Modeling Reentrancy in Smart Contracts through Noninterference

Authors:
Carla Piazza, Sabina Rossi, Lorenzo Benetollo, Dalila Ressi,
Alvise Spanò and Semia Guesmi

UniUd & UniVe
30/05/2025

Motivation

⚠️ The Problem

- Many tools flag contracts as reentrancy-vulnerable based on syntactic patterns.
- But: Not all syntactically vulnerable contracts are exploitable.

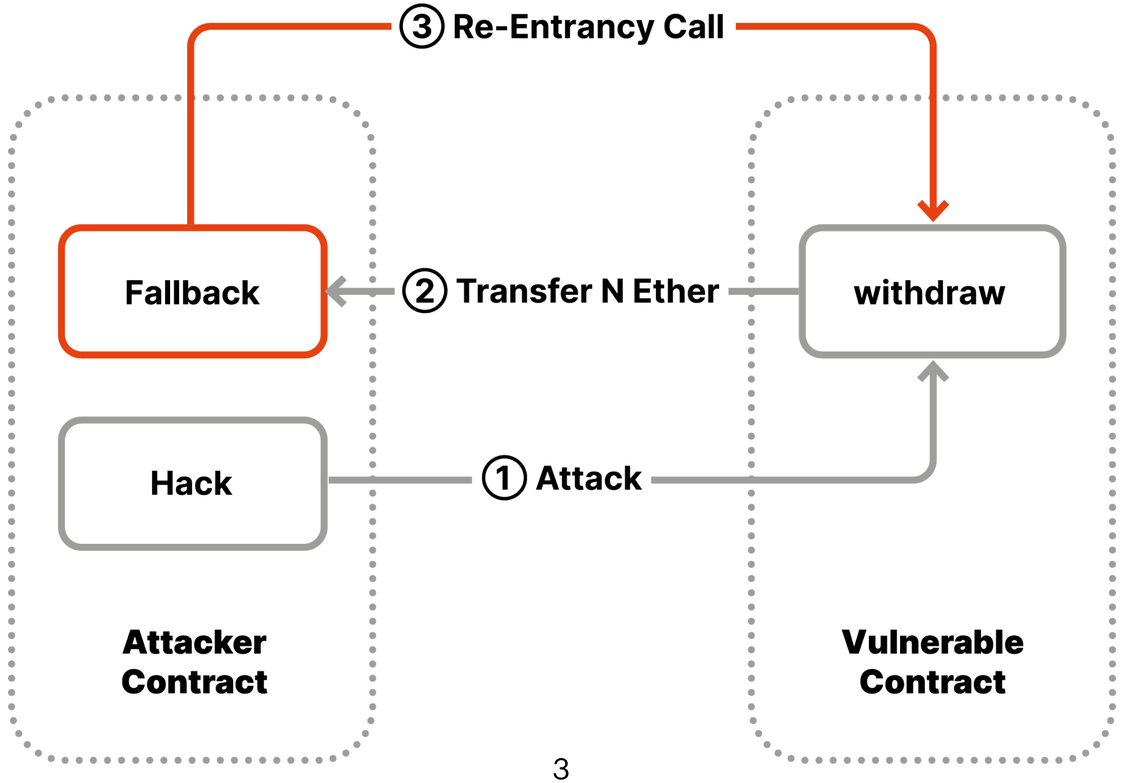
"A contract may exhibit reentrant behaviours at the code level, yet remain unexploitable under realistic execution scenarios.

🎯 Our objective:

Use a **semantics-driven** method that can more accurately differentiate **benign** from **truly dangerous** behaviors, especially in the context of **complex inter-contract interactions** and dynamic **asset transfers**.

What is Reentrancy?

Reentrancy vulnerabilities in Solidity smart contracts occur when a contract allows external entities to call back into itself or into other contracts it interacts with, before the initial function execution is completed.



Our collaboration

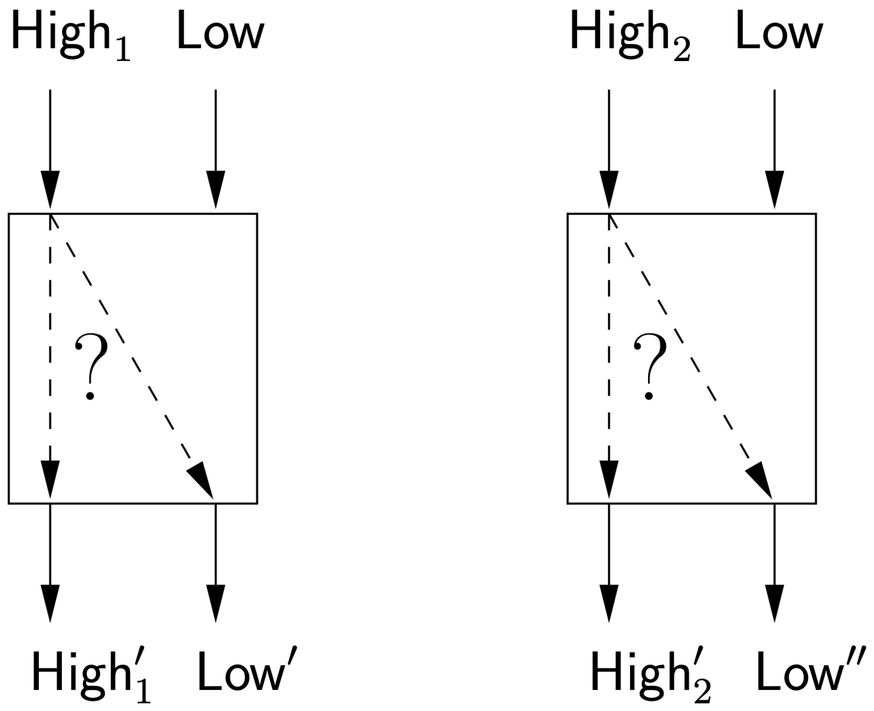
We offer a **novel perspective** based on the application of **formal verification methods** to detect and understand reentrancy vulnerabilities in smart contracts.

Our approach:

- Formalizes smart contracts using a **concurrent imperative language**.
- Applies **noninterference theory** and **unwinding conditions** to identify and localize illicit information flows.

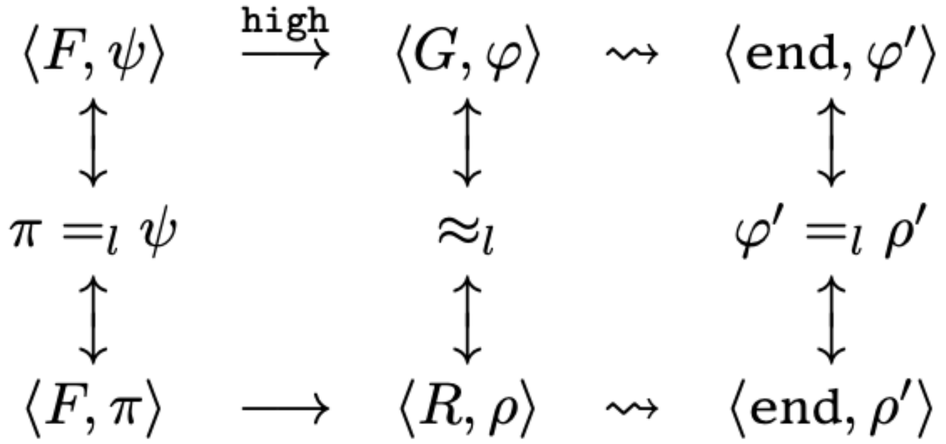
Noninterference

A security property introduced in multi-level secure systems. It ensures that no information flows from high-level entities (e.g., administrators) to low-level entities (e.g., standard users), preventing unintended disclosure of confidential information.



The program is non-interfering if **Low′ = Low″** for all possible High inputs and a fixed Low input.

Generalized unwinding condition



The unwinding condition is a formal method used to verify **noninterference** by analyzing a program's behavior in isolation.

It ensures that:

> **High-level variable modifications do not cause observable changes in low-level variables**, regardless of external high-level interactions.

Key Idea:

If a program P behaves the same from a low-level perspective, no matter how high-level data changes, then it **satisfies** the unwinding condition.

This approach allows:

Precise the exact points in the program where information flow might arise.

Case Study – Auction Contract

```
3  contract ReentrantAuction {
4
5      address payable public seller;
6      uint public endTime;
7      address public highestBidder;
8      uint public highestBid;
9
10     mapping(address => uint) public bids;
11
12     constructor(uint _startingBid, uint _duration) {
13         seller = payable(msg.sender);
14         highestBid = _startingBid;
15         endTime = block.timestamp + (_duration * 1 seconds);
16     }
17
18     function bid() external payable {
19         require(block.timestamp < endTime, "Bidding time expired");
20         require(msg.value > highestBid, "Value must be greater than highest");
21         require(bids[msg.sender] == 0, "You have already placed a bid");
22         bids[msg.sender] = msg.value;
23         highestBidder = msg.sender;
24         highestBid = msg.value;
25     }
26
27     function withdraw() public {
28         require(block.timestamp >= endTime, "Auction not ended");
29         uint amt = bids[msg.sender];
30         (bool success, ) = payable(msg.sender).call{value: amt}("");
31         bids[msg.sender] = 0;
32         require(success, "Transfer failed.");
33     }
34
35     function end() external {
36         require(msg.sender == seller, "Only the seller");
37         require(block.timestamp >= endTime, "Auction not ended");
38         (bool success, ) = seller.call{value: highestBid}("");
39         require(success, "Transfer failed.");
40     }
41 }
```

Imperative concurrent language

Syntax	Description
Z	Integer numbers
T	{true, false}
H	High-level locations
L	Low-level locations
B_H	High-level balance locations
B_L	Low-level balance locations
$a ::= n \mid X \mid B_C \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 * a_1$	Aexp
$b ::= \text{true} \mid \text{false} \mid (a_0 = a_1) \mid (a_0 \leq a_1) \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$	Bexp
$C ::= \langle P_1, .., P_n \rangle$	Contracts
$P ::= S \mid P_0; P_1 \mid \text{if}(b)\{P_0\}\,\text{else}\,\{P_1\} \mid \text{while}(b)\{P\}$	Programs
$S ::= \text{skip} \mid X := a \mid S_0; S_1$	Statements
$\langle P, \sigma \rangle \xrightarrow{\epsilon} \langle P', \sigma' \rangle$	State transition

Case Study: An Auction Contract

In Our Model, a Contract is Represented as a Tuple of Programs:

$$\textsc{ReentrantAuction} \equiv \langle \textsc{Bid}, \textsc{Withdraw} \rangle .$$

```
function bid() external payable {
    require(block.timestamp < endTime, "Bidding time expired");
    require(msg.value > highestBid, "Value must be greater than highest");
    require(bids[msg.sender] == 0, "You have already placed a bid");
    bids[msg.sender] = msg.value;
    highestBidder = msg.sender;
    highestBid = msg.value;
}

function withdraw() public {
    require(block.timestamp >= endTime, "Auction not ended");
    uint amt = bids[msg.sender];
    (bool success, ) = payable(msg.sender).call{value: amt}("");
    bids[msg.sender] = 0;
    require(success, "Transfer failed.");
}
```

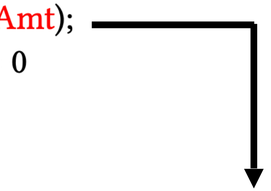
Case Study: An Auction Contract

```
function bid() external payable {
    require(block.timestamp < endTime, "Bidding time expired");
    require(msg.value > highestBid, "Value must be greater than highest");
    require(bids[msg.sender] == 0, "You have already placed a bid");
    bids[msg.sender] = msg.value;
    highestBidder = msg.sender;
    highestBid = msg.value;
}
```

```
1: Program Bid
2:     PrevBid := Bids[Sender];
3:     if (CallValue ≤ HighestBid ∨ PrevBid ≠ 0) then
4:         skip
5:     else
6:         HighestBidder := Sender;
7:         HighestBid := CallValue;
8:         Bids[Sender] := CallValue
```

```
function withdraw() public {
    require(block.timestamp >= endTime, "Auction not ended");
    uint amt = bids[msg.sender];
    (bool success, ) = payable(msg.sender).call{value: amt}("");
    bids[msg.sender] = 0;
    require(success, "Transfer failed.");
}
```

```
1: Program Withdraw
2:     Amt := Bids[Sender];
3:     call(Receive, Amt);
4:     Bids[Sender] := 0
```

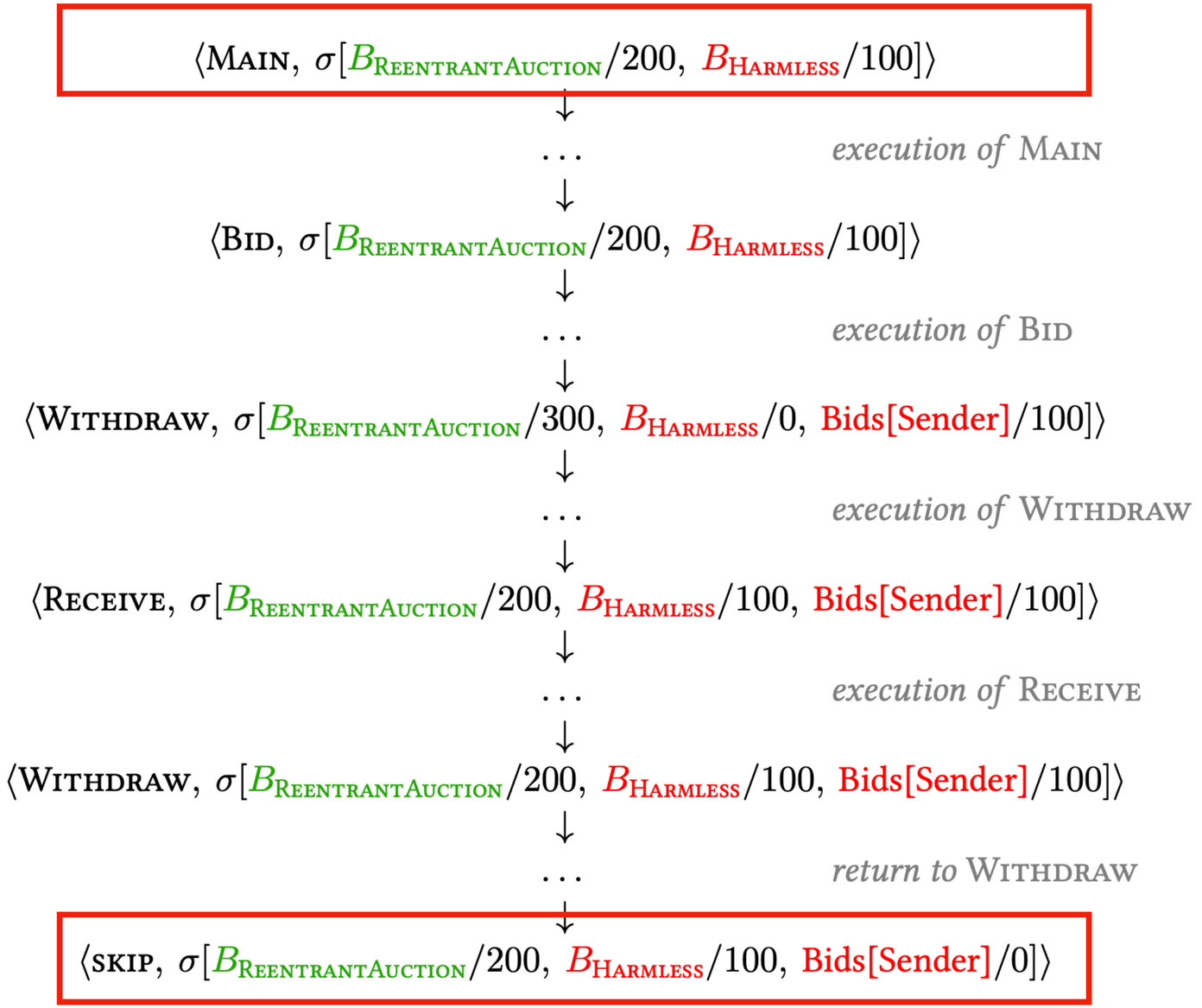


▷ B_{caller} -= Amt, B_{callee} += Amt

A Harmless Interaction

$\textsc{Harmless} \equiv \langle \textsc{Main}, \textsc{Receive} \rangle$

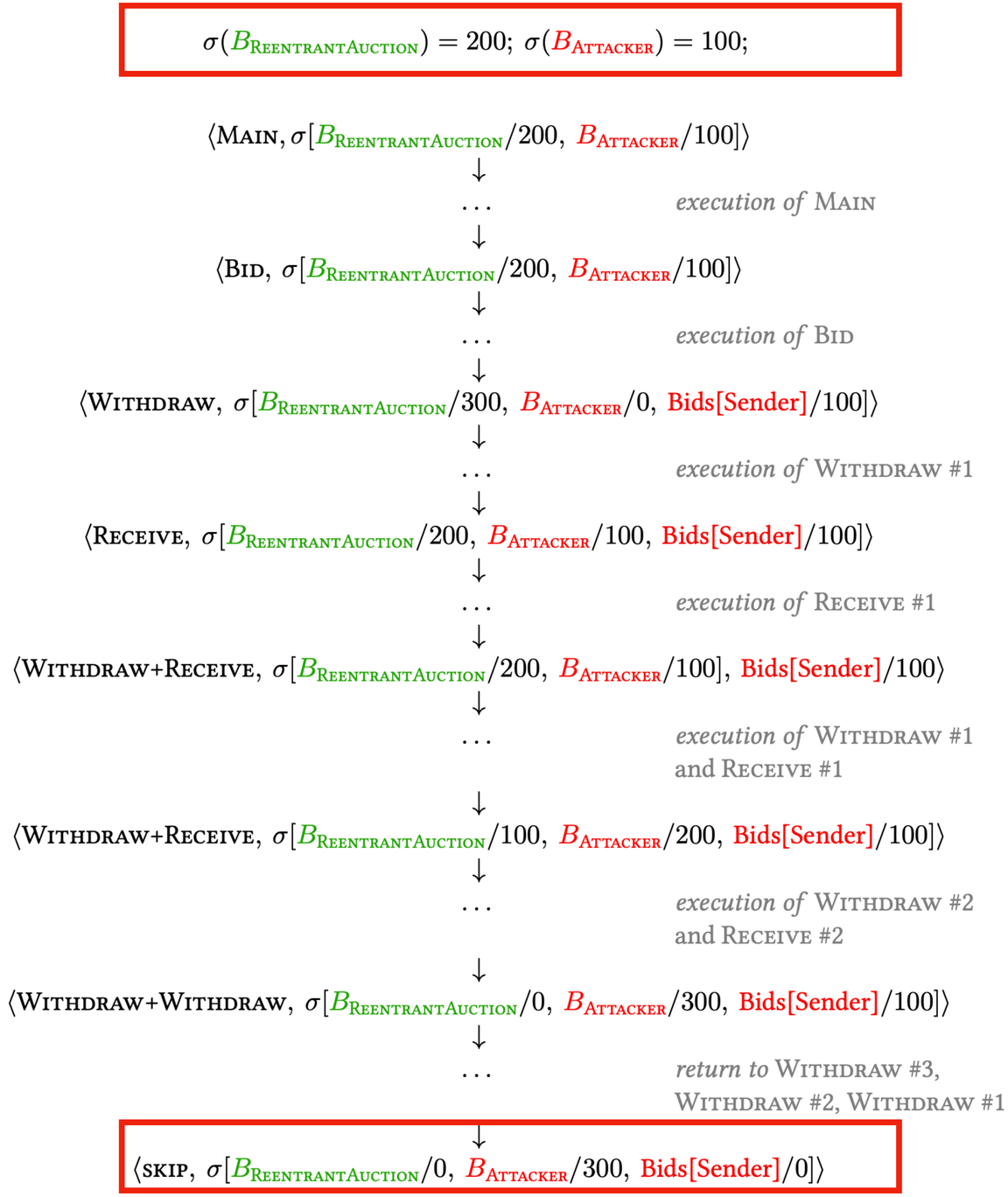
1: **Program** Main
2: `call(`Bid, 100`);`
3: `call(`Withdraw, 0`)`
1: **Program** Receive
2: `skip`



A Reentrancy Attack

$\textsc{Attacker} \equiv \langle \textsc{Main}, \textsc{Receive} \rangle$

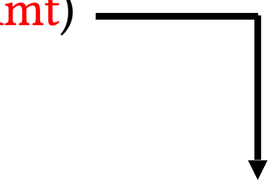
1: **Program** RECEIVE
2: `call`(WITHDRAW, 0)



A Non-Reentrant Auction

```
1  function withdraw() public {
2      require(block.timestamp >= endTime, "Auction not ended");
3      uint amt = bids[msg.sender];
4      bids[msg.sender] = 0;   // attackers reach this before and prevent further payments
5      (bool success, ) = payable(msg.sender).call{value: amt}("");
6      require(success, "Transfer failed.");
7  }
```

1: **Program** WITHDRAW
2: Amt := Bids[Sender];
3: Bids[Sender] := 0;
4: call(RECEIVE, Amt)



▷ $B_{SAFEAUCTION}$ -= Amt, $B_{ATTACKER}$ += Amt

A Non-Reentrant Auction

$$\textsc{SafeAuction} \equiv \langle \textsc{Bid}, \textsc{Withdraw} \rangle$$

Downgrading mechanism

The concept of downgrading refers to the intentional and controlled **release** of sensitive (high-level) information.

- Helps **distinguish** between secure and potentially vulnerable scenarios.
- Allows for explicitly authorized, limited information flows from high to low.

Downgrading Raises Key Questions:

- Who is authorized to perform downgrading?
- What information can be downgraded?
- Where and when is it permissible?

A case of Downgrading

- The **withdraw()** function includes an access control check that restricts its execution to the contract's owner.
- Additionally, the function transfers the entire contract balance in a single call, leaving no residual funds that an attacker could repeatedly steal through reentrant invocations.
- As a result, despite following a pattern commonly associated with reentrancy vulnerabilities, **the contract is secure**.

```
1  pragma solidity ^0.8.28;
2
3  contract Crowdfund {
4
5      bool open;     // flag that closes the Crowdfund
6      address receiver;   // receiver of the donated funds
7      address owner;
8
9      constructor (address payable receiver_) {
10         receiver = receiver_;
11         owner = msg.sender;
12         open = true;
13     }
14
15     function donate() public payable {
16         require (open);
17     }
18
19     function withdraw() public {
20         require (open);
21         require (msg.sender == owner);
22         (bool succ,) = receiver.call{value: address(this).balance}("");
23         open = false;
24         require(succ);
25     }
26 }
```

A case of Downgrading

$\textsc{CrowdFund} \equiv \langle \textsc{Donate}, \textsc{Withdraw} \rangle$

Programs Main and Receive of the Owner and Receiver Contracts:

$$\textsc{Owner} \equiv \langle \textsc{Main} \rangle$$

$$\textsc{Receiver} \equiv \langle \textsc{Receive} \rangle$$

```
1: Program WITHDRAW
2:     if (open = false ∨ Sender ≠ Owner) then
3:         skip
4:     else
5:         Balance := B_CrowdFund;
6:         call(RECEIVE, downgrade(Balance));
7:         open := false;
```

▷ $B_{\textsc{CrowdFund}}$ -= Balance, $B_{\textsc{Receiver}}$ += Balance

Future Work

- Develop a proof system for analyzing smart contracts to formally verify the absence of reentrancy vulnerabilities.
- Implement a tool based on the non-interference framework to enable automated smart contract analysis, enhancing security and reliability in blockchain applications.

🙏 Thank you for your attention!

📧 Contact: semia.guesmi@unicam.it