CS251 Fall 2020

(cs251.stanford.edu)

Solidity

Dan Boneh

Recap

World state: set of accounts identified by 160-bit 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
- **gasPrice, gasLimit**: Tx fees (later)
- **data:** what contract function to call & arguments

  if To = 0: create new contract **code = (init, body)**
- **[signature]:** if Tx initiated by an owned account

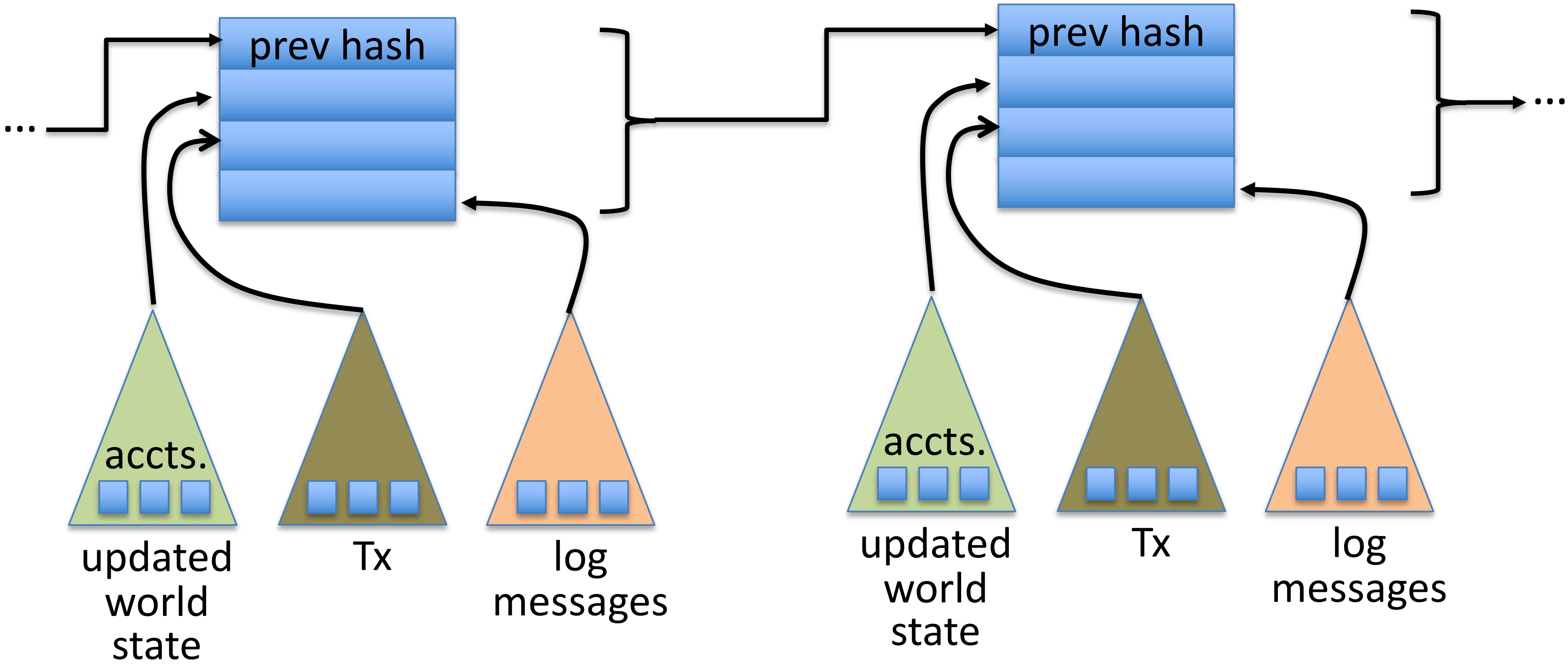
Recap: Blocks

Miners collect Tx from users:

$\Rightarrow$ run them sequentially on current world state

$\Rightarrow$ new block contains updated world state
and Tx list and log msgs

The Ethereum blockchain: abstractly



EVM mechanics: execution environment

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

- max stack depth = 1024
- program aborts if stack size exceeded; miner keeps gas
- contract can create or call 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) instruction: write data to log

Gas prices: examples

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

SUICIDE: kill current contract. 24,000 gas refund

Refund is given for reducing size of blockchain state

Gas calculation

Tx fees (gas) prevents submitting Tx that runs for many steps

Every EVM instruction costs gas:

- Tx specifies **gasPrice**: conversion: gas → Wei

  gasLimit: max gas for Tx

Gas calculation

Tx specifies **gasPrice**: conversion gas ⇾ Wei
gasLimit: max gas for Tx

(1) if **gasLimit** × **gasPrice** > msg.sender.balance: abort

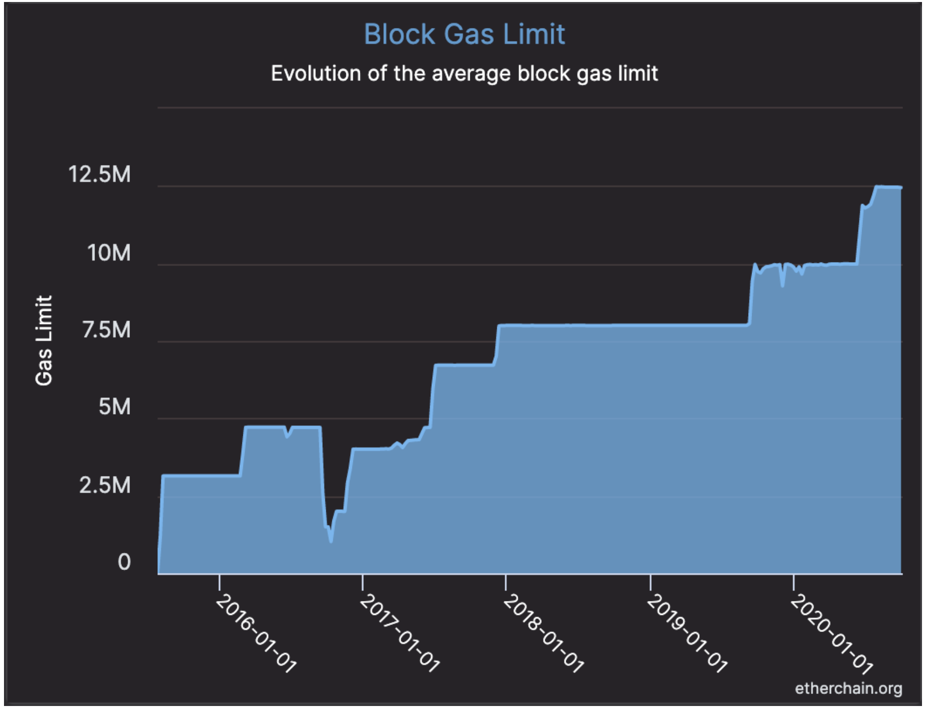
(2) deduct **gasLimit** × **gasPrice** from msg.sender.balance

(3) set Gas = gasLimit

(4) execute Tx: deduct gas from Gas for each instruction
if (Gas < 0): abort, miner keeps **gasLimit** × **gasPrice**

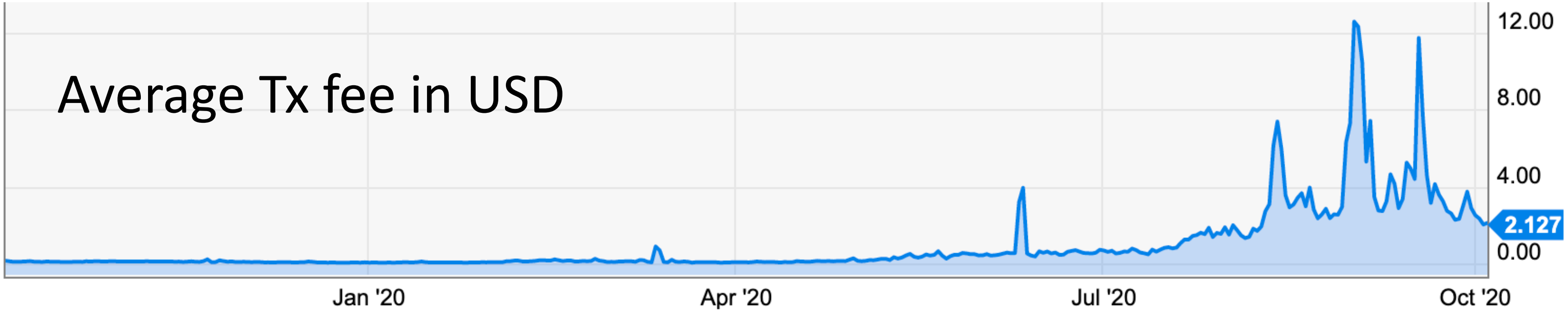
(5) Refund **Gas** × **gasPrice** to msg.sender.balance

Transactions are becoming more complex



GasLimit is increasing over time ⇒ each Tx takes more instructions to execute

Gas prices: spike during congestion



Solidity

docs: https://solidity.readthedocs.io/en/v0.7.2/

IDE: https://remix-ide.readthedocs.io/en/latest/#

Contract structure

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

Value types

- uint256
- address (bytes20)
    - _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

```
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, .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”)

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.

  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; }
```

Using imports

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

- 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; }
  - uint256 a = b.safeAdd(c);

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 contains its own ledger of balances.
- A standard interface allows other smart contracts to interact with every ERC20 tokens, rather than using 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 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;
   }
}
```

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.

```
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 to 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 <= 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 gas per non-zero byte, 4 gas per zero byte).
    - All else equal, a function with more arguments (and larger calldata) 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.

  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 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?
- 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: crypto economics