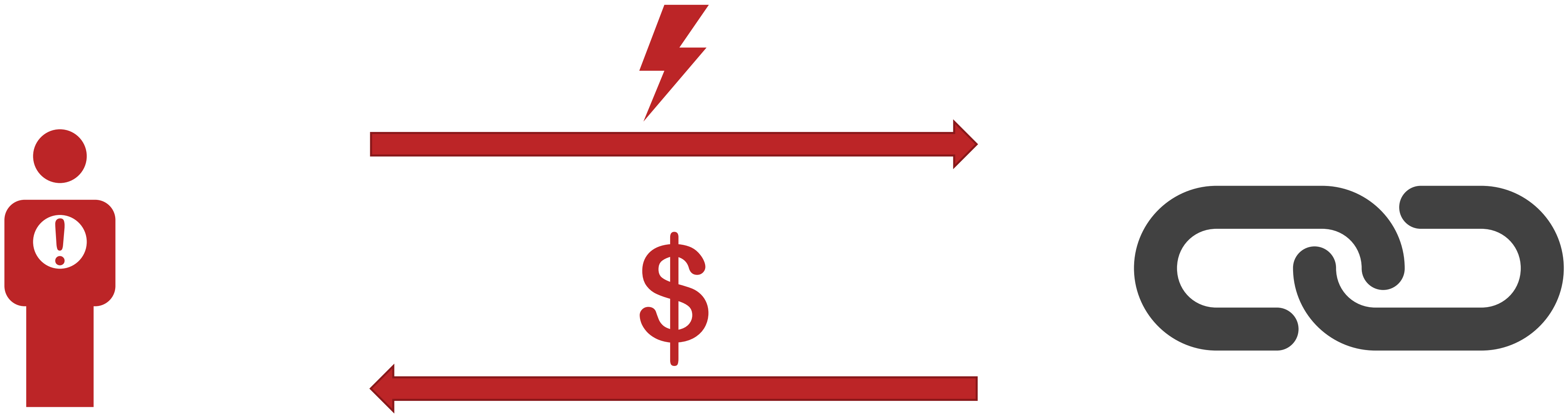
teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts

Johannes Krupp, Christian Rossow

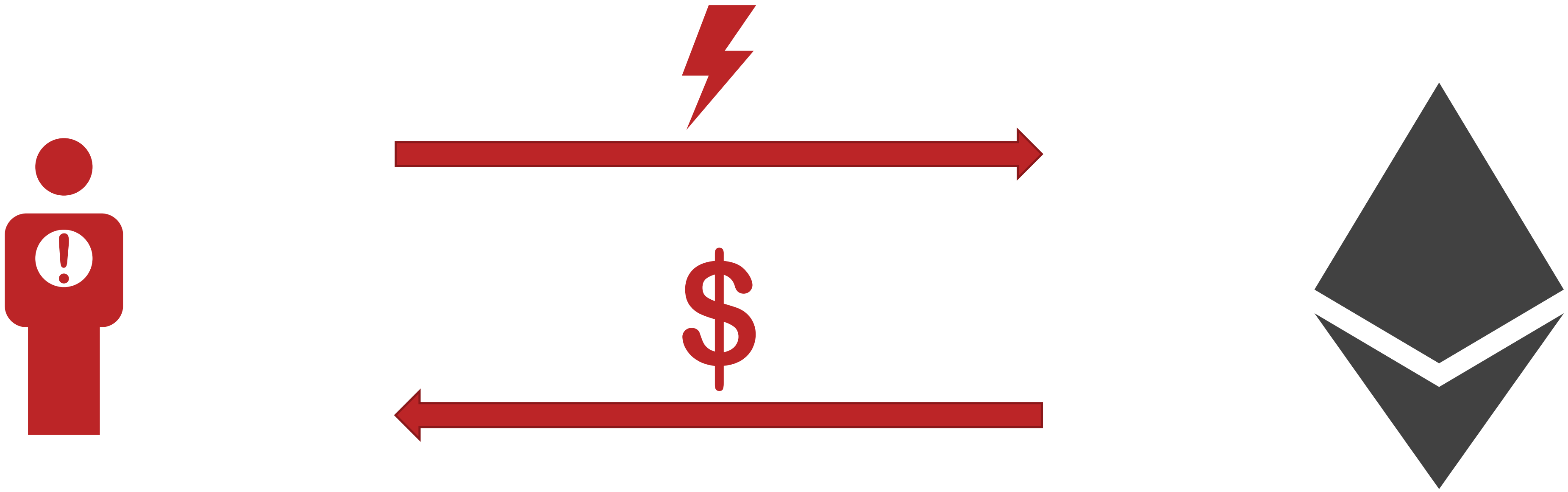
European Commission

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 700176

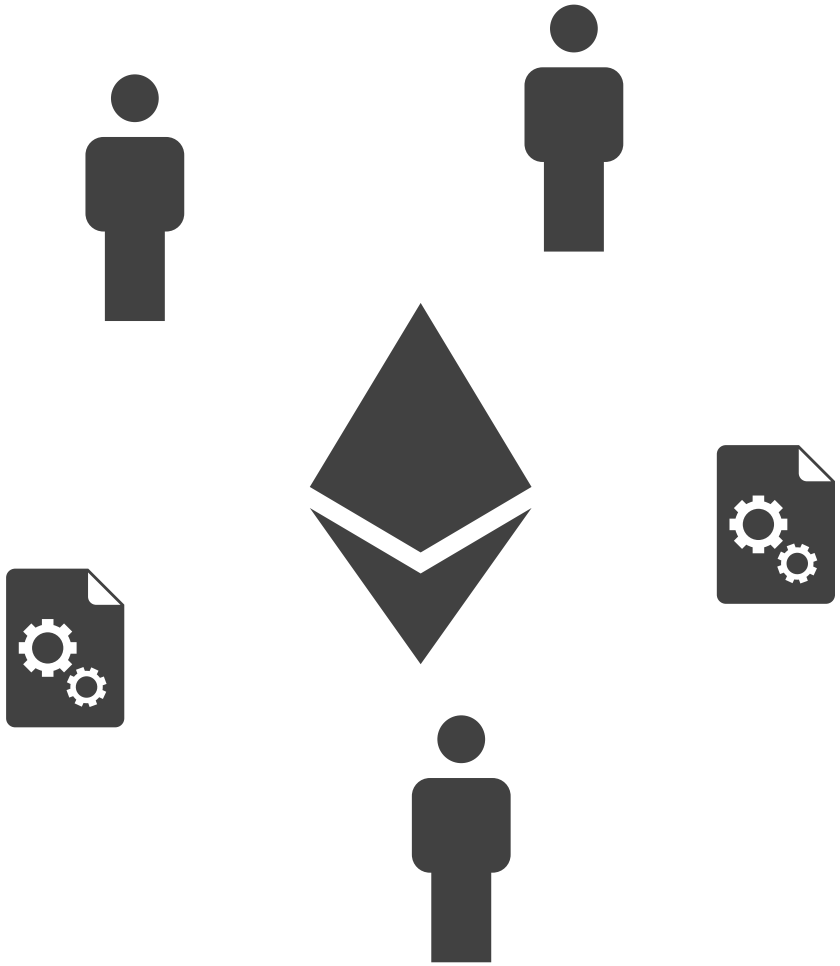
In a nutshell

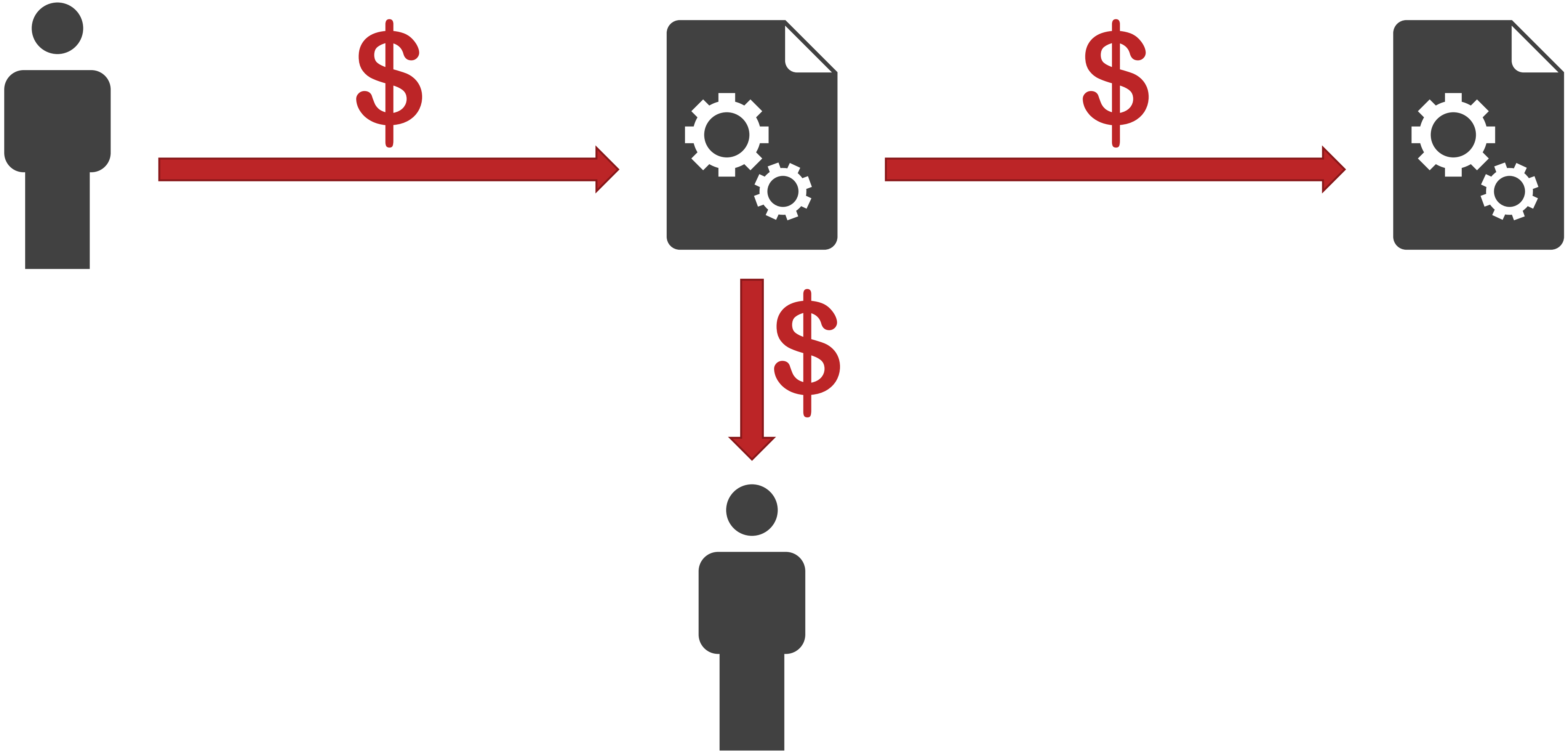


In a nutshell

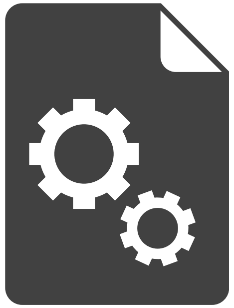


Ethereum

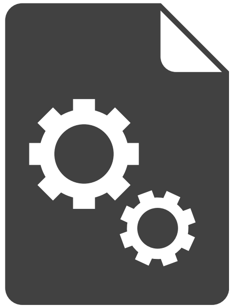




Smart Contracts



- Ethereum Virtual Machine (EVM) bytecode
- executed on incoming transaction
- otherwise like regular account:
  - address
  - balance
- use cases:
  - crowdfunding schemes
  - shared wallets
  - games
  - …



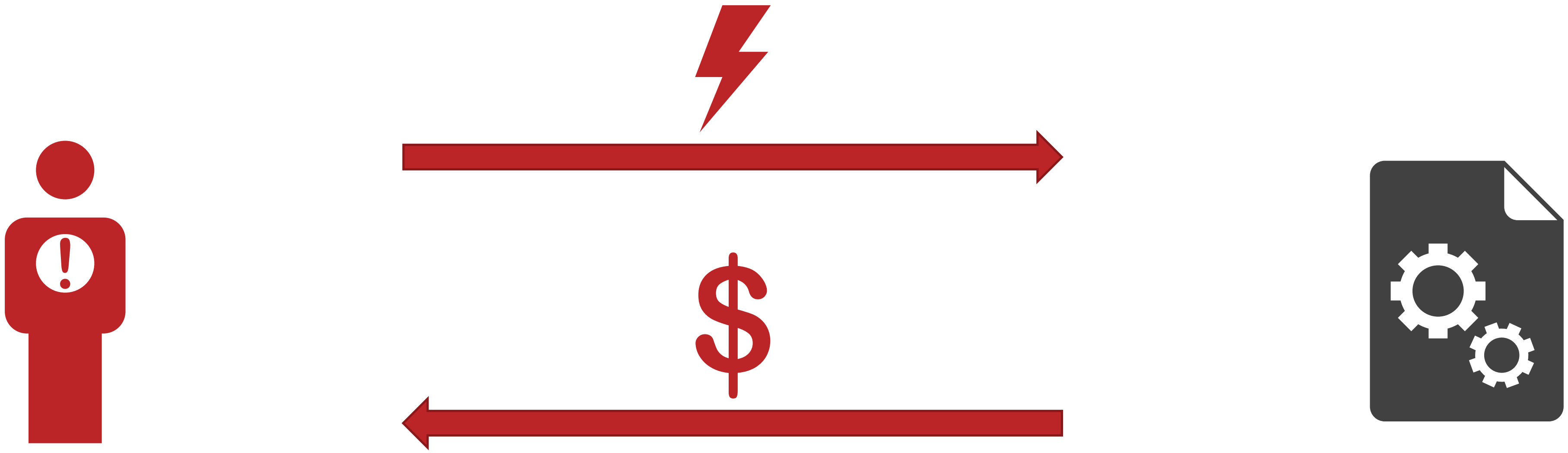
- Ethereum Virtual Machine (EVM) bytecode
- executed on incoming transaction
- otherwise like regular account:
  - address
  - balance

may contain bugs

cannot be updated

goal: find & exploit bugs

Smart Contract Exploitation



Transactions

- `from` sender
- `to` recipient
- `value` transferred amount
  - may also be zero
- `gas` „transaction fee“
- `data` input data
  - may be empty

Ethereum Virtual Machine (EVM)

- stack machine
- 256 bit wordsize
- ~70 instructions
  - arithmetic
  - logic
  - control flow
  - blockchain interaction

Challenges

- control flow graph recovery

Control Flow Instructions

- **`JUMP`**
  - unconditional jump
  - jump to **`target`**

- **`JUMPI`**
  - conditional jump
  - jump to **`target`** if **`test`** is non-zero

- **`JUMPDEST`**
  - marks valid jump target
  - no op

CFG Recovery

600934600757005b565b00

JUMP

JUMPI

JUMPDEST

CFG Recovery

```
0   6009   PUSH1 09
2   34     CALLVALUE
3   6007   PUSH1 07
5   57     JUMPI

6   00     STOP

7   5b     JUMPDEST
8   56     JUMP

9   5b     JUMPDEST
a   00     STOP
```

CFG Recovery

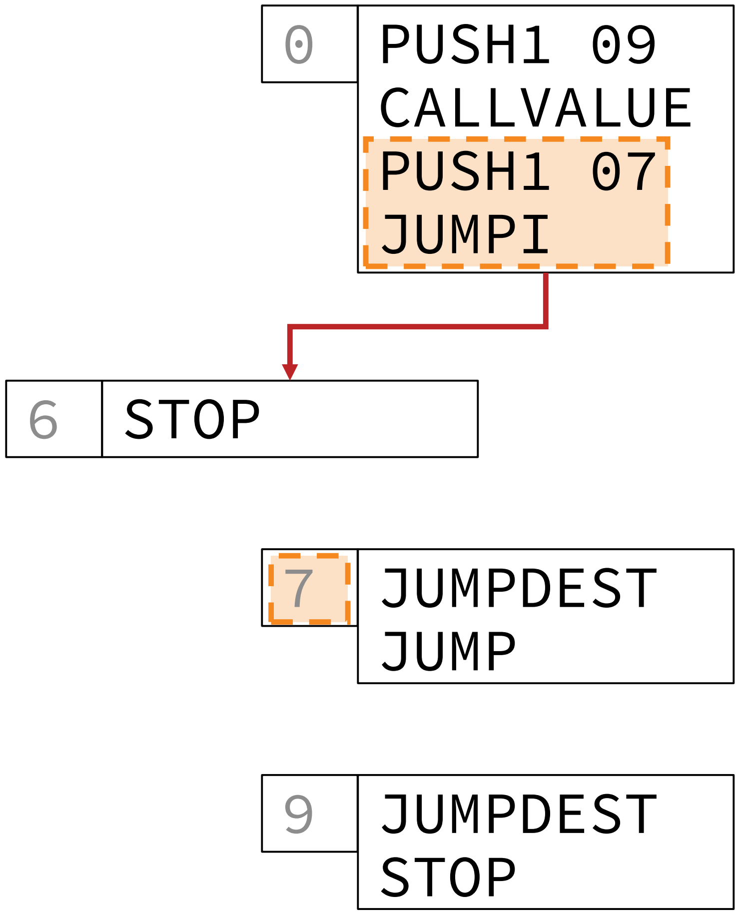
```
0  PUSH1 09
   CALLVALUE
   PUSH1 07
   JUMPI
```

```
6  STOP
```

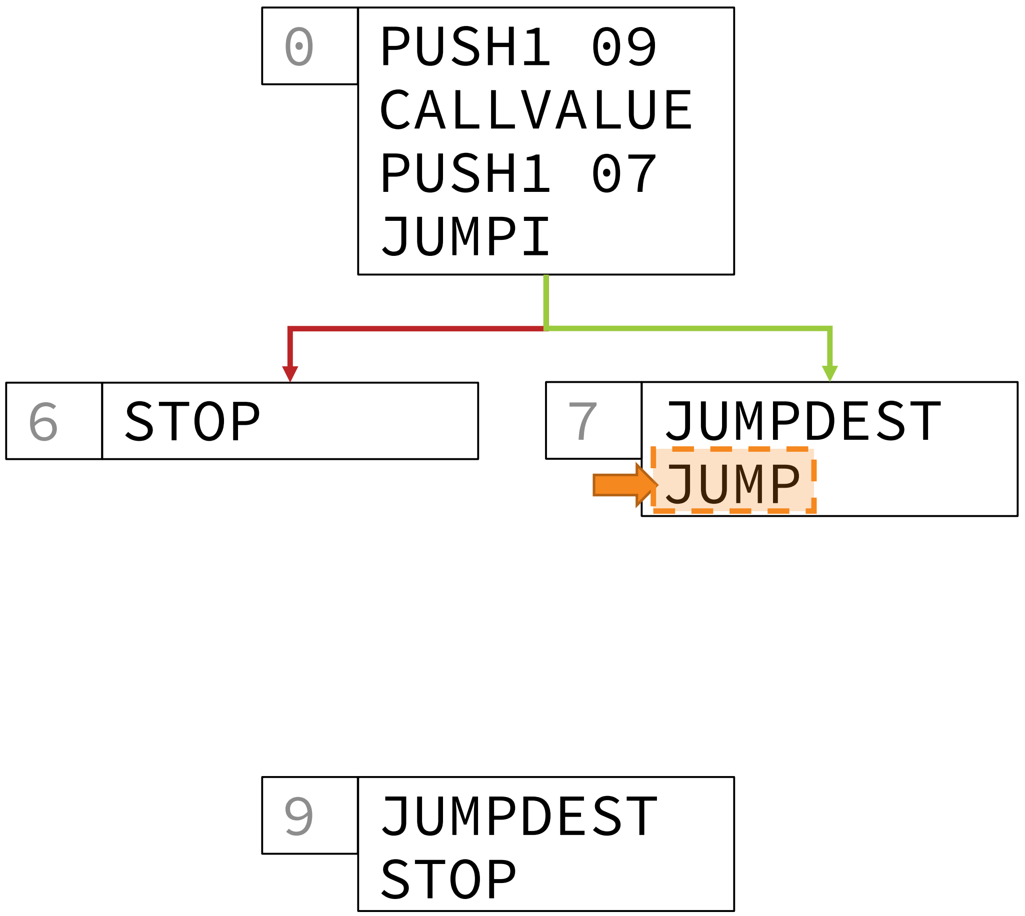
```
7  JUMPDEST
   JUMP
```

```
9  JUMPDEST
   STOP
```

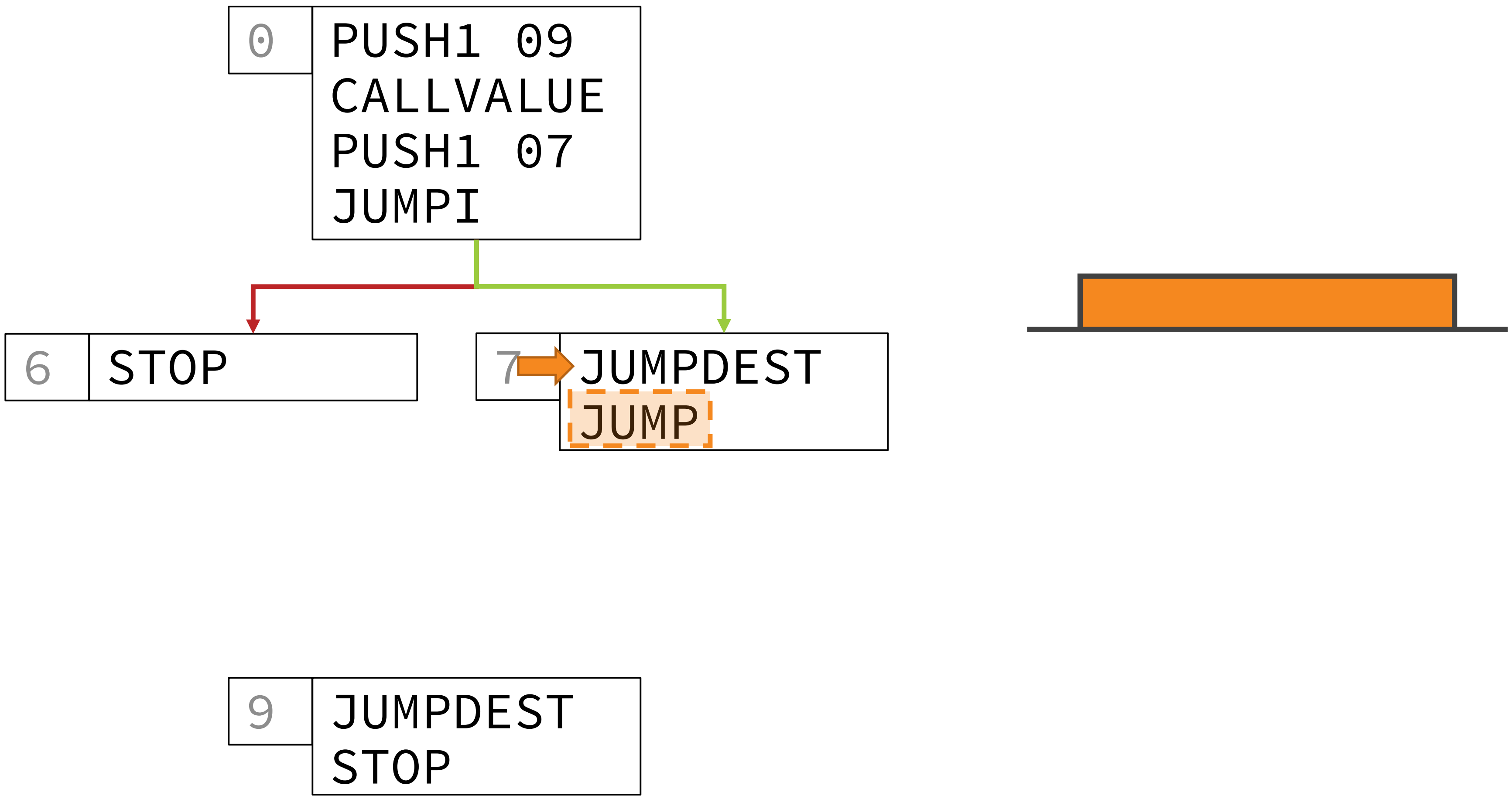
CFG Recovery

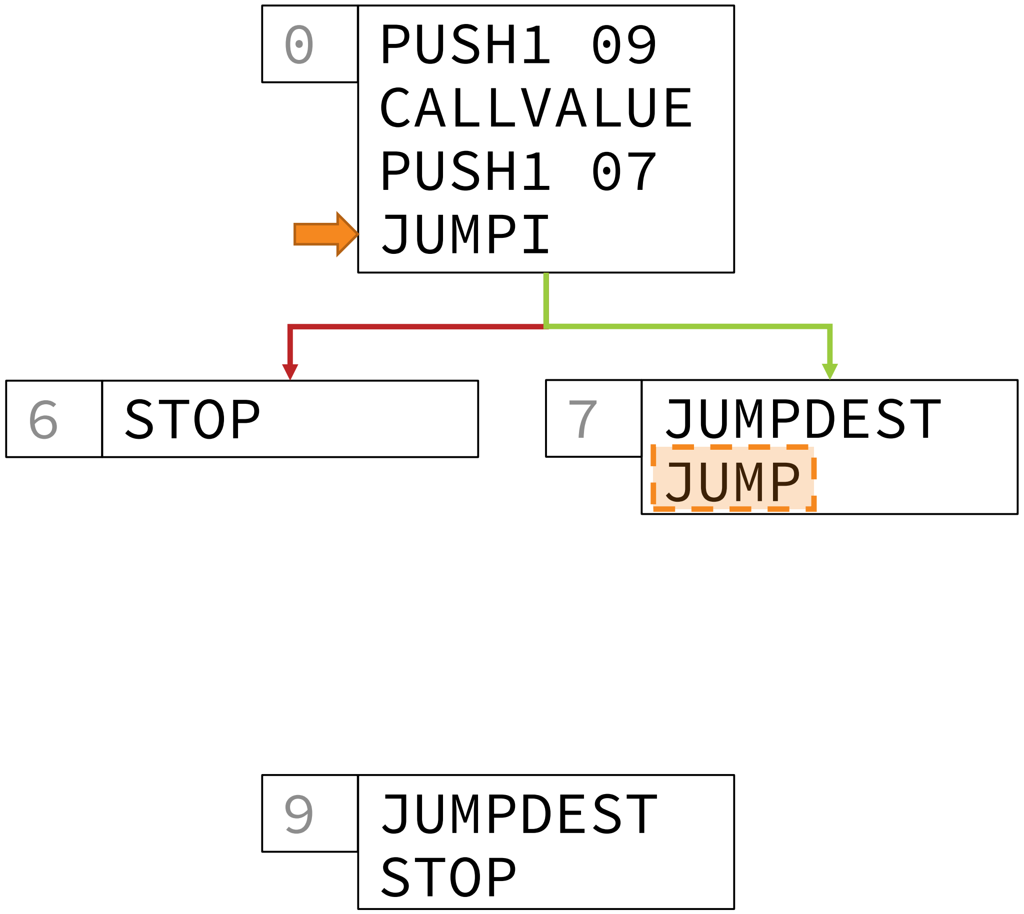


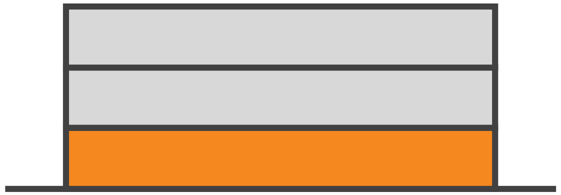
CFG Recovery



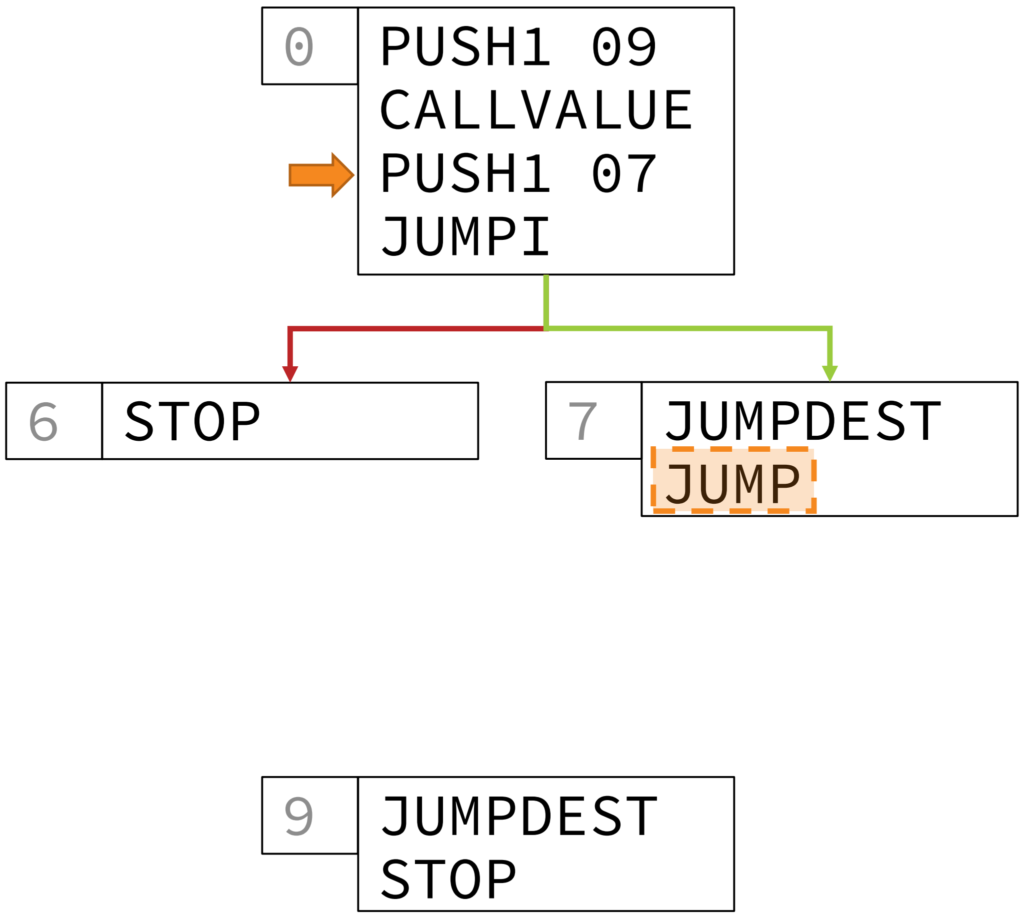
CFG Recovery

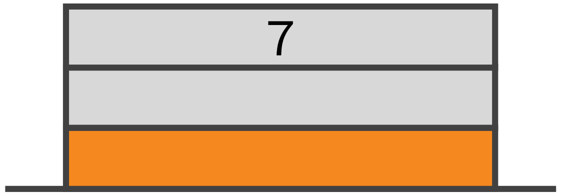



0
PUSH1 09
CALLVALUE
PUSH1 07
JUMPI
6
STOP
7
JUMPDEST
JUMP
9
JUMPDEST
STOP

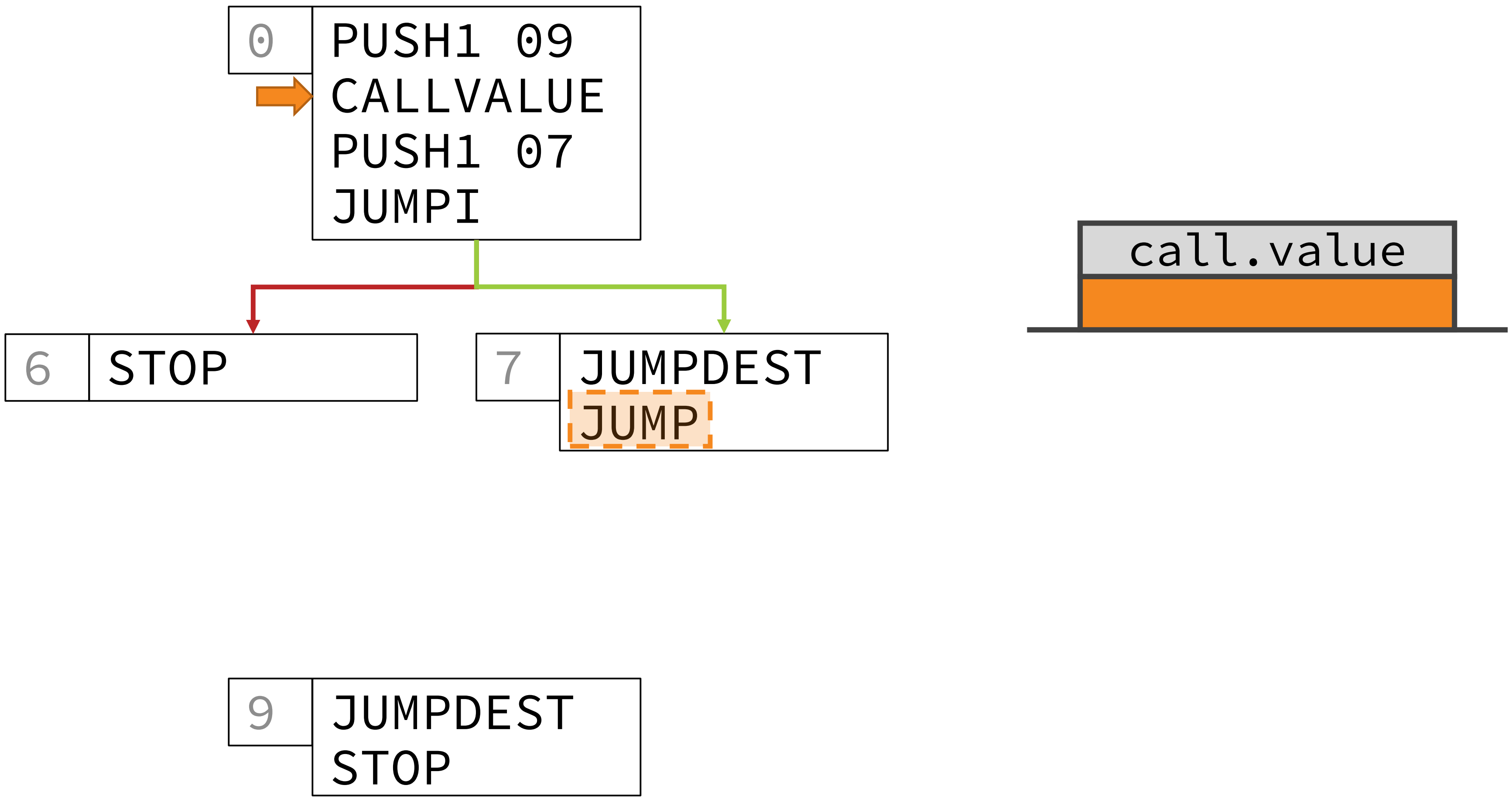


CFG Recovery

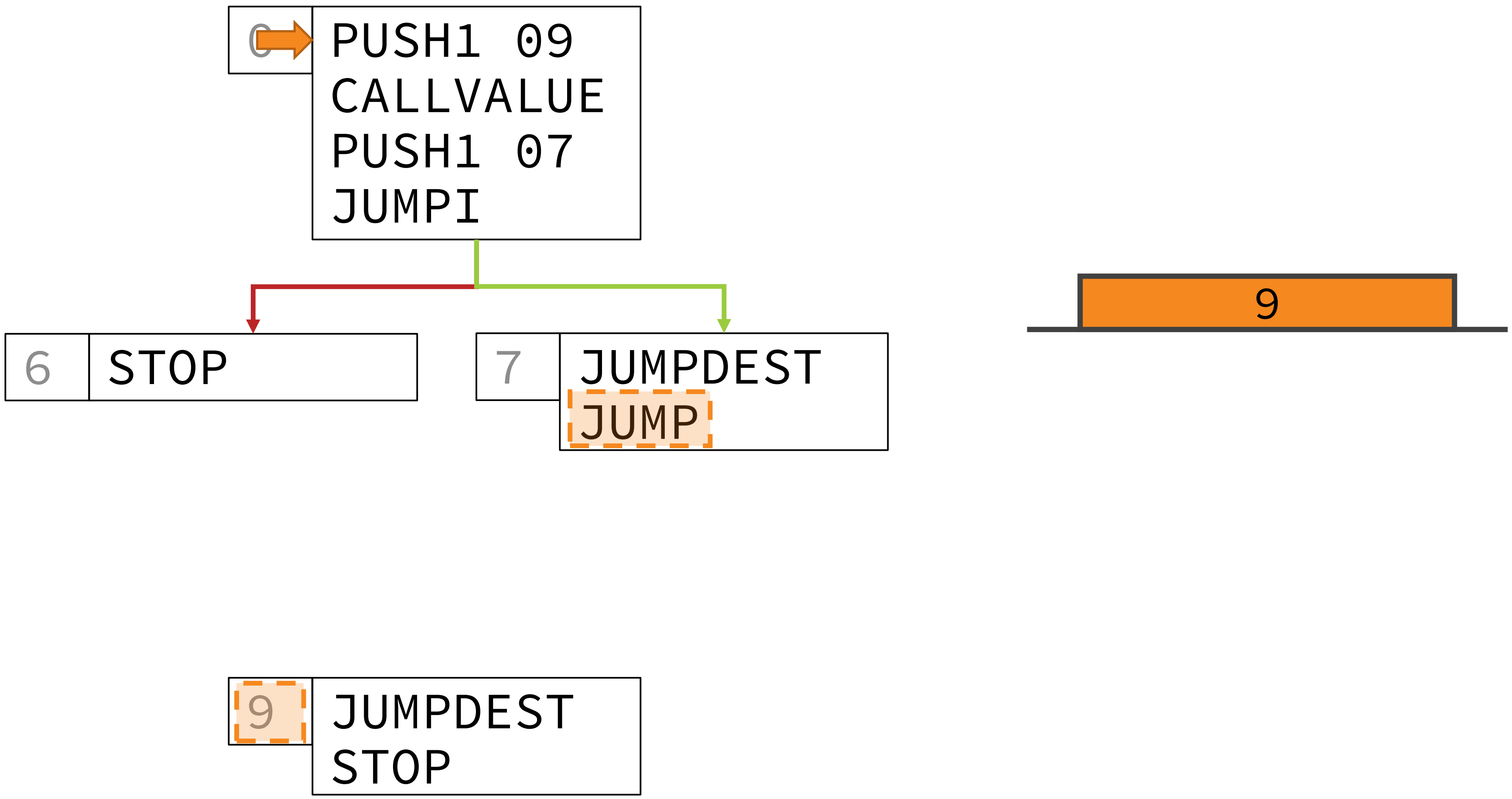




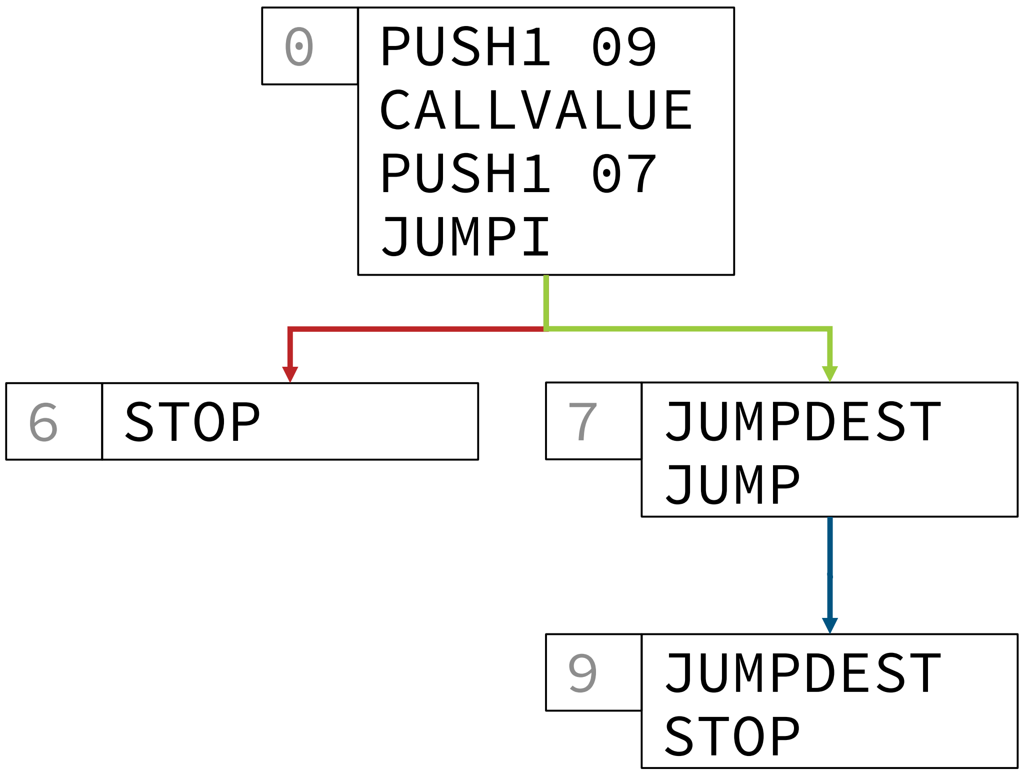
CFG Recovery



CFG Recovery



CFG Recovery



Challenges

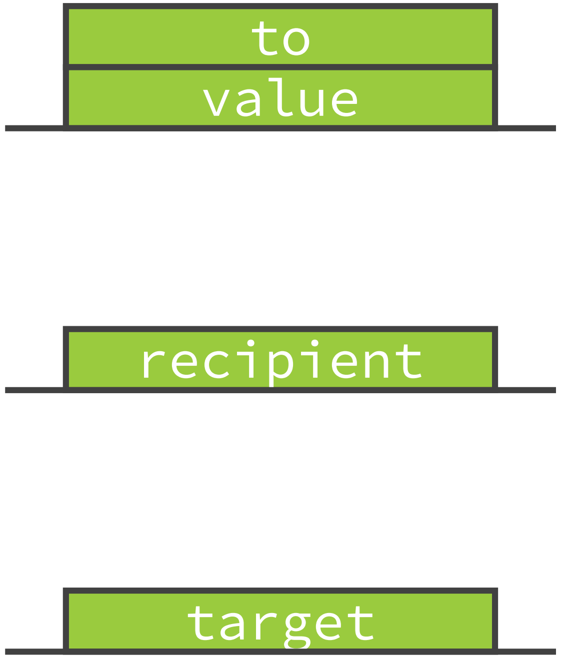
- control flow graph recovery ✓

how can we get money from a contract?

Ethereum Virtual Machine (EVM)

- stack machine
- 256 bit wordsize
- ~70 instructions
  - arithmetic
  - logic
  - control flow
  - **blockchain interaction**

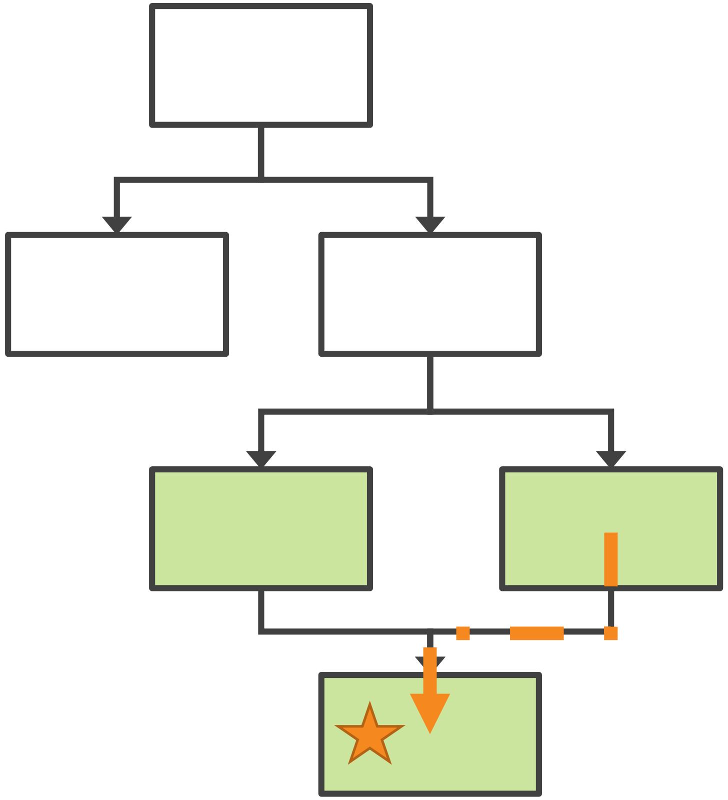
Critical Instructions



- **`CALL`**
  - regular transaction
  - transfer **`value`** to **`to`**
- **`SELFDESTRUCT`**
  - contract destruction
  - transfer funds to **`recipient`**
- **`CALLCODE`** / **`DELEGATECALL`**
  - execute code of **`target`**
  - „code injection“

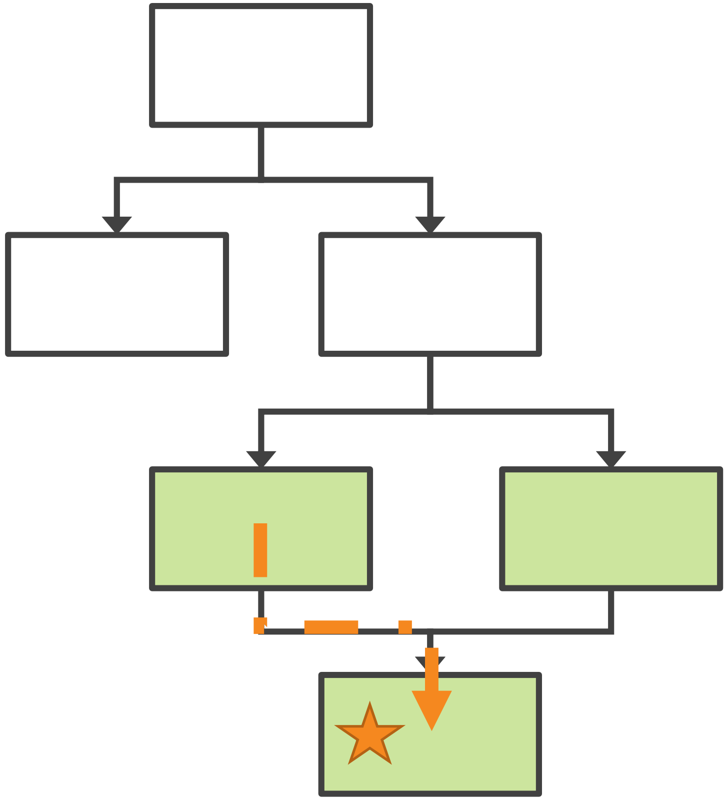
must execute one of these

Critical Paths

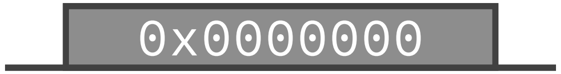


1. locate critical instructions
2. compute backward slices of argument(s)
3. filter for attacker controlled slices

Critical Paths

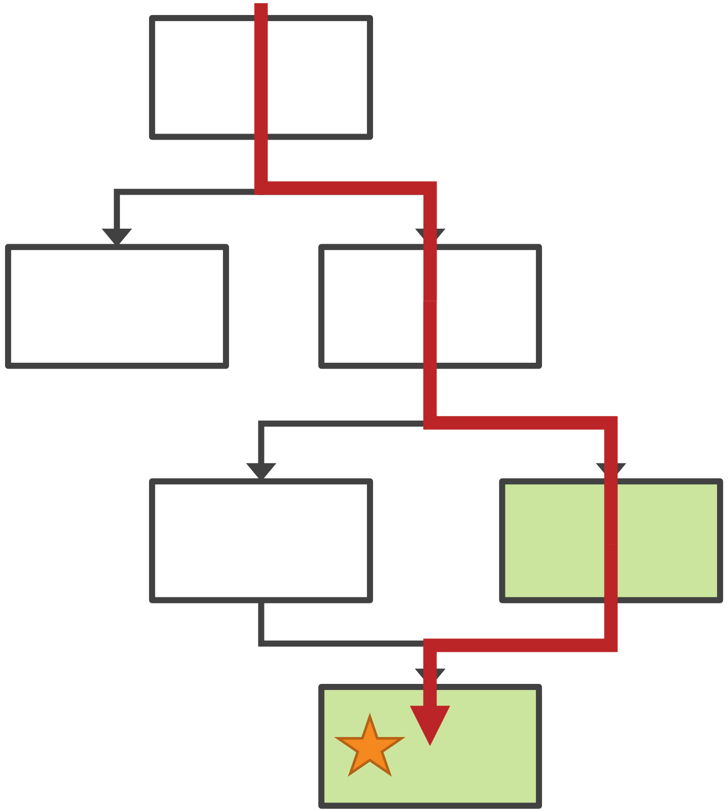


1. locate critical instructions
2. compute backward slices of argument(s)
3. filter for attacker controlled slices



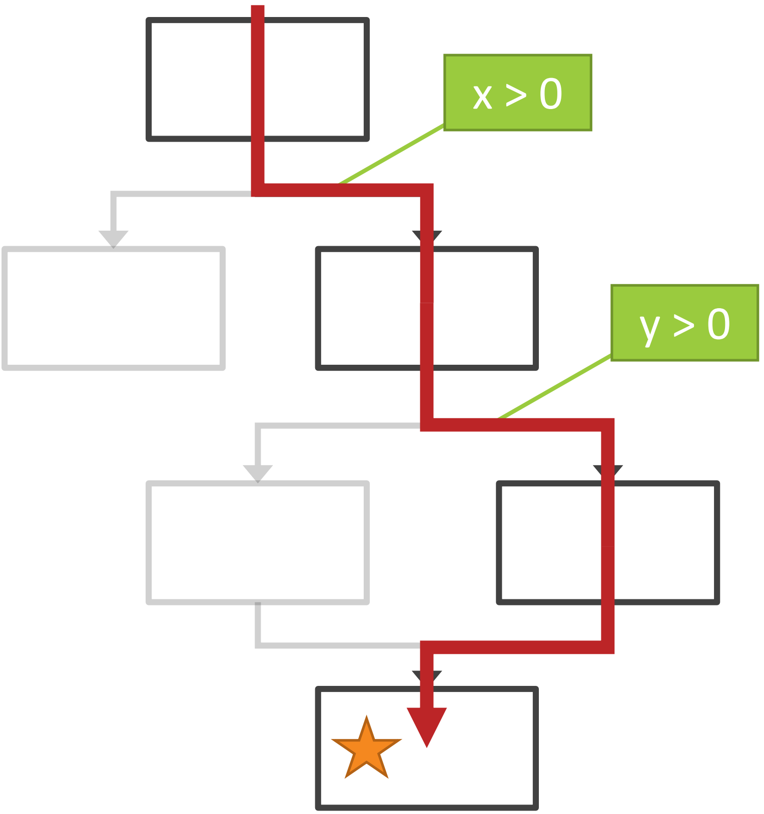
Critical Paths

CISPA HELMHOLTZ-ZENTRUM i. G.



1. locate critical instructions
2. compute backward slices of argument(s)
3. filter for attacker controlled slices
4. generate path through a slice

Exploit Generation – General Approach



1. locate critical instructions
2. compute backward slices of argument(s)
3. filter for attacker controlled slices
4. generate path through a slice
5. execute path symbolically
    - collect path constraints
6. use constraint solver
    - unsatisfiable: generate next path
    - satisfiable: done

Challenges

- control flow graph recovery ✓
- contract state
- **SHA3** instruction

Contract State

```
contract Stateful{

  bool vulnerable = false;

  function exploit(address attacker){
    require(vulnerable);
    attacker.transfer(this.balance);
  }

  function makeVulnerable(){
    vulnerable = true;
  }

}
```

state at bytecode level?

EVM – Memory Model

CISPA HELMHOLTZ-ZENTRUM i. G.

Stack

- stack
- 256 bit words
- volatile

Memory

- array
- byte-addressable
- volatile

Storage

- map/dictionary
- 256 bit keys, 256 bit values
- persistent

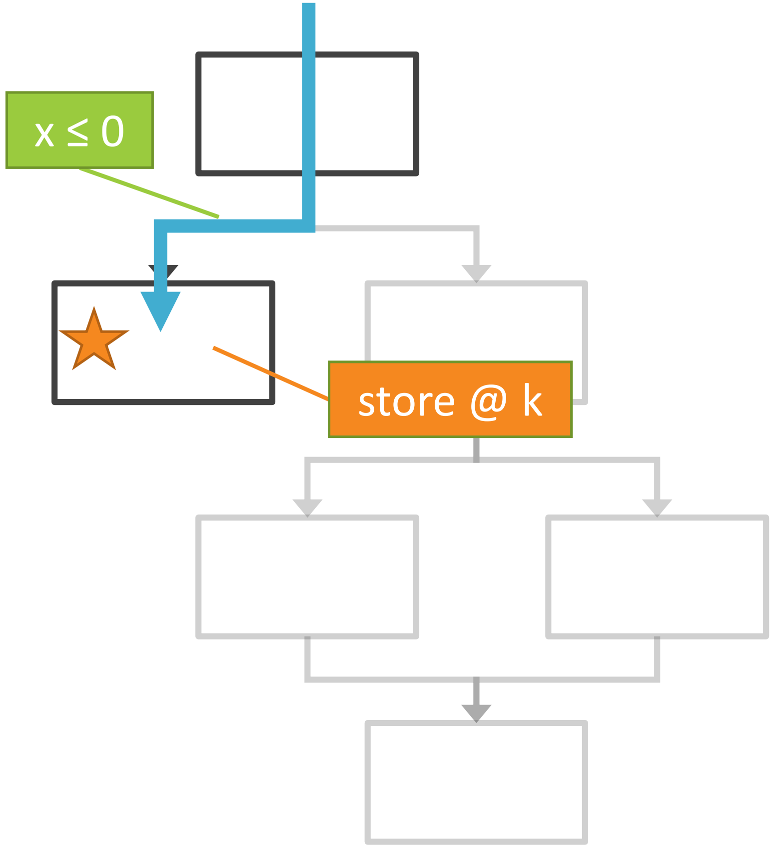
state change = storage change

Storage Instructions

- **`SLOAD`**
  - load value for **`key`**

- **`SSTORE`**
  - store **`value`** at **`key`**

State Changing Paths



1. locate **SSTORE** instructions
2. compute backward slices of argument(s)
3. generate path through a slice
4. execute path symbolically
   - collect path constraints C
   - collect storage reads R & writes W

$$C = \{x \leq 0\}$$

$$R = \emptyset$$

$$W = \{k\}$$

Path Stitching

- combine n state changing paths + 1 critical path

$$C_{\color{blue}\rightarrow} = \{x \leq 0\}$$

$$R_{\color{blue}\rightarrow} = \emptyset$$

$$W_{\color{blue}\rightarrow} = \{k\}$$

$$C_{\color{red}\rightarrow} = \{x > 0, y > 0\}$$

$$R_{\color{red}\rightarrow} = \{k\}$$

$$W_{\color{red}\rightarrow} = \emptyset$$

$$C_{\color{blue}\rightarrow\color{red}\rightarrow} = C^*_{\color{blue}\rightarrow} \cup C^*_{\color{red}\rightarrow} = \{x_0 \leq 0, x_1 > 0, y_1 > 0\}$$

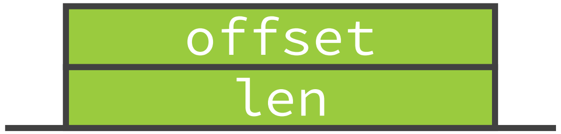
$$R_{\color{blue}\rightarrow\color{red}\rightarrow} = R_{\color{red}\rightarrow} \setminus W_{\color{blue}\rightarrow} \cup R_{\color{blue}\rightarrow} = \emptyset$$

$$W_{\color{blue}\rightarrow\color{red}\rightarrow} = W_{\color{blue}\rightarrow} \cup W_{\color{red}\rightarrow} = \{k\}$$

Challenges

- control flow graph recovery ✓
- contract state ✓
- **SHA3** instruction

SHA3 Instruction



- **SHA3**
  - compute Keccak-256 hash over `memory[offset : offset + len]`
  - used to implement Solidity's `mapping` type

```
function check(bytes32 data, bytes32 check){
   require(data == "1337" && sha3(data) == check)
   //…
```

$$C = \{data = \text{"1337"}, sha3(data) \ = \ check\}$$

How to solve $sha3(data) \ = \ check$?

SHA3 Instruction

dependent expression

$$C = \{data = "1337", sha3(data) = check\}$$

dependent constraint

---

1. remove dependent constraints
2. solve reduced set
3. compute hash values
4. replace dependent constraints
5. repeat

$$C' = \{data = "1337"\}$$

$$sha3(data) \rightarrow 0x985d..$$

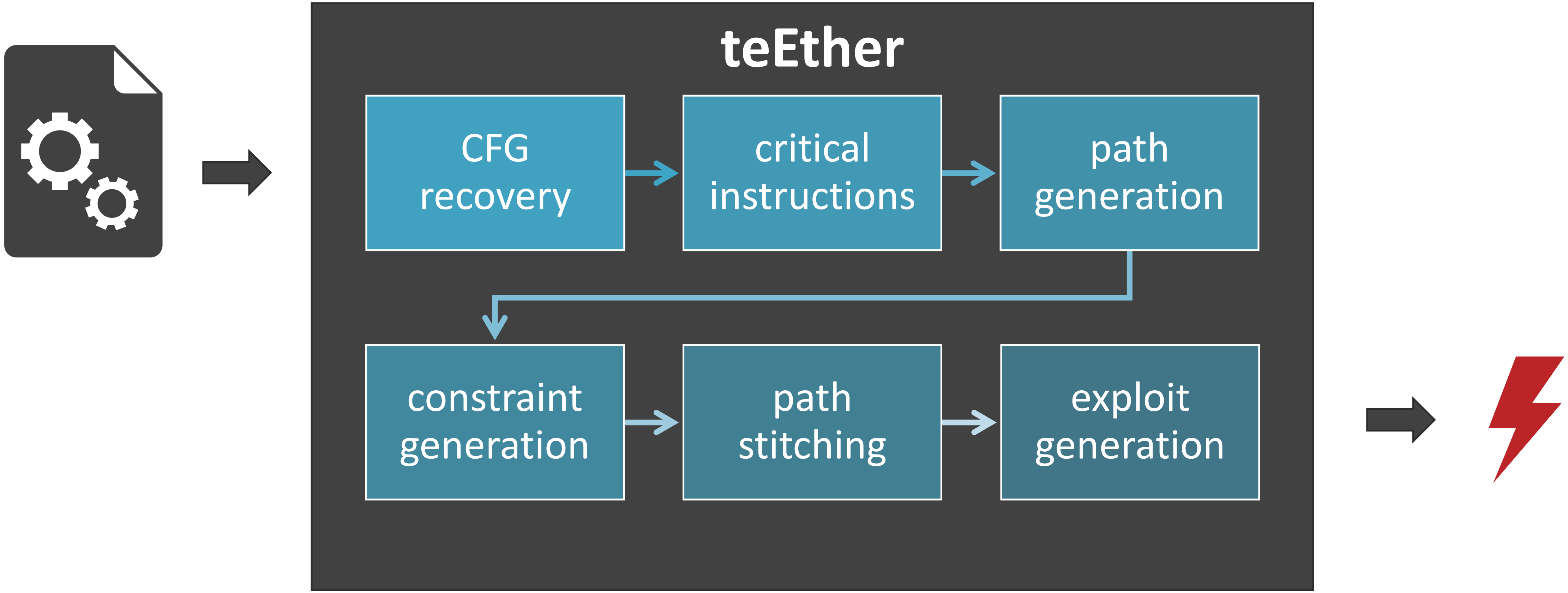
$$C' = \{data = "1337", 0x985d.. = check\}$$

independent

Challenges

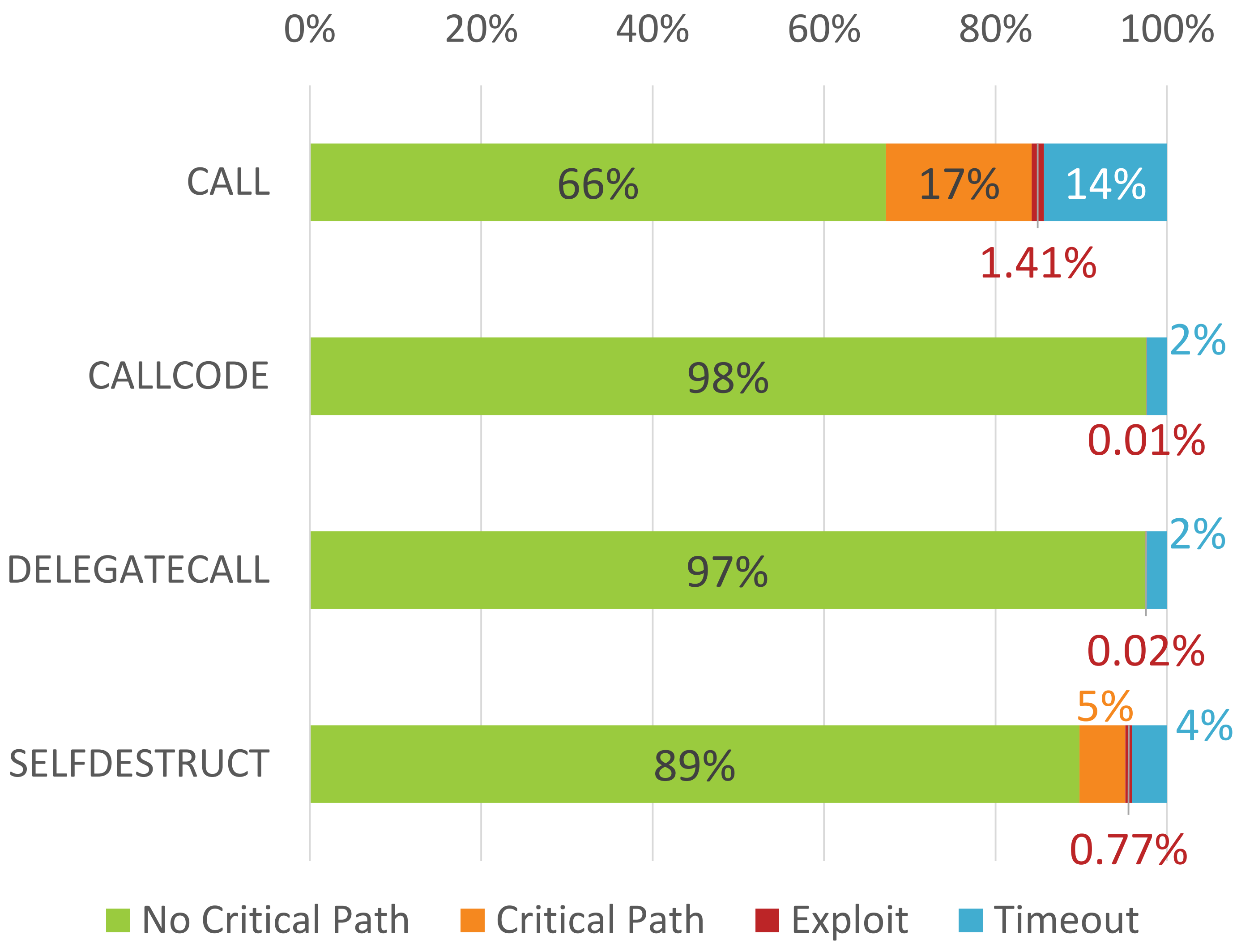
- control flow graph recovery ✓
- contract state ✓
- **SHA3** instruction ✓

teEther


teEther
CFG recovery
critical instructions
path generation
constraint generation
path stitching
exploit generation

Evaluation

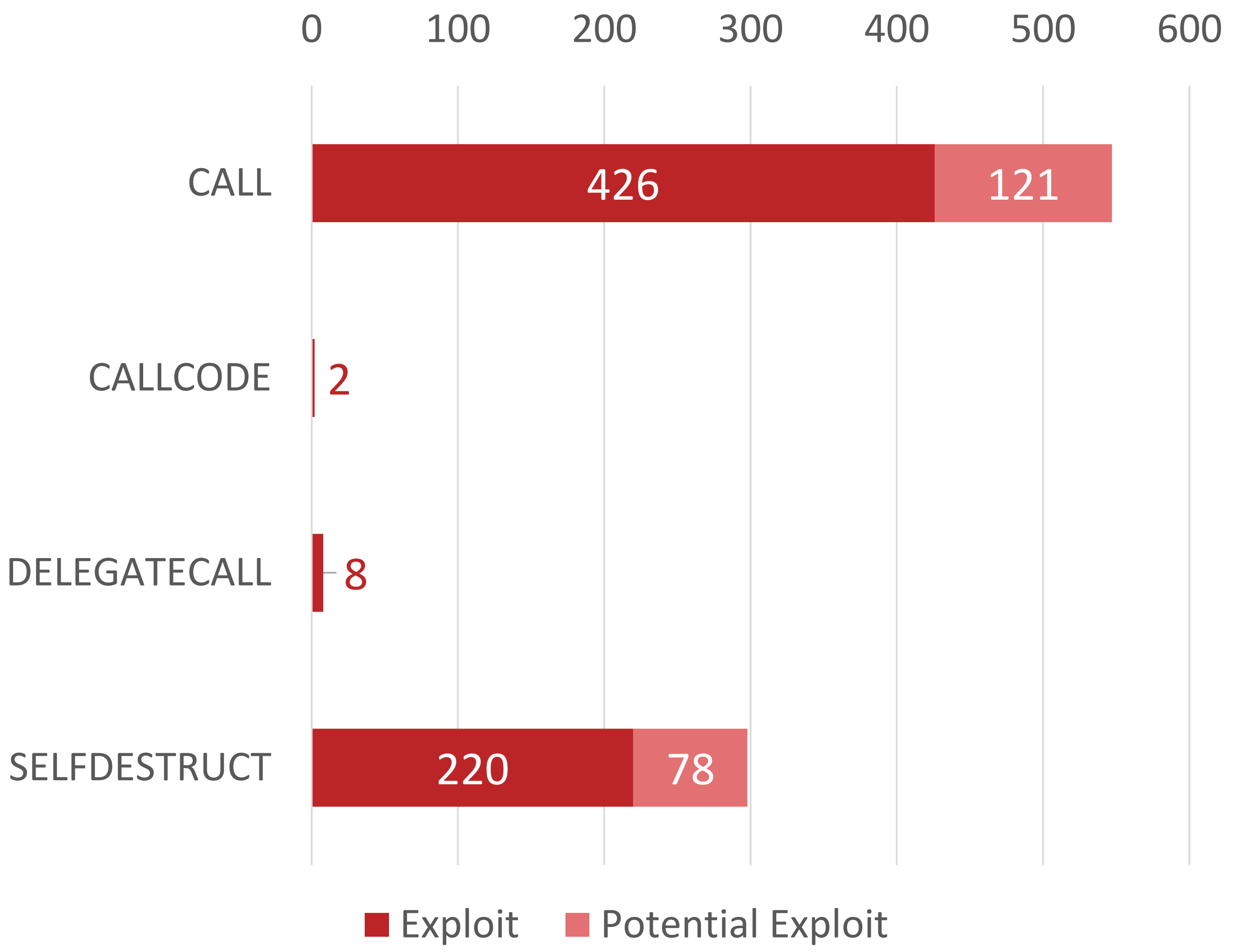
- contracts from blockchain
- 784,344 total
- 38,757 unique
- 30 min CFG recovery +
  30 min exploit generation



Evaluation - Exploits

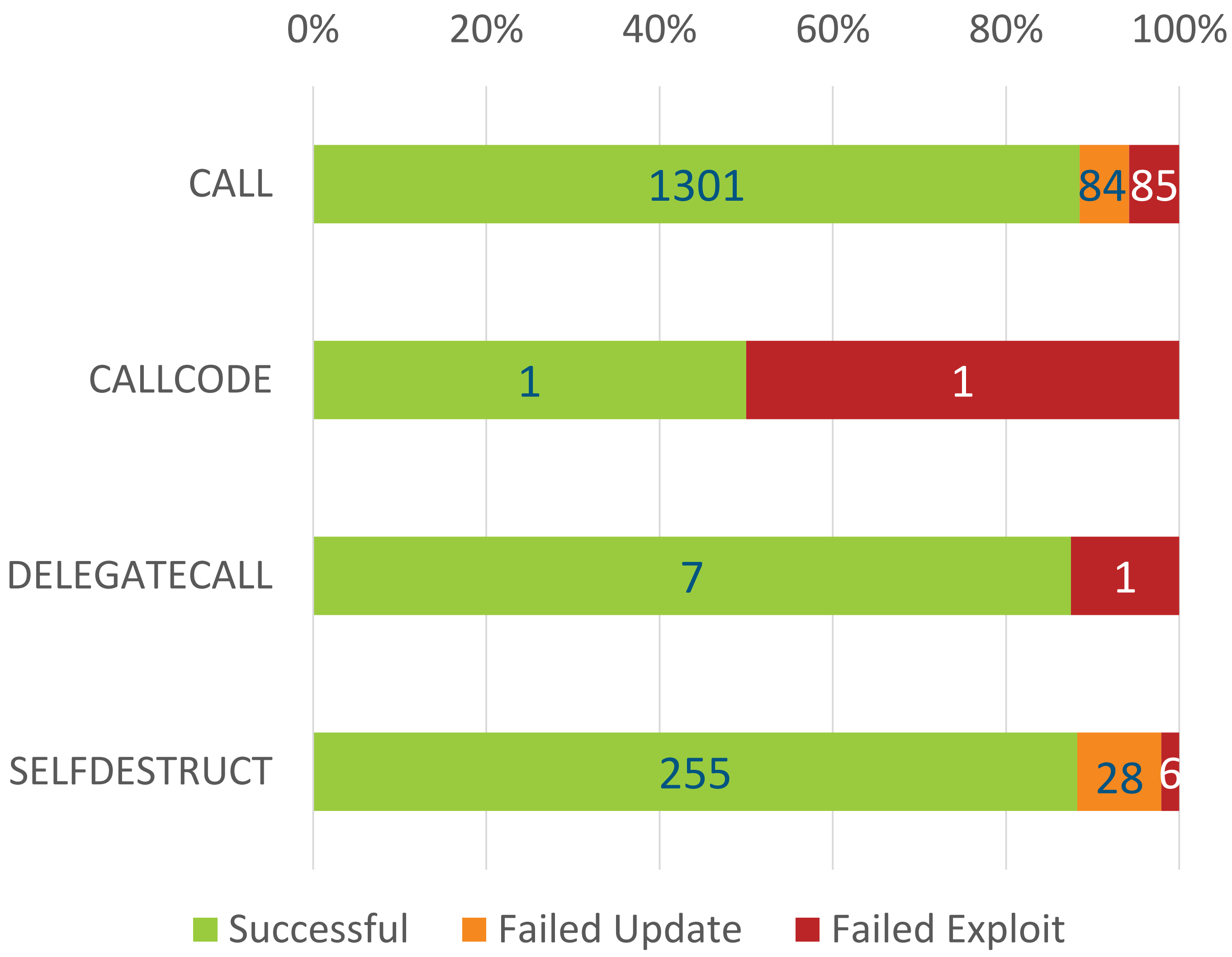
- contracts from blockchain
- 784,344 total
- 38,757 unique
- 30 min CFG recovery +
  30 min exploit generation

- 630 unique exploits
- 1,731 affected contracts
- 1,769 total exploits



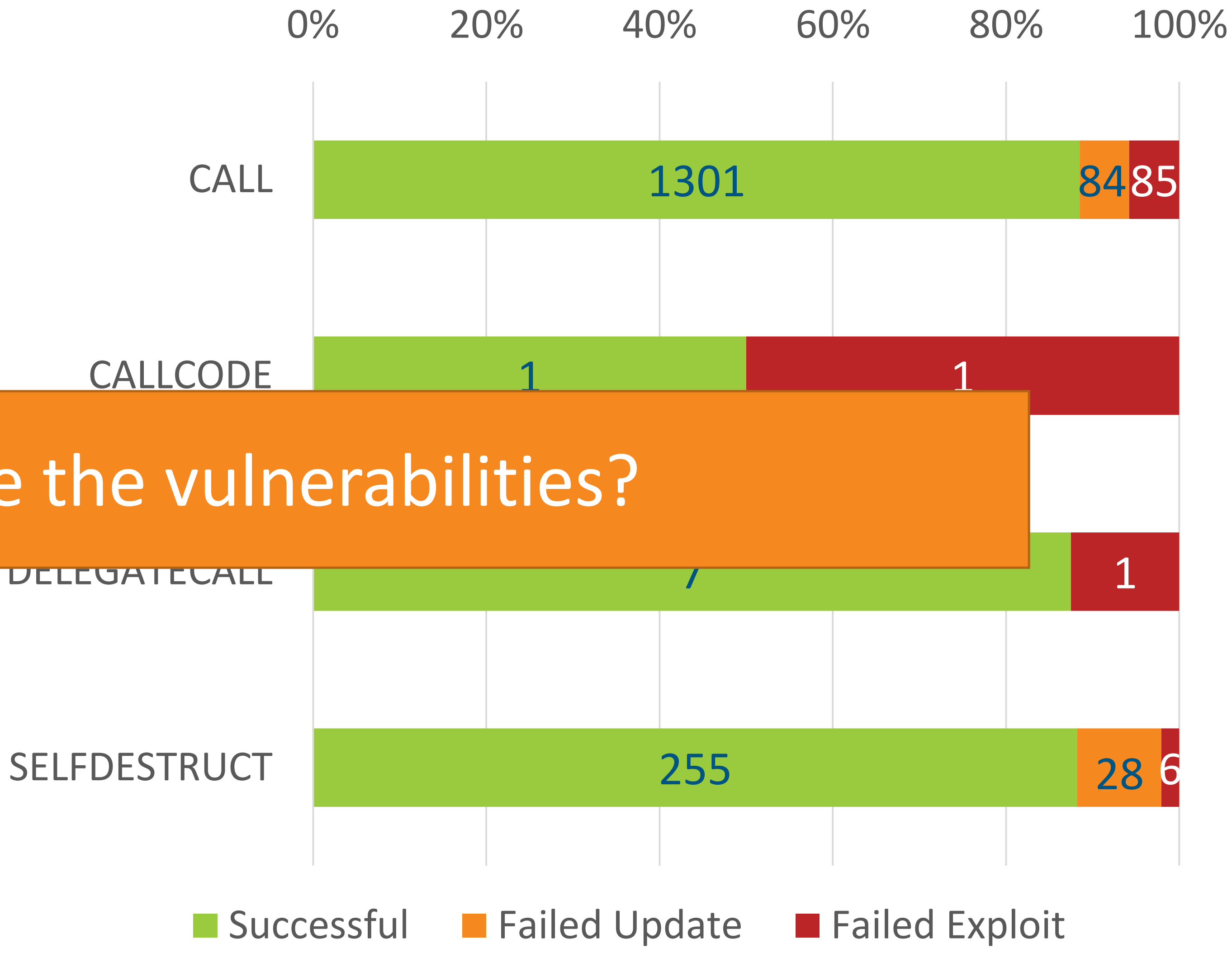
Validation

- local test network
- three accounts:
  - target contract
  - attacker
  - „shellcode“ contract
- two step validation:
  - update exploit to reflect target storage
  - replay exploit



Validation

- local test network
- three accounts:
  - target contract
  - attacker
  - „shellcode“ contract
- two ste[...]
  - upd[...] target storage
  - replay exploit

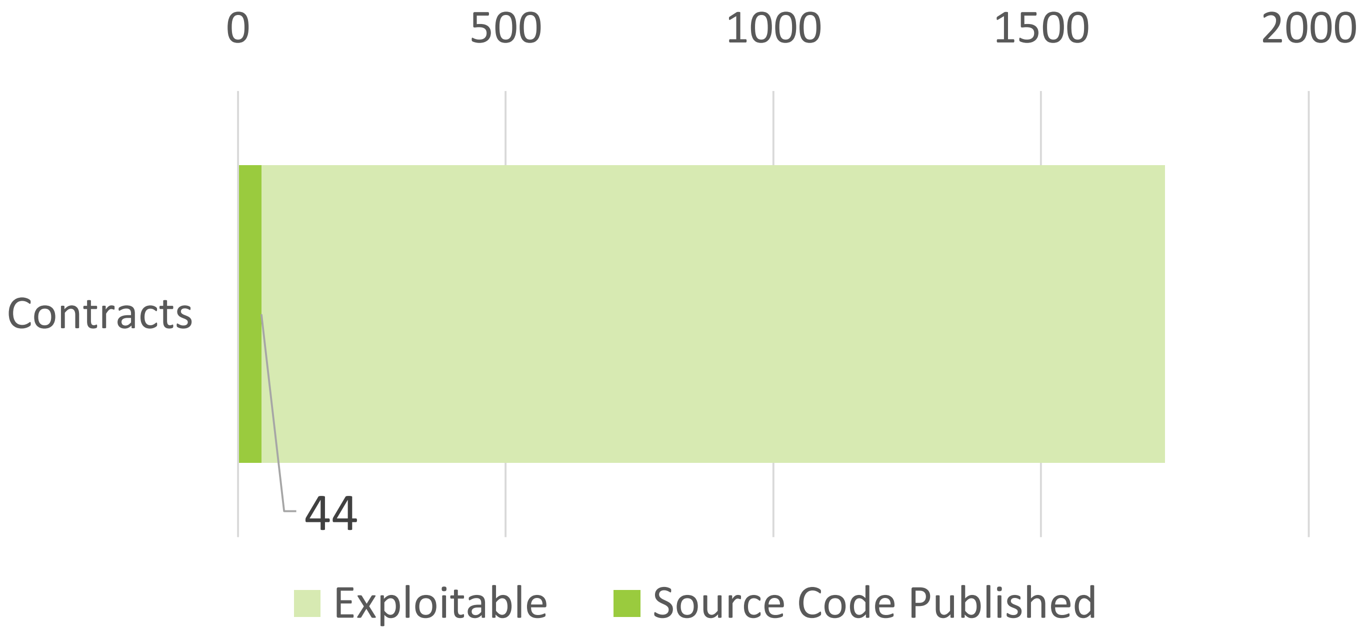


What are the vulnerabilities?

Vulnerabilities

- reverse engineering infeasible
- ~~source code unavailable~~
- OSINT: „publish & verify“ on etherscan.io
- manual analysis

ETHERSCAN
The Ethereum Block Explorer



Vulnerabilities

- logic bugs

```
modifier onlyowner() {
  require(msg.sender != owner);
  _;
}
```

Vulnerabilities

- logic bugs
- semantic confusion

`msg.value`	value of current transaction
`this.balance`	balance of account

Vulnerabilities

- logic bugs
- semantic confusion
- visibility errors

Vulnerabilities – Visibility

```
contract Bet{
  function play() {
    if(bet1 > bet2){
      win(player1);
    }else if(bet2 > bet1){
      win(player2);
    }else{
      draw(player1, player2);
    }
  }
  …
}
```

Vulnerabilities – Visibility

CISPA HELMHOLTZ-ZENTRUM i.G.

```
contract Bet{

  …

  function win(address winner) internal {
    winner.transfer(AMOUNT_WIN);
  }

  function draw(address player1, address player2) {
    player1.transfer(AMOUNT_DRAW);
    player2.transfer(AMOUNT_DRAW);
  }

}
```

default visibility: **public**

call **draw(attacker, attacker)**

Vulnerabilities

- logic bugs
- semantic confusion
- visibility errors
- constructor errors

Vulnerabilities – Constructor

```
contract Owned{
  function Owned() {
    owner = msg.sender;
  }

  ...

}
```

- constructor
- executed only once
- `msg.sender` = contract creator

```
contract Owned{
  function owned() {
    owner = msg.sender;
  }

  ...

}
```

- regular function
- can be called by anyone
- `msg.sender` = anyone

Vulnerabilities

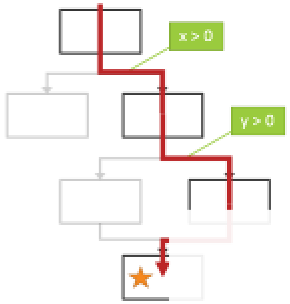
- logic bugs
- semantic confusion
- visibility errors
- constructor errors

caused by Solidity syntax? (semantic confusion, visibility errors, constructor errors)

Solidity partially at fault

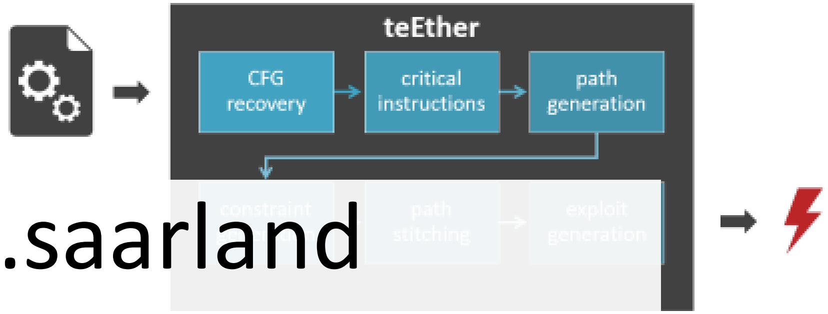
Conclusion

Exploit Generation – General Approach



1. locate critical instructions
2. compute backward slices of argument(s)
3. filter for attacker controlled slices
4. generate path through a slice
5. execute path symbolically
   - collect path constraints

teEther



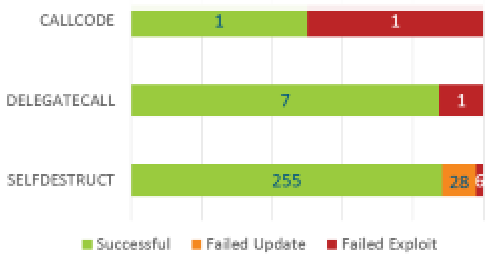
johannes.krupp@cispa.saarland

@KruppJohannes

teEther will be open sourced!

Validation

- local test network
- three accounts:
  - target contract
  - attacker
  - „shellcode" contract
- two step validation:
  - update exploit to reflect target storage
  - replay exploit



Solidity partially at fault