Automated Inference on Financial Security of Ethereum Smart Contracts

Wansen Wang, Wenchao Huang, Zhaoyi Meng, Yan Xiong, Fuyou Miao, Xianjin Fang, Caichang Tu, Renjie Ji

USENIX Security 2023

Presenter: Wansen Wang
Background

Wide usage
- financial industry
- Internet of Things
- ...

High value
- managing assets
- market cap of ethers keeps growing

Attractive for attackers
- June 2016, DAO, $150M
- July 2017, Parity wallet, $30M
- August 2021, Poly Network, $27M

It is necessary to guarantee the financial security of Ethereum smart contracts
Existing Security Analyzers

• Automated bug-finding tools
  ➢ support automated analysis on a great amount of smart contracts
  ➢ based on pre-defined patterns and not accurate enough

• Semi-automated verification frameworks

• Automated verifiers
Existing Security Analyzers

• Automated bug-finding tools

• Semi-automated verification frameworks
  ➢ formally verify the correctness or security of smart contracts
  ➢ require manually-defined properties

• Automated verifiers
Existing Security Analyzers

- Automated bug-finding tools
- Semi-automated verification frameworks
- Automated verifiers
  - try to provide sound and automated verification of pre-defined properties for smart contracts
  - eThor does not aim for the financial security of smart contracts
  - SECURIFY does not support solving numerical constraints
  - ZEUS has soundness issues in transforming contracts into IR
Example 1

```
contract Ex1{
    mapping(address=>uint) balances;
    constructor() public{
        balances[0x12] = 100;
    }
    function transfer (address to, uint value) public{
        uint val1 = balances[msg.sender] - value;
        uint val2 = balances[to] + value;
        balances[msg.sender] = val1;
        balances[to] = val2;
        return;
    }
}
```

- **Normal case:**
  
  \[ \text{balances[msg.sender]} -= \text{value}, \text{balances[to]} += \text{value} \]
Example 1

```solidity
contract Ex1{
    mapping(address=>uint) balances;
    constructor() public{
        balances[0x12] = 100;
    }
    function transfer(address to,uint value) public{
        uint val1 = balances[msg.sender] - value;
        uint val2 = balances[to] + value;
        balances[msg.sender] = val1;
        balances[to] = val2;  // overwrite the result of line 9
        return;
    }
}
```

- **Abnormal case:**
  
  ```
  msg.sender=to, balances[to]+=value
  ```
Questions

• How to generate properties automatically?

• How to translate contracts into models automatically?

• How to verify the properties against the models automatically?
Challenge

• There is no uniform standard for the security requirements of contracts

• Most existing automated tools define patterns or properties according to known vulnerabilities
  - The vulnerabilities that can be covered are limited to known ones
  - Even a variant of a known vulnerability may evade their detection
Automated Property Generation

Observation

• Most of the contracts are finance-related (related to ethers or tokens)

Our goal

• Analyze the financial security of smart contracts

Focus on

• ethers and tokens
Automated Property Generation

Method

• Categories
  ➢ ether-related
  ➢ token-related
  ➢ indirect-related
  ➢ non-finance-related
Automated Property Generation

Method

• Identification
  ➢ ether-related: *transfer, send, call, payable*
  ➢ token-related: *balances, ownedTokenCount*

(most token contracts use similar variable names to denote token balances)
Automated Property Generation

Method

• Property generation

  ➢ Invariant property (token-related):

\[
\sum_{a \in A_1} \text{balances} = C_1
\]
Automated Property Generation

Method

• Property generation

  ➢ Equivalence property (ether-related, token-related):

  given two sequences $A$ and $B$ consisting of the same transactions

  \[
  \begin{align*}
  \text{balances}_A(\text{adv}) &= \text{balances}_B(\text{adv}) \\
  \land \\
  \text{balance}_A(\text{adv}) &= \text{balance}_B(\text{adv})
  \end{align*}
  \]
Example: invariant property

```solidity
contract Ex1{
  mapping(address=>uint) balances;
  constructor() public{
    balances[0x12] = 100;
  }
  function transfer (address to,uint value) public{
    uint val1 = balances[msg.sender] - value;
    uint val2 = balances[to] + value;
    balances[msg.sender] = val1;
    balances[to] = val2;
    return;
  }
}
```

- Abnormal case:
  ```solidity```
  ```
  msg.sender=to, balances[to]+=value
  ```
Example: invariant property

```solidity
contract Ex1{
    mapping(address=>uint) balances;
    constructor() public{
        balances[0x12] = 100;
    }
    function transfer (address to,uint value) public{
        uint val1 = balances[msg.sender] - value;
        uint val2 = balances[to] + value;
        balances[msg.sender] = val1;
        balances[to] = val2;
        return;
    }
}
```

- Abnormal case:
  \[
  \sum \text{balances} += \text{value}
  \]
Automated Property Generation

Advantage of our properties

• Cover 6 types of vulnerabilities
  ➢ Invariant property: overflow/underflow, transferMint
  ➢ Equivalence property: reentrancy, gasless send, TD, TOD

• Not limited to known vulnerabilities
  ➢ transferMint (not supported by automated tools in our evaluation)
2-step modeling

- Generates different models according to different properties
  - Invariant property: 1-safety
  - Equivalence property: 2-safety

- Independent modeling module generates partial models of smart contracts (Written in Solidity language)

- Complementary modeling module modifies the models according to different properties
2-step modeling

- We prove the soundness of translation from Solidity language to our models based on KSolidity (a custom semantics of Solidity, IEEE S&P 2022)

**Theorem 1** (Soundness). If an invariant property (or equivalence property) holds in the complementary model of FASVERIF, it holds in real-world transactions interpreted by KSolidity semantics.
Automated Modeling and Verification

Verification

Input a property and a model

Searching for a finished execution of the model

Search fail?

yes

no

Z3

Solving numerical constraints

Constraints Satisfied?

yes

no

Current execution does not exist

Tamarin prover

Output that the property is valid

Output that the property is not valid
Evaluation

Dataset

• Vulnerability dataset: 549 contracts collected from public datasets of other works
  - transaction order dependency (TOD)
  - timestamp dependency (TD)
  - Reentrancy
  - gasless send
  - overflow/underflow
  - transferMint

• Real-world dataset: 30577 contracts crawled from Etherscan
Evaluation

Statistical analysis

- 27858/30577 finance-related contracts
- the accuracy of our method to identify token contracts is higher than 98%

<table>
<thead>
<tr>
<th>threshold</th>
<th>70</th>
<th>75</th>
<th>80</th>
<th>85</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acc(%)</td>
<td>98.31</td>
<td>98.32</td>
<td>98.32</td>
<td>98.50</td>
<td>98.46</td>
</tr>
<tr>
<td>F1(%)</td>
<td>98.13</td>
<td>98.14</td>
<td>98.14</td>
<td>98.31</td>
<td>98.27</td>
</tr>
</tbody>
</table>
• FASVERIF achieves higher accuracy and F1 values than other automated tools
• Only FASVERIF can detect all of the 6 types of vulnerabilities
Evaluation

Analysis of 1700 real-world contracts

- 10 contracts with transferMint, 3 contracts with TD

```solidity
contract Ex1{
    mapping(address=>uint) balances;
    constructor() public{
        balances[0x12] = 100;
    }
    function transfer(address to, uint value) public{
        uint val1 = balances[msg.sender] - value;
        uint val2 = balances[to] + value;
        balances[msg.sender] = val1;
        balances[to] = val2;
        return;
    }
}
```
Limitations *(Still working on them)*

- The average time to analyze a contract using FASVERIF is longer than the one using other automated tools.

- There are still some financial security properties and financial vulnerabilities that are unsupported by FASVERIF

- Solidity language is not fully supported.

- ...
Thank you for listening!

Presenter: Wansen Wang

wangws@mail.ustc.edu.cn