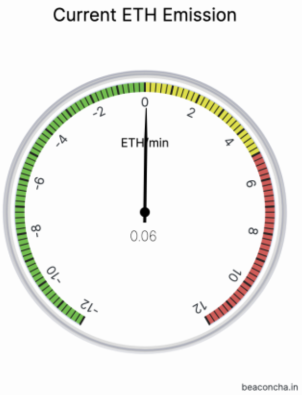
Burn rate (1h)
0.06 ETH/min
$101.19 / min

Burn rate (24h)
0.14 ETH/min
$251.09 / min

Gas target
17929741

Block utilization
50.6 %





Update in: 8s

Number	Gas Target	Gas Used	Reward	Txs	Age	Base Fee	Burned ETH
22334538	17929741	10674334	0.01686 ETH	124	11 mins ago	0.68 GWei	0.00722 ETH
22334537	17947267	33699919	0.01869 ETH	182	12 mins ago	0.61 GWei	0.02054 ETH
22334536	17964810	19121257	0.03018 ETH	225	12 mins ago	0.60 GWei	0.01156 ETH
22334535	17947284	1535098	0.00195 ETH	39	12 mins ago	0.68 GWei	0.00105 ETH
22334534	17964827	30017637	0.03586 ETH	194	12 mins ago	0.63 GWei	0.01890 ETH
22334533	17982387	25937029	0.02533 ETH	160	12 mins ago	0.60 GWei	0.01548 ETH
22334532	17999965	12252551	0.02630 ETH	134	13 mins ago	0.62 GWei	0.00762 ETH
22334531	17982405	10773799	0.00896 ETH	158	13 mins ago	0.65 GWei	0.00705 ETH
22334530	17999982	17913057	0.04856 ETH	162	13 mins ago	0.65 GWei	0.01173 ETH

Solidity

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

Contract structure

```
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)  {
        …  implentation …
}    }
```

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

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

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

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")

A → B → C → D:
at D: msg.sender == C
tx.origin == A

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

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

Inheritance

```
// SPDX-License-Identifier: MIT
// Compatible with OpenZeppelin Contracts ^5.0.0
pragma solidity ^0.8.27;

import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";

contract MyToken is ERC20, Ownable {
    constructor(address initialOwner)
        ERC20("MyToken", "MTK")
        Ownable(initialOwner)
    {}

    function mint(address to, uint256 amount) public onlyOwner {
        _mint(to, amount);
    }
}
```

ERC20 tokens

- https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
- 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.

```
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.
  - **tokenContract**.transfer(_to, _value);

Stack variables

- Stack variables generally cost the least gas
  - can be used for any simple types (anything that is <= 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 <u>memory</u> **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 → Bank.addToBalance(75)

(2) Attacker → Bank.withdrawBalance →
Attacker.fallback → Bank.withdrawBalance →
Attacker.fallback → Bank.withdrawBalance → …

withdraw 75 Wei at each recursive step

How to fix?

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

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

END OF LECTURE

Next lecture: DeFi contracts