Solidity

Dan Boneh
Recap

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

Two types of accounts:

1. **owned accounts**: address = H(PK)

2. **contracts**: address = H(CreatorAddr, CreatorNonce)
Recap: Transactions

- **To**: 32-byte address (0 → create new account)
- **From**: 32-byte address
- **Value**: # Wei being sent with Tx
- **Tx fees** ([EIP 1559](https://eips.ethereum.org/EIPS/eip-1559)): `gasLimit, maxFee, maxPriorityFee`
- **data**: what contract function to call & arguments
  
  if To = 0: create new contract  
  `code = (init, body)`
- **[signature]**: if Tx initiated by an owned account
Recap: Blocks

Miners collect Tx from users:

⇒ run them sequentially on current world state

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

[Diagram showing the blockchain structure with layers of prev hash, accounts, updated world state, transactions (Tx), logs, and messages.]

prev hash

updated world state

log messages

prev hash

updated world state

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

⇒ compile to EVM bytecode
   (recent projects use WASM or BPF bytecode)

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

Stack machine (like Bitcoin) but with JUMP

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) instruction: write data to log
Every EVM instruction costs gas

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

- zero → non-zero: 20,000 gas
- non-zero → non-zero: 5,000 gas
- non-zero → zero: 15,000 gas refund

Refund is given for reducing size of blockchain state

**SELFDESTRUCT** `addr`: kill current contract. 24,000 gas refund

**CREATE**: 32,000 gas

**CALL** `gas`, `addr`, `value`, `args`
Why charge gas?

- Tx fees (gas) prevents submitting Tx that runs for many steps.
- During high load: miners choose Tx from the mempool that maximize their income.

Old EVM: (prior to EIP1559, live on 8/2021)

- Every Tx contains a gasPrice "bid" (gas $\rightarrow$ Wei conversion price)
- Miners choose Tx with highest gasPrice ($\max \text{ sum(gasPrice} \times \text{gasLimit})$)

$\Rightarrow$ not an efficient auction mechanism (first price auction)
Gas prices spike during congestion

GasPrice in Gwei:
86 Gwei = $86 \times 10^{-9}$ ETH

Average Tx fee in USD
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) $\implies$ base fee goes up 12.5%
- earlier blocks empty $\implies$ base fee decreases by 12.5%

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

EIP1559 Tx specifies three parameters:

• **gasLimit**: max total gas allowed for Tx
• **maxFee**: maximum allowed gas price (max gas ⇽ Wei conversion)
• **maxPriorityFee**: additional “tip” to be paid to miner

Computed **gasPrice** bid:

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

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

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 $\text{gasLeft} \leftarrow \text{gasLimit}$
5. Execute Tx: deduct gas from $\text{gasLeft}$ for each instruction
   - If at end ($\text{gasLeft} < 0$): Tx is invalid (miner keeps $\text{gasLimit} \times \text{gasPrice}$)
6. Refund $\text{gasLeft} \times \text{gasPrice}$ to $\text{msg.sender.balance}$

7. $\text{gasUsed} \leftarrow \text{gasLimit} - \text{gasLeft}$
   7a. BURN $\text{gasUsed} \times \text{baseFee}$
   7b. Send $\text{gasUsed} \times (\text{gasPrice} - \text{baseFee})$ to miner
Burn results in practice

block reward (2ETH) minus Total baseFee burned in block

... sometimes burn exceeds block rewards $\implies$ ETH deflation

high baseFee period

watchtheburn.com
Impact on mining rewards

Daily fee mining rewards paid to miners

https://etherchain.org/charts/feeMiningReward
Why burn ETH ???

**EIP1559 goals (informal):**

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

Suppose no burn (i.e., baseFee given to miners):

\[\Rightarrow\] in periods of low Tx volume miners would try to increase volume by offering to refund the baseFee *off chain* to users.
Note: transactions are becoming more complex

Total Gas Usage

Evolution of the total gas used by the Ethereum network per day

Gas usage is increasing ⇒ each Tx takes more instructions to execute
Let’s look at the Ethereum blockchain etherscan.io:

**Latest Blocks**

<table>
<thead>
<tr>
<th>Bk</th>
<th>Time</th>
<th>Miner</th>
<th>Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>13405036</td>
<td>39 secs ago</td>
<td>F2Pool Old</td>
<td>125 txns in 19 secs</td>
</tr>
<tr>
<td>13405035</td>
<td>58 secs ago</td>
<td>Ethermine</td>
<td>188 txns in 2 secs</td>
</tr>
<tr>
<td>13405034</td>
<td>1 min ago</td>
<td>2Miners: PPLNS</td>
<td>85 txns in 3 secs</td>
</tr>
<tr>
<td>13405033</td>
<td>1 min ago</td>
<td>F2Pool Old</td>
<td>269 txns in 61 secs</td>
</tr>
</tbody>
</table>

**From/to address**

<table>
<thead>
<tr>
<th>From Address</th>
<th>Tx value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8875f31e963d42f2f18…</td>
<td>0 Eth</td>
</tr>
<tr>
<td>0xda1c7f958d2ee523a2…</td>
<td>0 Eth</td>
</tr>
<tr>
<td>0xcac29d3938e3a8330e…</td>
<td>0 Eth</td>
</tr>
<tr>
<td>0x7be8076f4ea4a4ad08…</td>
<td>0.09378 Eth</td>
</tr>
<tr>
<td>0xe37b696deffbc7e2639…</td>
<td>0.18 Eth</td>
</tr>
<tr>
<td>0xd9e1ce17f2641f24ae8…</td>
<td>0.18 Eth</td>
</tr>
<tr>
<td>0x578b076f33c021ca8ec…</td>
<td>0.18 Eth</td>
</tr>
<tr>
<td>0x7be8076f4ea4a4ad08…</td>
<td>0.18 Eth</td>
</tr>
</tbody>
</table>
Let's look at a transaction...

Transaction ID: 0xe3b0c8104edca4d07a00a842e05b4aa1ea80b13286c8699f...

From: 0x628ebe4e3fe7386da04a6f9a37ccb5e980c22ffc

To: Contract 0x1a2a1c938ce3ec39b6d47113c7955baa9dd454f2
(Axie Infinity: Ronin Bridge)

Value: 0.167 Ether ($583.16)

Data: Function: depositEthFor
[0]: d256119bb3ca86c7c9fcd4daba95bd233150e6

Contract generated a virtual Tx to 0xC02aaA39b... value=0.167 ETH
Let’s look at the To contract ...

Contract 0x1a2a1c938ce3ec39b6d47113c7955baa9dd454f2
(Axie Infinity: Ronin Bridge)

Balance: 240.527684887998961173 Ether

Code: 588 lines of solidity

```solidity
address public admin;
bool public paused;

modifier onlyAdmin { require(msg.sender == admin); _; }

function pause() public onlyAdmin whenNotPaused {
    paused = true; emit Paused(); }
```

anyone can read code snippet
Remember: contracts cannot keep secrets!

Contract 0x1a2a1c938ce3ec39b6d47113c7955baa9dd454f2
(Axie Infinity: Ronin Bridge)

Anyone can read contract state in storage array
⇒ never store secret keys in contract!

Solidity variables stored in S[] array
Solidity

docs: https://solidity.readthedocs.io/en/v0.8.9/

IDE: https://remix-ide.readthedocs.io/en/latest/#
contract IERC20Token {
    function transfer(address _to, uint256 _value) external returns (bool);
    function totalSupply() external view returns (uint256);
    ...
}

contract ERC20Token is IERC20Token {  // inheritance
    address owner;
    constructor() public { owner = msg.sender; }
    function transfer(address _to, uint256 _value) external returns (bool) {
        ...
    }
}
Value types

- uint256
- address (bytes32)
  - _address.balance, _address.send(value), _address.transfer(value)
  - call: send Tx to another contract
    
    ```
    bool success = _address.call(data).value(amount).gas(amount);
    ```
  - 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
class Person {
    uint128 age;
    uint128 balance;
    address addr;
}

Person[10] public people;
```
Globally available variables

- **block**: .blockhash, .coinbase, .difficulty, .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.
  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; }
```
Using imports

- Inheritance
  - contract A is SafeMath {}  
  - uint256 a = safeAdd(b, c);
  - SafeMath code is compiled into the A contract

```solidity
contract SafeMath {
    function safeAdd(uint256 a, uint256 b) internal pure returns (uint256 c) {
        c = a + b;
        require(c >= a, "UINT256_OVERFLOW");
    }
}
```
Using imports

- Inheritance
  - contract A is SafeMath {}
  - uint256 a = safeAdd(b, c);
  - SafeMath code is compiled into the A contract

- Libraries
  - contract A { using SafeMath for uint256; }
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:
  ```solidity
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`\( (\text{address \_to}, \text{uint256 \_value}) \) external returns \( (\text{bool}) \)
- function `transferFrom`\( (\text{address \_from}, \text{address \_to}, \text{uint256 \_value}) \) external returns \( (\text{bool}) \)
- function `approve`\( (\text{address \_spender}, \text{uint256 \_value}) \) external returns \( (\text{bool}) \)
- function `totalSupply`\( () \) external view returns \( (\text{uint256}) \)
- function `balanceOf`\( (\text{address \_owner}) \) external view returns \( (\text{uint256}) \)
- function `allowance`\( (\text{address \_owner}, \text{address \_spender}) \) external view returns \( (\text{uint256}) \)
How are ERC20 tokens transferred?

contract ERC20Token is IERC20Token {
    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.
  - In the case of `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. ABI encoding simple types means left padding each argument to 32 bytes.

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

- Everything costs gas, including processes that are happening under the hood (ABI decoding, copying variables to memory, etc).

Considerations in reducing gas costs:
- How often do we expect a certain function to be called? Is the bottleneck the cost of deploying the contract or the cost of each individual function call?
- Are the variables being used in calldata, the stack, memory, or storage?
Stack variables

- Stack variables are generally the cheapest to use and 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
  
  \[(68 \text{ gas per non-zero byte, } 4 \text{ gas per zero byte})\].

  - All else equal, a function with more arguments (and larger caldata) will cost more gas.

- It is cheaper to load variables directly from calldata, rather than copying them to memory.

  - For the most part, this can be accomplished by marking a function as `external`. 
Memory

- Memory is a byte array.
- Complex types (anything > 32 bytes such as structs, arrays, and strings) must be stored in memory or in storage.

```c
string memory name = "Alice";
```

- Memory is cheap, but the cost of memory grows quadratically.
Storage

- 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: 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?
- 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
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