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., **De**centralized **Fi**nance, food supply chain (IBM Food Trust).
- Send transactions by invoking functions in smart contracts.

```
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

balance[X] = 20,
balance[Y] = 0

X sends 5 tokens to Y.

transfer(Y, 5)
with X=msg.sender

balance[X] = 15,
balance[Y] = 5

Importance of Securing Smart Contracts

- **Immutable** once deployed.
- **Huge financial damage** once exploited.

(2016)

KLINT FINLEY 06.18.16 04:30 AM

A $50 Million Hack Just Showed That the DAO Was All Too Human

(2018)

ETHEREUM › TECHNOLOGY

BatchOverflow Exploit Creates Trillions of Ethereum Tokens, Major Exchanges Halt ERC20 Deposits

Sam Town · April 25, 2018 at 10:38 pm UTC · 3 min read

(2017)

WILLIAM SUBERG NOV 08, 2017

'Accidentally Killed It': Parity Grapples With $280 Mln Locked ETH

Parity is dealing with another code vulnerability which allowed a user to block access to almost $300 mln ETH.

(2020)

DeFi Protocol bZx Hacked Again: $8 Million Worth of ETH, LINK, Stablecoins Drained (Updated)

Author: Himadri Saha • Last Updated Sep 14, 2020 @ 17:20

In yet another full-blown attack, hackers made away with crypto funds worth more than $8 million from DeFi lending protocol bZx.

Goal: Find Vulnerabilities with Concrete Scenarios

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

```
1   contract Example {
2     address owner;
3     mapping (address => uint) balance;
4     mapping (address => mapping (address => uint)) allowed;
5     uint totalSupply;
6
7     constructor () public {
8       owner = msg.sender;
9       totalSupply = 0;
10    }
11
12    function mintToken (address target, uint amount) public {
13      require (owner == msg.sender);
14      balance[target] += amount;
15      totalSupply += amount;
16    }
17
18    function approve(address spender, uint value)
19    public returns (bool) {
20      allowed[msg.sender][spender] = value;
21      return true;
22    }
23
24    function burnFrom (address from, uint value)
25    public returns (bool) {
26      require (balance[from] >= value);
27      require (allowed[from][msg.sender] >= value);
28      balance[from] -= value;
29      allowed[from][msg.sender] -= value;
30      totalSupply -= value;
31      return true;
32    }
33  }
```

Underflow (line 30)

Goal: Find Vulnerabilities with Concrete Scenarios

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

```
1   contract Example {
2     address owner;
3     mapping (address => uint) balance;
4     mapping (address => mapping (address => uint)) allowed;
5     uint totalSupply;
6
7     constructor () public {
8       owner = msg.sender;
9       totalSupply = 0;
10    }
11
12    function mintToken (address target, uint amount) public {
13      require (owner == msg.sender);
14      balance[target] += amount;
15      totalSupply += amount;
16    }
17
18    function approve(address spender, uint value)
19    public returns (bool) {
20      allowed[msg.sender][spender] = value;
21      return true;
22    }
23
24    function burnFrom (address from, uint value)
25    public returns (bool) {
26      require (balance[from] >= value);
27      require (allowed[from][msg.sender] >= value);
28      balance[from] -= value;
29      allowed[from][msg.sender] -= value;
30      totalSupply -= value;
31      return true;
32    }
33  }
```

All elements are initially zeros.

Cannot pass guards if value>0.
(since balance[from]=0, allowed[from][msg.sender]=0)

Underflow
(line 30)

Goal: Find Vulnerabilities with Concrete Scenarios

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

```
1   contract Example {
2     address owner;
3     mapping (address => uint) balance;
4     mapping (address => mapping (address => uint)) allowed;
5     uint totalSupply;
6
7     constructor () public {
8       owner = msg.sender;
9       totalSupply = 0;
10    }
11
12    function mintToken (address target, uint amount) public {
13      require (owner == msg.sender);
14      balance[target] += amount;
15      totalSupply += amount;
16    }
17
18    function approve(address spender, uint value)
19    public returns (bool) {
20      allowed[msg.sender][spender] = value;
21      return true;
22    }
23
24    function burnFrom (address from, uint value)
25    public returns (bool) {
26      require (balance[from] >= value);
27      require (allowed[from][msg.sender] >= value);
28      balance[from] -= value;
29      allowed[from][msg.sender] -= value;
30      totalSupply -= value;
31      return true;
32    }
33  }
```

Underflow (line 30)

mintToken (A, 1)
with msg.sender = owner

totalSupply = 1,
balance[A] = 1

mintToken (B, 2^{256} - 1)
with msg.sender = owner

totalSupply = 0,
balance[B] = 2^{256} - 1

Overflow at line 15

Can pass line 26

approve (C, 1)
with msg.sender = B

allowed[B][C] = 1

Can pass line 27

burnFrom (B, 1)
with msg.sender = C

Pass line 26, 27
Underflow at line 30

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.

```
1   contract Example {
2     address owner;
3     mapping (address => uint) balance;
4     mapping (address => mapping (address => uint)) allowed;
5     uint totalSupply;
6
7     constructor () public {
8       owner = msg.sender;
9       totalSupply = 0;
10    }
11
12    function mintToken (address target, uint amount) public {
13      require (owner == msg.sender);
14      balance[target] += amount;
15      totalSupply += amount;
16    }
17
18    function approve(address spender, uint value)
19    public returns (bool) {
20      allowed[msg.sender][spender] = value;
21      return true;
22    }
23
24    function burnFrom (address from, uint value)
25    public returns (bool) {
26      require (balance[from] >= value);
27      require (allowed[from][msg.sender] >= value);
28
29
30
31
32    }
33  }
```

mintToken (A, 1)
with msg.sender = owner

↓ totalSupply = 1, balance[A] = 1

mintToken (B, 2^{256} - 1)
with msg.sender = owner

↓ **totalSupply = 0,** balance[B] = 2^{256} - 1

Overflow at line 15

Can pass line 26

approve (C, 1)
with msg.sender = B

↓ allowed[B][C] = 1

Can pass line 27

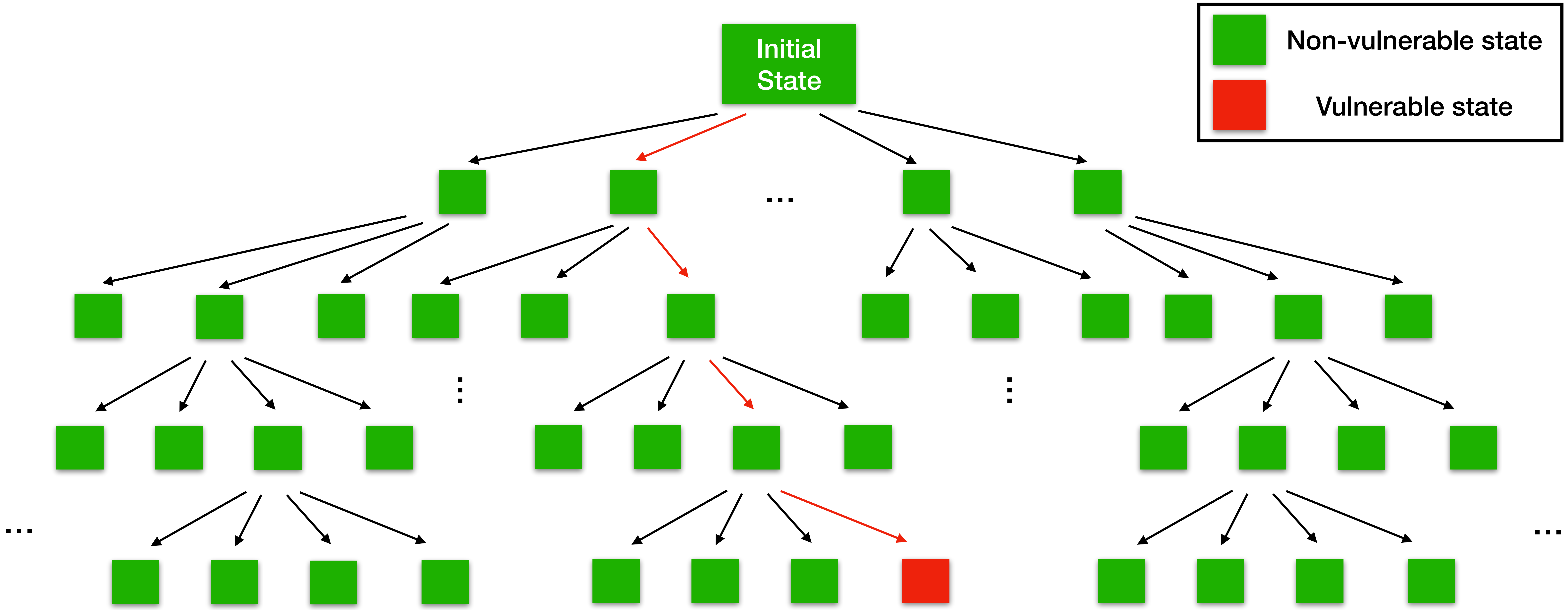
ass line 26, 27
derflow at line 30

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.

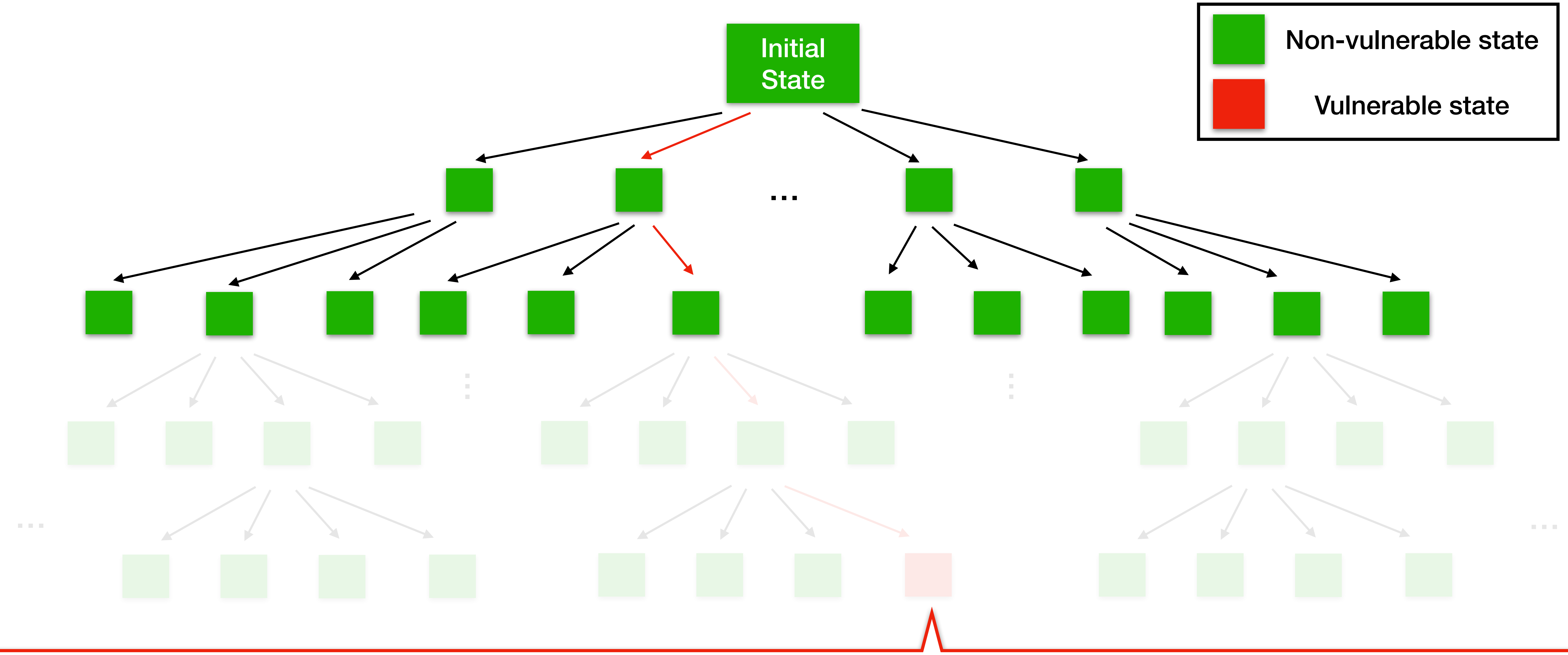
Challenge: Path Explosion

- Huge search space for transaction sequences.



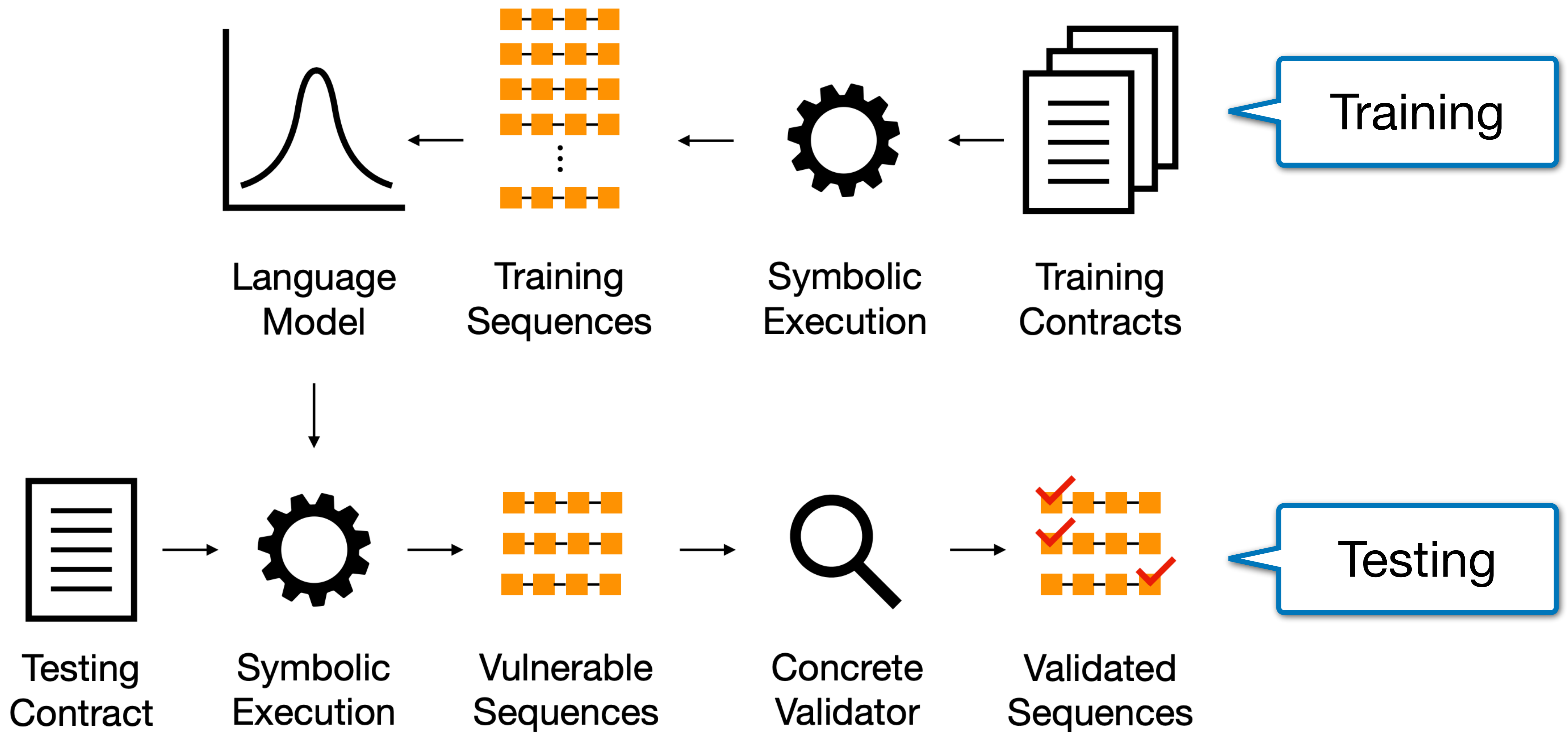
Challenge: Path Explosion

- 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.

Detail: Learning Language Model

- **Goal**: construct a training corpus.
- **Issue**: how to abstract transaction sequences for effective generalization.
- **Our Idea**: use type information.

Transaction

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

Mapping from types to
the number of occurrences
(in the collected sequences)

abstraction with
type frequency table

Type	Frequency
mapping (address=>uint)	2,100
uint	1,400
address	1,200

Word

$$\langle \underbrace{0, 0, 1}_{\textbf{Def}}, \underbrace{0, 0, 1}_{\textbf{Use}}, \ldots \rangle$$

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

Detail: Using Language Model

Transaction Sequence

$$t_0 \cdot t_1 \cdot t_2$$

Word Sequence

$$\langle s \rangle \cdot \langle s \rangle \cdot w_0 \cdot w_1 \cdot w_2$$

Pseudo-start word

Probability of Being Vulnerable

$$P(w_0 \mid s \cdot s) \times P(w_1 \mid s \cdot w_0) \times P(w_2 \mid w_0 \cdot w_1)$$

- 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

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

90.5%

35.1%

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

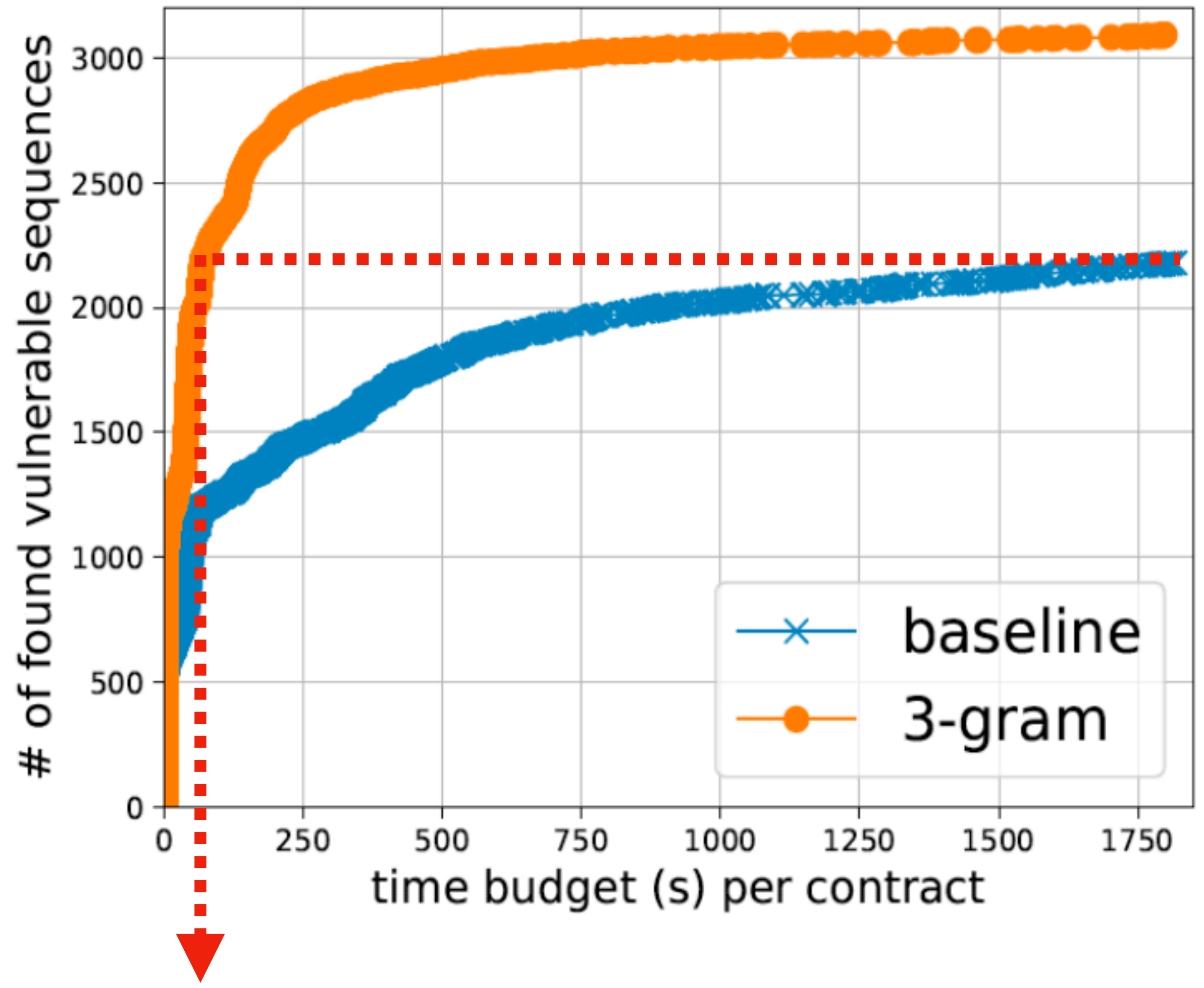
Tool	Ether-leaking (90 contracts)			Suicidal (53 contracts)		
	#Contract	#Function	#Line	#Contract	#Function	#Line
SmarTest	**81**	**111**	**111**	**51**	**51**	**51**
ILF [CCS '19]	75	101	n/a	50	50	n/a
Maian [ACSAC '18]	58	n/a	n/a	43	n/a	n/a
teEther [USENIX Security '18]	37	n/a	n/a	n/a	n/a	n/a
Mythril (ConsenSys)	7	8	8	19	19	19
Manticore (Trail of Bits)	9	9	9	3	3	3

Table 2. Results on Leaking-Suicidal dataset. Found and validated (when available) vulnerable sequences.

Evaluation 2: Effectiveness of Using Language Model

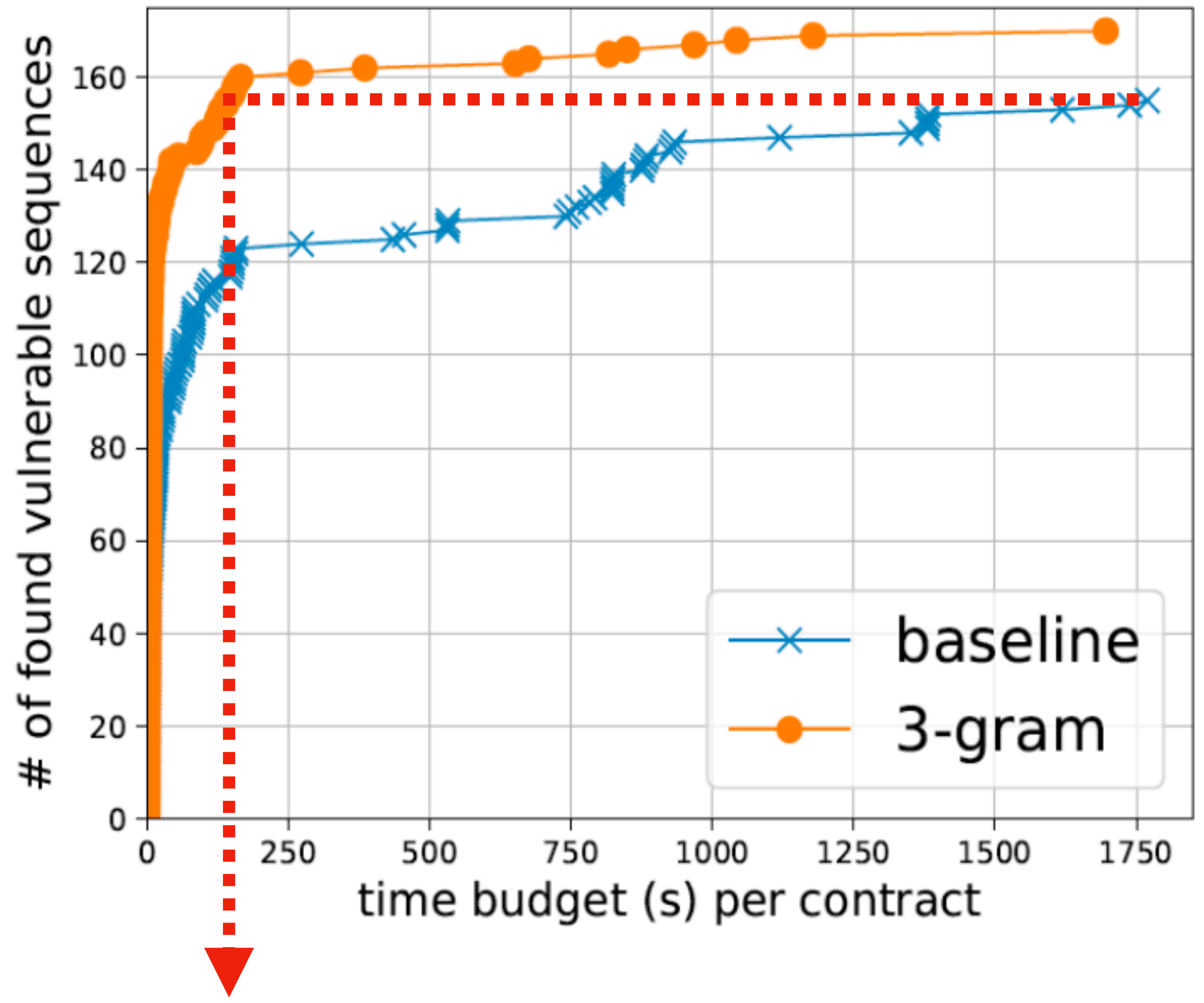
- Baseline: SmarTest without language model (i.e., prioritize short sequences)
- 3-gram: SmarTest with 3-gram language model

CVE dataset



68s (vs. 1,817s)

Leaking-Suicidal dataset



140s (vs. 1,770s)

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.

```
1  contract AToken {
2    /* Constructor function */
3    function BToken () public {
4      balance[msg.sender] = 10000000000;
5      totalSupply = 10000000000;
6    }
7  ...
```

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

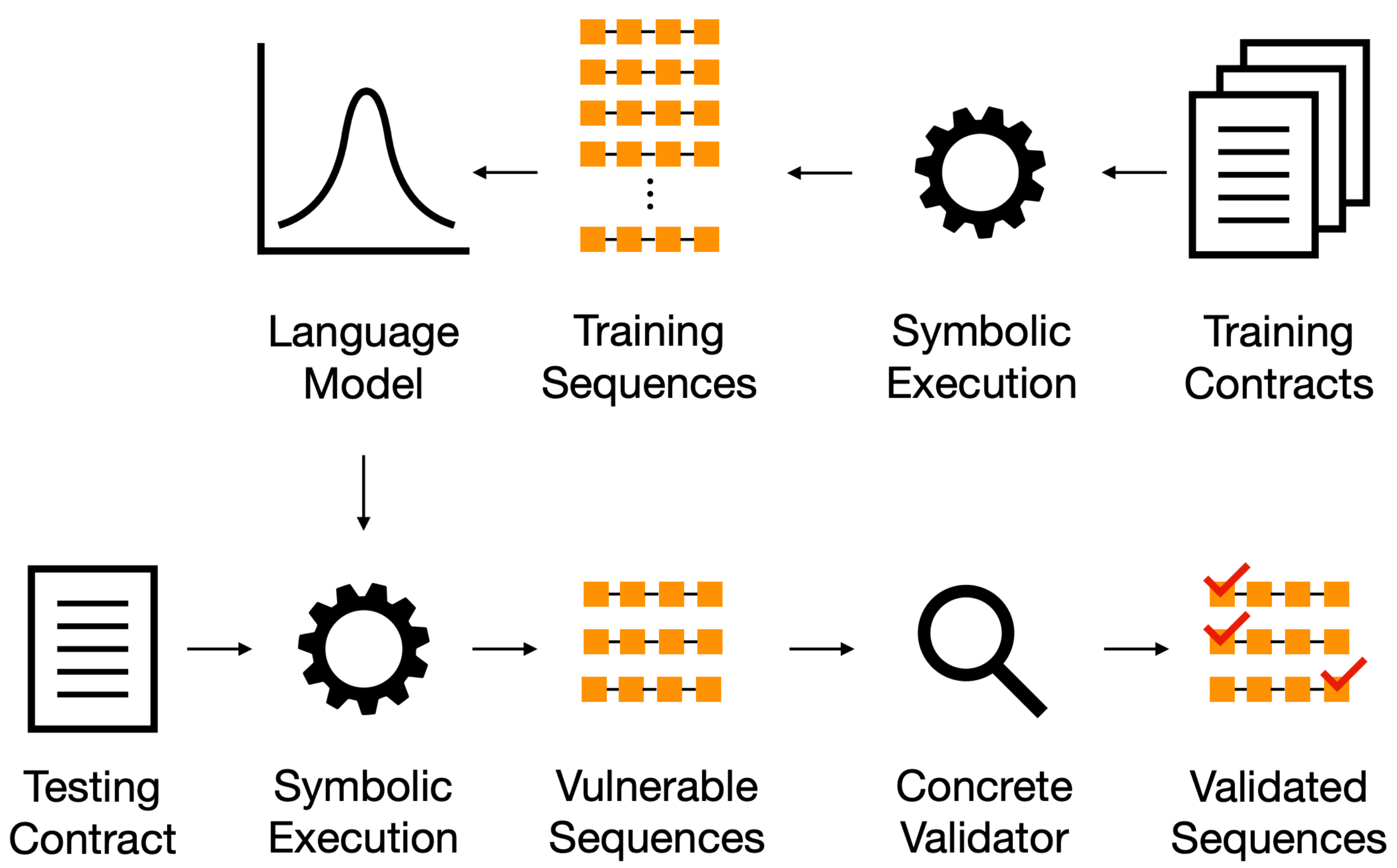
Pattern 2: Unrestricted token transfer by unauthorized users.

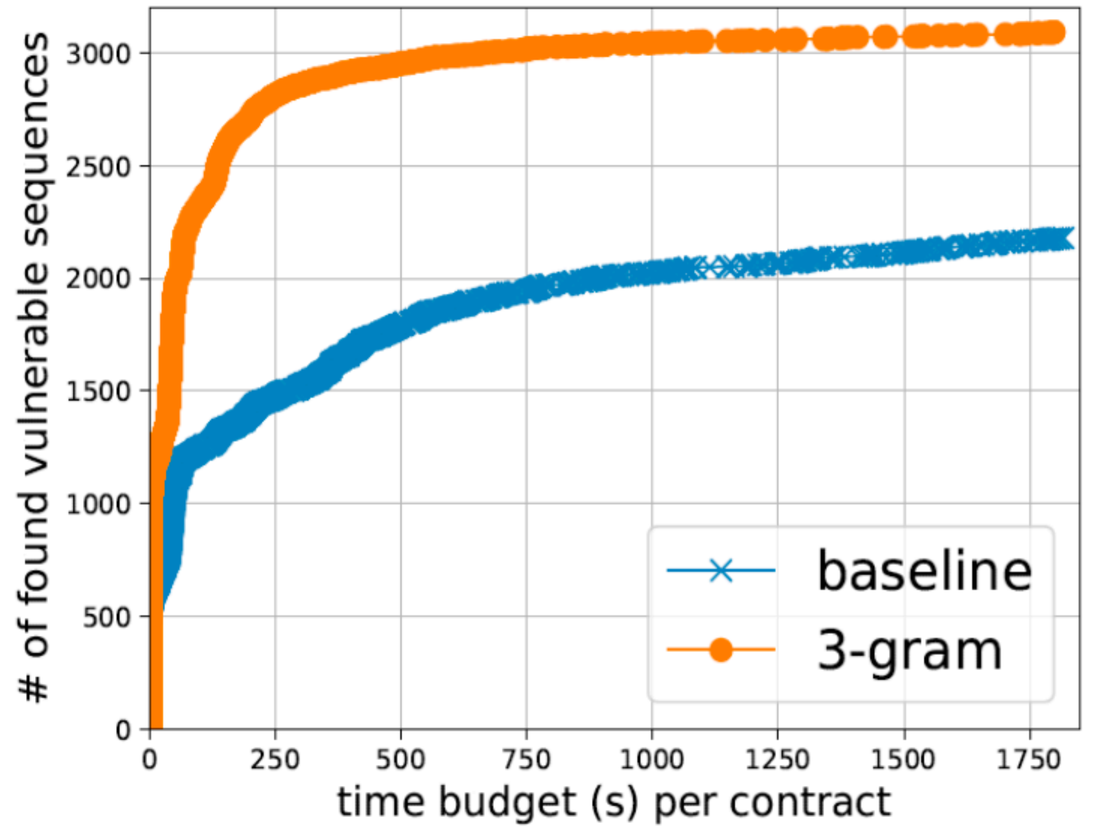
```
function transferFrom (address from, address to,
  uint value) public returns (bool) {
  require(balance[from] >= value);
  require(balance[to] + value >= balance[to]);
  require(allowed[from][msg.sender] >= value);
  balance[from] -= value;
  balance[to] += value;
  return true;
}
```

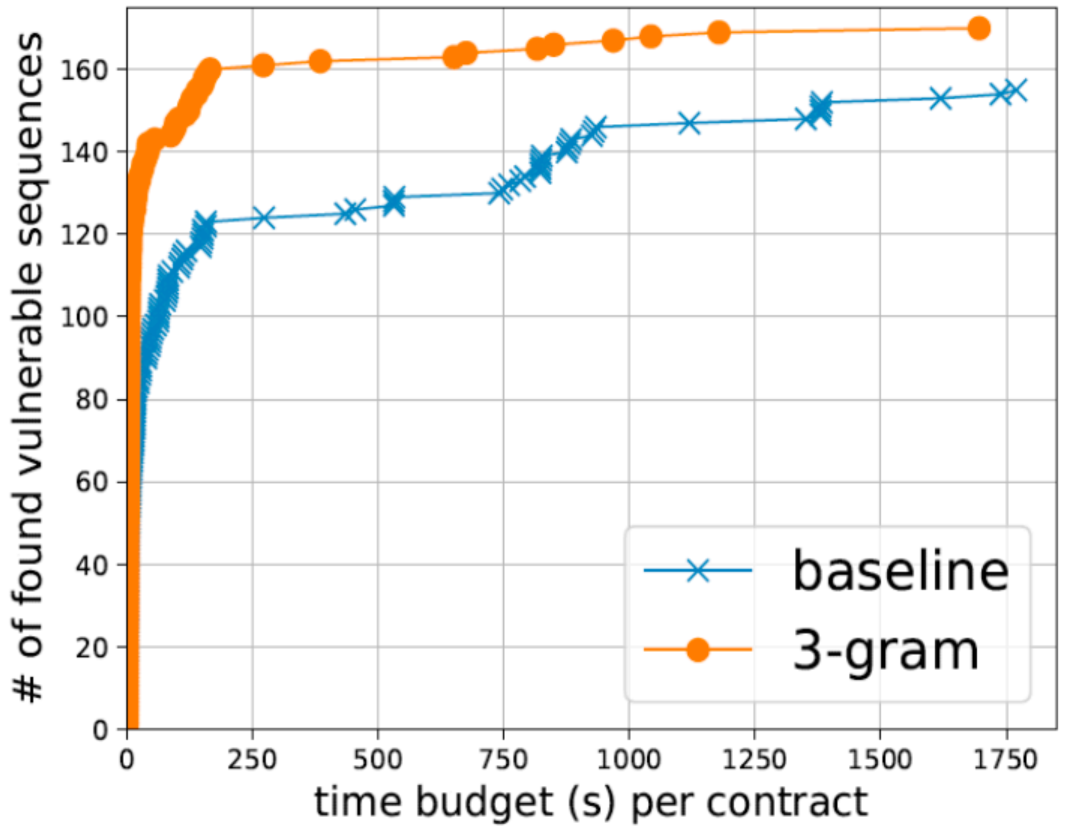
Red-colored part is absent in the original code.

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
Benchmark: https://github.com/kupl/VeriSmart-benchmarks

Thank you!