SAILFISH:

Vetting Smart Contract State-Inconsistency Bugs in Seconds

Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna

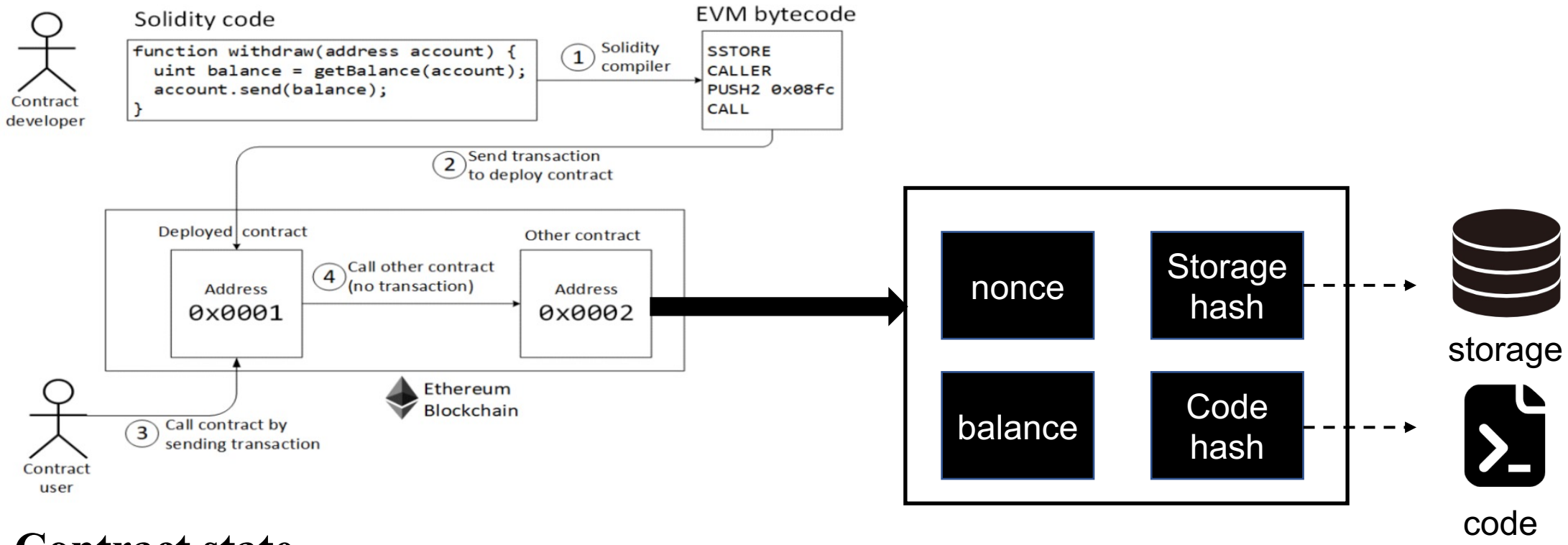
University of California, Santa Barbara

43th S&P 2022

Background

Smart Contract

Written in high-level languages (e.g. solidity), Complied down to EVM bytecode, Delpoyed to Ethereum.



Contract state

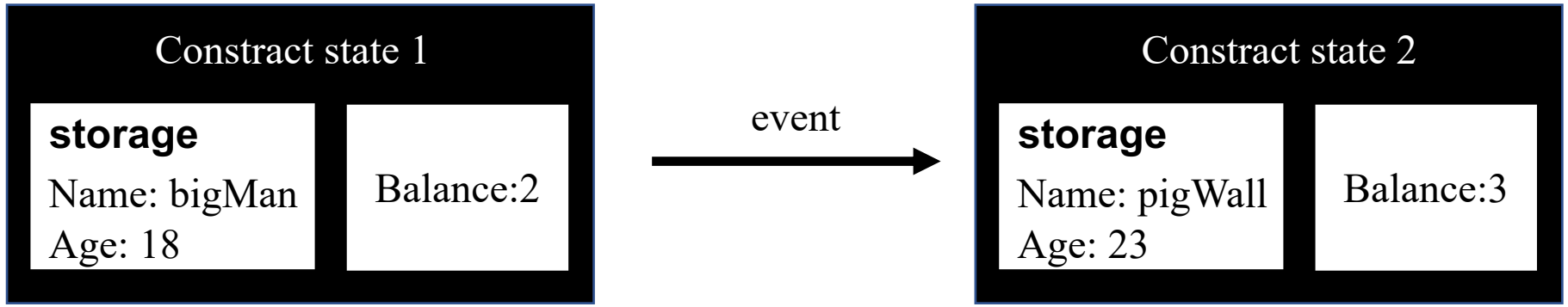
1. A set of all the storage variables of storage
2. balance

Background

Event

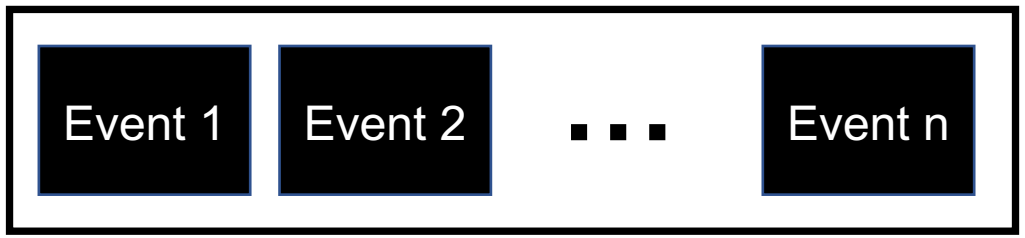
Public/External methods of a contract, can be invoke in two ways:

1. Transaction
2. Other contract



Schedule

A schedule H is a valid sequence of events that can be executed by the EVM.

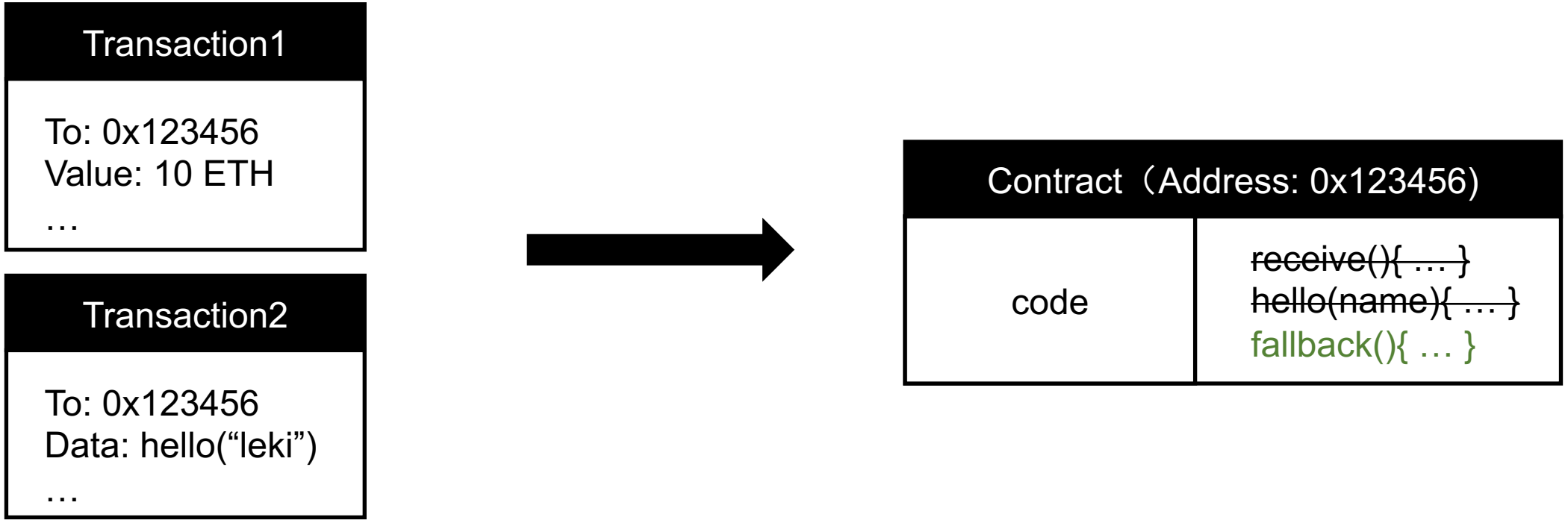


Background

fallback

The **fallback()** is executed on a call to the contract if **none of the other functions match the given function signature**, or if no data was supplied at all and there is **no receive()**.

fallback() can execute complex operations.



Background

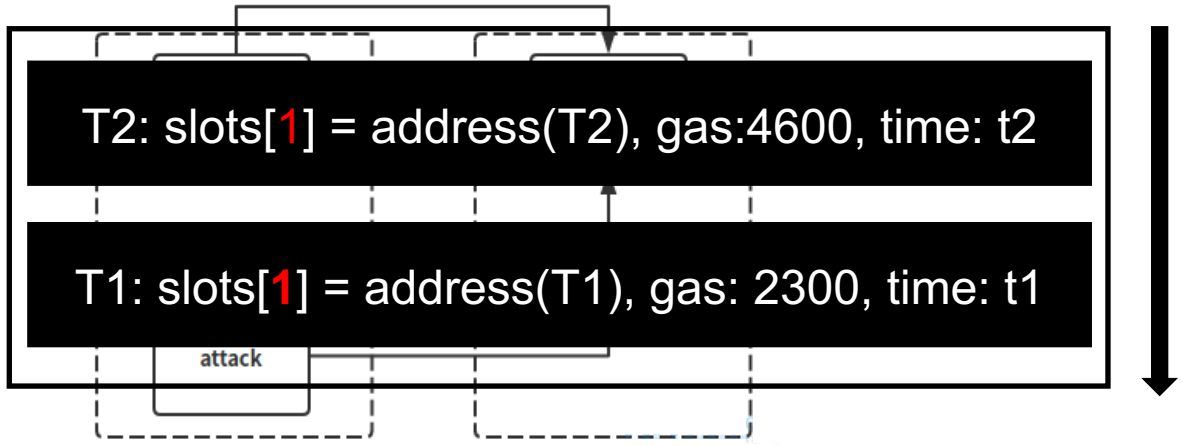
Reentrancy

If a contract A calls another contract B, the Ethereum protocol allows B to call back to any public/external method m of A in the same transaction before even finishing the original invocation.

Transaction Order Dependence (TOD)

By the time a transaction T1 (scheduled at time t1) is picked up by a miner, the network and the contract states might change due to another transaction T2 (scheduled at time t2) getting executed beforehand, though t1 < t2

```
1  contract Queue {
2    function reserve(uint256 slot){
3      if (slots[slot] == 0) {
4          slots[slot] = msg.sender;
5      }
6    }
7  }
```



(2) Transaction Order Dependence（TOD）

(1) Reentrancy

Background

State inconsistency (SI)

Two schedules individually operate on the same initial state Δ, but yield different final states,

Motivation

TABLE I: Comparison of smart-contract bug-finding tools.

Tool	Cr.	Haz.	Scl.	Off.
SECURIFY [54]	○	○	◐	●
VANDAL [23]	○	○	●	●
MYTHRIL [3]	○	○	○	●
OYENTE [46]	○	○	◐	●
SEREUM [50]	●	○	●	○
SAILFISH	●	●	●	●

● Full ◐ Partial ○ No support. **Cr.**: Cross-function, **Haz.**: Hazardou
access, **Scl.**: Scalability, **Off.**: Offline detection

```
1  // [Step 1]: Set split of 'a' (id = 0) to 100(%)
2  // [Step 4]: Set split of 'a' (id = 0) to 0(%)
3  function updateSplit(uint id, uint split) public{
4     require(split <= 100);
5     splits[id] = split;
6  }
7
8  function splitFunds(uint id) public {
9     address payable a = payee1[id];
10    address payable b = payee2[id];
11    uint depo = deposits[id];
12    deposits[id] = 0;
13
14    // [Step 2]: Transfer 100% fund to 'a'
15    // [Step 3]: Reenter updateSplit
16    a.call.value(depo * splits[id] / 100)("");
17
18    // [Step 5]: Transfer 100% fund to 'b'
19    b.transfer(depo * (100 - splits[id]) / 100);
20 }
```

Cross-function

Motivation

TABLE I: Comparison of smart-contract bug-finding tools.

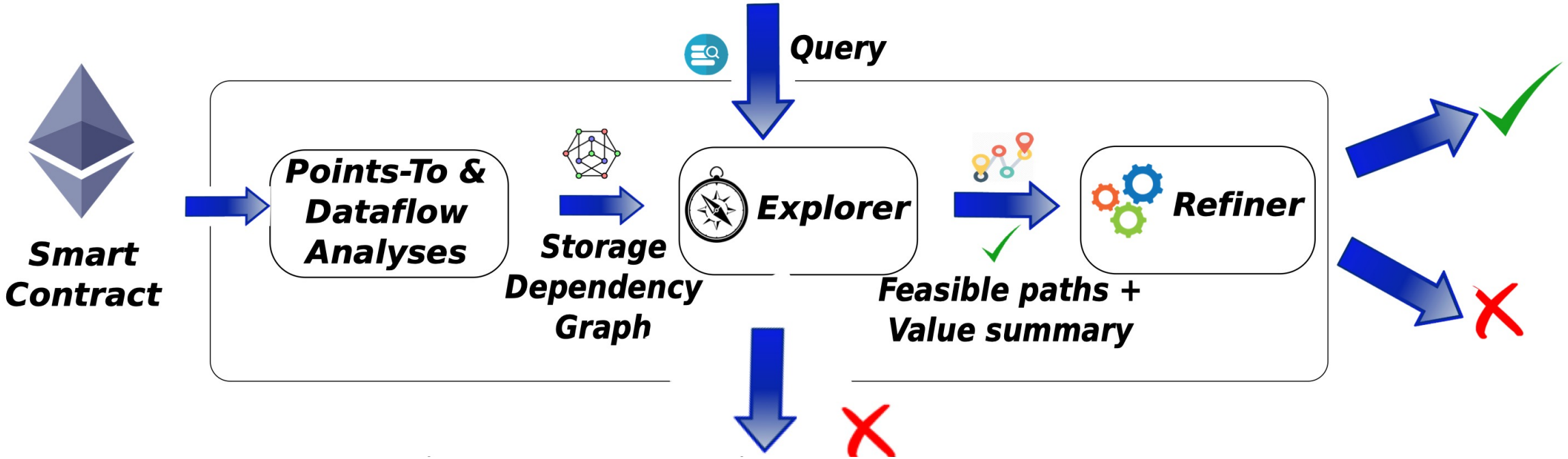
Tool	Cr.	Haz.	Scl.	Off.
SECURIFY [54]	○	○	◐	●
VANDAL [23]	○	○	●	●
MYTHRIL [3]	○	○	○	●
OYENTE [46]	○	○	◐	●
SEREUM [50]	●	○	●	○
SAILFISH	●	●	●	●

● Full ◐ Partial ○ No support. **Cr.**: Cross-function, **Haz.**: Hazardous access, **Scl.**: Scalability, **Off.**: Offline detection

```
1  function withdrawBalance(uint amount) public {
2      //[Step 1]: Enter when mutex is false
3      //[Step 4]: Early return, since mutex is true
4      if (mutex == false) {
5        //[Step 2]: mutex = true prevents re-entry
6        mutex = true;
7        if (userBalance[msg.sender] > amount) {
8           //[Step 3]: Attempt to reenter
9           msg.sender.call.value(amount)("");
10          userBalance[msg.sender] -= amount;
11       }
12       mutex = false;
13     }
14 }
```

Hazardous acess

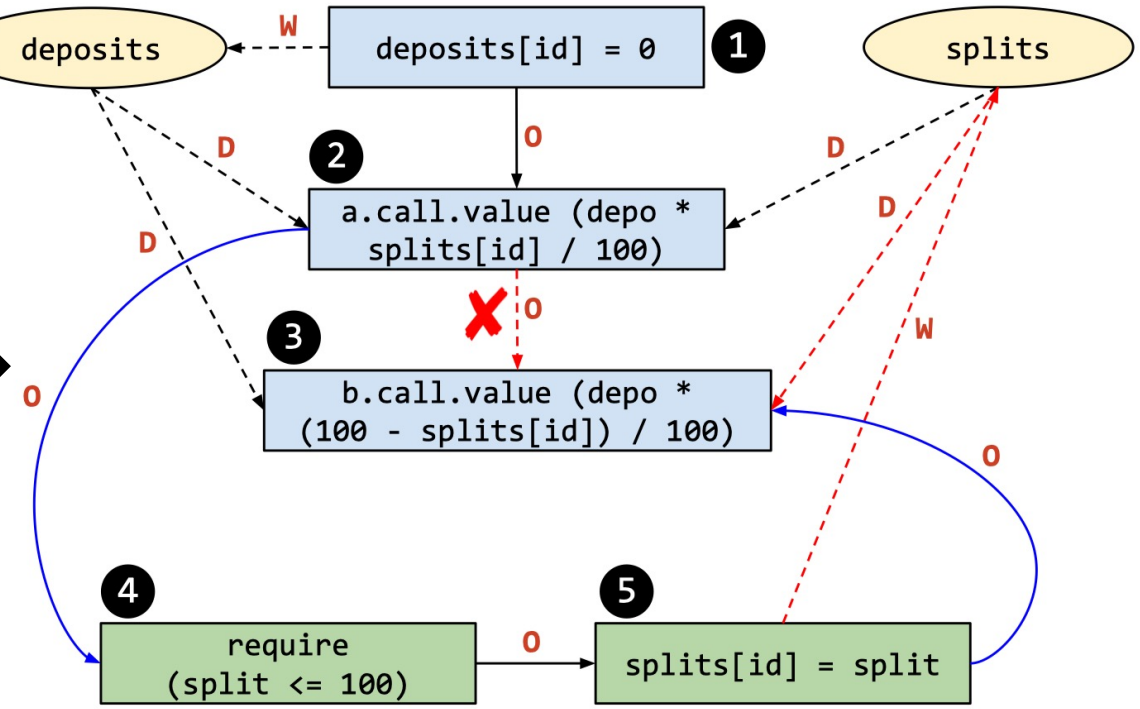
Overview of SAILFISH



Explorer: SDG

```
1  // [Step 1]: Set split of 'a' (id = 0) to 100(%)
2  // [Step 4]: Set split of 'a' (id = 0) to 0(%)
3  function updateSplit(uint id, uint split) public{
4      require(split <= 100);
5      splits[id] = split;
6  }
7
8  function splitFunds(uint id) public {
9      address payable a = payee1[id];
10     address payable b = payee2[id];
11     uint depo = deposits[id];
12     deposits[id] = 0;
13
14     // [Step 2]: Transfer 100% fund to 'a'
15     // [Step 3]: Reenter updateSplit
16     a.call.value(depo * splits[id] / 100)("");
17
18     // [Step 5]: Transfer 100% fund to 'b'
19     b.transfer(depo * (100 - splits[id]) / 100);
20 }
```

Construct SDG



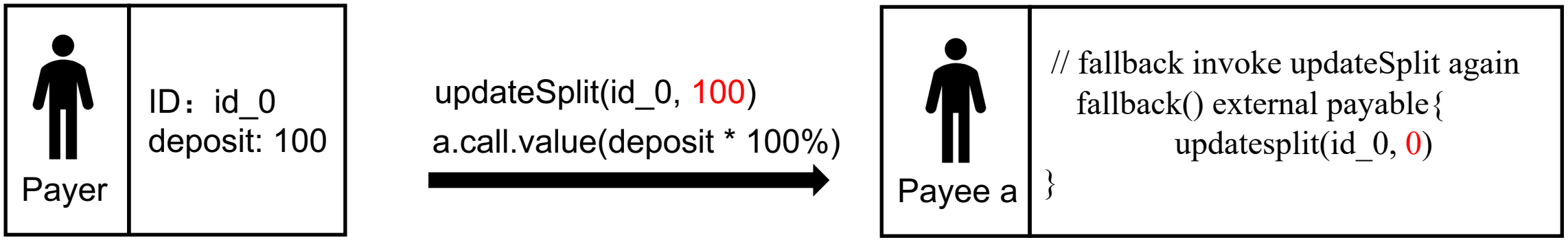
Node

- Storage variable V or
- A statement S operating on a storage variable

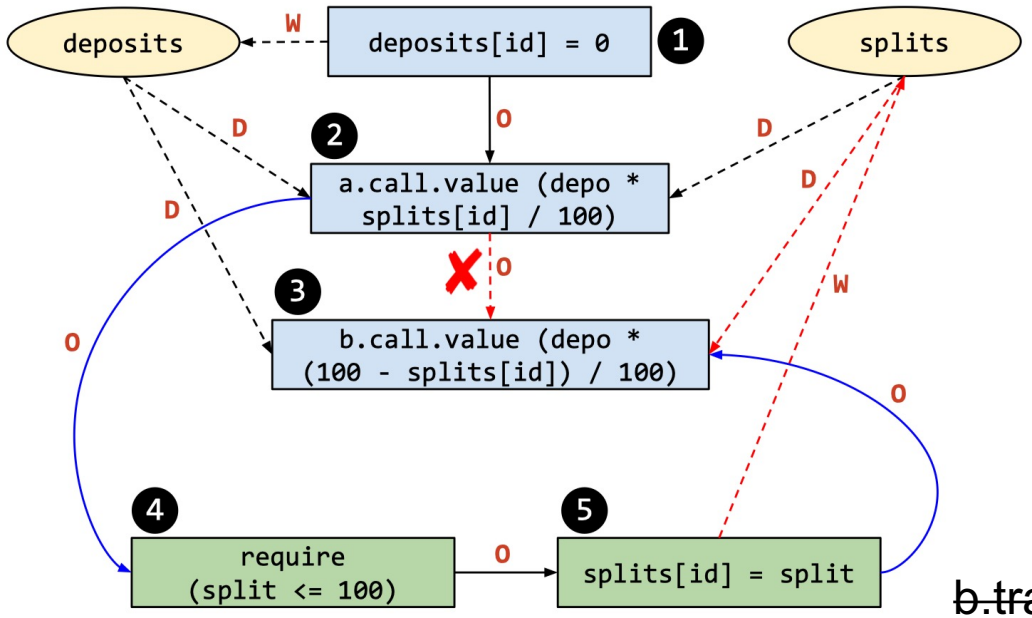
Edge <u,v>

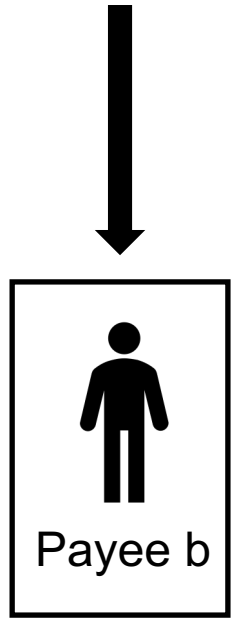
- D: u ∈ V, v ∈ S, and statement v is data-dependent on the state variable u
- W: u ∈ S, v ∈ V, and the state variable v is written by the statement u
- O: u ∈ S, v ∈ S, and statement u precedes statement v in the control-flow graph.

Explorer: SDG




```
1  // [Step 1]: Set split of 'a' (id = 0) to 100(%)
2  // [Step 4]: Set split of 'a' (id = 0) to 0(%)
3  function updateSplit(uint id, uint split) public{
4      require(split <= 100);
5      splits[id] = split;
6  }
7
8  function splitFunds(uint id) public {
9      address payable a = payee1[id];
10     address payable b = payee2[id];
11     uint depo = deposits[id];
12     deposits[id] = 0;
13
14     // [Step 2]: Transfer 100% fund to 'a'
15     // [Step 3]: Reenter updateSplit
16     a.call.value(depo * splits[id] / 100)("");
17
18     // [Step 5]: Transfer 100% fund to 'b'
19     b.transfer(depo * (100 - splits[id]) / 100);
20 }
```





~~b.transfer(deposit * 0%)~~
b.transfer(deposit * 100%)

Explorer: SI Detection

SI bug

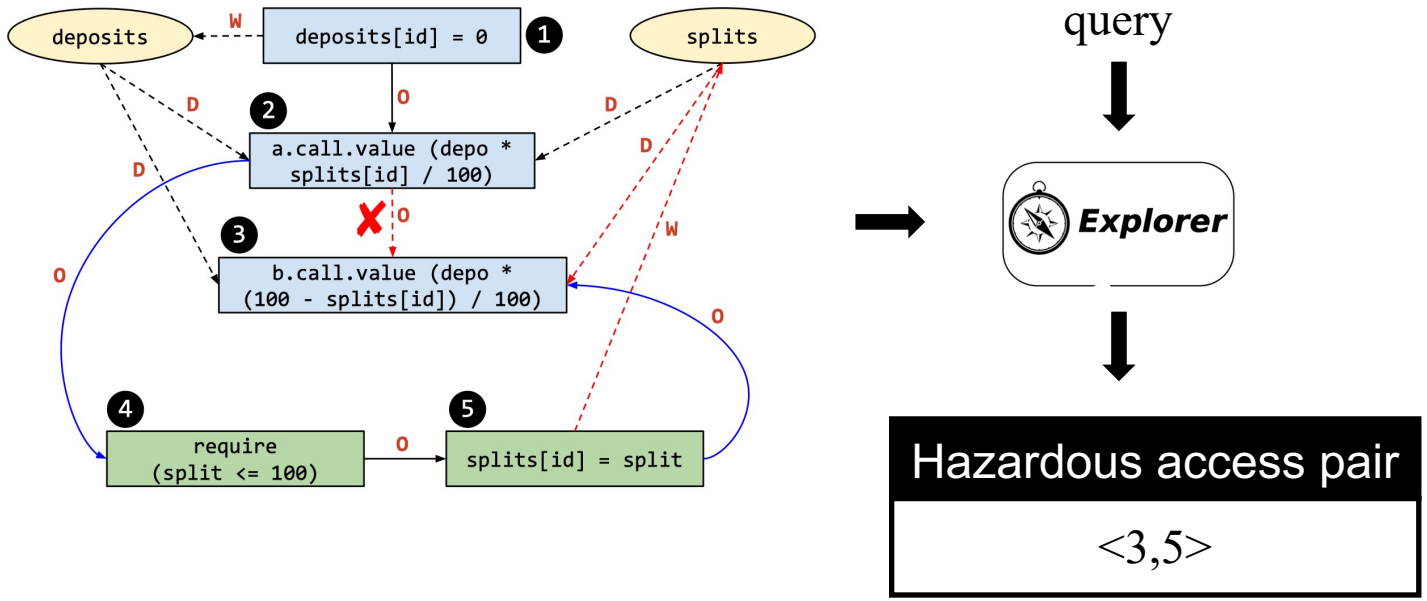
- There exist two operations statement, where at least one is a write access, on a common storage variable.

  Hazardous access pair <s1, s2>

- The relative order of such operations differ in two schedules.

Reentrancy detection

looks for a hazardous access pair <s1 , s2> such that both s1 and s2 are **reachable** from an external call in the SDG, and executable by an attacker.



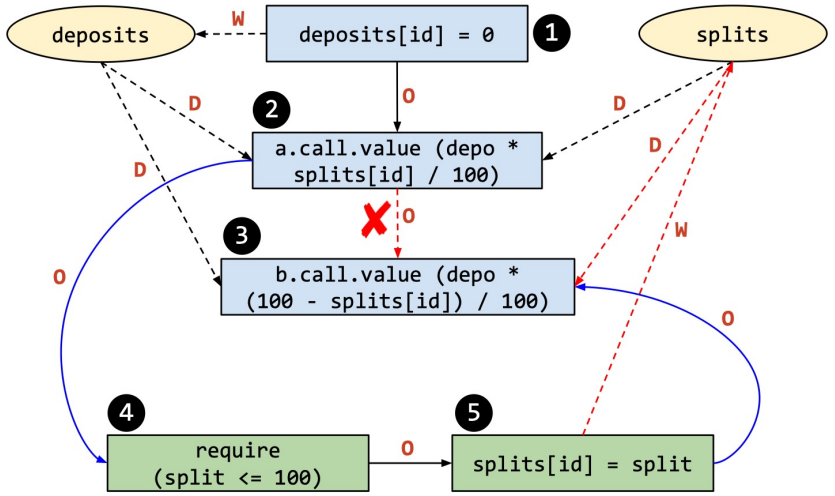
Explorer: Counter-example

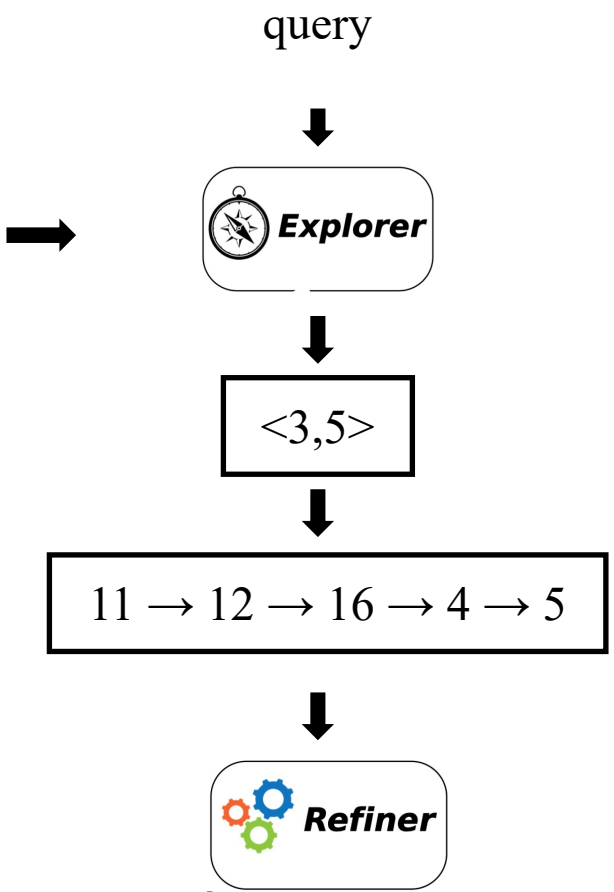
Counter-example: path in ICFG, mapped from SDG, final output of Explorer.

SDG slice: 1 → 2 → 4 → 5 → 3

Counter-example returned by the EXPLORER is： 11 → 12 → 16 → 4 → 5 .

```
1  // [Step 1]: Set split of 'a' (id = 0) to 100(%)
2  // [Step 4]: Set split of 'a' (id = 0) to 0(%)
3  function updateSplit(uint id, uint split) public{
4      require(split <= 100);
5      splits[id] = split;
6  }
7
8  function splitFunds(uint id) public {
9      address payable a = payee1[id];
10     address payable b = payee2[id];
11     uint depo = deposits[id];
12     deposits[id] = 0;
13
14     // [Step 2]: Transfer 100% fund to 'a'
15     // [Step 3]: Reenter updateSplit
16     a.call.value(depo * splits[id] / 100)("");
17
18     // [Step 5]: Transfer 100% fund to 'b'
19     b.transfer(depo * (100 - splits[id]) / 100);
20 }
```



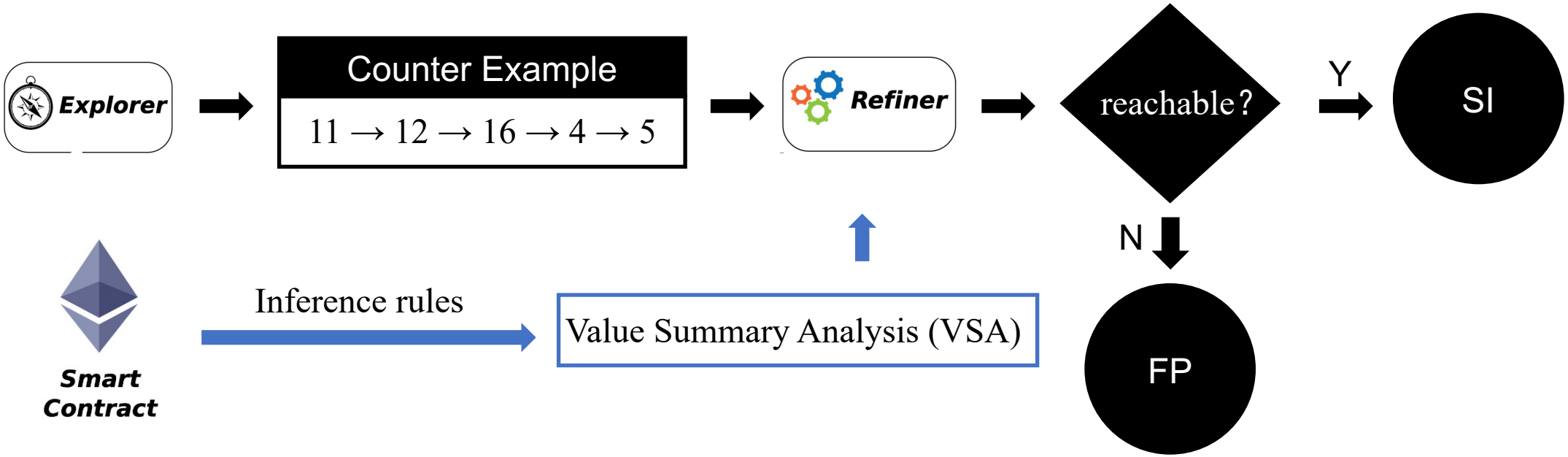


Explorer: False Positive

Due to the over-approximated nature of our SDG that ignores all path conditions, a valid path in SDG does not always map to a feasible execution path in the original ICFG.

```
1  function withdrawBalance(uint amount) public {
2      //[Step 1]: Enter when mutex is false
3      //[Step 4]: Early return, since mutex is true
4      if (mutex == false) {
5        //[Step 2]: mutex = true prevents re-entry
6        mutex = true;
7        if (userBalance[msg.sender] > amount) {
8           //[Step 3]: Attempt to reenter
9           msg.sender.call.value(amount)("");
10          userBalance[msg.sender] -= amount;
11       }
12       mutex = false;
13     }
14 }
15
```

Refiner



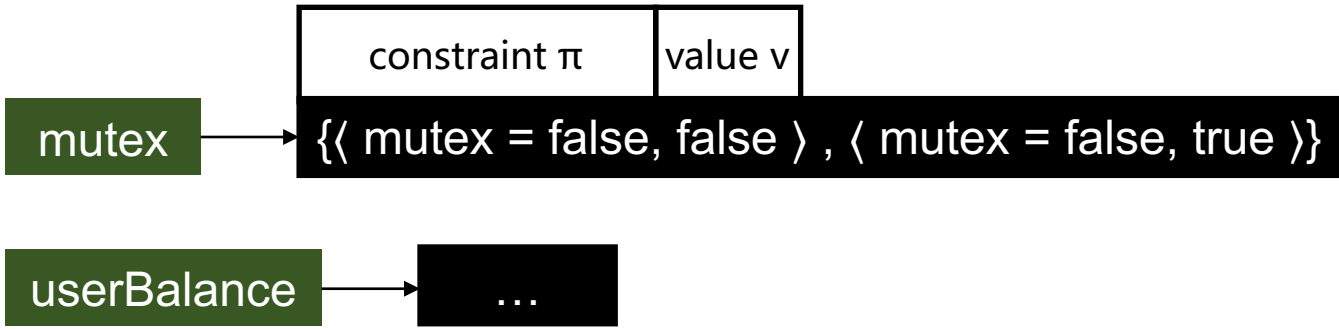
However, a naive symbolic evaluation whose storage variables are completely **unconstrained** will still raise false positives.

Refiner

Value summary analysis (VSA)

For each storage variable s, VSA maps it to a set of pairs ⟨ π, v ⟩ where v is the value of s under the constraint π.

```
1  function withdrawBalance(uint amount) public {
2      //[Step 1]: Enter when mutex is false
3      //[Step 4]: Early return, since mutex is true
4      if (mutex == false) {
5        //[Step 2]: mutex = true prevents re-entry
6        mutex = true;
7        if (userBalance[msg.sender] > amount) {
8           //[Step 3]: Attempt to reenter
9           msg.sender.call.value(amount)("");
10          userBalance[msg.sender] -= amount;
11       }
12       mutex = false;
13     }
14 }
15
```



After invoking any sequence of public functions, if pre-condition *mutex* = *false* holds, mutex can be updated to *true* or *false*.

Refiner

Symbolic Evaluation

- symbolic state σ: when an assignment statement is encountered, add a mapping to σ
- path constraint π: when a conditional statement is encountered, update π

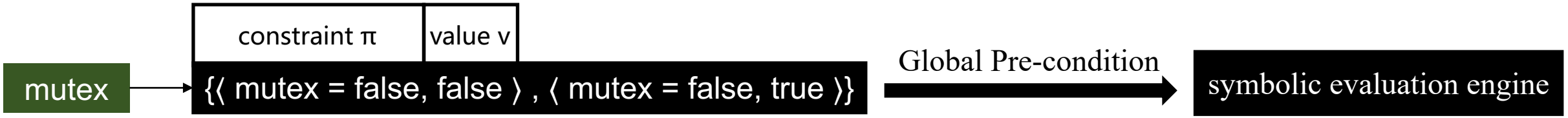
```
1 int f() {
2           ...
3           y = read();                          σ = {y → y0}  π  = true
4           z = y * 2;                           σ = {y → y0, z → 2*y0}  π  = true
5           if (z == 12) {
6                       fail();                  σ = {y → y0, z → 2*y0}  π  = (2*y0 = 12)
7           } else {
8                       printf("OK");            σ = {y → y0, z → 2*y0}  π  = (2*y0 ≠ 12)
9           }
10 }
```

The test input is obtained by solving the constraint path π using the constraint solver.

Refiner

Symbolic Evaluation

① Value summary analysis . Applying the VSA rules to the contract, generates the summary for storage variable mutex.



② Symbolic Evaluation. Applying symbolic checker on the *withdrawBalance()* for the first time, generates the path condition π

```
1  function withdrawBalance(uint amount) public {
2      //[Step 1]: Enter when mutex is false
3      //[Step 4]: Early return, since mutex is true
4      if (mutex == false) {
5        //[Step 2]: mutex = true prevents re-entry
6        mutex = true;
7        if (userBalance[msg.sender] > amount) {
8          //[Step 3]: Attempt to reenter
9          msg.sender.call.value(amount)("");
10         userBalance[msg.sender] -= amount;
11       }
12       mutex = false;
13     }
14 }
15
```

- symbolic state σ: { mutex → true, ... }
- path constraint π = (mutex = false) ∧ (userBalance[msg.sender] > amount)

③ Based on the value summary of mutex in Step 1, the pre-condition to set mutex to false is mutex = false.
However, the pre-condition is not satisfiable under the current state σ { mutex → true, ... }
Thus, Refiner discards the reentrancy report as false positive.

Evalution

Expr 1: Setup

All 91,921 contracts from Etherscan, which cover a period until October 31, 2020.

Code line number	[0,500]	[500,1000]	[1000,∞]
Tag	small	medium	large
Amout	73433	11730	4690

Celery v4.4.4 cluster consisting of six identical machines
Ubuntu 18.04.3 Server
Intel(R) Xeon(R) CPU E52690 v2@3.00 GHz processor (40 core)
256 GB memory.

Evalution

Expr 1: Result

Bug	Tool	Safe	Unsafe	Timeout	Error
Reentrancy	SECURIFY	72,149	6,321	10,581	802
	VANDAL	40,607	45,971	1,373	1,902
	MYTHRIL	25,705	3,708	59,296	1,144
	OYENTE	26,924	269	0	62,660
	SAILFISH	83,171	2,076	1,211	3,395
TOD	SECURIFY	59,439	19,031	10,581	802
	OYENTE	23,721	3,472	0	62,660
	SAILFISH	77,692	7,555	1,211	3,395

TABLE II: Comparison of bug finding abilities of tools

- **safe**: no vulnerability was detected
- **unsafe**: a **potential** state inconsistency bug was detected
- **timeout**: the analysis failed to converge within the time budget (20 minutes)
- **error**: the analysis aborted due to infrastructure issues, e.g., unsupported Solidity version, or a framework bug, etc.

Tool	Small	Medium	Large	Full
SECURIFY	85.51	642.22	823.48	196.52
VANDAL	16.35	74.77	177.70	30.68
MYTHRIL	917.99	1,046.80	1,037.77	941.04
OYENTE	148.35	521.16	675.05	183.45
SAILFISH	9.80	80.78	246.89	30.79

TABLE IV: Analysis times (in seconds) on four datasets.

Evalution

Expr 2

Manual analysis on a randomly sampled subset of 750 contracts ranging up to 3, 000 lines of code, out of a total of 6,581 contracts successfully analyzed by all five static analysis tools, without any timeout or error.

Tool	Reentrancy			TOD		
	TP	FP	FN	TP	FP	FN
SECURIFY	9	163	17	102	244	8
VANDAL	26	626	0	–	–	–
MYTHRIL	7	334	19	–	–	–
OYENTE	8	16	18	71	116	39
SAILFISH	26	11	0	110	59	0

TABLE III: Manual determination of the ground truth

- TP: Correct Detection
- FP: Incorrect Detection
- FN: Missed detection

Evalution

zero-day bugs

Out of total 401 reentrancy-only and 721 TOD-only contracts, we manually selected 88 and 107 contracts, respectively. We limited our selection effort only to contracts that contain at most 500 lines of code, and are relatively easier to reason about in a reasonable time budget. Our manual analysis confirms **47 contracts are exploitable** (not just vulnerable)—meaning that they can be leveraged by an attacker to accomplish a malicious goal.

Limitations

- Source-code dependency
- Potential unsoundness: VSA(value-summary analysis) not expressive enough to model all the complex aspects