Securify: Practical Security Analysis of Smart Contracts

https://securify.ch

Dr. Petar Tsankov
Scientific Researcher, ICE center, ETH Zurich
Co-founder and Chief Scientist, ChainSecurity AG
http://www.ptsankov.com/
@ptsankov
Inter-disciplinary and inter-department research center at ETH Zurich

http://ice.ethz.ch
Research @ ICE

Programmable networks

Blockchain security

Safe and interpretable AI

Security and privacy
Research @ ICE

Programmable networks

Blockchain security

Safe and interpretable AI

Security and privacy
What is a Smart Contract?

- Small programs that handle cryptocurrencies
- Written in high-level languages (e.g., Solidity, Vyper)
- Executed on the blockchain (e.g., Ethereum)
- Usually no patching after release

What can happen when programs handle billions worth of USD?
Smart Contract **Security Bugs** in the News

- **The DAO Attack**
  - $60 Million Ether Frozen
  - Wallet bug freezes more than $150 million worth of Ethereum

- **Ethereum**

- **ICO**
  - **The DAO**
  - **Critically Bug Found in ICON Smart Contract**
  - **Token Transfers Disabled**
  - Latest news update: June 16, 2018

- **CNBC**
  - **Cybersecurity**
  - **Breaking News: $32 Million Worth of Digital Currency Ether Stolen by Hackers**
  - Latest news update: July 20, 2017

- **Security**
  - **1 Million Worth of Ethereum another hacker attack**
  - **Company Parity**
  - **Warning of vulnerability in its wallet software**
  - Latest news update: July 19, 2017

**2 days ago**
June 2016: The DAO hack
The DAO hack: Reentrancy

Can the user contract withdraw more than its balance?

**User Contract**

```solidity
function moveBalance() {
    bank.withdraw();
    ...
}
```

**Bank Contract**

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

**Can the user contract withdraw more than its balance?**

- **Yes**: The user contract could withdraw more than its balance.

**Explanation**

When the user contract calls the `withdraw()` function, it transfers ether to the contract, and the `balances[msg.sender]` is zeroed after the transfer, indicating the balance is zeroed after the ether transfer.

**Diagram**

The diagram illustrates the reentrancy attack where the user contract calls the `bank.withdraw()` function, which in turn calls the `withdraw()` function of the user contract, allowing for the withdrawal of more ether than the current balance.

**Calls the default "fallback" function**

**Balance is zeroed after ether transfer**
The DAO hack: Reentrancy

User Contract

```solidity
function moveBalance() {
    bank.withdraw();
    ...
    function () payable {
        bank.withdraw();
    }
}
```

Bank Contract

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

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

An attacker used this bug to steal 3.6M ether (> 1B USD today)
July 2017: Parity Multisig Bug 1
Parity Multisig **Bug 1**: Unprivileged Write to Storage

Wallet Contract

```solidity
address owner = ...;

function initWallet(address _owner) {
    owner = _owner;
}

function withdraw(uint _amount) {
    if (msg.sender == owner) {
        msg.sender.transfer(_amount);
    }
}
```

Any user may change the wallet’s owner

Only the owner can withdraw ether

An attacker used a similar bug to **steal $30M** in July 2017
Four months later... Parity Multisig Bug 2
Parity Multisig **Bug 2**: Frozen Wallets

Wallet Contract

```solidity
address walletLibrary = ... // address

function() payable {
    walletLibrary.delegatecall(msg.data);
}

function withdraw(uint amount) {
    walletLibrary.delegatecall(msg.data);
}
```

Wallet Library

```solidity
function() payable {
    // fallback
}

function withdraw(uint amount) {
    // withdraw funds
}
```

However, in Ethereum, smart contracts can be killed!
Parity Multisig Bug 2: Frozen Wallets

A user froze $170M by deleting the wallet library

An attacker deleted the library

No withdraws are possible
### Relevant Security Properties...

<table>
<thead>
<tr>
<th>Icon</th>
<th>Security Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>🎈</td>
<td>Unexpected ether flows</td>
</tr>
<tr>
<td>🐛</td>
<td>Insecure coding, such as unprivileged writes</td>
</tr>
<tr>
<td>🕵️‍♂️</td>
<td>Use of unsafe inputs (e.g., reflection, hashing, ...)</td>
</tr>
<tr>
<td>⌚️</td>
<td>Reentrant method calls (e.g., DAO bug)</td>
</tr>
<tr>
<td>⬉️</td>
<td>Manipulating ether flows via transaction reordering</td>
</tr>
</tbody>
</table>

Many of these are nontrivial trace-/hyper-properties
Automated Security Analysis of Smart Contracts: Challenges and Gaps
Security Analysis (high-level view)

Minor issue 😊: Smart contracts are written in Turing-complete languages
Automated Security Solutions

Truffle
- Testing
  - Report true bugs
  - Can miss bugs

Oyente, Mythril, MAIAN
- Dynamic (symbolic) analysis
  - Report true bugs
  - Can miss bugs

WANTED: Automated Verifier
- Can report false alarms
- No missed bugs

Bug finding
Verification
Domain-Specific Insight:

When contracts satisfy/violate a property, they often also satisfy/violate a much simpler property.
Example: The DAO Hack

Security property
No state changes after call instructions

Hard to verify in general

Compliance pattern
No writes to storage may follow call instructions

Verifies 91% of all deployed contracts

Easier to check automatically

Violation pattern
A write to storage must follow call instructions

```solidity
function withdraw() {
    uint amount = balances[msg.sender];
    msg.sender.call.value(amount)();
    balances[msg.sender] = 0;
}
```
Classifying Behaviors using Compliance and Violation Patterns

All unsafe behaviors are reported
A practical **verifier** for Ethereum smart contracts:
- fully-automated
- extensible
- scalable
- precise
- publicly available
Beta version released in Fall 2017
- Regularly used by auditors to perform professional security audits

**New release** coming up very soon

- 95% positive feedback
- >8K uploaded smart contracts
- >800 users signed up for updates

Interesting discussions on Reddit

---

Seems almost too good to be true :) What are the limitations and how exactly does it work under the hood?

It's great that the authors of the tool are aware they are pushing a set of behaviors in the growing direction. That's the way to handle safety properties without false-negatives. I'm interested how they compare their EVM semantics against other EVM implementations in the wild.

Please, someone, humour a layman: how can a Turing complete language be formally verified?

I thought formally verifiable languages were necessarily not Turing complete, and we can therefore not formally verify Solidity.
Securify: Under the Hood

EVM Bytecode

<table>
<thead>
<tr>
<th>EVM Bytecode</th>
<th>Intermediate Representation</th>
<th>Semantic Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00: 60</td>
<td>x = Balance</td>
<td>MemTag(0x20, Balance)</td>
</tr>
<tr>
<td>02: 5b</td>
<td>y = 0x20</td>
<td>MemTag(0x40, Const)</td>
</tr>
<tr>
<td>04: 42</td>
<td>If (x == 0x00)</td>
<td>VarTag(z, Const)</td>
</tr>
<tr>
<td>06: 80</td>
<td>MStore(y, x)</td>
<td>VarTag(k, Gas)</td>
</tr>
<tr>
<td>08: 90</td>
<td>z = y</td>
<td>Assign(s, 0x20)</td>
</tr>
<tr>
<td>0a: 56</td>
<td>goto 0x42</td>
<td>Call(s, k)</td>
</tr>
</tbody>
</table>

Fully automated, sound, scalable, extensible
Securify: Under the Hood

EVM Bytecode

Decompile

Intermediate Representation

00: x = Balance
02: y = 0x20
04: If (x == 0x00)
06: MStore(y, x)
08: z = y
0a: goto 0x42

:
From EVM to CFG over SSA

Decompiling EVM bytecode:

- Convert into **static single assignment form** (each variable is assigned once)
- Perform **partial evaluation** (to resolve jump destination, memory/storage offsets)
- Identify and inline methods (to enable context-sensitive analysis)
- Construct **control-flow graph**
Which facts are relevant for verifying smart contracts?
## Semantic Facts

Many properties can be checked on the contract’s dependency graph

<table>
<thead>
<tr>
<th><strong>Flow dependencies</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>MayFollow($l, l'$)</td>
</tr>
<tr>
<td>MustFollow($l, l'$)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Data dependencies</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>MayDepOn($x, t$)</td>
</tr>
<tr>
<td>DetBy($x, t$)</td>
</tr>
</tbody>
</table>

A tag can be an instruction (e.g. Caller) or a variable

The inference of all semantic facts is declaratively specified in Datalog
Example: *MayFollow*

\[ \text{MayFollow}(i, j) \leftarrow \text{Follow}(i, j) \]
\[ \text{MayFollow}(i, j) \leftarrow \text{Follow}(i, k), \text{MayFollow}(k, j) \]

Datalog input:

1: \( x := 10 \)
2: \( y := x + 20 \)
3: \( y \leftarrow \)
4: return
5: \( y := 0 \)
6: return

Datalog fixpoint:

MayFollow(1,2)
MayFollow(1,3)
MayFollow(1,4)
MayFollow(1,5)
MayFollow(1,6)
MayFollow(2,3)
MayFollow(3,4)
MayFollow(2,5)
MayFollow(5,6)
Deriving MayDepOn

1: \( x := \text{Balance} \)
2: \( \text{Mstore}(0x20, x) \)
3: \( y := \text{MLoad}(0x20) \)
4: \( z := x + y \)

MayDepOn\((x, t)\) ← Assign\((x, t)\)
MayDepOn\((x, t)\) ← \(\text{Op}(\_, x, x'), \text{MayDepOn}(x', t)\)
MayDepOn\((x, t)\) ← \(\text{MLoad}(l, x, o), \text{isConst}(l, o), \text{MemTag}(l, o, t)\)
MayDepOn\((x, t)\) ← \(\text{MLoad}(l, x, o), \neg \text{isConst}(l, o), \text{MemTag}(l, \_, t)\)

MemTag\((l, o, t)\) ← \(\text{MStore}(l, o, x), \text{isConst}(o), \text{MayDepOn}(x, t)\)
MemTag\((l, T, t)\) ← \(\text{MStore}(l, o, x), \neg \text{isConst}(o), \text{MayDepOn}(x, t)\)
MemTag\((l, o, t)\) ← \(\text{Follows}(l, l'), \text{MemTag}(l', o, t), \neg \text{MStore}(l', o, \_)\)
Securify: Under the Hood

Semantic Representation

- MemTag(0x20, Balance)
- MemTag(0x40, Const)
- VarTag(z, Const)
- VarTag(k, Gas)
- Assign(s, 0x20)
- Call(s, k)

Check patterns
Patterns DSL

(Labels) \[ l ::= \text{(labels)} \]

(Vars) \[ x ::= \text{(variables)} \]

(Tags) \[ t ::= l \mid x \]

(Instr) \[ n ::= \text{Instr}(l, x, \ldots, x) \]

(Facts) \[ f ::= \text{MayFollow}(l, l) \mid \text{MustFollow}(l, l) \]
\[ \quad \mid \text{MayDepOn}(x, t) \mid \text{DetBy}(x, t) \]

(Patterns) \[ p ::= f \mid \forall n: p \mid \exists n: p \mid p \land p \mid \neg p \]
Detecting the DAO Hack

Security property: No state changes after call instructions

Compliance pattern: \( \text{Call}(l, \_ \_ \_, \_ \_)_ : \neg \exists \text{SStore}(l', \_ \_ \_, \_ \_). \text{MayFollow}(l, l') \)

Violation pattern: \( \text{Call}(l, \_ \_ \_, \_ \_)_ : \exists \text{SStore}(l', \_ \_ \_, \_ \_). \text{MustFollow}(l, l') \)

Proofs establish a formal logical relation between the property and its patterns
Detecting Unrestricted Writes

Security property: No storage offset is writable by all users

Compliance pattern

Violation pattern

Unrestricted write

Formalized as a hyperproperty

```
address owner = ...;

function initWallet(address _owner) {
    owner = _owner;
}
```
How well does this approach work in practice?
Securify vs. Existing Solutions

State-of-the-art security checkers for Ethereum smart contracts
- Oyente
- Mythril

Dataset
- 80 open-source smart contracts

Experiment
- Run contracts using Securify, Oyente, and Mythril
- Manually inspect each reported vulnerability
Securify vs. Oyente vs. Mythril

-60 %
-40 %
-20 %
0%
20%
40%
60%

Tr ue war nings
False war nings
Vô lation
Unrepo rted vulner abilité s

Securify
Oyente
Securify
Oyente
Mythril
Securify
Oyente
Mythril
Securify
Mythril

Transaction reordering
Reentrancy
Handled exception
Restricted transfer
Securing the blockchain

https://chainsecurity.com

Research

- SECURE, RELIABLE, INTELLIGENT SYSTEMS LAB
  - http://ai2.ethz.ch
- ISC
  - http://securify.ch
- DEGUARD
  - http://apk-deguard.com
- JSNICE
  - http://isnice.org
- PSI SOLVER
  - http://psisolver.org
- EVENTRACER
  - http://eventracer.org

Start-ups

- CHAINSECURITY
  - Securing the blockchain
  - https://chainsecurity.com
  - jobs@chainsecurity.com
  - contact@chainsecurity.com
  - @chain_security