

TxSpector: Uncovering Attacks in Ethereum from Transactions







Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin,

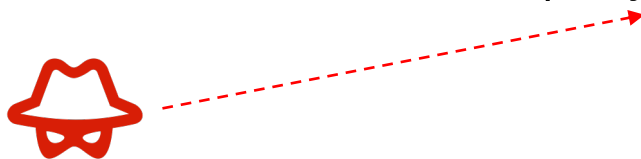
The Ohio State University

USENIX Security 2020

Background

Three types of transactions in Ethereum

-  transfer *Ether* → 
EOA EOA
-  deploy → 0x0 → 
EOA Smart Contract
-  execute a function → 
EOA deployed SC



1. developed using high-level programming languages
2. compiled into bytecode, which are executed in the EVM
3. bytecode consist of OPCODEs

```
function transferProxy(address _from, address _to,  
uint256 _value, uint256 _feeSmt... {  
    if(balances[_from] < _feeSmt + _value) revert();  
    balances[_to] += _value;  
    Transfer(_from, _to, _value);  
    balances[msg.sender] += _feeSmt;  
    Transfer(_from, msg.sender, _feeSmt);  
    balances[_from] -= _value + _feeSmt;...}
```

Source code

```
PUSH1; 0x60  
PUSH1; 0x40  
MSTORE  
CALLDATASIZE; 0x144  
ISZERO  
PUSH2; 0x20e  
JUMPI
```

OPCODEs

Problems & Research Gap

Two features have made smart contracts more vulnerable to software attacks than traditional software programs:

1. Smart contracts are immutable once deployed.
2. Ethereum is driven by cryptocurrency. (huge financial losses)

Systems	Tx Order Dependence	State Dependence	Mishandled Exception	Re-entrancy	Restricted Transfer	Failed Send	Unsecured Balance	Misuse-of-origin	Integer Overflow	Suicidal	Denial-of-Service
OYENTE [31]	▲	▲	▲	▲							
ZEUS [29]	▲	▲	▲	▲	▲		▲	▲			
SECURIFY [44]	▲		▲	▲	▲						
VANDAL [3]			▲	▲		▲	▲			▲	
GIGAHORSE [23]				▲						▲	▲
MAIAN [35]					▲					▲	
SLITHER [20]		▲	▲	▲	▲	▲	▲			▲	
MYTHRIL [7]	▲	▲	▲	▲		▲	▲	▲			
ETHBMC [21]						▲				▲	
SEREUM [38]				★							
ECFCHECKER [24]				★							
TXSPECTOR	★	★	★		★	★	★		★	★	

Detecting vulnerabilities using static analysis has limitations:

1. difficult to achieve completeness and accuracy simultaneously
2. could not be used to inspect and understand real-world Ethereum attacks.

Dynamic tools such as SEREUM and ECFCHECKER can detect Ethereum attacks, but they only target re-entrancy attacks.

Goals & Challenges

Goals:

1. a generic analysis framework for Ethereum transactions to identify real-world attacks against smart contracts in transactions
 2. and enable the forensic analysis of the attacks. (the pattern and statistics of the attacks, addresses used by attackers, and addresses of victims)
-

Challenges:

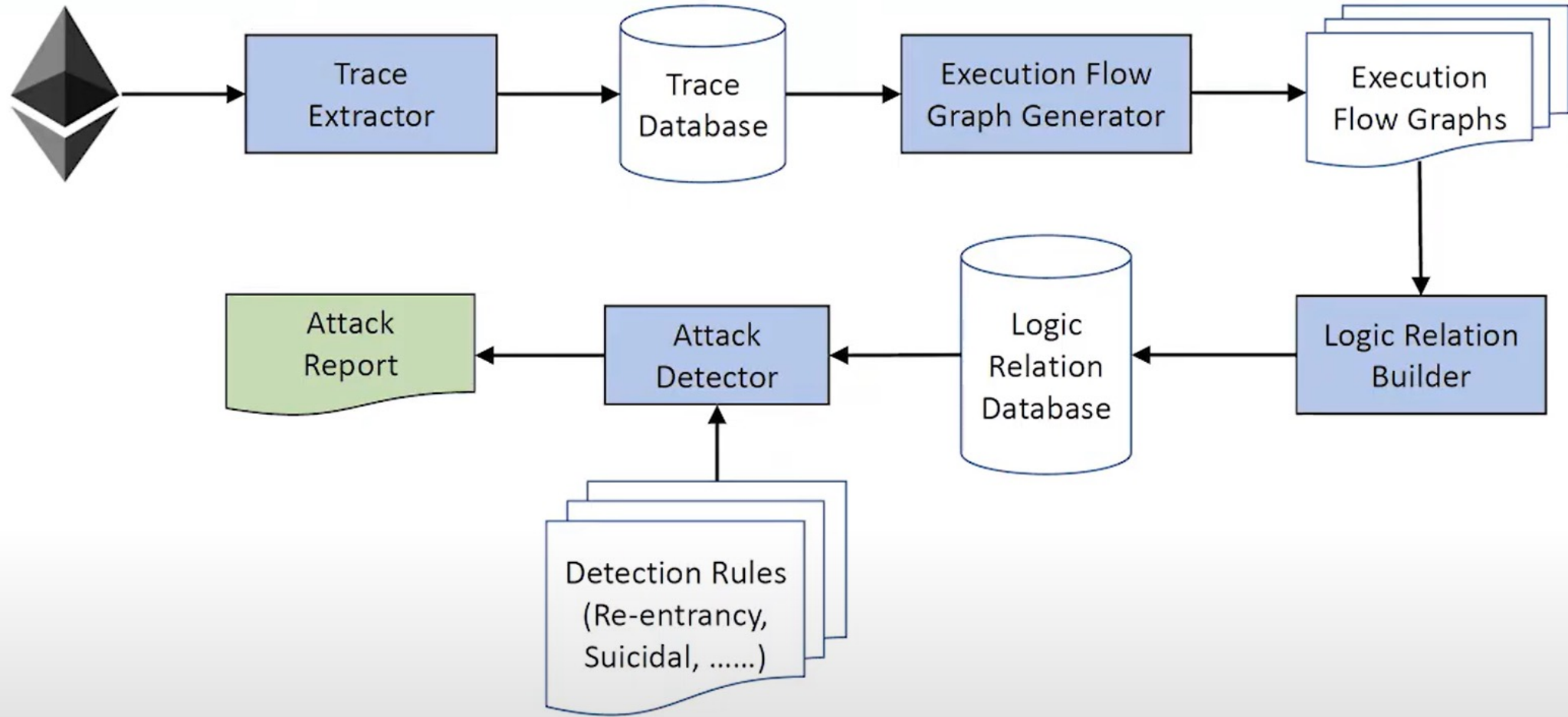
1. new methods need to be developed to extract data and control dependencies in Ethereum transactions and encode them into logic relations.
2. transaction volumes can be huge. Therefore, tracing and analyzing Ethereum transactions requires innovative approaches to optimize the performance.

Addressing Challenges

Challenges:

1. new methods need to be developed to extract data and control dependencies in Ethereum transactions and encode them into logic relations.
constructs Execution Flow Graphs (EFGs) to encode the control and data dependencies
2. transaction volumes can be huge. Therefore, tracing and analyzing Ethereum transactions requires innovative approaches to optimize the performance.
replays transactions on the blockchain and stores the traces into databases
extracts logic relations from the EFGs and stores them into databases.

Overview



1 Trace Extractor

Method: Record bytecode-level traces when transactions are executed.

- A trace contains metadata and executed OPCODEs.
- Metadata: the address of the transaction receiver, the timestamp, etc.
- Record OPCODEs: {<PC>, <OPCODE>, <ARGS>}

```
0; PUSH1; 0x60
2; PUSH1; 0x40
4; MSTORE
5; CALLDATASIZE; 0x144
6; ISZERO
7; PUSH2; 0x20e
10; JUMPI
```

Listing 1: Trace Snippet

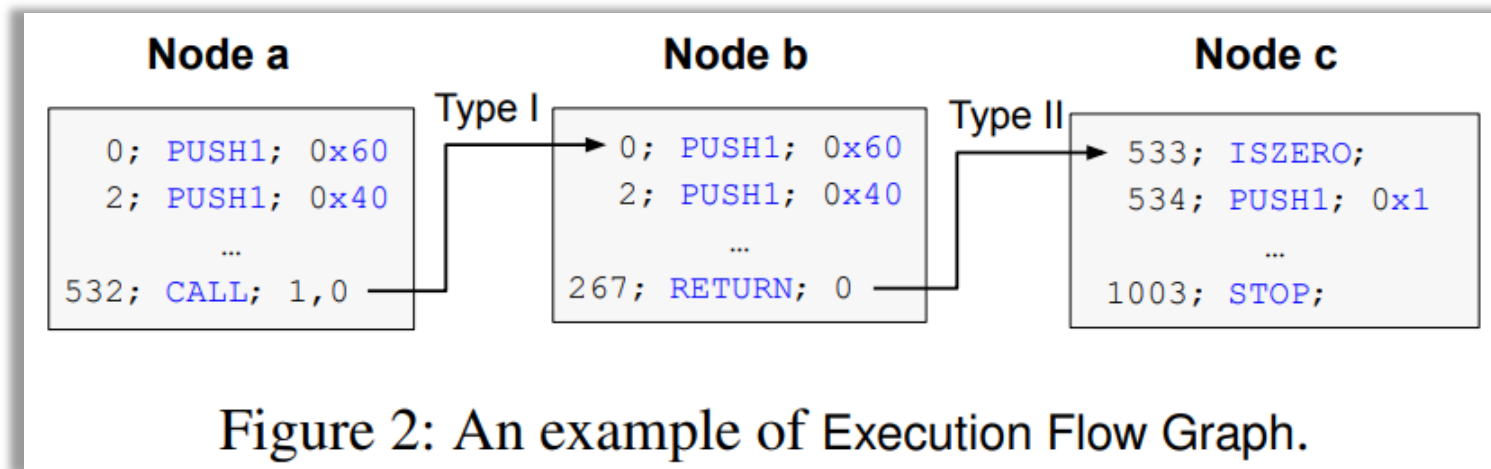
Implementation: modify the EVM to log related information of the OPCODEs.

2 Execution Flow Graph Generator

Goal: Since control-flow information is needed to detect attacks, the control-flow needs to be expressed more explicitly.

Method: Execution Flow Graph Generator builds Execution Flow Graphs (EFGs) that encode the control and data-flow information of the traces into graphs.

- Node: create when execution flow is altered from one Smart contract to another.
- Edge: represents the control flow between two nodes.



3 Logic Relation Builder

Goal: extracts the logic relations that express the semantics of the transactions.

Idea: abstract the semantics (control, data-flow) for the next step of attack detection

Method:

1. parses the EFGs to construct intermediate representation (IR) suitable for the analysis. IR replaces the stack operations with registers:

```
0; PUSH1; 0x60  
2; PUSH1; 0x40  
4; MSTORE  
5; CALLDATASIZE; 0x144  
6; ISZERO  
7; PUSH2; 0x20e  
10; JUMPI
```

Listing 1: Trace Snippet

```
0: V0 = 0x60  
2: V1 = 0x40  
4: M[0x40] = 0x60  
5: V2 = 0x144  
6: V3 = ISZERO 0x144  
7: V4 = 0x20e  
10: JUMPI 0x20e 0x0
```

Listing 2: IR Snippet

Benefits:

the values of registers are updated accordingly and all of the intermediate values are properly recorded.

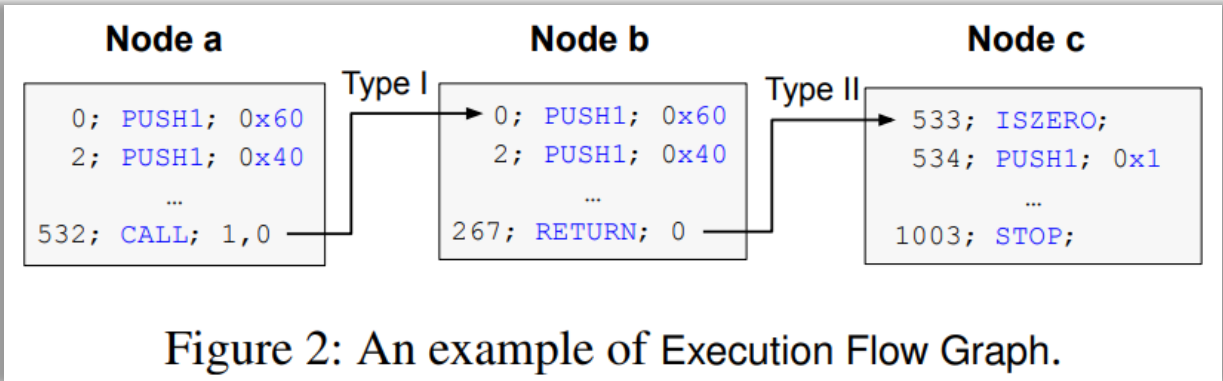
3 Logic Relation Builder

Goal: extracts the logic relations that express the semantics of the transactions.
Idea: abstract the semantics (control, data-flow) for the next step of attack detection

Method:

The logic relations express the register transfer language of an EVM bytecode program.

- 2. extracts the logic relations that express the semantics of the transactions by defining logic rules.



PC	Register	Idx	Depth	Callnum
0	V1	1	1	0
2	V2	2	1	0
0	V89	245	2	1
2	V90	246	2	1
534	V285	1,072	1	1

Table 2: An example of PUSH1 logic relations.

4 Attack Detector

Goal: Users can write detection rules to detect the attacks they want to detect

Method:

1. takes user-specified query rules (dubbed Detection Rules) as inputs
 2. queries the Logic Relation DB generated by Logic Relation Builder
 3. The outputs are not simple yes or no answers, instead, detailed information.
-

Taking *Suicidal* Attacks as an example:

```
自毁函数(参数) public {  
    if (CALLER有权限) {  
        SELFDESTRUCT ;  
        (将ETH发往参数指定的地址)  
    }  
    else ;  
}
```

```
Suicidal(args) :-  
    op_SELFDESTRUCT(_, _, sdIdx, 1, _),  
    op_CALLER(_, callerAddr, callerIdx, 1, _),  
    !jumpiDep(jumpiIdx, 1, callerIdx, callerAddr).
```

Figure 7: The Detection Rules for detecting *Suicidal*.

Evaluation

Dataset:

1. focus on the transactions starting from the 7,000,000-th block
2. contains 16,485,279 transactions (9,662,675 after filtering)
3. covering the transactions between January 2019 to February 2019

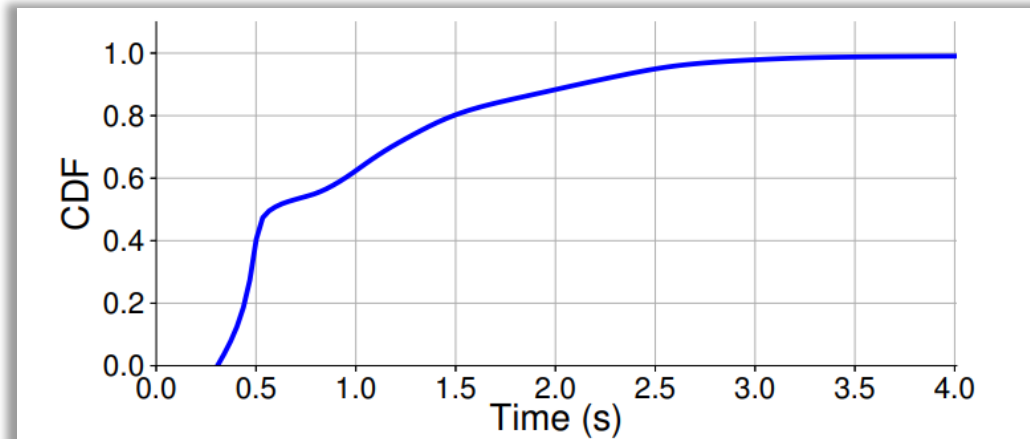


Figure 8: Time distribution on generating logic relations.

Results of *Suicidal* Attacks:

1. TXSPECTOR flagged 23 transactions.
2. did not produce false positives.

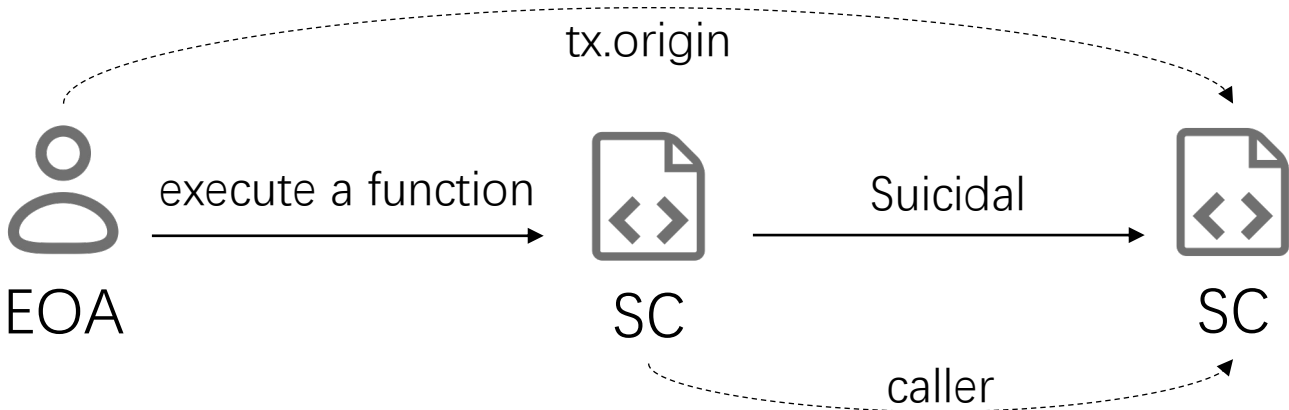
Comparison with Other Tools:

Vulnerability	System	# Total	# Timeout or Error	# Remaining	# Flagged
Suicidal	TXSPECTOR	9,661,593	327,208	9,334,385	23
	VANDAL	105,535	1,187	104,348	349
	GIGAHORSE	105,535	N/A	N/A	383

Application

Forensic Analysis of Suicidal:

- 1. the reasons behind the 23 Suicidal transactions
 - 20 — No permission check at all
 - 3 — Mistakes in checks



2. Beneficiaries

Beneficiary Address	Tx Count
0x3a91b432b27eb9a805c9fd32d9f5517e9dd42aa4	3
0x6e226310db63ac3701f657bcc62c153c1aaa3004	2
0x15202d3d183708649451878f50982d5c1bb4d01b	2

Table 7: Common beneficiary addresses.

Discussion

1. Time cost.

TXSPECTOR is designed as a forensic analysis framework on transactions, but not intended to be used as a real-time attack detection tool.

2. Transaction vs. bytecode.

for forensic analysis, analyzing transactions is more meaningful than studying smart contract bytecode.

3. Reactive approach vs. proactive approach.

meaning that attacks can only be detected *after* they have occurred

studying transactions can reveal *true* attacks happened in the past, and learn from them in a forensic perspective.

Conclusion

