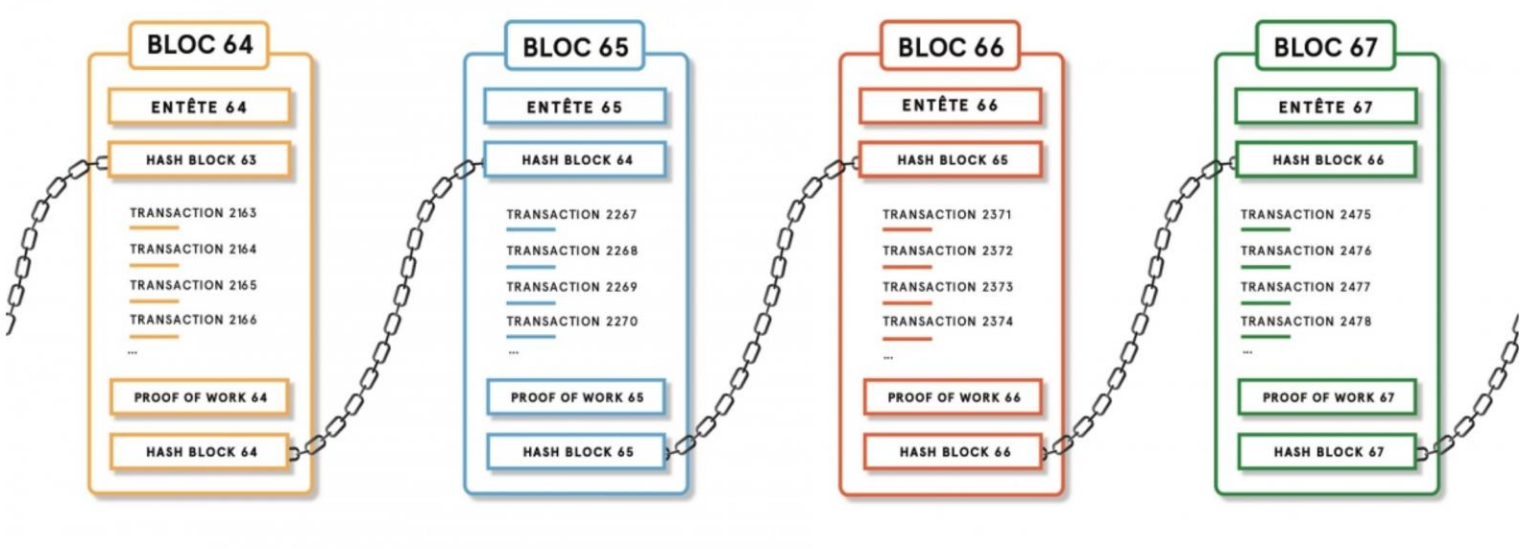
Ethereum Smart Contracts

Shuai Wang

香 港 科 技 大 學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

A Very Holistic View of Ethereum Blockchain (Blockchain 2.0)



Bitcoin

- Unregulated digital currency
- Bitcoin transactions are stored on Blockchain
- Each anonymous address on the Blockchain acted as a simple bank account.

"traditional mobile phone"

Ethereum

- Unregulated digital currency and computing system
- Smart contracts: programs executed on the blockchain
- Each anonymous address on the blockchain could be a user or a smart contract.

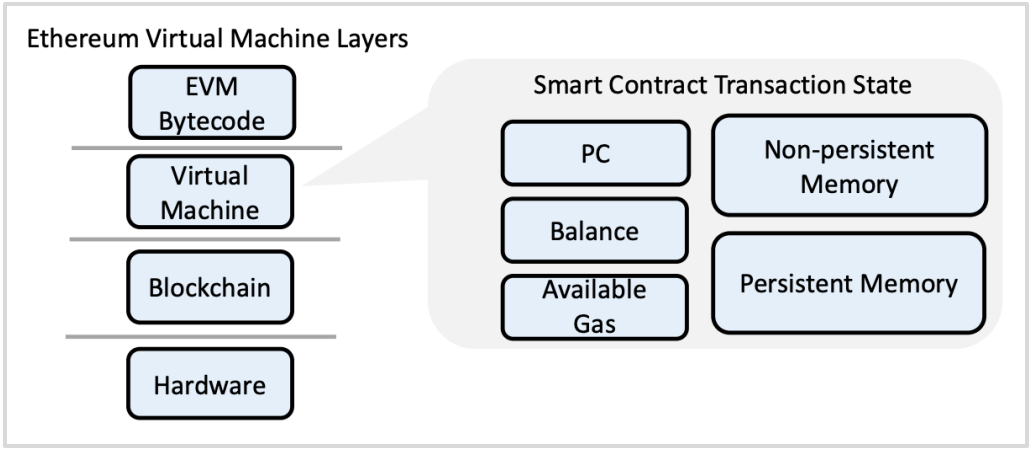
"smart phone app" → smart contract

Blockchain figure from: https://blog.theodo.com/2018/01/deploy-first-ethereum-smart-contract-blockchain/

Smart Contract

- Small programs handling cryptocurrencies
- Written in Solidity, Vyper
- Executed on the Blockchain
- No patching after release
- What if there is infinite loop in the code?
  - That's why we need gas.
  - Generally speaking, prevent abusing the miner who picks this transaction.

```
mapping(address => uint) balances;

function withdraw() {
   uint amount = balances[msg.sender];
   msg.sender.call.value(amount);
   balances[msg.sender] = 0;
}
```



Smart Contract

Think about how you define a Java/C++/Python Class: {public function; private function; private data; local data}

```
contract Intermediary {
  uint256 public fee;
  address public seller;
  address public owner;

  function Intermediary() {
    owner = msg.sender;
    seller initialization is omitted
    fee = 10;
  }
```

A sample smart contract.

```
  function purchase() {
// msg.value is how much Ether was sent by user
// transfer pays (msg.value-fee) to the seller
    owner.transfer(msg.value - fee);
  }
  function setFe      t256 _fee) {
    if (msg.sender      ner)
      fee = _fee;
  }
}
```

A sample smart contract (cont'd).

- Transaction: each transaction typically executes one public function.
- Global variables: kept in persistent storage across different transactions.
- Local variables (not in the example): temporarily used within the current transaction.
- Payment statement: contract external calls.

Create Your Own Crypto Tokens -- ICO

Besides **Ether**, we can customize our own tokens within Ethereum Smart Contracts

ERC-20 Token Standard

function name() **public** view returns (string)
function totalSupply() **public** view returns (uint256)
function balanceOf(address _owner) **public** view returns (uint256 balance)
function transfer(address _to, uint256 _value) **public** returns (bool success)
…

But why? (it's believed over 140K different Crypto Tokens have been created so far)

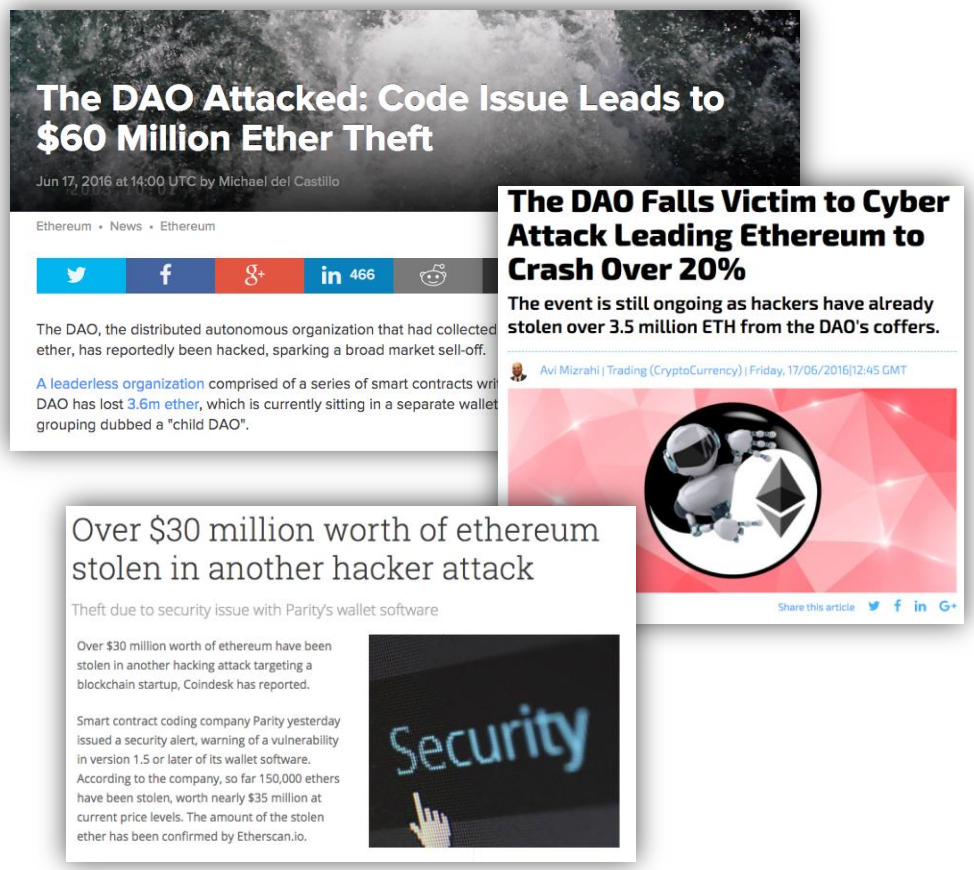
- An easy way of getting super rich? → Initial Coin Offering (ICO) (similar to IPO)
- Or lost your reputation? → completely unregulated
- …

ERC-20 has become the technical standard used for all smart contracts on the Ethereum blockchain for token implementation. https://eips.ethereum.org/EIPS/eip-20

https://www.investopedia.com/terms/i/initial-coin-offering-ico.asp

Motivation

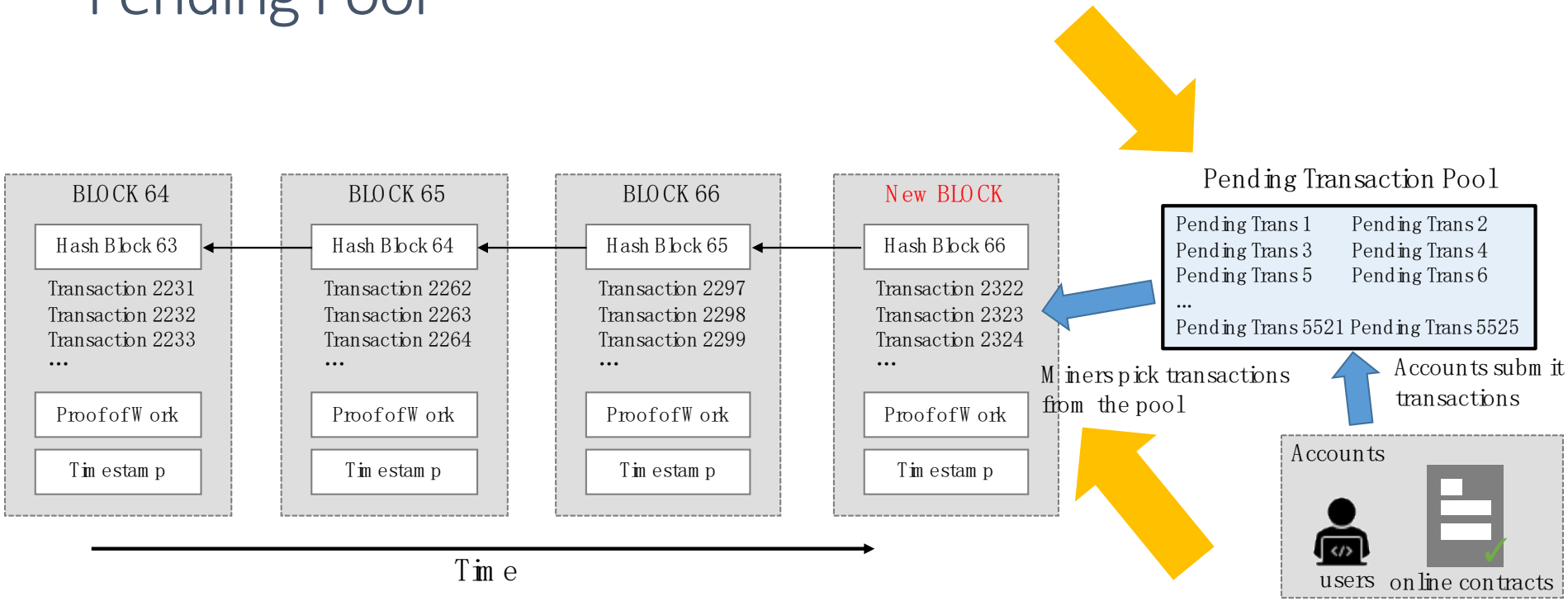
- Handle a lot of money
- Difficult to write bug-free smart contracts (really difficult)
  - Lack of fully understanding of subtle design choice of Smart contracts and its execution model



The “non-determinism” of Smart Contract Execution Model

Actually not “unique” to Ethereum but particularly harmful to Ethereum/smart contracts!!!

The Ethereum Blockchain and the Transaction Pending Pool



- Ethereum account: primary users of Ethereum; could be controlled by private keys or controlled by contract code
- Each transaction waits in the transaction pending pool, until it gets picked and executed by an miner.
- Miners are free to include any transactions into the New BLOCK in any order
  - → a source of "nondeterminism" since order of transactions execution are unpredictable to users.

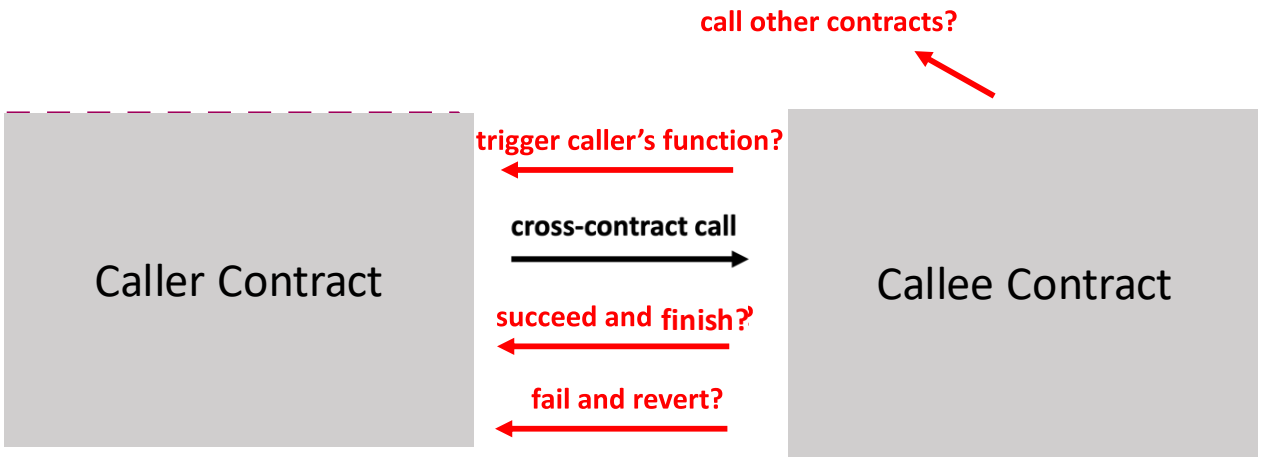
Payment Statement (External Contract Call)

```
contract Intermediary {
  uint256 public fee;
  address public seller;
  address public owner;

  function Intermediary() {
    owner = msg.sender;
// seller initialization is omitted
    fee = 10;
  }
```

A sample smart contract.

```
  function purchase() {
// msg.value is how much Ether was sent by user
// transfer pays (msg.value-fee) to the seller
    owner.transfer(msg.value - fee);
  }
  function setFe      256 _fee) {
    if (msg.sender      ner)
      fee = _fee;
  }
}
```

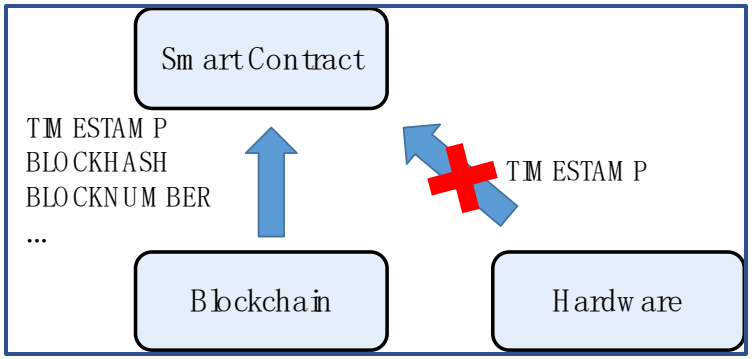
A sample smart contract (cont'd).



External contract call → another source of "nondeterminism", since callee behavior is unpredictable (e.g., could "re-enter" caller functions).

Block and Transaction State Dependency

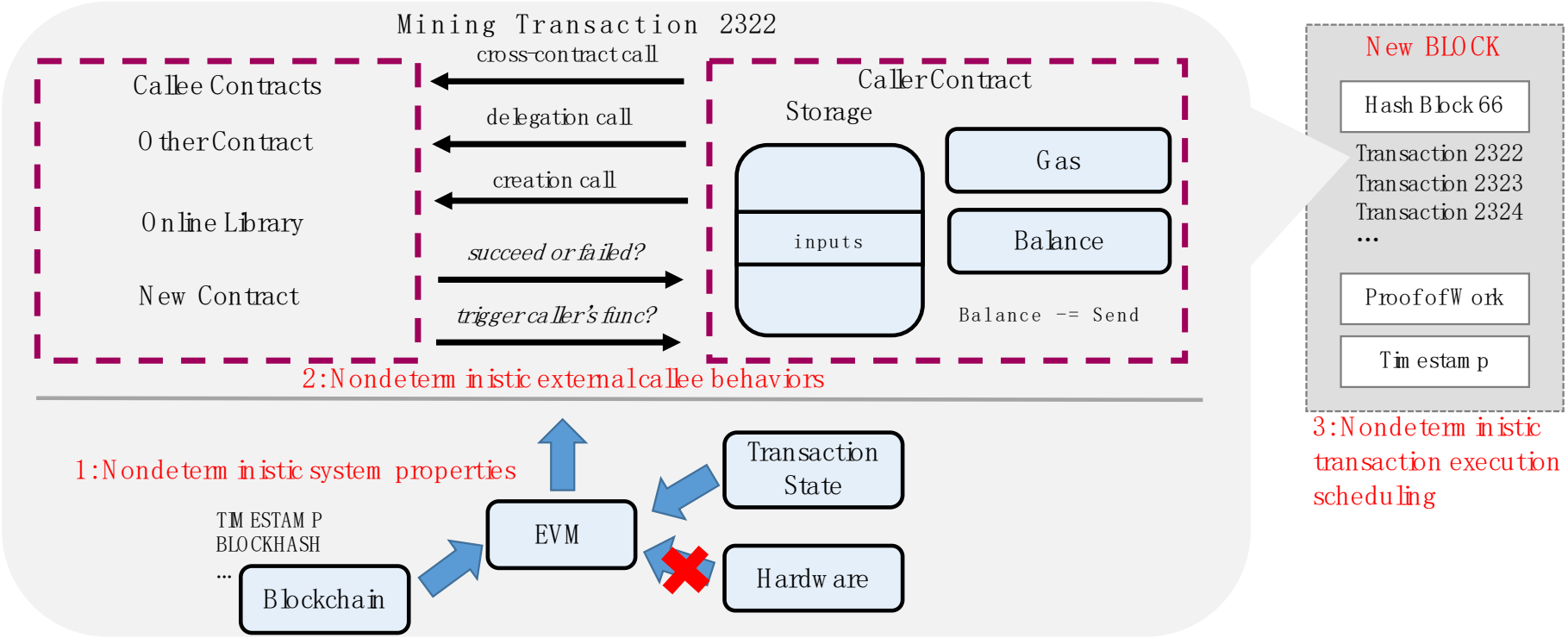
```
contract RandomReward {
  uint256 constant private salt = block.timestamp;
  uint256 constant private threshold = 1000;

  function buggy_reward(uint256 bet) public {
      uint256 t = salt * block.timestamp/(salt % 5) ;
      if (t > threshold)
        msg.sender.send.value(bet * 100)().
  }
}
```



Compute random number

- Easy for a unified design and keep consensus across different physical machine.
- But not real random!
  - Another source of nondeterminism since, for instance, system properties can be manipulated by (malicious) miners.

Non-Determinism are Everywhere in Ethereum



And it is the **ROOT CAUSE** of much confusions to the user and even enable **many common payment bugs**!

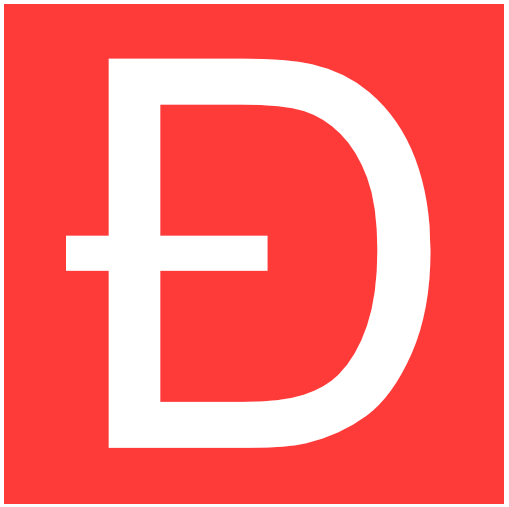
At the end of the day, doing smart contract program is not the same as writing JS/Python.

Common Attacks toward Smart Contracts

DAO

(Decentralized Autonomous Organization)

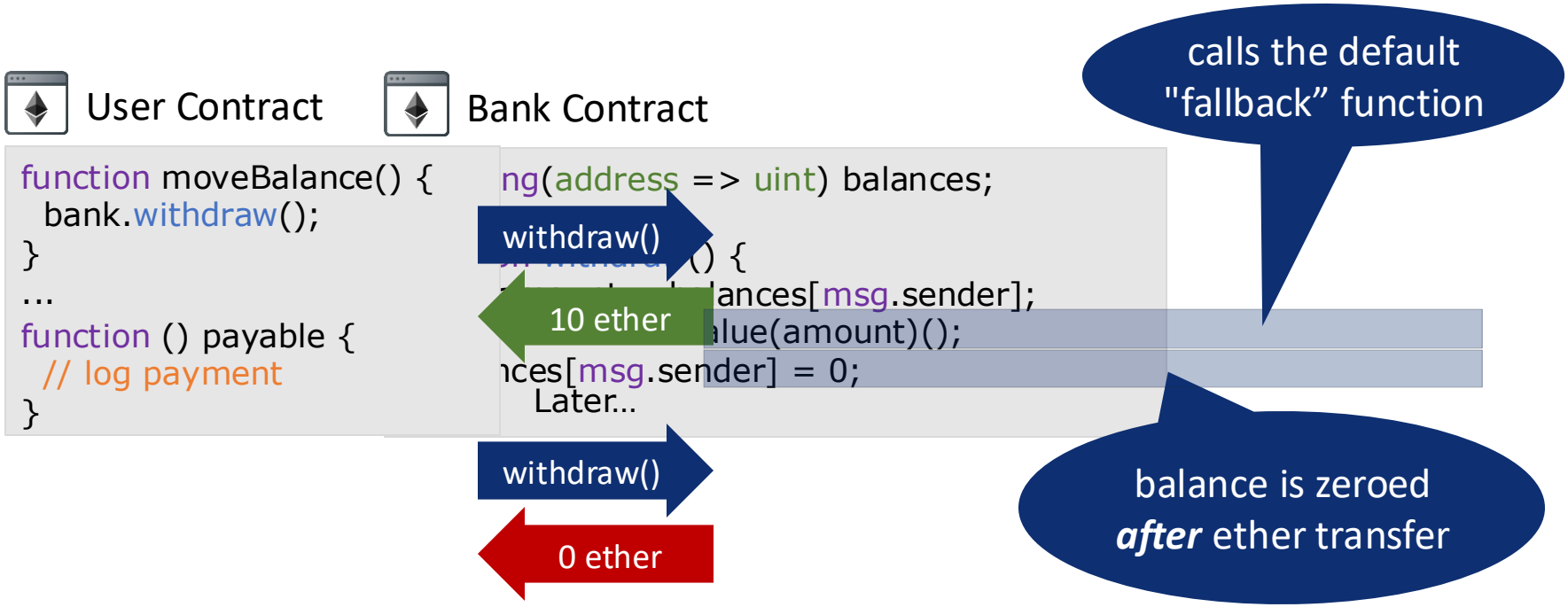
- Fully automated investor-directed venture capital fund
- Largest crowd fund in history[1]
- Held ETH 11.5 million
- Open-source



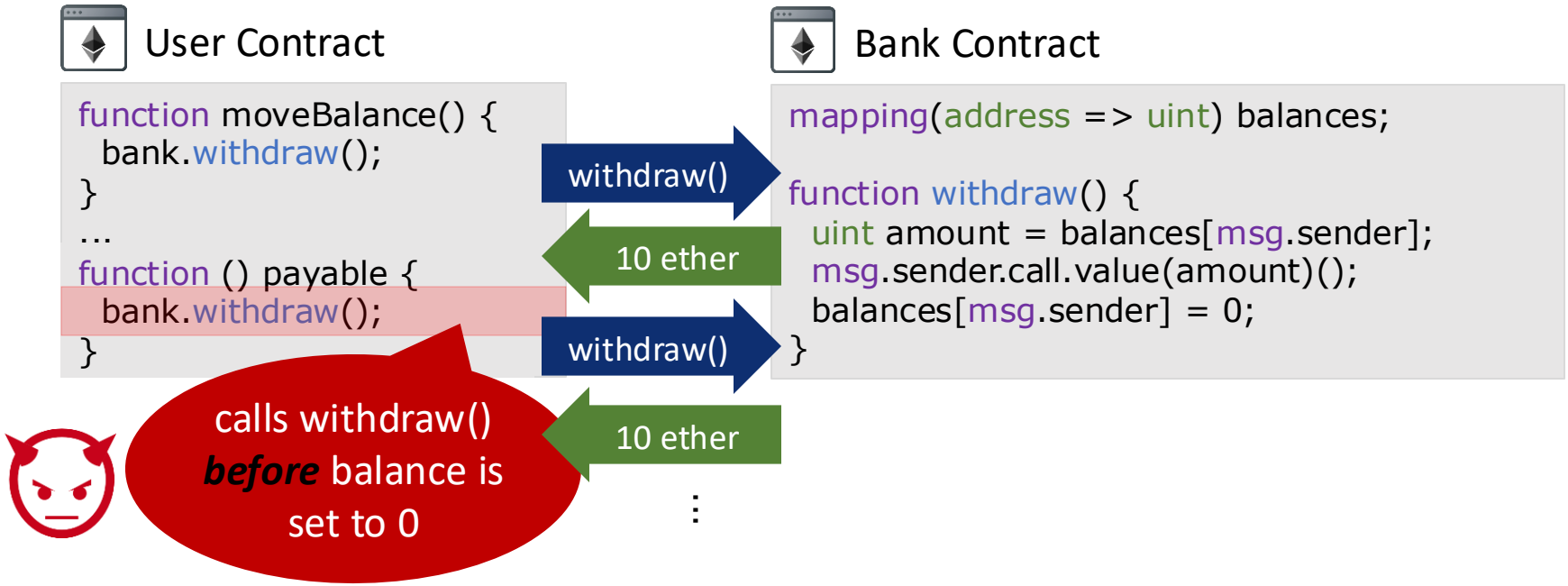
[1]https://www.cnbc.com/2016/05/17/automated-company-raises-equivalent-of-120-million-in-digital-currency.html

Source: https://en.wikipedia.org/wiki/File:The-dao-logo.png

DAO Attack

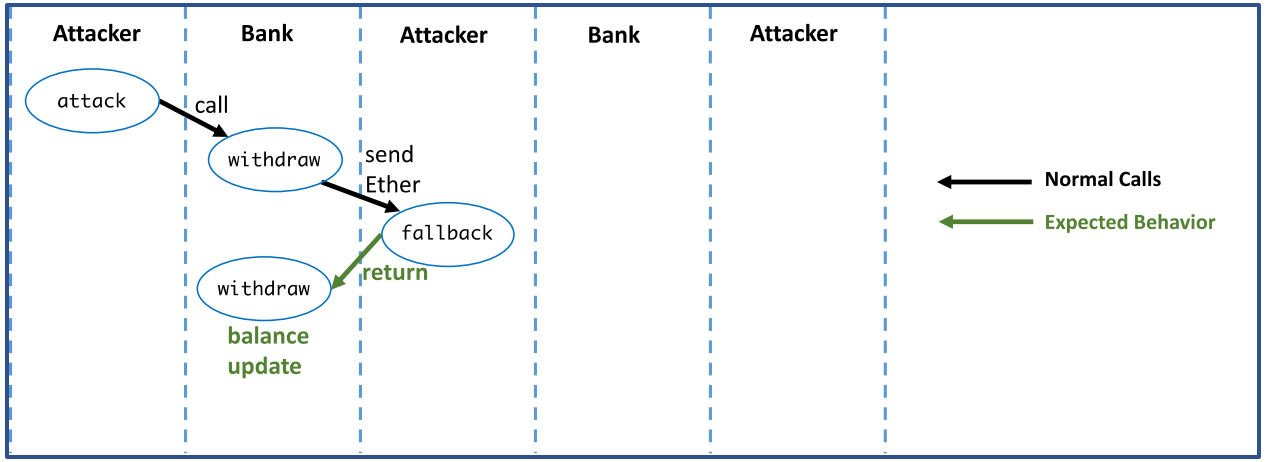

User Contract
Bank Contract
function moveBalance() {
bank.withdraw();
}
...
function () payable {
// log payment
}
withdraw()
10 ether
withdraw()
0 ether
calls the default "fallback" function
balance is zeroed *after* ether transfer

DAO Attack

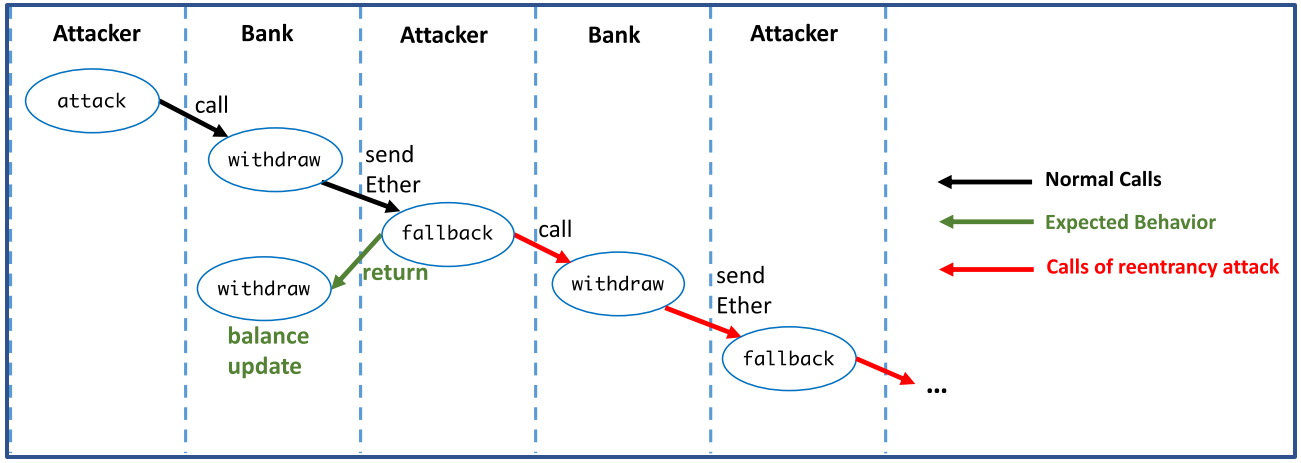

User Contract
function moveBalance() {
bank.withdraw();
}
...
function () payable {
bank.withdraw();
}
Bank Contract
mapping(address => uint) balances;
function withdraw() {
uint amount = balances[msg.sender];
msg.sender.call.value(amount)();
balances[msg.sender] = 0;
}
withdraw()
10 ether
withdraw()
10 ether
calls withdraw() before balance is set to 0

Comparison on the Workflow

What the bank is anticipating…



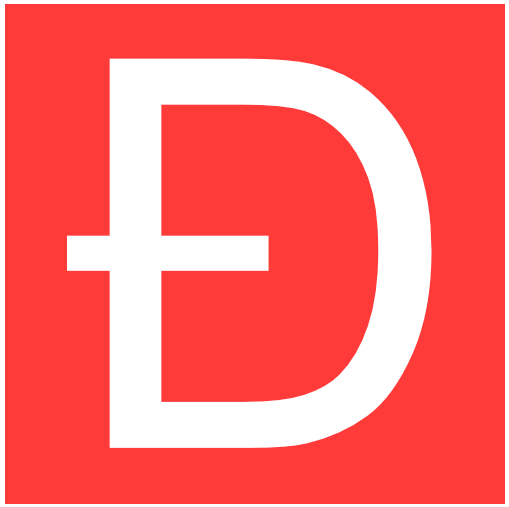
What's really going on…



DAO Attack

(June 2016)

- Recursive reentrancy attack
- Stole 3.6M Ether
- Resulted in market crash
- Let to a hard fork



Transaction Ordering Dependence (TOD) Attack

```
contract TransactionOrdering {
 uint256 price; address owner;

 function estimate(uint256 amount) {
   cost = price * amount
   return cost;
 }

 function setPrice(uint256 _price) {
   // owner can set the price.
   if (msg.sender == owner)
     price = _price;
 }

 function purchase(uint256 money) {
   return money / price;
 }
}
```

What can go wrong?

The “transactionOrdering” is just a demo of TOD attack; don’t take its functionality too seriously.

Block State Dependence Attack

```
contract RandomReward {
  uint256 constant private salt = block.timestamp;
  uint256 constant private threshold = 1000;

  function buggy_reward(uint bet) public {
      //get the best seed for randomness?
      uint256 t = salt * block.timestamp/(salt % 5)
      if (t > threshold)
        msg.sender.send.value(bet * 100)().
  }
}
```

A malicious miner can control the “timestamp”

Therefore can decide which callee should be rewarded.

Unexpected Revert Attack

Also known as “Failed External Call”

```
address[] private refundAddresses;
mapping (address => uint) public refunds;

function refundAll() public {
  for(uint x; x < refundAddresses.length; x++) {
    // now a single failure on send will hold up all funds
    require(refundAddresses[x].send(refunds[refundAddresses[x]]))
  }
}
```

What could go wrong?

You should not let the results of one money transfer to affect other transfers, unless you really know what you are doing.

Ownership Transfer

- No built-in privilege checks
- Ownership can be transferred to anyone
- Parity #1 hack stole $30M

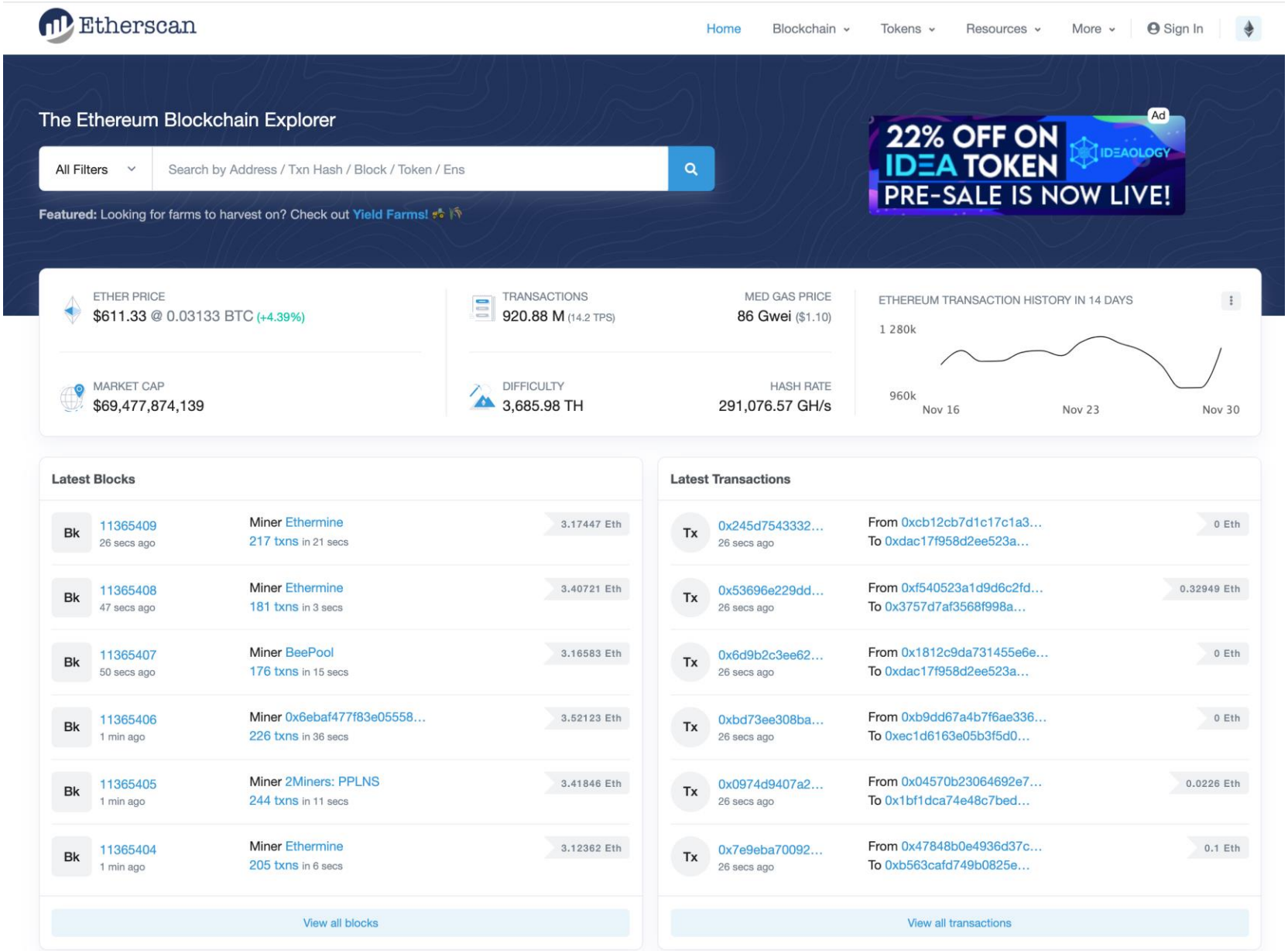
```
contract OwnableWallet {
  address owner;

  // called by the constructor
  function initWallet(address _owner) {
    owner = _owner; // any user can change owner
    // more setup
  }

  // function that allows the owner to withdraw ether
  function withdraw(uint _amount) {
    if (msg.sender == owner) {
      owner.transfer(_amount);
    }
  }
  // ...
}
```

Functionality Delegation

- Reliance on library for critical functionality
- Can freeze assets
- Parity bug #2 froze $280M

```
contract Wallet {
  // fixed address of the wallet library
  address constant walletLibrary = ...;

  // function that receives ether
  function deposit() payable {
    log(msg.sender, msg.value);
  }

  // function for withdrawing ether
  function withdraw() {
    walletLibrary.delegatecall(msg.data);
  }

  // ...
} // No guaranteed ether transfer
```

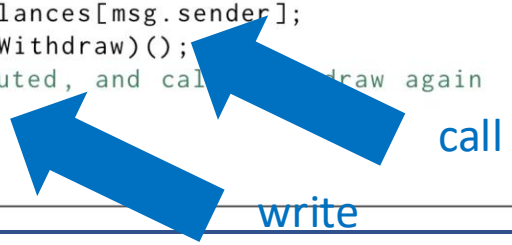
Well, once you deploy, you cannot “patch” the code, that’s the issue.



https://etherscan.io/

Existing Work: "Vulnerability Checklist"-Based Detection

```
contract Attack {
  function attack() { bank.withdraw(); }
  function () public payable { bank.withdraw(); }
}

contract Bank {
  mapping (address => uint) private userBalances;

  function withdraw() public {
    uint amountToWithdraw = userBalances[msg.sender];
    msg.sender.call.value(amountToWithdraw)();
    // the attacker's code is executed, and cal      draw again
    userBalances[msg.sender] = 0;
  }
}
```



- "No Writes After Calls (NW)"
- "Path condition for the execution before the CALL is executed. We then check if such condition with updated variables"
- …

Problem of "Vulnerability Checklist"-Based Detection

But "vulnerability checklist" based approach can miss subtle issues and require expert's continues effort to update the list

- New reentrancy vulnerabilities do not follow "no write after call" pattern → after Constantinople upgrade of Ethereum
- Unknown vulnerabilities are definitely not included until security breaches.

Still an open research problem…