

Smart Contracts and Ethereum

50.037 Blockchain Technology
Paweł Szalachowski

Contracts

	Traditional
specification	Natural language + “legalese”
identity & consent	Signatures
dispute resolution	Judges, arbitrators
nullification	By judges
payment	Carried out by parties separately
escrow	Trusted third party, settled in \$

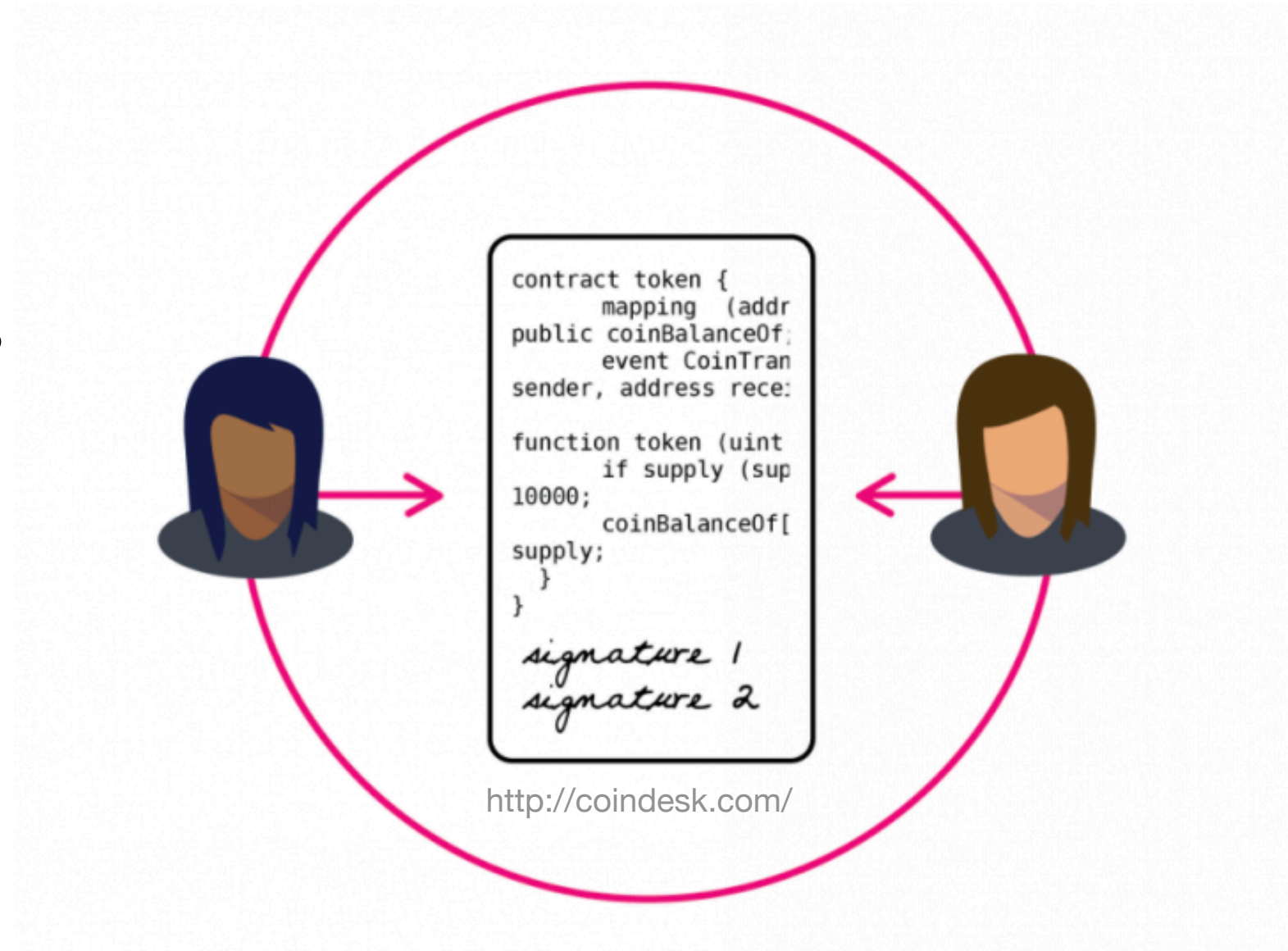
Smart Contracts

“A smart contract is a computerized transaction protocol that executes the terms of a contract. The general objectives are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitrations and enforcement costs, and other transaction costs.”

Nick Szabo, 1994

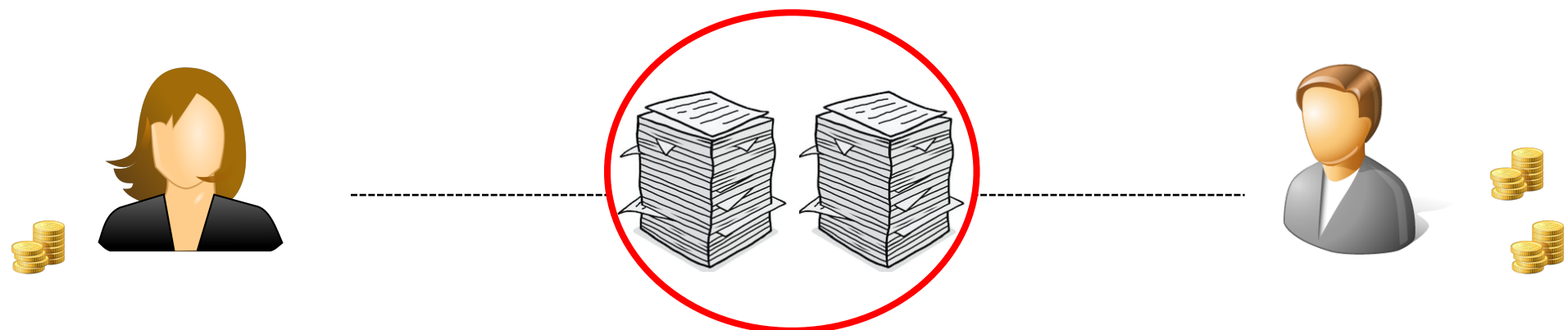
Smart Contracts

- Any agreement that can be represented by code (*code is law*)
- Insurances
- Financial Contracts
- Gambling
- Voting
- ...



Example

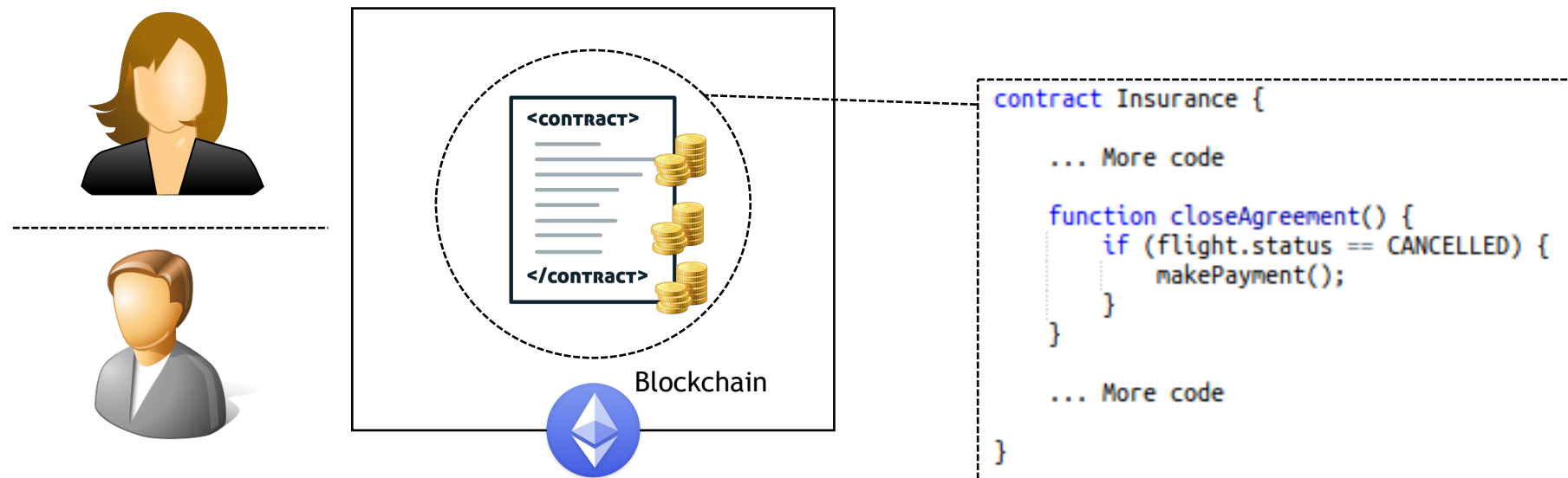
- Example: flight insurance
- Expensive and long (potentially annoying) process
- Asymmetric
- Manual process
- Can we make it fair and automated?



Example

- Represent insurance as *code* executed over the blockchain

- Immutable
- Deterministic
- Decentralized
- Explicit
- Built-in monetary transfers



Bitcoin & Smart Contracts

- The idea has been asleep for decades
 - How to implement smart contracts practically?
- Run code (programs) over blockchain
- Why do not use Bitcoin pubkey and sig scripts?
 - Encode more sophisticated logic than token transfer

Example: Coin Flipping Online

Round 1:

Each party picks a large random string — Alice picks x , Bob picks y , and Carol picks z .

The parties publish $H(x)$, $H(y)$, $H(z)$ respectively.

Each party checks that $H(x)$, $H(y)$, $H(z)$ are all distinct values (otherwise aborts the protocol).

Round 2:

The three parties reveal their values, x , y , and z .

Each party checks that the revealed values agree with the hashes published in Round 1.

The outcome is $(x + y + z) \% 3$.

What if someone does not reveal their commitment?

Alice can make a *timed commitment* with a bond spendable to Bob if

Case 1: it is signed by Alice & Bob

Case 2: or, only by Alice but revealing x (the input)

Then they create a time locked transaction ($nLockTime$) paying the bond to Bob after some time t

```
scriptPubKey:
  OP_IF
    <AlicePubKey> OP_CHECKSIGVERIFY <BobPubKey> OP_CHECKSIG
  OP_ELSE
    <AlicePubKey> OP_CHECKSIGVERIFY OP_HASH <H(x)> OP_EQUAL
  OP_ENDIF

scriptSig for Case 1:
  <BobSignature> <AliceSignature> 0
scriptSig for Case 2:
  x <AliceSignature> 1
```


Bitcoin & Smart Contracts



- If we need new features (opcodes) we can add them
 - See Namecoin
 - Still application-specific
- The scripting language is not Turing complete and limited
 - Growing code size
 - Limited capabilities
 - Miners rejecting non-standard scripts
- Can we have a general purpose with cryptocurrency?

Ethereum



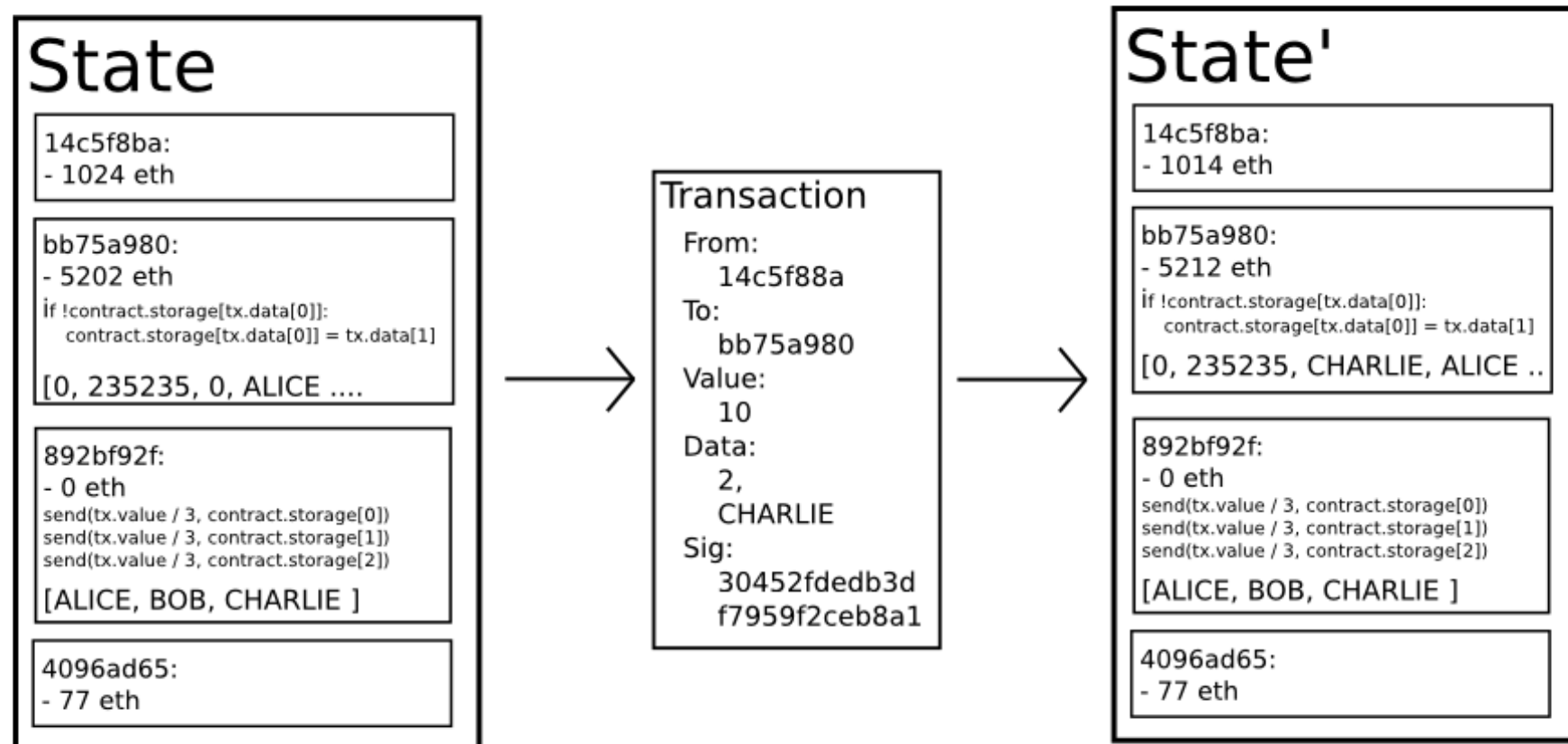
ETHEREUM

- *General-purpose* cryptocurrency, 2014
- Crowdfunded
- The second largest cryptocurrency
- Flexible and Turing complete scripting languages
- Native currency: Ether (1 Ether = $1e18$ Wei)

#	Name	Market Cap
1	 Bitcoin	\$109,375,449,973
2	 Ethereum	\$21,101,591,960

High-level Overview

- Miners run a consensus protocol
 - On the top: monetary transfers, code, and code calls
- Miners keep the current state
 - State transition



Ethereum

- Code is published on blockchain
 - Visible, immutable, ...
 - Anyone can interact (by sending transactions)
- Anything that can be coded
 - With monetary transfers
- Distributed Applications (DApps)
 - Smart contract + Front-end + ...
 - Built-in payments, auditable, ...
 - Limited development lifecycle (code cannot be updated, bugs are painful)

Smart Contracts

	Traditional	Smart
specification	Natural language + “legalese”	Code
identity & consent	Signatures	Digital signatures
dispute resolution	Judges, arbitrators	Decentralized platform
nullification	By judges	????
payment	Carried out by parties separately	built-in
escrow	Trusted third party, settled in \$	built-in

Namecoin in Ethereum

- Namecoin (recap)
 - Bitcoin's fork
 - New OPCODES (name register, update, ...)
- Ethereum (pseudocode)

```
def register(name, value):  
    if !self.storage[name]:  
        self.storage[name] = value  
    return 1  
return 0
```

Ethereum Virtual Machine (EVM)

- Runtime environment for smart contracts
 - Isolated (no access to network, filesystem, processes, ...)
- Accounts (all treated equally by EVM)
 - External: controlled by public-private key pairs (users)
 - Derived from public keys
 - Contract: controlled by the code
 - Derived from creator address and the number (nonce) of txs of that address
- Every account has a persistent key-value storage and balance
- Contracts can have memory, freshly cleared for every call

Transactions

- A message sent from one account to another
- Can have payload and Ether
 - If the target account has code, it is executed with the payload (as the input)
- If the target account = 0, a new contract is created
 - Payload is bytecode, executed and stored permanently
 - Contract's address is derived from sender and nonce

Gas

Recommended Gas Prices
(based on current network conditions)

Speed	Gas Price (gwei)
SafeLow (<30m)	1.7
Standard (<5m)	1.7
Fast (<2m)	14

- What prevents from: while(1) do(); ?
- Pre-paid cost expressed in gas
 - Paid for contract creation, execution, and storage
- Consumed by EVM while execution
- gas_price set by sender: $\text{cost} = \text{gas_price} * \text{gas}$
 - Paid from the sending account (leftovers are refunded)
- Out-of-gas exception
 - Reverts all modifications of the call (except the gas used)

Name	Value	Description*
G_{zero}	0	Nothing paid for operations of the set W_{zero} .
G_{base}	2	Amount of gas to pay for operations of the set W_{base} .
$G_{verylow}$	3	Amount of gas to pay for operations of the set $W_{verylow}$.
G_{low}	5	Amount of gas to pay for operations of the set W_{low} .
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid} .
G_{high}	10	Amount of gas to pay for operations of the set W_{high} .
$G_{extcode}$	700	Amount of gas to pay for operations of the set $W_{extcode}$.
$G_{balance}$	400	Amount of gas to pay for a BALANCE operation.
G_{sload}	200	Paid for a SLOAD operation.
$G_{jumpdest}$	1	Paid for a JUMPDEST operation.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	5000	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{selfdestruct}$	24000	Refund given (added into refund counter) for self-destructing an account.
$G_{selfdestruct}$	5000	Amount of gas to pay for a SELFDESTRUCT operation.
G_{create}	32000	Paid for a CREATE operation.
$G_{codedeposit}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
G_{call}	700	Paid for a CALL operation.
$G_{callvalue}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$G_{callstipend}$	2300	A stipend for the called contract subtracted from $G_{callvalue}$ for a non-zero value transfer.
$G_{newaccount}$	25000	Paid for a CALL or SELFDESTRUCT operation which creates an account.
G_{exp}	10	Partial payment for an EXP operation.
$G_{expbyte}$	50	Partial payment when multiplied by $\lceil \log_{256}(exponent) \rceil$ for the EXP operation.
G_{memory}	3	Paid for every additional word when expanding memory.
$G_{txcreate}$	32000	Paid by all contract-creating transactions after the <i>Homestead</i> transition.
$G_{txdatazero}$	4	Paid for every zero byte of data or code for a transaction.
$G_{txdatanonzero}$	68	Paid for every non-zero byte of data or code for a transaction.
$G_{transaction}$	21000	Paid for every transaction.
G_{log}	375	Partial payment for a LOG operation.
$G_{logdata}$	8	Paid for each byte in a LOG operation's data.
$G_{logtopic}$	375	Paid for each topic of a LOG operation.
G_{sha3}	30	Paid for each SHA3 operation.
$G_{sha3word}$	6	Paid for each word (rounded up) for input data to a SHA3 operation.
G_{copy}	3	Partial payment for *COPY operations, multiplied by words copied, rounded up.
$G_{blockhash}$	20	Payment for BLOCKHASH operation.
$G_{quaddivisor}$	100	The quadratic coefficient of the input sizes of the exponentiation-over-modulo precompiled contract.

Calls

- A client sends a transaction that contains a message
 - source, target, payload, Ether, gas, ...
- Contracts can call other contracts
- Contracts can create other contracts
- Delegatecall / Callcode
 - Load code dynamically from a different address
- Selfdestruct operation
 - Removes the contract (from the state) and refunds remaining Ether

TxHash: 0x2f4abe39fa1ca7a2eec687cc13b0aeedbc158179d21c2ab9ec17430d5c51a6fe

TxReceipt Status: **Success**

Block Height: [6522631](#) (2 Block Confirmations)

TimeStamp: 1 min ago (Oct-16-2018 12:07:55 AM +UTC)

From: [0x824ada524ad4dd041036160f352a6f38411edf0b](#)

To: [Contract 0xd4f5bf184bebfd53ac276ec6e091d051d0ed459e](#) 
TRANSFER 0.023 Ether From [0xd4f5bf184bebf...](#) To → [0xa9ba80d13cc2...](#)

Value: 0 Ether (\$0.00)

Gas Limit: 150000

Gas Used By Transaction: 67541

Gas Price: 0.00000002 Ether (20 Gwei)

Actual Tx Cost/Fee: 0.00135082 Ether (\$0.28)

Nonce & {Position}: 52245 | {37}

Input Data:

```
Function: sendMultiSig(address toAddress, uint256 value, bytes data, uint256  
expireTime, uint256 sequenceId, bytes signature) ***
```

MethodID: 0x39125215

```
[0]: 00000000000000000000000000000000a9ba80d13cc25a4914437127b716dbb7e715b7cf  
[1]: 00000000000000000000000000000000000000000000000000000000000000051b660cdd58000  
[2]: 0000000000000000000000000000000000000000000000000000000000000000c0  
[3]: 0000000000000000000000000000000000000000000000000000000000000005bce661c
```

View Input As ▾

Solidity

- The most popular smart-contract-oriented language
 - Other languages: Vyper, Serpent, Bamboo, LLL, ...
- Turing complete, neither low-level nor high-level
- JavaScript-like, statically typed, under heavy development
- Compiled to bytecode and run in EVM
 - Bytecode published on blockchain (src code can be published too)
- Interactions via transactions (more precisely, messages)
 - Application Binary Interface (ABI)

Example: Simple Storage

- Pragma
- Contract (like class)
 - Code + data
- Storage
 - State variables

```
pragma solidity ^0.4.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```


Example: Subcurrency

- Address
 - Associates external actors
- Public
 - Creates a getter function
 - (always readable from the blockchain)
- Mappings
- Events
 - Listeners can listen to them
- Constructors

```
pragma solidity ^0.4.21;

contract Coin {
    // The keyword "public" makes those variables
    // readable from outside.
    address public minter;
    mapping (address => uint) public balances;

    // Events allow light clients to react on
    // changes efficiently.
    event Sent(address from, address to, uint amount);

    // This is the constructor whose code is
    // run only when the contract is created.
    function Coin() public {
        minter = msg.sender;
    }

    function mint(address receiver, uint amount) public {
        if (msg.sender != minter) return;
        balances[receiver] += amount;
    }

    function send(address receiver, uint amount) public {
        if (balances[msg.sender] < amount) return;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```


Memory

Forced data location:

- parameters (not return) of external functions: calldata
- state variables: storage

Default data location:

- parameters (also return) of functions: memory
- all other local variables: storage

```
pragma solidity ^0.4.0;

contract C {
    uint[] x; // the data location of x is storage

    // the data location of memoryArray is memory
    function f(uint[] memoryArray) public {
        x = memoryArray; // works, copies the whole array to storage
        var y = x; // works, assigns a pointer, data location of y is storage
        y[7]; // fine, returns the 8th element
        y.length = 2; // fine, modifies x through y
        delete x; // fine, clears the array, also modifies y
        // The following does not work; it would need to create a new temporary /
        // unnamed array in storage, but storage is "statically" allocated:
        // y = memoryArray;
        // This does not work either, since it would "reset" the pointer, but there
        // is no sensible location it could point to.
        // delete y;
        g(x); // calls g, handing over a reference to x
        h(x); // calls h and creates an independent, temporary copy in memory
    }

    function g(uint[] storage storageArray) internal {}
    function h(uint[] memoryArray) public {}
}
```

Read doc

- Modifiers, events, structs, enums, booleans, integers, calls, arrays, function types,
- Payable, fallback, functions...
 - Read warnings carefully!

`this` (current contract's type):

the current contract, explicitly convertible to `Address`

`selfdestruct(address recipient)`:

destroy the current contract, sending its funds to the given `Address`

`suicide(address recipient)`:

deprecated alias to `selfdestruct`

- `block.blockhash(uint blockNumber)` returns `(bytes32)`: hash of the given block - only works for 256 most recent, excluding current, blocks - deprecated in version 0.4.22 and replaced by `blockhash(uint blockNumber)`.
- `block.coinbase` (`address`): current block miner's address
- `block.difficulty` (`uint`): current block difficulty
- `block.gaslimit` (`uint`): current block gaslimit
- `block.number` (`uint`): current block number
- `block.timestamp` (`uint`): current block timestamp as seconds since unix epoch
- `gasleft()` returns `(uint256)`: remaining gas
- `msg.data` (`bytes`): complete calldata
- `msg.gas` (`uint`): remaining gas - deprecated in version 0.4.21 and to be replaced by `gasleft()`
- `msg.sender` (`address`): sender of the message (current call)
- `msg.sig` (`bytes4`): first four bytes of the calldata (i.e. function identifier)
- `msg.value` (`uint`): number of wei sent with the message
- `now` (`uint`): current block timestamp (alias for `block.timestamp`)
- `tx.gasprice` (`uint`): gas price of the transaction
- `tx.origin` (`address`): sender of the transaction (full call chain)

`<address>.balance (uint256):`

balance of the [Address](#) in Wei

`<address>.transfer(uint256 amount):`

send given amount of Wei to [Address](#), throws on failure, forwards 2300 gas stipend, not adjustable

`<address>.send(uint256 amount) returns (bool):`

send given amount of Wei to [Address](#), returns `false` on failure, forwards 2300 gas stipend, not adjustable

`<address>.call(...) returns (bool):`

issue low-level `CALL`, returns `false` on failure, forwards all available gas, adjustable

`<address>.callcode(...) returns (bool):`

issue low-level `CALLCODE`, returns `false` on failure, forwards all available gas, adjustable

`<address>.delegatecall(...) returns (bool):`

issue low-level `DELEGATECALL`, returns `false` on failure, forwards all available gas, adjustable

`keccak256(...) returns (bytes32):`

compute the Ethereum-SHA-3 (Keccak-256) hash of the [\(tightly packed\) arguments](#)

`sha256(...) returns (bytes32):`

compute the SHA-256 hash of the [\(tightly packed\) arguments](#)

`sha3(...) returns (bytes32):`

alias to `keccak256`

`ripemd160(...) returns (bytes20):`

compute RIPEMD-160 hash of the [\(tightly packed\) arguments](#)

`ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)`

recover the address associated with the public key from elliptic curve signature or return zero on error ([example usage](#))

Design Challenges

- Correctness and availability of smart contracts
- Their code, all state, and interactions are publicly visible
- Can be resource consuming
- Calls reordering and delays
- Limited by the underlying consensus protocol
 - Forks, scalability, ...

Security

- Smart contracts have to be deterministic
 - How to implement randomness?

```
// Won if block number is even
// (note: this is a terrible source of randomness, please don't use
this with real money)
```

```
bool won = (block.number % 2) == 0;
```

```
// Compute some *almost random* value for selecting winner from
current transaction.
```

```
var random = uint(sha3(block.timestamp)) % 2;
```

```
function random(uint64 upper) public returns (uint64 randomNumber) {
    _seed = uint64(sha3(sha3(block.blockhash(block.number), _seed),
now));
    return _seed % upper;
}
```

Security

- Re-entrancy
 - Contract A calling contract B passes its control to B, that can call A again (think of money transfers)

```
contract Fund {  
    /// Mapping of ether shares of the contract.  
    mapping(address => uint) shares;  
    /// Withdraw your share.  
    function withdraw() public {  
        if (msg.sender.send(shares[msg.sender]))  
            shares[msg.sender] = 0;  
    }  
}
```

```
contract Fund {  
    /// Mapping of ether shares of the contract.  
    mapping(address => uint) shares;  
    /// Withdraw your share.  
    function withdraw() public {  
        var share = shares[msg.sender];  
        shares[msg.sender] = 0;  
        msg.sender.transfer(share);  
    }  
}
```

Security

- Ether transfers are rejected when sent to contracts w/o a fallback function
- Fallback functions get gas stipend via `send()`
 - Can be increased (`transfer(x)`, `call.value(x)()`)
- Max call stack = 1024
 - When exceeded an exception is thrown (`send()` does not throw an exception, returns `False` instead)
- `msg.sender != tx.origin`

Security

- `for (var i = 0; i < 2000; i++) { ... }`
 - Out-of-gas exception, why?

- Again: **read warnings!**

Warning

The type is only deduced from the first assignment, so the loop in the following snippet is infinite, as `i` will have the type `uint8` and the highest value of this type is smaller than

```
2000. for (var i = 0; i < 2000; i++) { ... }
```

- Integer overflows
- Pragma, fail-safe modes, verification
- Use secure templates when possible

Reading

- Textbook 10.7
- <https://github.com/ethereum/wiki/wiki/White-Paper>
- <https://solidity.readthedocs.io>
- <https://ethereum.github.io/yellowpaper/paper.pdf>
- ... and inline references