CS251 Fall 2025

(cs251.stanford.edu)

Solidity

Dan Boneh

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

Recap

World state: set of accounts identified by 20-byte address.

Two types of accounts:

(1) owned accounts (EOA): address = H(pk)

(2) contracts: address = H(CreatorAddr, CreatorNonce)

every contract has its own storage array **S[bytes32] → bytes32**

Recap: Transactions

- **To:** 20-byte address (0 → create new account)
- **From**: 20-byte address
- **Value**: # Wei being sent with Tx (1 Wei = 10^{-18} ETH, 1 GWei = 10^{-9} ETH)
- Tx fees (EIP 1559)**: gasLimit, maxFee, maxPriorityFee**
- **calldata:** what contract function to call & arguments

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

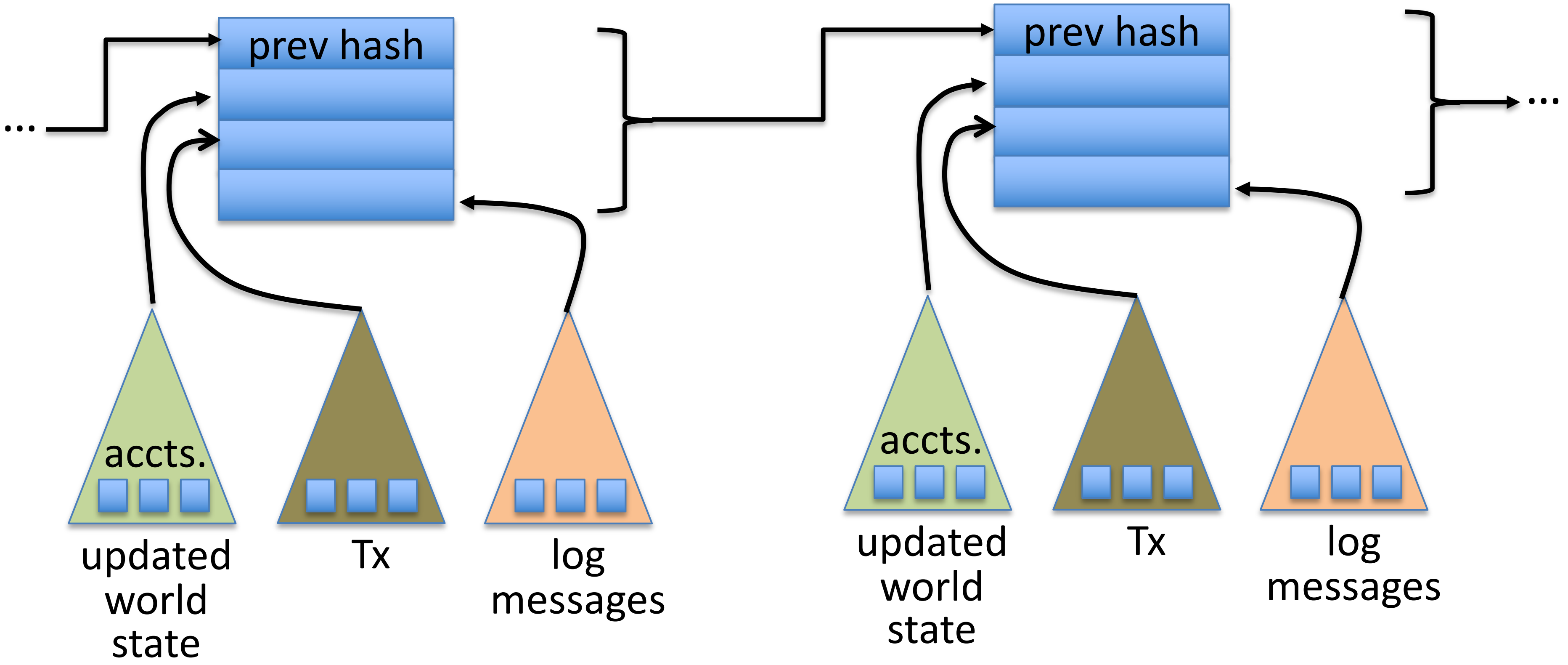
Recap: Blocks

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



EVM mechanics: execution environment

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

⇒ compile to EVM bytecode, e.g., using **solc** compiler

⇒ validators run the contract's EVM bytecode in response to a Tx

The EVM

The EVM

see https://www.evm.codes

Stack machine (like Bitcoin) but with JUMP

- max stack depth = 1024
- program aborts if stack size exceeded; block proposer keeps gas

A contract can <u>create</u> or <u>call</u> another contract

- There are several ways to call another contract
- Using the CALL instruction to call another contract
  creates a new execution frame that is deleted on return

The EVM

see https://www.evm.codes

The EVM maintains three types of zero initialized memory per contract.
All three are private to the contract that owns them (e.g., nameCoin)

- **Persistent storage** (on blockchain): SLOAD, SSTORE (expensive)
- **Volatile memory** (lives for a single Tx, one per execution frame):
  MLOAD, MSTORE (very cheap, 3 Gas)
- **Transient memory** (lives for a single Tx, one per contract):
  TLOAD, TSTORE (cheap, 100 Gas)

- LOG0(data): write data to log (easily read by a block explorer)

- Tx calldata (16 gas/byte): read-only, readable by EVM only in current Tx

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.

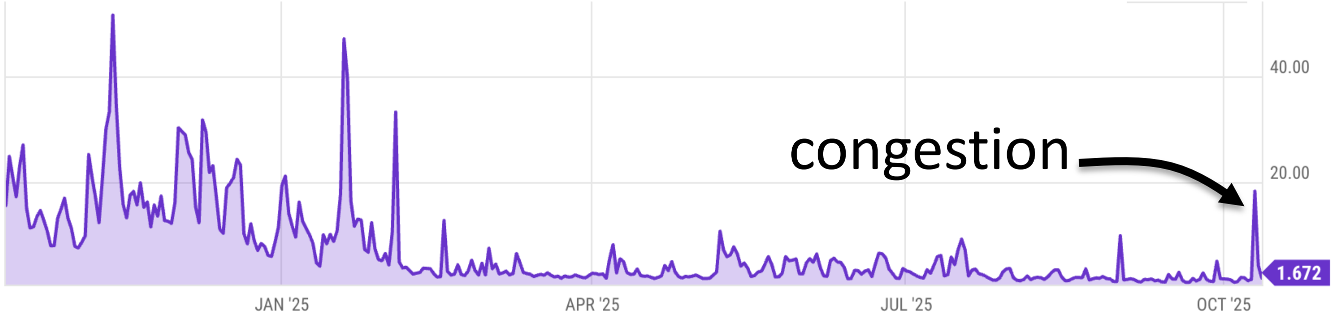
if **gasUsed ≥ gasLimit**: block proposer keeps gas fees (from Tx originator)

calculated by EVM

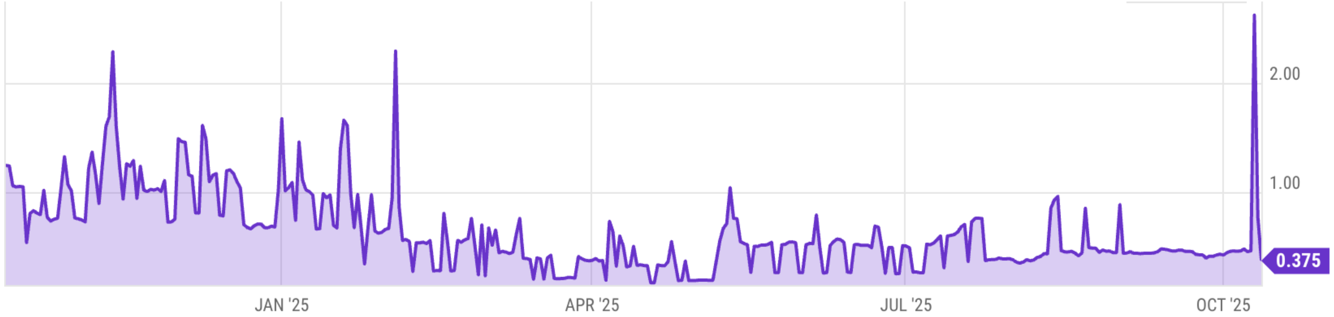
specified in Tx

Gas prices spike during congestion

GasPrice in Gwei: 1.672 Gwei = 1.672×10^{-9} ETH



Average Tx fee in USD:



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 (45M 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" (22.5M gas) $\Longrightarrow$ baseFee does not change

Gas calculation

A transaction specifies three parameters:

- **gasLimit**: max total gas allowed for Tx

bid

- **maxFee:** maximum allowed gas price
- **maxPriorityFee**: additional "tip" to be paid to block proposer

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

gasPrice ⇽ min(**maxFee, baseFee** + **maxPriorityFee)**

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

Gas calculation (informal)

gasUsed ← gas used by Tx

Send **gasUsed** × (**gasPrice** – **baseFee**) to block proposer

BURN **gasUsed** × **baseFee** 

⇒ total supply of ETH can decrease

Gas calculation

(1) if **gasPrice** < **baseFee**: 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 (leftover change)

---

(7) **gasUsed** ⇽ **gasLimit – Gas**

(7a) BURN **gasUsed** × **baseFee**
(7b) Send **gasUsed** × (**gasPrice – baseFee**) to block producer

Example baseFee and effect of burn

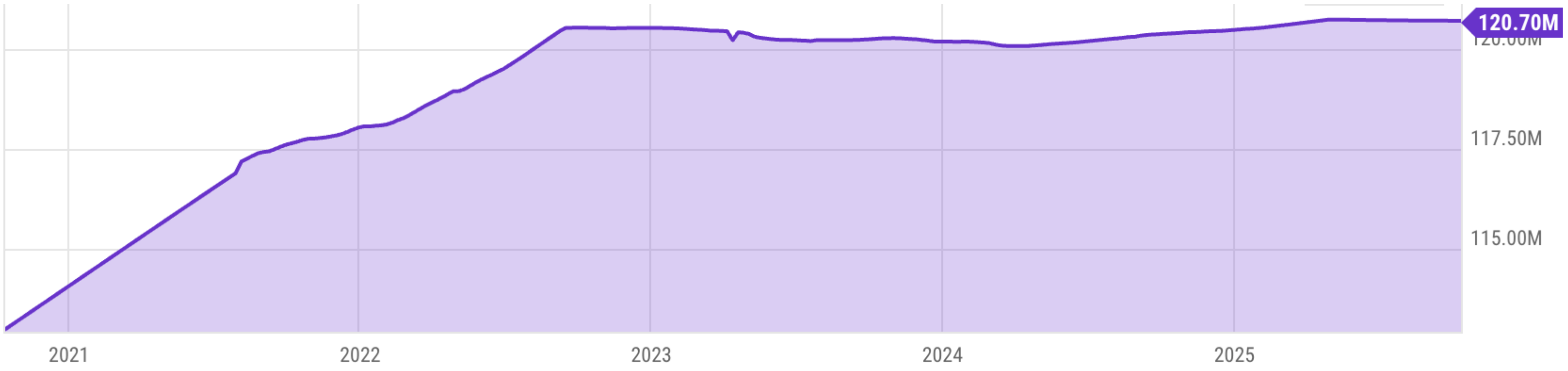
block #	gasUsed		baseFee (Gwei)	ETH burned
23573600	**10,243,205**		0.12574 ↓	0.001288
23573599	**19,388,322**	(<22.5M)	0.12794	0.002481
23573598	26,354,665		0.12525 ↑	0.003301
23573597	23,873,377	(>22.5M)	0.12429 ↓	0.002967
23573596	**21,155,077**	(<22.5M)	0.12523 ↑	0.002649
23573595	44,988,950	(>22.5M)	0.11132	0.005008

≈ gasUsed × baseFee

new issuance > burn ⇒ ETH inflates

new issuance < burn ⇒ ETH deflates

Eth total supply (last 5 years)



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):

$\Longrightarrow$ in periods of low Tx volume proposer would try to increase volume by offering to refund the baseFee to users.

Let’s look at the Ethereum blockchain

etherscan.io:

From/to address   Tx value

Latest Blocks

Bk	15778674 7 secs ago	Fee Recipient Fee Recipient: 0x6d2...766 138 txns in 12 secs
Bk	15778673 19 secs ago	Fee Recipient Lido: Execution Layer Re… 111 txns in 12 secs
Bk	15778672 31 secs ago	Fee Recipient Flashbots: Builder 313 txns in 12 secs
Bk	15778671 43 secs ago	Fee Recipient Lido: Execution Layer Re… 34 txns in 12 secs

From		To	Value
0x39feb77c9f90fae6196…	→	0x52de8d3febd3a06d3c…	0.088265 Ether
areyougay.eth	→	0x404f5a67f72787a6dbd…	0.2 Ether
Optimism: State Root Pr…	→	Optimism: State Commit…	0 Ether
0xb3336d324ed828dbc8…	→	Uniswap V3: Router 2	0 Ether
0x1deaf9880c1180b023…	→	Uniswap V3: Router 2	0.14 Ether
0x10c5a61426b506dcba…	→	Uniswap V2: Router 2	0 Ether
defiantplatform.eth	→	0x617dee16b86534a5d7…	0 Ether

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

anyone can read

```
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
}
```

code snippet

Remember: contracts cannot keep secrets!

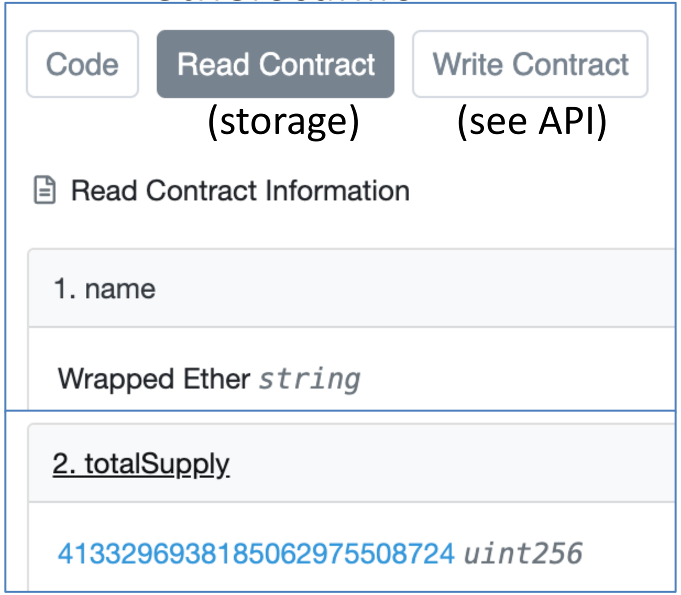
Contract 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2

(Wrapped ETH)

Anyone can read contract state in storage array

⟹ never store secrets in contract!

etherscan.io



Solidity variables stored in S[] array

Solidity

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

Several IDE's available

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)  {
        … implementation …
}    }
```

Value types

- uint256
- address (bytes20)   // address is a 20-byte value
  - _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 => uint256) **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(bytes), sha256(bytes)
- **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.

  ```
  function foo(bytes memory data) public {}   // data is always copied from calldata to memory
  function foo(bytes data) public {}   // data is not copied to memory, therefore read-only
  ```

- **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

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

- <u>Inheritance</u>

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

code of contract "owned" is compiled into contract Destructable

---

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

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");
        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)**

Calling other contracts

- Addresses can be cast to contract types:

```
address  _token = 0x2b34aced3456781243651234de348791;
ERC20Token  tokenContract = ERC20Token(_token);    // type cast
```

- When calling a function on an external contract, Solidity will automatically handle ABI encoding, copying to memory, and copying return values.

```
// call the `transfer` function at address tokenContract
(bool success,) = tokenContract.transfer(_to,  _value);
```

this causes the EVM to send a message from origin contract to tokenContract.

ABI encoding and decoding

When calling a contract, the **calldata** in the transaction is an ABI encoding of:

(1) First 4 bytes of calldata: a **function selector** indicating what function in the contract to run

- The function selector is the first 4 bytes of the hash of the function signature:

  for \`transfer\`, this looks like **bytes4(keccak256("transfer(address,uint256)"));**

(2) The rest of calldata is an ABI encoding of the function arguments

Contracts can also have two special ``last resort'' functions: **receive** and **fallback**

- These functions are called if no function in the contract matches the selector in calldata

```
receive() external payable { code }      // called if calldata is empty (i.e., pure ETH transfer)
fallback() external payable { code }     // called if calldata is not empty
```

An example ABI encoding

ERC20Token **tokenContract** = ERC20Token(_token);

address _to;

The easy way to call the transfer function:

```
    // Solidity compiler creates the ABI encoding
    (bool success,) = tokenContract.transfer( _to,   1 ether);
```

the function selector for `transfer`

Or you can do the ABI encoding yourself:

```
    action = abi.encodeWithSelector( tokenContract.transfer.selector,  _to,  1 ether);
    (bool success,) = tokenContract.call(action);
```

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 done by marking a function as \`external\` or \`public\` without marking the argument as \`memory\`

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 gas: 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?

Reentrancy bugs

```
contract Bank {
    mapping(address => uint256) public userBalances;

    function getUserBalance(address user) constant public returns(uint256) {
        return userBalances[user];    }

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

    function withdrawBalance() public {     // customer withdraws its entire balance
        uint256 amountToWithdraw = userBalances[msg.sender];
        // send ETH from Bank contract to caller ... vulnerable!
        (bool succ,) = msg.sender.call{value: amountToWithdraw}("");
        require(succ, "Withdraw failed");
        userBalances[msg.sender] = 0;    // if success, clear user's balance
}}
```

```
contract Attacker {
    Bank bank;

    constructor(address bankAddress) payable {
        bank = Bank(bankAddress);
        bank.addToBalance{value: 75}();     // Deposit 75 Wei
    }

    function triggerAttack() public {  bank.withdrawBalance();  }

    receive() external payable {
        //  repeat as long as we have enough gas and the bank still has funds
        if (gasleft() > 10000  &&  address(bank).balance >= 75) {
            bank.withdrawBalance();
        }
} }
```

ETH balance of Bank contract

Why is this an attack?

step 1: Attacker ⇾ Bank.addToBalance(75)

step 2: Attacker ⇾ Bank.withdrawBalance ⇾
Attacker.receive ⇾ Bank.withdrawBalance ⇾
Attacker.receive ⇾ Bank.withdrawBalance ⇾ …

Withdraw 75 Wei from Bank contract at each recursive step !!

- Need to ensure overall transaction does not fail; ensured by if

How to fix: method 1

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

    // checks
    require(amountToWithdraw > 0, "No balance to withdraw");

    // effects
    userBalances[msg.sender] = 0;      // clear user’s balance

    // interactions
    (bool success, ) = msg.sender.call{value:amountToWithdraw}("");
    require(success);          //  revert transaction on failure
}
```

How to fix: method 2

```
bool transient locked;        // a flag in the contract's transient memory
```

```
modifier nonReentrant {
    require(!locked, "Reentrancy attempt");      // function prologue
    locked = true;
    _;
    locked = false;    // reset guard in function epilogue
}
```

```
function withdrawBalance() public nonReentrant {    // a protected function
    …
}
```

END OF LECTURE

Next lecture: DeFi contracts