SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution

Sunbeom So, Seongjoon Hong, Hakjoo Oh

Korea University

USENIX Security 2021
Smart Contract

• Digital contract written in programming languages.
  
  - E.g., Decentralized Finance, food supply chain (IBM Food Trust).

• Send transactions by invoking functions in smart contracts.

```solidity
function transfer (address to, uint value) public returns (bool) {
    require (balance[msg.sender] >= value);
    balance[msg.sender] -= value;
    balance[to] += value;
    return true;
}
```

Solidity Function
Importance of Securing Smart Contracts

• **Immutable** once deployed.

• **Huge financial damage** once exploited.
Goal: Find Vulnerabilities with Concrete Scenarios

- How to demonstrate the integer underflow at line 30?
  - Sending a single transaction `burnFrom (..., 1)` will fail.

```solidity
contract Example {
    address owner;
    mapping (address => uint) balance;
    mapping (address => mapping (address => uint)) allowed;
    uint totalSupply;

    constructor () public {
        owner = msg.sender;
        totalSupply = 0;
    }

    function mintToken (address target, uint amount) public {
        require (owner == msg.sender);
        balance[target] += amount;
        totalSupply += amount;
    }

    function approve(address spender, uint value) public returns (bool) {
        allowed[msg.sender][spender] = value;
        return true;
    }

    function burnFrom (address from, uint value) public returns (bool) {
        require (balance[from] >= value);
        require (allowed[from][msg.sender] >= value);
        balance[from] -= value;
        allowed[from][msg.sender] -= value;
        totalSupply -= value;
        return true;
    }
}
```
Goal: Find Vulnerabilities with Concrete Scenarios

- How to demonstrate the integer underflow at line 30?
  - Sending a single transaction `burnFrom (... , 1)` will fail.

```solidity
contract Example {
    address owner;
    mapping (address => uint) balance;
    mapping (address => mapping (address => uint)) allowed;
    uint totalSupply;

    constructor () public {
        owner = msg.sender;
        totalSupply = 0;
    }

    function mintToken (address target, uint amount) public {
        require (owner == msg.sender);
        balance[target] += amount;
        totalSupply += amount;
    }

    function approve(address spender, uint value) public returns (bool) {
        allowed[msg.sender][spender] = value;
        return true;
    }

    function burnFrom (address from, uint value) public returns (bool) {
        require (balance[from] >= value);
        require (allowed[from][msg.sender] >= value);
        balance[from] -= value;
        allowed[from][msg.sender] -= value;
        totalSupply -= value;
        return true;
    }
}
```

Underflow (line 30)
Goal: Find Vulnerabilities with Concrete Scenarios

• Need a transaction sequence (a sequence of function invocations) of length at least 4.

In addition to simply reporting vulnerable locations, we aim to find vulnerable sequences that can demonstrate the flaws.
Goal: Find Vulnerabilities with Concrete Scenarios

- Need a transaction sequence (a sequence of function invocations) of length at least 4.

```solidity
contract Example {
    address owner;
    mapping (address => uint) balance;
    mapping (address => mapping (address => uint)) allowed;
    uint totalSupply;

    constructor () public {
        owner = msg.sender;
        totalSupply = 0;
    }

    function mintToken (address target, uint amount) public {
        require (owner == msg.sender);
        balance[target] += amount;
        totalSupply += amount;
    }

    function approve(address spender, uint value) public returns (bool) {
        allowed[msg.sender][spender] = value;
        return true;
    }

    function burnFrom (address from, uint value) public returns (bool) {
        require (balance[from] >= value);
        require (allowed[from][msg.sender] >= value);
        balance[from] -= value;
        allowed[from][msg.sender] -= value;
        totalSupply -= value;
        return true;
    }
}
```

Without sequence information, manual bug triage is needed.

In addition to simply reporting vulnerable locations, we aim to find vulnerable sequences that can demonstrate the flaws.
Challenge: Path Explosion

- Huge search space for transaction sequences.
Challenge: Path Explosion

- Huge search space for transaction sequences.

Existing approaches (e.g., Mythril, Manticore) fail to find vulnerabilities that require long sequences.
SmarTest: Language Model-Guided Symbolic Execution

- **Key Idea**: guide symbolic execution with language models, towards likely paths.
  - **Training**: using unguided symbolic execution, obtain vulnerable sequences and learn a model.
  - **Testing**: according to a learned model, prioritize sequences likely to be vulnerable.
**Goal:** construct a training corpus.

**Issue:** how to abstract transaction sequences for effective generalization.

**Our Idea:** use type information.

---

### Transaction

```solidity
function setOwner (address newOwner) {
    require (owner == msg.sender);
    owner = newOwner;
}
```

---

### Word

A variable that has address type (3rd ranked) is defined and used.

---

### Table: Type Frequency

<table>
<thead>
<tr>
<th>Type</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapping (address=&gt;uint)</td>
<td>2,100</td>
</tr>
<tr>
<td>uint</td>
<td>1,400</td>
</tr>
<tr>
<td>address</td>
<td>1,200</td>
</tr>
</tbody>
</table>

---

Mapping from types to the number of occurrences (in the collected sequences)
• Prioritize the transaction sequences, according to the computed probabilities.
Evaluation Setup

• Benchmark (Solidity): CVE (443 contracts) + Leaking-Suicidal dataset (104 contracts)
  - https://github.com/kupl/VeriSmart-benchmarks

• Compared with 5 recently-developed tools that can generate vulnerable sequences.
  - ILF [CCS ’19], Maian [ACSAC ’18], teEther [Security ’18], Mythril (ConsenSys), Manticore (Trail of Bits)

• Used 3-gram language model.

• 4-fold cross validation.
# Evaluation 1: SmarTest vs. 5 Tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Integer Overflow</th>
<th>Division by Zero</th>
<th>Assertion Violation</th>
<th>ERC20 Standard Violation</th>
<th>CVE Vulnerability Detection (Sample: 242)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmarTest</td>
<td>1982</td>
<td>203</td>
<td>77</td>
<td>654</td>
<td>219</td>
</tr>
<tr>
<td>Mythril (ConsenSys)</td>
<td>460</td>
<td>73</td>
<td>25</td>
<td>n/a</td>
<td>85</td>
</tr>
<tr>
<td>Manticore (Trail of Bits)</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>n/a</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1. Results on CVE dataset. Found and validated vulnerable sequences.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Ether-leaking (90 contracts)</th>
<th>Suicidal (53 contracts)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Contract</td>
<td>#Function</td>
</tr>
<tr>
<td>SmarTest</td>
<td>81</td>
<td>111</td>
</tr>
<tr>
<td>ILF [CCS ’19]</td>
<td>75</td>
<td>101</td>
</tr>
<tr>
<td>Maian [ACSAC ’18]</td>
<td>58</td>
<td>n/a</td>
</tr>
<tr>
<td>teEther [USENIX Security ’18]</td>
<td>37</td>
<td>n/a</td>
</tr>
<tr>
<td>Mythril (ConsenSys)</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Manticore (Trail of Bits)</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 2. Results on Leaking-Suicidal dataset. Found and validated (when available) vulnerable sequences.
Evaluation 2: Effectiveness of Using Language Model

- Baseline: SmarTest without language model (i.e., prioritize short sequences)
- 3-gram: SmarTest with 3-gram language model

Language model significantly improves the vulnerability-finding capability of symbolic execution.
Evaluation 3: Finding Zero-day Bugs

- Ran SmarTest on 2,743 contracts from Etherscan.
- Manually confirmed 7 critical vulnerabilities (ERC20 Standard Violation).

Pattern 1: Due to mistakenly named constructors, anyone can tokens for free.

```solidity
contract AToken {
    /* Constructor function */
    function BToken () public {
        balance[msg.sender] = 10000000000;
        totalSupply = 10000000000;
    }
    ...
}
```

“BToken” at line 3 should be changed to “AToken”.

Pattern 2: Unrestricted token transfer by unauthorized users.

```solidity
function transferFrom (address from, address to,
    uint value) public returns (bool) {
    require(balance[from] >= value);
    require(balance[to] + value >= balance[to]);
    require(allowed[from][msg.sender] >= value);
    balance[from] -= value;
    balance[to] += value;
    return true;
}
```

Red-colored part is absent in the original code.
Summary

- **Goal**: effectively find vulnerable transaction sequences.
- **Key Idea**: guide symbolic execution with language models, towards likely paths.

SmarTest: [http://prl.korea.ac.kr/smartest](http://prl.korea.ac.kr/smartest)
Benchmark: [https://github.com/kupl/VeriSmart-benchmarks](https://github.com/kupl/VeriSmart-benchmarks)

Thank you!