Symbolic Analysis Tool to find DoS Vulnerabilities in Smart Contract

Deepak Yadav

Under the supervision of

Prof. Sandeep Sukla

IIT Kanpur

Outline

- Introduction
- DoS Vulnerabilities detected in Smart Contracts
- Basics
- Symbolic Analysis Tool
- Design Choices
- Results
- Limitations

Ethereum

- Launched in 2015.
- Ethereum is the world’s programmable blockchain.
- Build new kinds of applications.
- Ethereum is the second largest cryptocurrency platform.
- Ether is the cryptocurrency generated by ethereum.
- EVM

Smart Contracts

- Computer programs
- Immutable
- Deterministic

Why vulnerable Smart Contract is a concern

- Source code is not available
- Property of immutability
- Trust Issues

DoS Vulnerabilities in Smart contract

- CREATE2 Information Leak Vulnerability
- Call to an address which can be set by external user

```
contract setThrone {
    address currentSuccessor;
    uint highestValue;

    function claimThrone() payable {
        require(msg.value > highestValue);
        require(currentSuccessor.send(highestValue)); // Refund the old Successor else revert
        currentSuccessor = msg.sender;
        highestValue = msg.value;
    }
}
```

DoS Vulnerabilities continued...

- Block Gas Limit
  - Adding garbage values to array, mapping which needs to be iterated in the code.
  - Directly changing loop condition variable

```
1   pragma solidity ^0.4.24;
2   contract BasicToken {
3       uint256[] balances;
4       function add(uint256 a) private  {
5           balances.push(a);
6       }
7       function balanceOf() public  view returns (uint256) {
8           uint c = 1;
9           add(i);
10          return c;
11      }
12      function print() public view returns (uint256){
13          uint256 temp = 0;
14          for(uint256 i = 0; i < balances.length; i++){
15              temp += balances[i];
16          }
```

Vulnerabilities Patterns

- CREATE2 opcode
- CALL opcode
- Symbolic loop condition

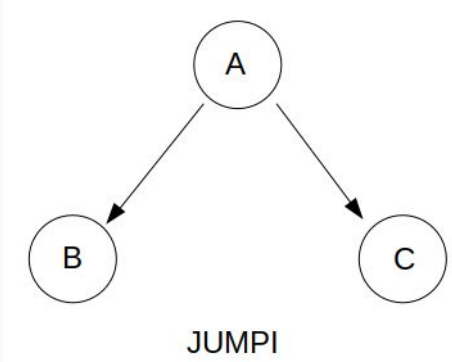
Some Basics

- JUMPDEST opcode

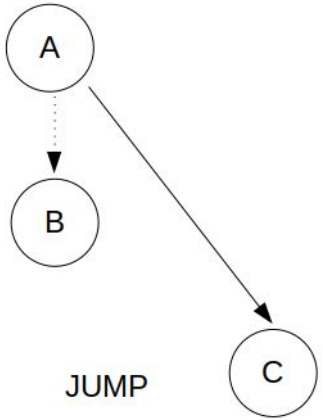
```
PUSH1 0x80
PUSH1 0x40
MSTORE
PUSH1 0x4
CALLDATASIZE
LT
PUSH1 0x3E
JUMPI
PUSH4 0xFFFFFFFF
PUSH29 0x100000000000000000000000000000000000000000000000000000000
PUSH1 0x0
CALLDATALOAD
DIV
AND
PUSH4 0x3E58C58C
DUP2
EQ
PUSH1 0x43
JUMPI
JUMPDEST
PUSH1 0x0
DUP1
REVERT
JUMPDEST
PUSH1 0x62
PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
PUSH1 0x4
CALLDATALOAD
AND
PUSH1 0x64
JUMP
JUMPDEST
STOP
JUMPDEST
PUSH1 0x0
SLOAD
PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
AND
CALLER
EQ
PUSH1 0x87
JUMPI
PUSH1 0x0
DUP1
REVERT
JUMPDEST
PUSH1 0x40
MLOAD
PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
DUP3
AND
SWAP1
CALLVALUE
DUP1
ISZERO
PUSH2 0x8FC
MUL
SWAP2
PUSH1 0x0
DUP2
DUP2
DUP2
DUP6
CALL
POP
POP
POP
JUMP
STOP
```

Some Basics

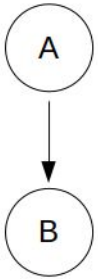
- Control Flow from one basic block to another
  - JUMP opcode
  - JUMPI opcode
  - Sequential execution enters next adjacent basic block
- Deploytime bytecode vs runtime bytecode



JUMPI



JUMP



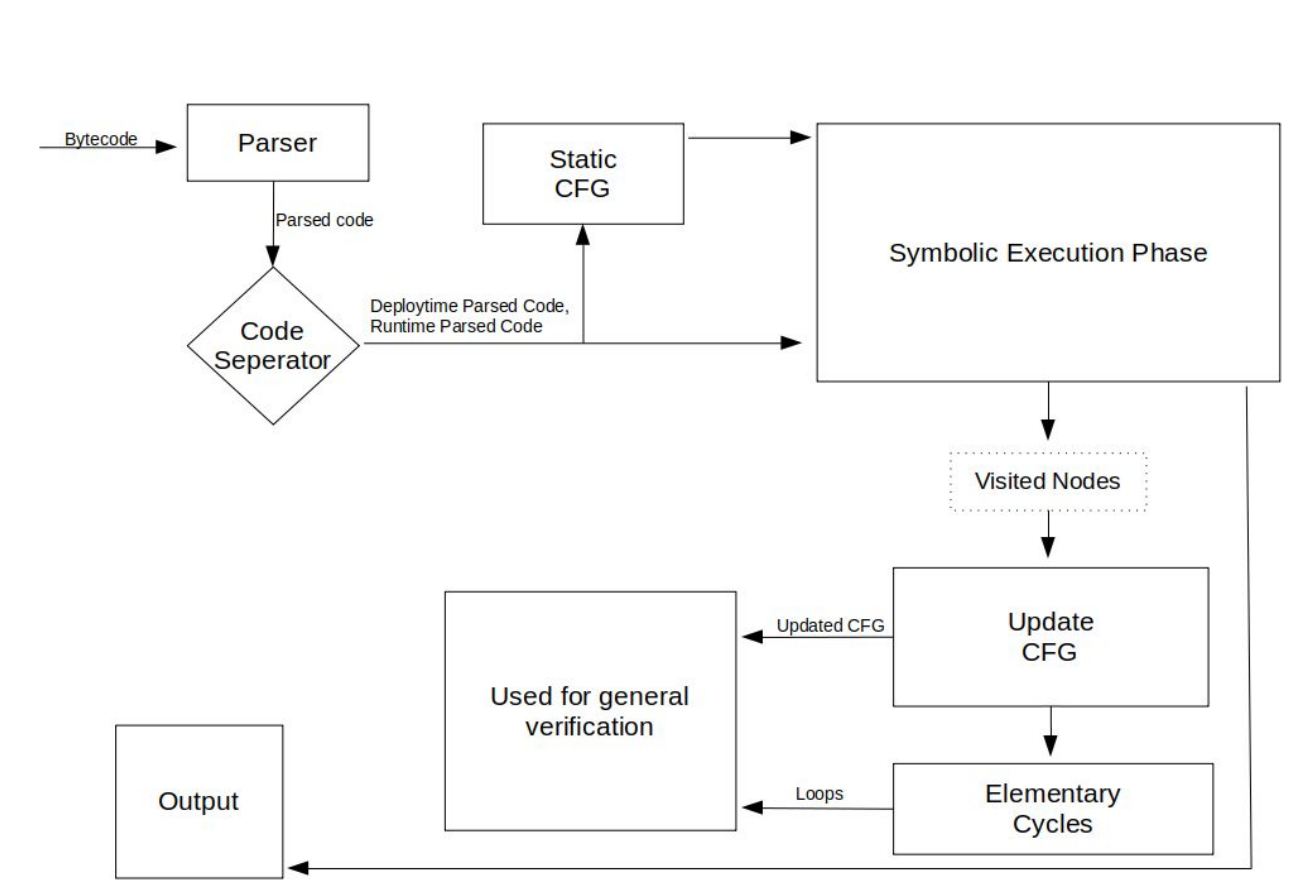
Sequential

Symbolic Analysis Tool

We can divide the whole tool into 3 phases

- Preprocessing the input
- Symbolic Execution Phase
- Results and limitations of tool

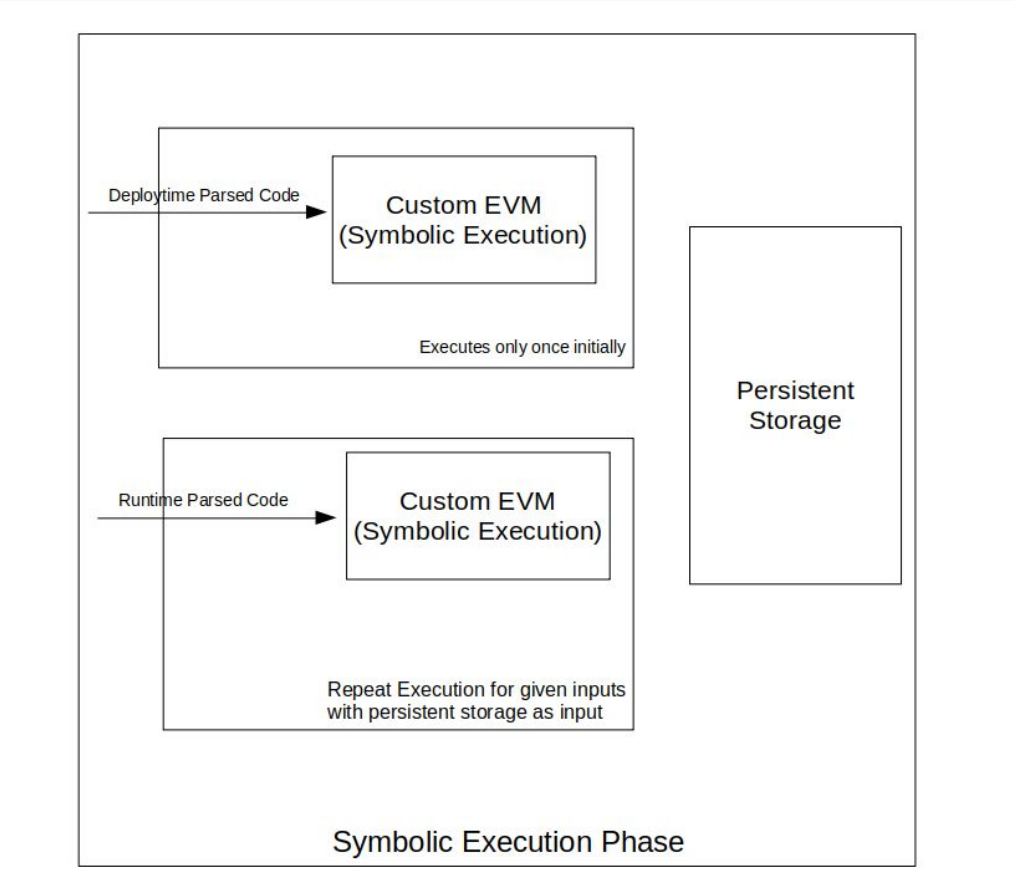
A flow chart of the tool



1. Preprocessing the input

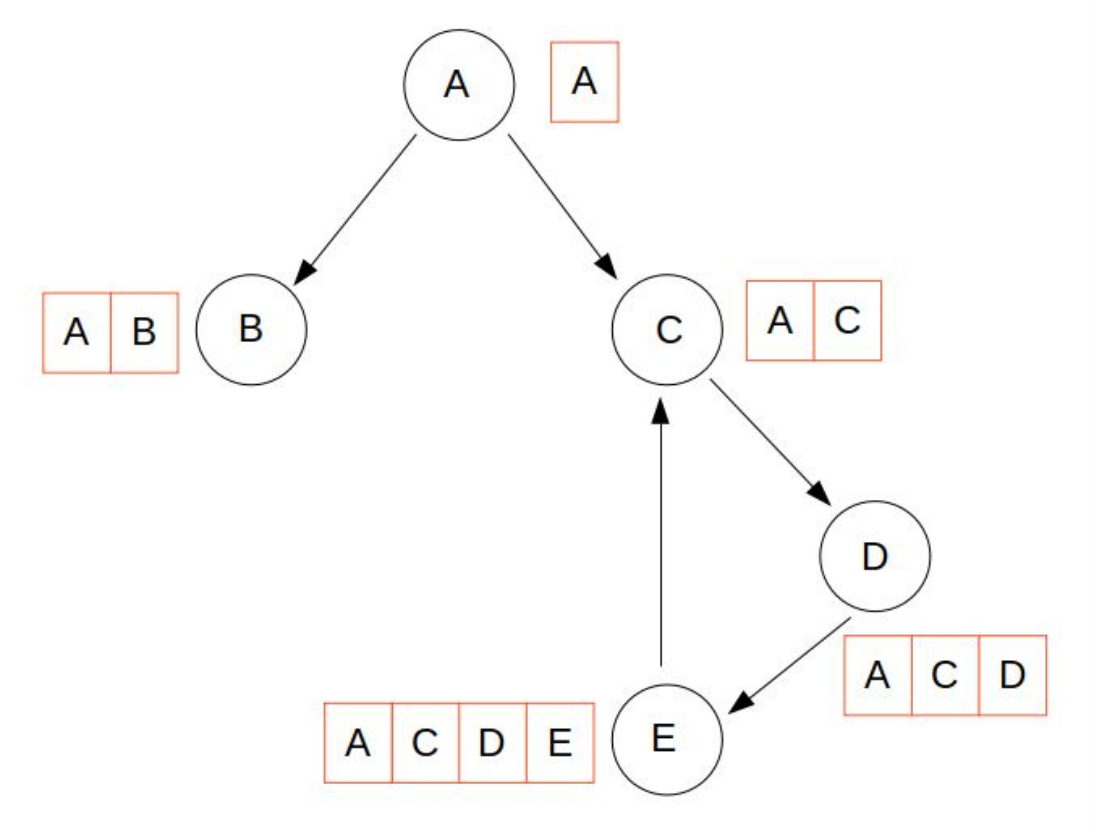
- Source code is compiled to Bytecode
- EVM contains predefined sets of opcodes/instructions.
- Every instruction/opcode have a corresponding hexadecimal values(2 bytes in size)
  - ‘0x60’ corresponds to PUSH1
  - ‘0x52’ corresponds to MSTORE…
- Before feeding data to Symbolic Execution Phase
  - **Parser** : Parser takes input the bytecode and return a list of opcode view
  - **Code Separator** : Separates opcode view list into two separate list
    - Deploytime bytecode
    - Runtime bytecode

2. Symbolic Analysis Phase



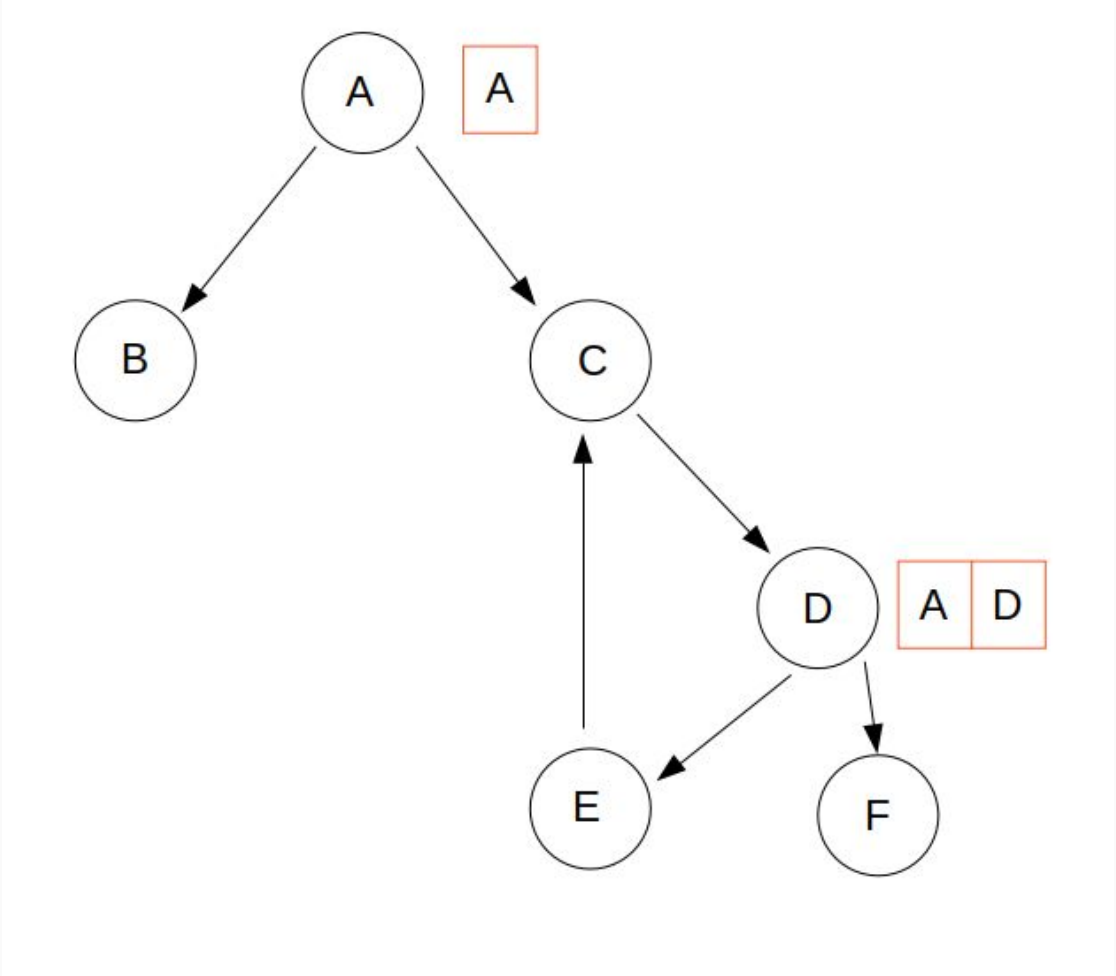
How Loops are Identified

- Maintain a list of visited basic blocks along the traversing path.
- For every basic block visited check if the list contains the same basic block or not.



Decide the loop is unbounded or not

- Store Stack State whenever a JUMPI with symbolic condition is found.
- Stack State is a map with key as basic block id and data as the list of all symbolic JUMPI within the basic block.
- Final check is to ensure that the jump to basic block should not be in the list of basic blocks of the current loop.



Design Choices

- **Problem 1** : Loop termination problem
  - During the Symbolic Execution we can detect loops using the stack.
  - But we cannot decide whether the loop will terminate in definite iterations or not.
  - Hence a decision is to be made how to terminate the loop.

- **Solution :** A max_path variable is set.
  - A max_path variable is given as argument to the tool
  - A path_length variable is maintained and incremented for every basic block in the execution path.
  - When path_length is equal to max_path we terminate the search along this path.

Design Choices continued...

Problem 2 : Variable is set to symbolic after execution of the path in which it can be detected.

First 4 bytes of keccak256 hash as external function signature.

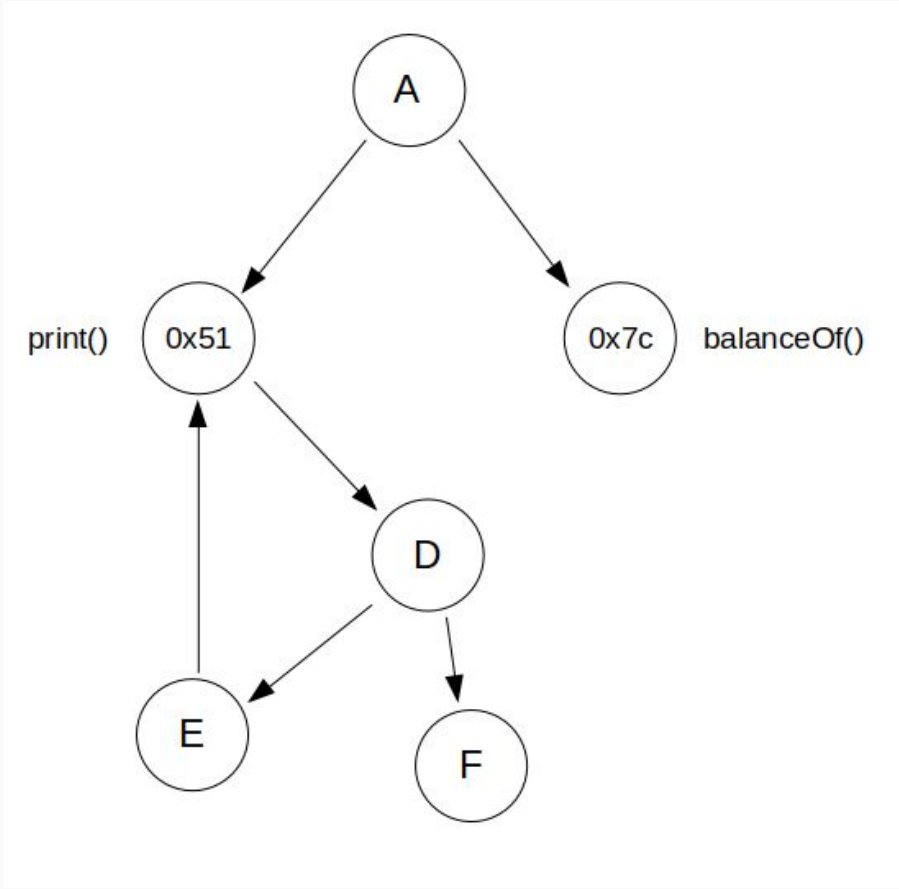
- print() = 13bdfacd
- balanceOf() = 722713f7

```
1   pragma solidity ^0.4.24;
2   contract BasicToken {
3       uint256[] balances;
4       function balanceOf() public  view returns (uint256) {
5           uint c = 1;
6           balances.push(a);
7           return c;
8       }
9       function print() public view returns (uint256){
10          uint256 temp = 0;
11          for(uint256 i = 0; i < balances.length; i++){
12              temp += balances[i];
13          }
14      return temp;
15      }
16  }
```

Problem 2 continued...

The assembly view of external functions in the smart contract above is

```
1        PUSH4 0x13BDFACD
2        EQ
3        PUSH2 0x51
4        JUMPI
5        DUP1
6        PUSH4 0x722713F7
7        EQ
8        PUSH2 0x7C
9        JUMPI
```



Design Choices continued...

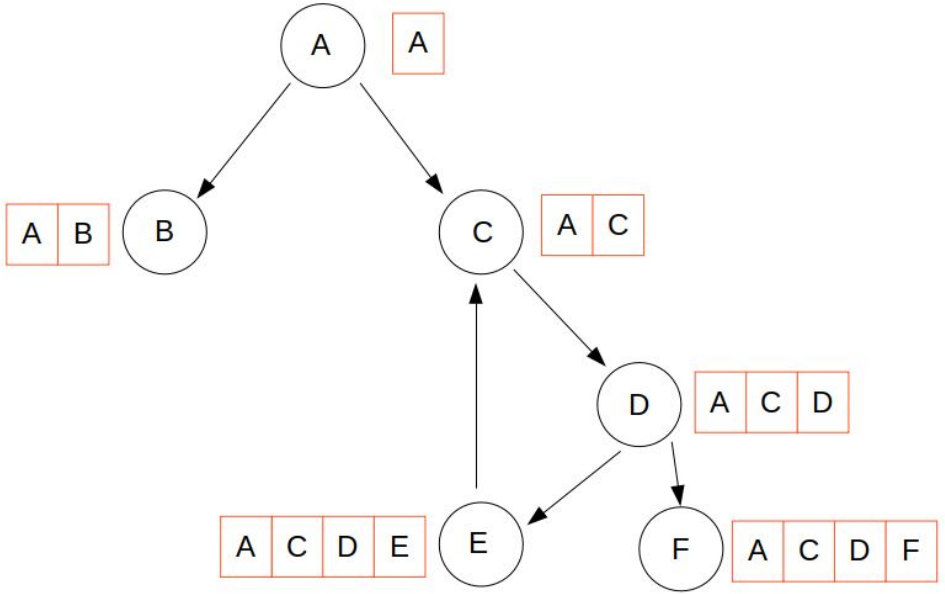
Solution : Multiple Invocations

- Max_invocation parameter is provided as an argument.
- Only runtime bytecode is run multiple times.
- Storage is permanent in ethereum and used to store the state of smart contracts between invocations.

Designed choices continued...

- **Problem 3** : Constant Loops with iterations more than MAXIMUM PATH LENGTH

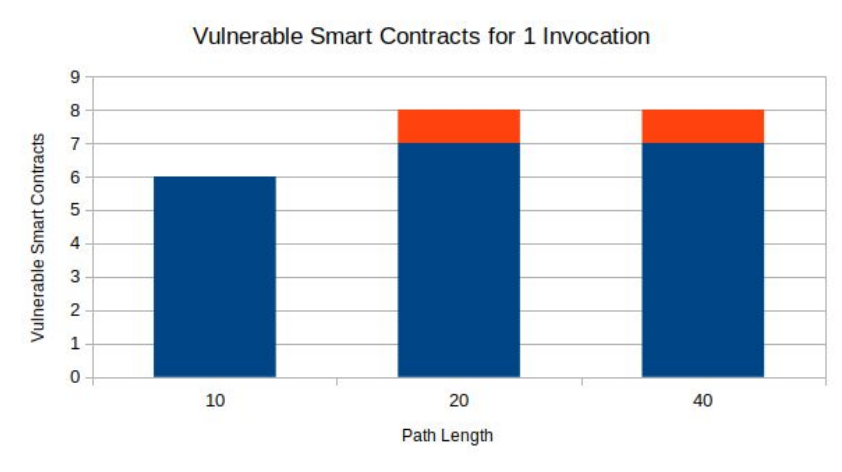
```
function print() public view returns (uint256){

    uint256 temp = 0;

    for(uint256 i = 0; i < 1000; i++){
        temp++;
    }

    for(i = 0; i < balances.length; i++){
        temp += balances[i];
    }

    return temp;
}
```

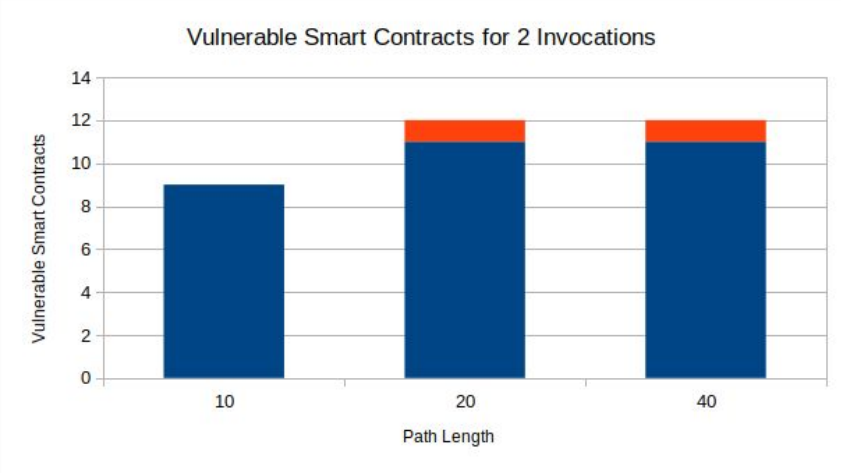


Results

- The tool is set up on a 64-bit Ubuntu 20.04 LTS, 7.7 GB of RAM and 8 Core Intel(R) Core(TM) i5-8250U CPU @ 1.60GHZ.
- A sample of 35 smart contracts is used for testing the tool.
- The tool is tested for different sets of {invocations, path_length}
- Invocations 1, 2 and path_length 10, 20, and 40 respectively.

Results


Vulnerable Smart Contracts for 1 Invocation
Vulnerable Smart Contracts
0
1
2
3
4
5
6
7
8
9
10
20
40
Path Length

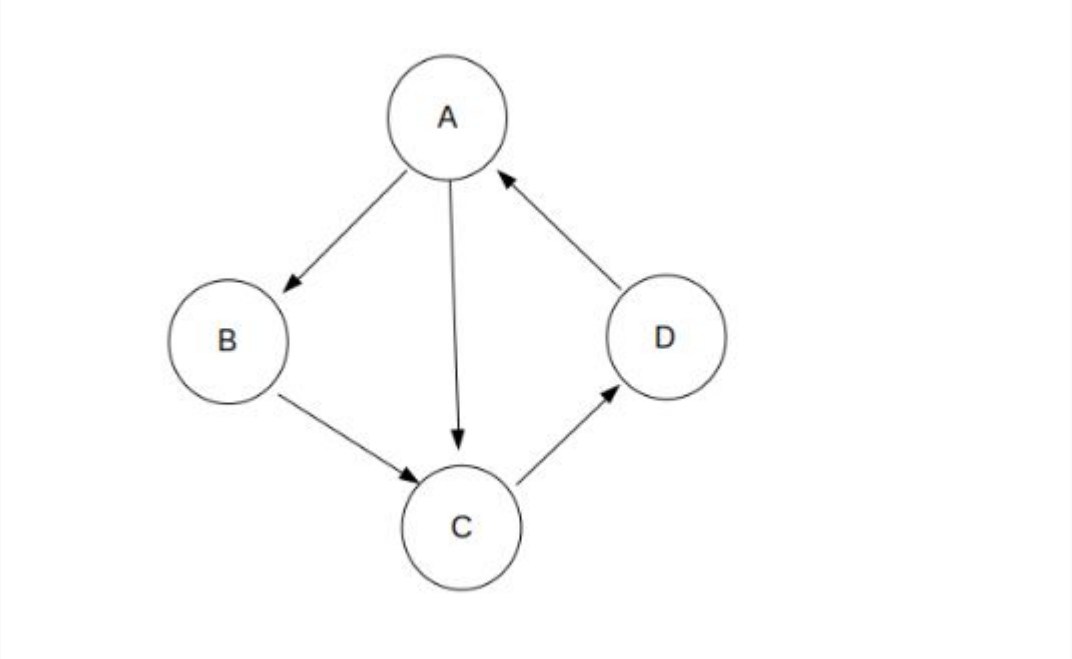

Vulnerable Smart Contracts for 2 Invocations
Vulnerable Smart Contracts
0
2
4
6
8
10
12
14
10
20
40
Path Length

False positives and limitation of tool

- Smt solver treats variables/constants as 256 bit number and ignores the real range of variables/constants

```
1   function nameFilter(string _input)
2       returns(bytes32)
3   {
4       uint256 _length = _temp.length;
5       require (_length <= 32 && _length > 0);
6       for (uint256 i = 0; i < _length; i++)
7       {
8           //some code
9       }
10  }
```

Thank You


A
B
C
D