USTC
University of Science and Technology of China

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

Example1

```
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**:

  balances[msg.sender]-=value, balances[to]+=value

Example1

```
1   contract Ex1{
2       mapping(address=>uint) balances;
3       constructor() public{
4           balances[0x12] = 100;
5       }
6       function transfer (address to,uint value) public{
7           uint val1 = balances[msg.sender] - value;
8           uint val2 = balances[to] + value;
9           balances[msg.sender] = val1;
10          balances[to] = val2;
11          return;
12      }
13  }
```

overwrite the result of line 9

- **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?

Automated Property Generation

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) :

$$\sum_{a \in A_1} balances = C_1$$

Automated Property Generation

Method

- Property generation
  - Equivalence property (ether-related, token-related):

given two sequences A and B consisting of the same transactions

$$balances_A(adv) = balances_B(adv)$$
$$\land$$
$$balance_A(adv) = balance_B(adv)$$

Example: invariant property

```
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**:

msg.sender=to, balances[to]+=value

Example: invariant property

```
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;
    }
}
```

The invariant property is violated

- **Abnormal case**:

$$\sum balances += value$$

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)

Automated Modeling and Verification

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

Automated Modeling and Verification

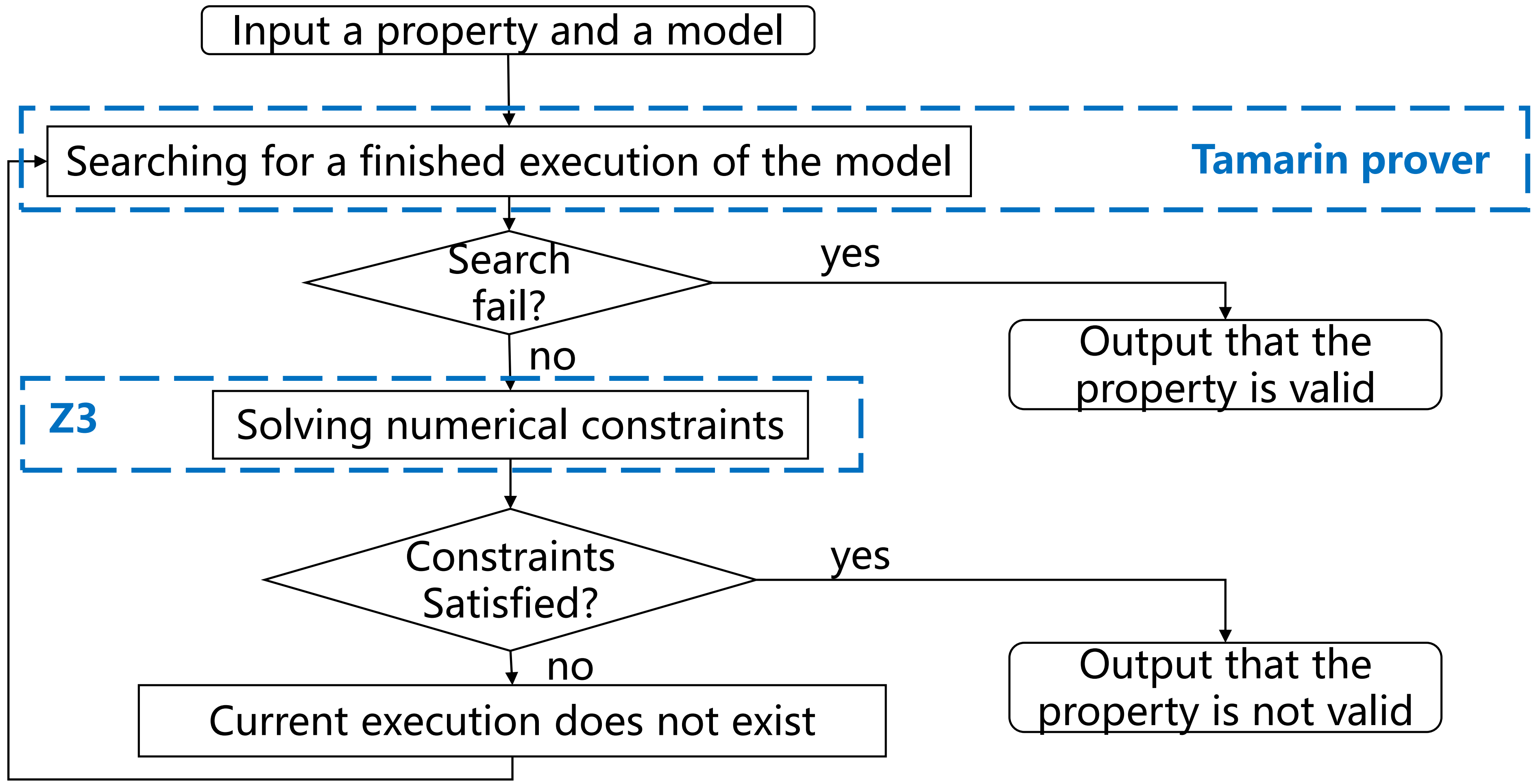
2-step modeling

- 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



Evaluation

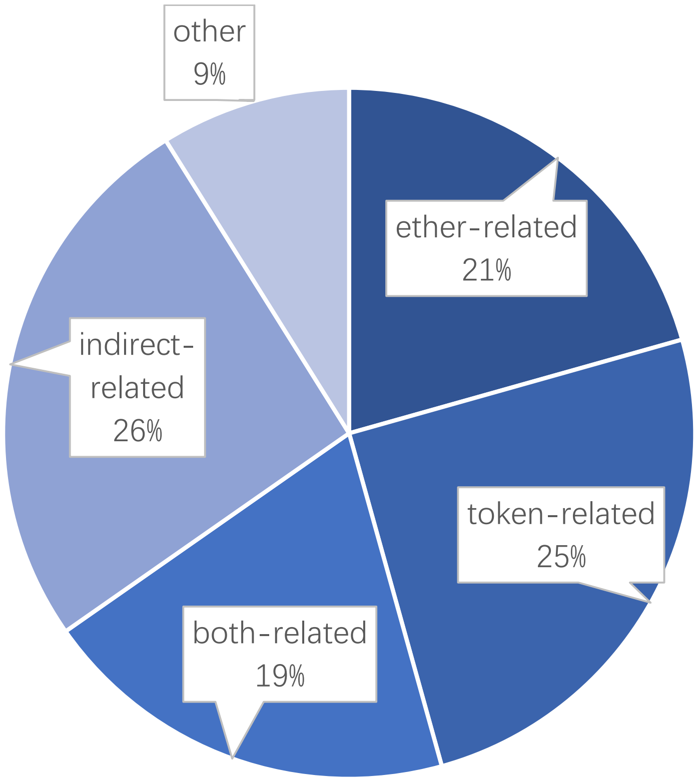
Dataset

- 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



threshold	70	75	80	85	90
Acc(%)	98.31	98.32	98.32	98.50	98.46
F1(%)	98.13	98.14	98.14	98.31	98.27

- the accuracy of our method to identify token contracts is higher than 98%

- 27858/30577 finance-related contracts

Evaluation

Comparison

Table 1: A comparison of representative automated analyzers for smart contracts. (Acc and F1 outside brackets correpsond to the finance-vulnerable contracts, while those inside brackets correpsond to the vulnerable contracts, * denote automated verifiers)

Types of Vulnerabilities	Osiris		SECURIFY*		Mythril		OYENTE		VERISMART		SmartCheck		Slither		Manticore		eThor*		FASVERIF *		
	Acc(%)	F1	Acc(%)	F1	Acc(%)	F1	Acc(%)	F1	Acc(%)	F1	Acc(%)	F1	Acc(%)	F1	Acc(%)	F1	Acc(%)	F1	Acc(%)	F1	U
TOD-eth	/	/	96.43	0.98	/	/	42.86	0.6	/	/	/	/	/	/	/	/	/	/	100	1	10
TOD-token	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	100	1	0
TD	71.60 (70.37)	0.83 (0.82)	/	/	45.68 (44.44)	0.62 (0.62)	76.54 (75.31)	0.87 (0.86)	/	/	/	/	16.05 (14.81)	0.26 (0.25)	24.69 (23.46)	0.38 (0.38)	/	/	95.06 (93.83)	0.97 (0.96)	33
reentrancy	66.67 (69.05)	0.79 (0.81)	78.57 (76.19)	0.85 (0.84)	71.42 (69.04)	0.81 (0.8)	73.81 (76.19)	0.85 (0.86)	/	/	73.81 (76.19)	0.85 (0.86)	85.71 (83.33)	0.91 (0.90)	38.09 (35.71)	0.41 (0.40)	83.72 (86.05)	0.92 (0.93)	90.48 (88.10)	0.94 (0.93)	2
gasless send	/	/	92.19	0.95	82.35	0.67	/	/	/	/	92.19	0.95	85.94	0.91	29.69	0.26	/	/	100	1	7
overflow/underflow	81.20 (81.20)	0.89 (0.89)	/	/	95.30 (95.30)	0.97 (0.97)	90.27 (90.27)	0.95 (0.95)	98.99 (98.99)	0.99 (0.99)	/	/	/	/	19.40 (19.40)	0.11 (0.11)	/	/	99.33 (99.33)	0.99 (0.99)	4
transferMint	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	100	1	0

- 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

```
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;
    }
}
```

- 10 contracts with transferMint, 3 contracts with TD

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