Securify: Practical Security Analysis of Smart Contracts
https://securify.ch
What is a smart contract?

- Small programs that handle cryptocurrencies
- Written in high-level languages (e.g., Solidity)
- Usually no patching after release

What can go wrong when programs handle billions worth of USD?
Smart contract **bugs** in the news
Smart contract **bugs** in the news
Smart contract **bugs** in the news
June 2016: The DAO hack
The DAO hack

**User Contract**

```javascript
function moveBalance() {
  dao.withdraw();
}
...
function () payable {
  // log payment
}
```

**DAO Contract**

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

withdraw() → 10 ether

Later...

withdraw() → 0 ether
The DAO hack

**User Contract**

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

function () payable {
    // log payment
}
```

**DAO Contract**

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

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

calls the default "fallback" function
balance is zeroed after transfer

withdraw() 10 ether

Later...

withdraw() 0 ether
The DAO hack

User Contract

function moveBalance() {
    dao.withdraw();
}

... function () payable {
    dao.withdraw();
}

calls withdraw() before balance is set to 0

DAO Contract

mapping(address => uint) balances;

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

In 2017, more than $300M have been lost due to these issues

<table>
<thead>
<tr>
<th>Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unexpected ether flows</td>
</tr>
<tr>
<td>Unprivileged writes</td>
</tr>
<tr>
<td>Use of unsafe inputs</td>
</tr>
<tr>
<td>Reentrant method calls</td>
</tr>
<tr>
<td>Transaction reordering</td>
</tr>
</tbody>
</table>
Wanted: Automated security analysis
The DAO hack

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

**Security property:** No state changes after call instructions

Can we automatically find all unsafe calls?
The DAO hack

Security property: No state changes after call instructions

Can we automatically find all unsafe calls?
No, smart contracts are Turing-complete
The DAO hack

Security property: No state changes after call instructions

Can we automatically find all unsafe calls?
No, smart contracts are Turing-complete

Existing solutions focus on bug finding and can miss issues
When contracts satisfy/violate a security property, they often satisfy/violate a simpler property.
The DAO hack

Security property: No state changes after call instructions

Violation pattern: A write always follows call.value()

Compliance pattern: No writes may follow call.value()

Verifies 91% of all calls

function withdraw() {
    uint amount = balances[msg.sender];
    msg.sender.call.value(amount)();
    balances[msg.sender] = 0;
}
Scalable and fully *automated verifier* for Ethereum smart contracts
Impact

Used daily by security auditors (29K+ contracts scanned so far)

1K+ subscribers

Grants:

New startup:
Securify: System overview

1. decompile

EVM bytecode

Intermediate representation

2. infer facts

Security report

Semantic representation

3. check patterns

push 0x04
datastore
push 0x08
jump
jumpdest
stop
jumpdest

1: a = 0x04
2: b = load(a)
3: abi_00(b)
4: stop
   abi_00(b)
5: c = 0x00
6: sstore(c,b)

assign(1, a, 0x04)
follow(2, 1)
mayDepOn(b, a)
load(2, b, a)
follow(3,2)
follow(5,3)

证券交易序：4

1. decompile

2. infer facts

Security report

Semantic representation
Step 1: Decompilation

1. decompile

EVM bytecode

- push 0x04
data_load
push 0x08
jump
jumpdest
stop
jumpdest

Intermediate representation

- Static single assignment form
- Control-flow graph recovery

1: a = 0x04
2: b = load(a)
3: abi_00(b)
4: stop
   abi_00(b)
5: c = 0x00
6: sstore(c, b)

...
Step 2: Inferring semantic facts

Intermediate representation

1: a = \text{0x04}
2: b = \text{load}(a)
3: \text{abi}_00(b)
4: \text{stop}
   \text{abi}_00(b)
5: c = \text{0x00}
6: \text{sstore}(c, b)

Semantic representation

\text{assign}(1, a, 0x04)
\text{follow}(2, 1)
\text{mayDepOn}(b, a)
\text{load}(2, b, a)
\text{follow}(3, 2)
\text{follow}(5, 3)

2. infer facts
Step 2: Inferring semantic facts

Scalable inference of semantic facts using Datalog solvers

Datalog program

```
MayFollow(i, j) ← Follow(i, j)
MayFollow(i, j) ← Follow(i, k), MayFollow(k, j)
```

Datalog input:

```
IR
```

1: \( a = 0x04 \)
2: \( b = \text{load}(a) \)
3: \( \text{abi}_00(b) \)
4: \( \text{stop} \)
5: \( c = 0x00 \)
6: \( \text{sstore}(c, b) \)

```
MayFollow(2, 1)
\( \text{Follow}(3, 2) \)
\( \text{Follow}(5, 3) \)
\( \text{Follow}(6, 5) \)
\( \text{Follow}(4, 6) \)
```

Datalog fixpoint:

```
MayFollow(2, 1)
\( \text{MayFollow}(3, 1) \)
\( \text{MayFollow}(4, 1) \)
\( \text{MayFollow}(5, 1) \)
\( \text{MayFollow}(6, 1) \)
\( \vdots \)
```
### Step 2: Inferring semantic facts

For real-world contracts, Securify infers 1 - 10M such facts

#### Relevant semantic facts

**Control-flow analysis**

<table>
<thead>
<tr>
<th>Relation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{mayFollow}(L_1, L_2)$</td>
<td>Instruction at label $L_1$ may follow that at label $L_2$</td>
</tr>
<tr>
<td>$\text{mustFollow}(L_1, L_2)$</td>
<td>Instruction at label $L_1$ must follow that at label $L_2$</td>
</tr>
</tbody>
</table>

**Data-flow analysis**

<table>
<thead>
<tr>
<th>Relation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{mayDepOn}(X, T)$</td>
<td>The value of $X$ may depend on tag $T$</td>
</tr>
<tr>
<td>$\text{eq}(X, T)$</td>
<td>The values of $X$ and $T$ are equal</td>
</tr>
<tr>
<td>$\text{detBy}(X, T)$</td>
<td>For different values of $T$ the value of $X$ is different</td>
</tr>
</tbody>
</table>

In IR → Datalog input → Datalog fixpoint
Step 3: Check patterns

Security report

Semantic representation

assign(1, a, 0x04)
follow(2, 1)
mayDepOn(b, a)
load(2, b, a)
follow(3, 2)
follow(5, 3)
⋯
Security patterns language

A *pattern* is a logical formula over semantic predicates:

\[ \varphi ::= \text{instr}(L, Y, X, \ldots, X) \]

\[ \mid \text{eq}(X, T) \mid \text{detBy}(X, Y) \mid \text{mayDepOn}(X, Y) \]

\[ \mid \text{follow}(L, L) \mid \text{mayFollow}(L, L) \mid \text{mustFollow}(L, L) \]

\[ \mid \exists X. \varphi \mid \exists L. \varphi \mid \exists T. \varphi \mid \neg \varphi \mid \varphi \land \varphi \]

see paper for details
Example: No writes after calls

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

Security property: $\varphi \equiv \text{“No state changes after call instructions”}$

**Compliance pattern** $\varphi_C \equiv \forall \text{call}(L_1, \_, \_). \neg \exists \text{sstore}(L_2, \_, \_). \text{mayFollow}(L_2, L_1)$

**Violation pattern** $\varphi_V \equiv \exists \text{call}(L_1, \_, \_). \exists \text{sstore}(L_2, \_, \_). \text{mustFollow}(L_2, L_1)$

We can (manually) prove that: $\varphi_C \Rightarrow \varphi$ and $\varphi_V \Rightarrow \neg \varphi$
Security report

All unsafe calls are reported as either violations or warnings.
## Patterns for relevant security 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 = $ callvalue $\land$ \text{Follow}($L_2$, $L_4$) $\land$ $L_3 \neq L_4$ $\land$ MustFollow($L_4$, $L_1$)</td>
</tr>
<tr>
<td></td>
<td>violation</td>
<td>some call($L_1$, _, _, Amount). Amount $\neq 0$ $\lor$ DetBy(Amount, data)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(some stop($L$). $\neg$MayDepOn($L$, callvalue)) $\land$ (all call(_, _, _, Amount). Amount = 0)</td>
</tr>
<tr>
<td>NW: No writes after call</td>
<td>compliance</td>
<td>all call($L_1$, _, _, _). all sstore($L_2$, _, _). $\neg$MayFollow($L_1$, $L_2$)</td>
</tr>
<tr>
<td></td>
<td>violation</td>
<td>some sstore(_, _, _, _). sstore($L_2$, _, _). MustFollow($L_1$, $L_2$)</td>
</tr>
<tr>
<td>RW: Restricted write</td>
<td>compliance</td>
<td>all sstore(_, $X$, _). DetBy($X$, caller)</td>
</tr>
<tr>
<td></td>
<td>violation</td>
<td>some sstore($L_1$, $X$, _). $\neg$MayDepOn($X$, caller) $\land$ $\neg$MayDepOn($L_1$, caller)</td>
</tr>
<tr>
<td>RT: Restricted transfer</td>
<td>compliance</td>
<td>all call(_, _, _, Amount). Amount $= 0$</td>
</tr>
<tr>
<td></td>
<td>violation</td>
<td>some call($L_1$, _, _, Amount). DetBy(Amount, data) $\land$ $\neg$MayDepOn($L_1$, caller) $\land$ $\neg$MayDepOn($L_1$, data)</td>
</tr>
<tr>
<td>HE: Handled exception</td>
<td>compliance</td>
<td>all call($L_1$, $Y$, _, _). some goto($L_2$, $X$, _). MustFollow($L_1$, $L_2$) $\land$ DetBy($X$, $Y$)</td>
</tr>
<tr>
<td></td>
<td>violation</td>
<td>some call($L_1$, $Y$, _, _). all goto($L_2$, $X$, _). MayFollow($L_1$, $L_2$) $\Rightarrow$ $\neg$MayDepOn($X$, $Y$)</td>
</tr>
<tr>
<td>TOD: Transaction ordering dependency</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>compliance</td>
<td>all call(_, _, _, Amount). $\neg$MayDepOn(Amount, sload) $\land$ $\neg$MayDepOn(Amount, balance)</td>
</tr>
<tr>
<td></td>
<td>violation</td>
<td>some sload(<em>, $Y$, $X$). some sstore(</em>, $X$, _). DetBy(Amount, $Y$) $\land$ isConst($X$)</td>
</tr>
<tr>
<td>VA: Validated arguments</td>
<td>compliance</td>
<td>all sstore($L_1$, _, $X$). MayDepOn($X$, arg)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\Rightarrow$ (some goto($L_2$, $Y$, _). MustFollow($L_2$, $L_1$) $\land$ DetBy($Y$, arg))</td>
</tr>
<tr>
<td></td>
<td>violation</td>
<td>some sstore($L_1$, _, $X$). DetBy($X$, arg)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\Rightarrow$ $\neg$(some goto($L_2$, $Y$, _). MayFollow($L_2$, $L_1$) $\land$ MayDepOn($Y$, arg))</td>
</tr>
</tbody>
</table>
Evaluation

1. Is Securify precise for relevant security properties?

2. How does Securify compare to other contract checkers?
How precise is Securify?

Dataset
- First 100 real-world contracts uploaded to https://securify.ch in 2018

Security properties
- 9 critical vulnerabilities (reentrancy, ...)

Experiment:
- Measure % of violations, safe behaviors, and warnings
- Manually classify warnings into true warnings and false warnings
How precise is Securify?

< 10% warnings for 6 out of 9 security properties
How precise is Securify?

< 10% warnings for 6 out of 9 security properties

> 90% verified

% of all potential vulnerabilities
How does Securify compare to other checkers?

![Bar chart comparing Securify to other checkers (Oyente, Mythril) on various security issues (TOD, Reentrancy, Unhandled exception, Unsafe transfer). The chart shows comparisons on violations, true warnings, false warnings, and unreported vulnerabilities.]
How does Securify compare to other checkers?

- Fewer false warnings
- > 50% false negatives

Oyente
Mythril

- Violations
- True warnings
- False warnings
- Unreported vulnerabilities
Summary

```
1: a = 0x04
2: b = load(a)
3: abi_00(b)
4: stop
5: c = 0x00
6: sstore(c,b)
```

Scalable automated analysis

```
push 0x04
data load
push 0x08
jump
jump dest
stop
```

Domain-specific patterns

Try online: https://securify.ch

High precision on real contracts