Modeling Reentrancy in Smart Contracts through Noninterference

Lorenzo Benetollo[1], Semia Guesmi[1], Carla Piazza[2], Dalila Ressi[3], Sabina Rossi[3,*] and Alvise Spanò[3]

[1]*Università degli Studi di Camerino*
[2]*Università degli Studi di Udine*
[3]*Università Ca' Foscari di Venezia*

Abstract

Reentrancy is a well-known vulnerability in smart contracts for blockchain platforms, allowing malicious actors to repeatedly call a contract before previous executions are completed, often leading to unexpected and harmful behavior. In this paper, we propose the use of the notion of noninterference to model and analyze reentrancy attacks. Originally developed to characterize unwanted information flows in multi-level security systems, noninterference provides a rigorous framework for reasoning about the absence of illicit interactions between components. Among the various formulations of noninterference, those based on unwinding conditions are particularly well-suited for our analysis, as they enable the precise localization of information flows within a system. We investigate how these conditions can be applied to detect and understand reentrancy vulnerabilities in smart contracts, offering a novel perspective and potential foundation for developing verification techniques against such attacks.

Keywords

Smart Contracts, Reentrancy, Noninterference, Unwinding Conditions

1. Introduction

Reentrancy remains one of the most critical and extensively studied vulnerabilities in the domain of smart contracts. It refers to a subtle yet highly dangerous class of bugs that arise when a contract issues an external call to another contract, and the callee, before the original execution flow is complete, calls back into the initiating contract. If the contract's internal state has not been properly updated prior to the external call, such re-entries can manipulate the state in unintended ways, leading to potentially severe consequences.

This vulnerability is particularly concerning in decentralized environments such as Ethereum, where smart contracts operate autonomously and are entrusted with managing significant financial assets without centralized oversight. The infamous DAO attack in 2016 stands as a stark example of the risks posed by reentrancy: an attacker exploited this very flaw to drain millions of dollars worth of Ether, severely undermining trust in the early Ethereum ecosystem. Since then, reentrancy has become a canonical example underscoring the importance of rigorous security practices in smart contract development.

In response to the threat of reentrancy, a broad spectrum of tools and techniques has been developed to detect and mitigate related vulnerabilities. These range from static analysis methods [1] to machine learning-based approaches [2], each aiming to identify potentially exploitable code patterns. Formal analyzers often employ techniques such as symbolic execution, taint analysis, fuzzing, and constraint solving [1]. Notable static analysis tools include Oyente [3], Securify [4], and Mythril [5], which inspect the contract's code for known reentrancy patterns. These are complemented by dynamic analysis tools like ContractFuzzer [6] and sFuzz [7], which test runtime behavior through fuzzing techniques. Among static analyzers, Ethor [8] stands out as the only tool that provides soundness guarantees, ensuring the

DLT2025: 7th Distributed Ledger Technology Workshop, June, 12-14 2025 - Pizzo, Italy
*Corresponding author.
sabina.rossi@unive.it (S. Rossi)

absence of false negatives. Ethor works by abstracting EVM bytecode semantics using Horn clauses to verify reachability properties. To better handle the complexity of inter-contract dependencies, more advanced analyzers like Clairvoyance [9] and SmartDagger [10] focus specifically on cross-contract interactions. More recent approaches have begun to incorporate Large Language Models (LLMs) into the analysis pipeline. For example, AdvScanner [11] uses static analysis in conjunction with LLMs to generate adversarial smart contracts that test the robustness of target contracts. However, LLMs on their own have shown limited effectiveness in vulnerability detection [12] and in code generation tasks more broadly [13]. On the machine learning front, graph-based representations of smart contracts have proven particularly effective [14, 15], especially when combined with multi-modal input representations that include both structural and semantic features [16, 17, 18].

However, distinguishing between contracts that are merely syntactically vulnerable and those that are truly exploitable remains a subtle and unresolved problem. A contract may exhibit reentrant behavior at the code level, yet remain unexploitable under realistic execution scenarios [19]. This gap highlights the growing need for rigorous, semantics-driven methods that can more accurately differentiate benign from truly dangerous behaviors, especially in the context of complex inter-contract interactions and dynamic asset transfers.

To address this gap, we investigate the application of formal verification methods to the problem of reentrancy detection, with a particular focus on noninterference, a foundational concept in security that ensures the absence of unauthorized information flow between distinct components of a system. Noninterference was originally introduced in the setting of multi-level secure systems [20], and has since evolved through various formalizations and extensions [21, 22, 23], adapting to new contexts such as distributed computing and programming language semantics.

Among the different approaches to verifying noninterference, one of the most tractable and compositional methods is based on unwinding conditions. These conditions offer a structured and modular framework for reasoning about security properties, providing sufficient criteria to ensure that no illicit flow of information occurs during system execution [24, 25, 26, 27]. This makes them particularly well-suited for analyzing complex behaviors such as reentrancy, where the interaction between components and the timing of state updates play a critical role.

In this work, we investigate how unwinding conditions can be effectively leveraged to reason about reentrancy vulnerabilities in smart contracts. To this end, we build on a simplified concurrent imperative language originally proposed in [25, 27], extending it to capture key features of account-based blockchain platforms such as Ethereum. In particular, we enrich the language with constructs that model contracts performing value transfers during execution, that is an essential aspect of smart contract semantics. Our abstraction deliberately omits low-level implementation details in order to focus on the core mechanisms of inter-contract interaction and monetary flow, enabling a clean and precise formal analysis of potential reentrancy behaviors.

As a concrete application of our approach, we present a formal case study based on an auction contract, a widely used and security-sensitive pattern in decentralized applications. This case study demonstrates how our formalization can be used to systematically identify potential reentrancy vulnerabilities, assess their exploitability, and analyze the effectiveness and limitations of common mitigation strategies. Our work builds on [28], which exploits similar noninterference-based reasoning in the context of Maximal Extractable Value (MEV) [29, 30] attacks. In contrast, we apply a similar formal approach to uncover and analyze reentrancy vulnerabilities, highlighting how these reasoning principles extend naturally to this distinct class of threats.

Our findings demonstrate that the noninterference framework, and in particular the application of unwinding conditions, offers a powerful and principled foundation for reasoning about reentrancy in smart contracts. By formalizing the contract semantics and systematically analyzing the conditions under which re-entrant behavior can occur, we gain deeper insight into the nature of this vulnerability and the structural safeguards required to mitigate it effectively.

Structure of the paper. The remainder of the paper is organized as follows. Section 2 introduces a simplified concurrent imperative language designed to model contracts that perform value transfers during execution. In Section 3, we present the notion of noninterference and explain how the concept of downgrading can be used to represent the intentional release of sensitive information. Section 4 provides a detailed formalization of the Auction contract within our language, illustrating how the noninterference property is instantiated and how downgrading helps distinguish between secure and potentially vulnerable scenarios. Finally, Section 5 concludes the paper.

2. The Language

In this section, we present an extension of the imperative concurrent language introduced in [27], aimed at capturing the behavior of account-based smart contracts, such as those written in Solidity.

Let $\mathbb{Z}$ be the set of integer numbers, $\mathbb{T} = \{\text{true}, \text{false}\}$ be the set of boolean values, $\mathbb{L}$ be a set of low-level locations and $\mathbb{H}$ be a set of high-level locations, with $\mathbb{L} \cap \mathbb{H} = \emptyset$. We also assume a distinguished family of special locations, denoted B_C, used to access the balance associated with a contract C. These balance variables are readable within the program but are not directly writable, their value can only be updated indirectly through external interactions, such as contract calls that transfer funds. As with standard storage locations, balance variables are divided into high-level and low-level subsets, denoted by $\mathbb{B}_H$ and $\mathbb{B}_L$ respectively, with $\mathbb{B}_L \cap \mathbb{B}_H = \emptyset$.

The set $\mathbf{Aexp}$ of arithmetic expressions is defined by the grammar:

$$a ::= n \mid X \mid B_C \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 * a_1$$

where $n \in \mathbb{Z}$, $X \in \mathbb{L} \cup \mathbb{H}$ and $B_C \in \mathbb{B}_L \cup \mathbb{B}_H$. We assume that arithmetic expressions are total. The set $\mathbf{Bexp}$ of boolean expressions is defined by:

$$b ::= \text{true} \mid \text{false} \mid (a_0 = a_1) \mid (a_0 \leq a_1) \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$$

where $a_0, a_1 \in \mathbf{Aexp}$.

We say that an arithmetic expression a is *confidential*, denoted by $a \in \texttt{high}$, if there is a high-level location which occurs in it. Otherwise we say that a is *public*, denoted by $a \in \texttt{low}$. Similarly, we say that a boolean expression b is *confidential*, denoted by $b \in \texttt{high}$, if there is a confidential arithmetic expression which occurs in it. Otherwise we say that b is *public*, denoted by $b \in \texttt{low}$. This notion of confidentiality, both for arithmetic and boolean expressions, is purely syntactic. Notice that a high-level expression can contain low-level locations, i.e., its value can depend on the values of low-level locations. This reflects the idea that a high-level user can read high- and low-level data.

The syntax of our language is organized into three primary categories: contracts, programs, and statements. A contract is defined as a tuple of programs, each representing a possible execution entry point. Programs, in turn, consist of sequences of statements. This structured syntax enables the modeling of smart contracts that perform value transfers and interact with other contracts via the `call` operator, effectively capturing the essential semantics of contract interaction and asset flow. Formally, the syntax is defined as:

$$\begin{array}{lcll}
C & ::= & \langle P_1, .., P_n \rangle & \textbf{contracts} \\
P & ::= & S \mid P_0; P_1 \mid \texttt{if}(b)\ \{P_0\}\ \texttt{else}\ \{P_1\} \mid \texttt{while}(b)\ \{P\} & \textbf{programs} \\
 & & \mid \texttt{await}(b)\ \{S\} \mid \texttt{co}\ P_1 \| \ldots \| P_n\ \texttt{oc} \mid \texttt{call}_P\ (P', a) & \\
S & ::= & \texttt{skip} \mid X := a \mid S_0; S_1 & \textbf{statements}
\end{array}$$

where $a \in \mathbf{Aexp}$, $X \in \mathbb{L} \cup \mathbb{H}$, and $b \in \mathbf{Bexp}$. Notice that, as in [31], in the body of the `await` operator only sequences of assignments are allowed.

The operational semantics of our language is based on the notion of *state*. A state σ is a function which assigns to each location, both standard and balance location, an integer, i.e., $\sigma : \mathbb{L} \cup \mathbb{H} \cup \mathbb{B}_L \cup \mathbb{B}_H \longrightarrow \mathbb{Z}$. Given a state σ, we denote by $\sigma[X/n]$ (resp., $\sigma[B_C/n]$) the state σ' such that $\sigma'(X) = n$ (resp., $\sigma'(B_C) = n$) and $\sigma'(Y) = \sigma(Y)$ for all $Y \neq X$ (resp., $Y \neq B_C$). Moreover, we denote by σ_L the restriction of σ to the low-level locations and, given the states σ and θ, we write $\sigma =_l \theta$ for $\sigma_L = \theta_L$.

$$\frac{}{\langle \texttt{skip}, \sigma\rangle \xrightarrow{\texttt{low}} \langle \mathrm{end}, \sigma\rangle} \qquad \frac{\langle a, \sigma\rangle \to n}{\langle X := a, \sigma\rangle \xrightarrow{\epsilon} \langle \mathrm{end}, \sigma[X/n]\rangle}\ a \in \epsilon$$

$$\frac{\langle P_0, \sigma\rangle \xrightarrow{\epsilon} \langle P_0', \sigma'\rangle}{\langle P_0; P_1, \sigma\rangle \xrightarrow{\epsilon} \langle P_0'; P_1, \sigma'\rangle}\ P_0' \not\equiv \mathrm{end} \qquad \frac{\langle P_0, \sigma\rangle \xrightarrow{\epsilon} \langle \mathrm{end}, \sigma'\rangle}{\langle P_0; P_1, \sigma\rangle \xrightarrow{\epsilon} \langle P_1, \sigma'\rangle}$$

$$\frac{\langle b, \sigma\rangle \to \mathrm{true}}{\langle \texttt{if}(b)\ \{P_0\}\ \texttt{else}\ \{P_1\}, \sigma\rangle \xrightarrow{\epsilon} \langle P_0, \sigma\rangle}\ b \in \epsilon \qquad \frac{\langle b, \sigma\rangle \to \mathrm{false}}{\langle \texttt{if}(b)\ \{P_0\}\ \texttt{else}\ \{P_1\}, \sigma\rangle \xrightarrow{\epsilon} \langle P_1, \sigma\rangle}\ b \in \epsilon$$

$$\frac{\langle b, \sigma\rangle \to \mathrm{true}}{\langle \texttt{while}(b)\ \{P\}, \sigma\rangle \xrightarrow{\epsilon} \langle P; \texttt{while}(b)\ \{P\}, \sigma\rangle}\ b \in \epsilon \qquad \frac{\langle b, \sigma\rangle \to \mathrm{false}}{\langle \texttt{while}(b)\ \{P\}, \sigma\rangle \xrightarrow{\epsilon} \langle \mathrm{end}, \sigma\rangle}\ b \in \epsilon$$

$$\frac{\langle b, \sigma\rangle \to \mathrm{true} \qquad \langle S, \sigma\rangle \overset{\epsilon_2}{\rightsquigarrow} \langle \mathrm{end}, \sigma'\rangle}{\langle \texttt{await}(b)\ \{S\}, \sigma\rangle \xrightarrow{\epsilon_1 \cup \epsilon_2} \langle \mathrm{end}, \sigma'\rangle}\ b \in \epsilon_1 \qquad \frac{\langle b, \sigma\rangle \to \mathrm{false}}{\langle \texttt{await}(b)\ \{S\}, \sigma\rangle \xrightarrow{\epsilon} \langle \texttt{await}(b)\ \{S\}, \sigma\rangle}\ b \in \epsilon$$

$$\frac{\langle P_i, \sigma\rangle \xrightarrow{\epsilon} \langle P_i', \sigma'\rangle}{\langle \mathrm{co}\ P_1 \| \ldots \| P_i \| \ldots \| P_n\ \mathrm{oc}, \sigma\rangle \xrightarrow{\epsilon} \langle \mathrm{co}\ P_1 \| \ldots \| P_i' \| \ldots \| P_n\ \mathrm{oc}, \sigma'\rangle} \qquad \frac{}{\langle \mathrm{co\ end} \| \ldots \| \mathrm{end} \| \ldots \| \mathrm{end\ oc}, \sigma\rangle \xrightarrow{\texttt{low}} \langle \mathrm{end}, \sigma\rangle}$$

$$\frac{\exists C.P \in C \quad \exists C'.P' \in C' \quad \sigma(B_C) \to n \quad \sigma(B_{C'}) \to n' \quad \langle a, \sigma\rangle \to n_0 \quad n \geq n_0}{\langle \texttt{call}_P(P',\ a),\ \sigma\rangle \xrightarrow{\epsilon} \langle P', \sigma[B_{C'}/n' + n_0][B_C/n - n_0]\rangle}\ a \in \epsilon$$

$$\frac{\exists C.P \in C \quad \exists C'.P' \in C' \quad \sigma(B_C) \to n \quad \langle a, \sigma\rangle \to n_0 \quad n < n_0}{\langle \texttt{call}_P(P',\ a),\ \sigma\rangle \xrightarrow{\epsilon} \langle \mathrm{end}, \sigma\rangle}\ a \in \epsilon$$

Table 1
The operational semantics.

Given an arithmetic expression $a \in \mathbf{Aexp}$ and a state σ, the evaluation of a in σ, denoted by $\langle a, \sigma\rangle \to n$ with $n \in \mathbb{Z}$, is defined in the standard way. Similarly, $\langle b, \sigma\rangle \to v$ with $b \in \mathbf{Bexp}$ and $v \in \{\mathrm{true}, \mathrm{false}\}$, denotes the evaluation of a boolean expression b in a state σ. In both cases, atomicity of the evaluation operation is assumed.

To model the behavior of Ethereum smart contracts, we assume that the execution of a contract C is initiated by an external user U through a call. In our formalism, contract execution begins with a designated entry-point program, typically named Main. We represent this interaction as an implicit call of the form $\texttt{call}_U(\texttt{Main}, a)$, where a is the argument passed to the contract. This models the external invocation of a contract, as commonly occurs in Ethereum when users send transactions to trigger contract logic. Since all contract behavior ultimately unfolds through the execution of individual programs, we focus our operational semantics on programs rather than entire contracts. This modular view allows us to capture the core dynamics of execution, including inter-contract interactions and value transfers, while maintaining a clear and tractable the semantic model.

Let $\mathbf{Prog}$ be the set of programs of our language. The operational semantics of programs is defined in terms of state transitions. A transition from a program P and a state σ has the form $\langle P, \sigma\rangle \xrightarrow{\epsilon} \langle P', \sigma'\rangle$ where P' is either a program or the special symbol end (denoting termination) and $\epsilon \in \{\texttt{high}, \texttt{low}\}$ stating that the transition is either confidential or public. The operation $\epsilon_1 \cup \epsilon_2$ returns low if both ϵ_1 and ϵ_2 are low otherwise it returns high. Let $\mathbb{P} = \mathbf{Prog} \cup \{\mathrm{end}\}$ and Σ be the set of all the possible states. The operational semantics of $\langle P, \sigma\rangle \in \mathbb{P} \times \Sigma$ is the *labelled transition system* (LTS) defined by structural induction on P according to the rules depicted in Table 1. Intuitively, the semantics of the sequential composition impose that a program of the form $P_0; P_1$ behaves like P_0 until P_0 terminates and then it behaves like P_1. To describe the semantics of a program of the form $\texttt{while}(b)\ \{P\}$ we have to distinguish two cases: if b is true, then the program is unravelled to $P; \texttt{while}(b)\ \{P\}$; otherwise it terminates. As far as the await operator is concerned, if b is true then $\texttt{await}(b)\ \{P\}$ terminates executing P in one indivisible action, i.e., it is not possible to observe the state changes internal to the execution of P, while if b is false then $\texttt{await}(b)\ \{P\}$ loops waiting for b to become true. In a parallel composition of the form $\texttt{co}\ P_0 \| \dots \| P_n \texttt{oc}$ any of the P_i can move, and the termination is reached only when all the P_i's have terminated. Finally, we define the call $\texttt{call}_P(P', a)$ operator, which describes how a program P can invoke some other program P' while transferring some amount of currency a from the caller's balance B_C to the callee's balance $B_{C'}$, where C and C' are the contracts to which P and P' belong respectively. In other words, the call command can be used to transfer some amount of currency a and invoke a program P' on another contract. In this case, the specified amount is subtracted from the caller's balance and added to the callee's balance. The execution then continues in the callee's context.

We use the following notations. We write $\langle P, \sigma\rangle \to \langle P', \sigma'\rangle$ to denote $\langle P, \sigma\rangle \xrightarrow{\epsilon} \langle P', \sigma'\rangle$ with $\epsilon \in \{\texttt{low}, \texttt{high}\}$ and $\langle P_0, \sigma_0\rangle \to^n \langle P_n, \sigma_n\rangle$ with $n \geq 0$ for $\langle P_0, \sigma_0\rangle \to \langle P_1, \sigma_1\rangle \to \cdots \to \langle P_{n-1}, \sigma_{n-1}\rangle \to \langle P_n, \sigma_n\rangle$. The notation $\langle P_0, \sigma_0\rangle \overset{\texttt{low}}{\rightsquigarrow} \langle P_n, \sigma_n\rangle$ stands for $\langle P_0, \sigma_0\rangle \to^n \langle P_n, \sigma_n\rangle$ for some $n \geq 0$ with all the n transitions labelled with $\texttt{low}$; similarly $\langle P_0, \sigma_0\rangle \overset{\texttt{high}}{\rightsquigarrow} \langle P_n, \sigma_n\rangle$ stands for $\langle P_0, \sigma_0\rangle \to^n \langle P_n, \sigma_n\rangle$ for some $n \geq 0$ with at least one of the n transitions labelled with $\texttt{high}$. Finally, we write $\langle P, \sigma\rangle \rightsquigarrow \langle P', \sigma'\rangle$ to denote $\langle P, \sigma\rangle \overset{\epsilon}{\rightsquigarrow} \langle P', \sigma'\rangle$ with $\epsilon \in \{\texttt{low}, \texttt{high}\}$.

Notice that the operational semantics defined in Table 1 is non-deterministic, since in the case of parallel composition there are many possible evolutions.

3. Noninterference and Downgrading

In [27], we introduced an imperative concurrent language, inspired by the one presented in [31], and we investigated several notions of noninterference aimed at ensuring secure information flow in multi-level systems. In such systems, users are classified into different security levels, typically high (e.g., system administrators) and low (e.g., standard users), and noninterference guarantees that no information flows from high-level entities to low-level ones. Such a flow would, in effect, result in the unintended disclosure of confidential information.

However, recognizing that a strict prohibition of all high-to-low flows can be overly rigid and impractical in many real-world scenarios, we also proposed a downgrading mechanism in [27]. This mechanism allows for explicitly authorized, limited information flows from high to low, enabling a more flexible and context-aware approach to enforcing security.

To keep the presentation simple and accessible, we avoid relying on complex observational equivalences such as bisimulation. Instead, we adopt a more direct approach by observing only the final values of low-level variables at the end of program execution.

3.1. Noninterference

Intuitively, when analyzing a program P that manipulates both low and high-level variables, our goal is to ensure that the behavior of P does not allow high-level activities to influence the low-level outcomes. In other words, no matter how the high-level variables are modified during execution, potentially due to interactions with high-level users or components, the values of the low-level variables should remain unaffected. If modifications to high-level variables result in observable changes to low-level ones, this constitutes an information flow, or more precisely, interference.

The key idea is to reason about the security of P in isolation, by showing that its execution is secure in any possible context, regardless of the actions performed by external high-level entities. This principle is formalized in [27] through the notion of a generalized unwinding condition, which characterizes classes of secure programs that are parametric with respect to:

- a binary relation $\doteq$ which equates two states if they are indistinguishable for a low-level observer;
- a binary reachability relation $\mathcal{R}$ on $\mathbb{P} \times \Sigma$ which associates to each pair $\langle P, \sigma\rangle$ all the pairs $\langle F, \psi\rangle$ which, in some sense, are reachable from $\langle P, \sigma\rangle$.
- a binary relation $\doteqdot$ which equates two pairs $\langle P, \sigma\rangle$ and $\langle Q, \theta\rangle$ if they are indistinguishable for a low-level observer;

A pair $\langle P, \sigma \rangle$ satisfies (an instance of) our unwinding framework (i.e., there are no flows of information from high to low) if any high-level step $\langle F, \psi \rangle \overset{\texttt{high}}{\to} \langle G, \varphi \rangle$ performed by a pair $\langle F, \psi \rangle$ reachable from $\langle P, \sigma \rangle$ has no effect on the observation of a low-level user. This is achieved by requiring that all the elements in the set $\{\langle F, \pi \rangle \mid \pi \doteq \psi\}$ (whose states are low-level equivalent) may perform a transition reaching an element of the set $\{\langle R, \rho \rangle \mid \langle R, \rho \rangle \dot{\div} \langle G, \varphi \rangle\}$ (whose elements are all indistinguishable for a low-level observer). We use the notation $\mathcal{R}(\langle P, \sigma \rangle)$ to denote the set of pairs reachable from $\langle P, \sigma \rangle$, i.e., $\mathcal{R}(\langle P, \sigma \rangle) = \{\langle F, \psi \rangle \mid \langle P, \sigma \rangle \mathcal{R} \langle F, \psi \rangle\}$.

Definition 1. (Generalized Unwinding) Let $\doteq$ be a binary relation over Σ, $\mathcal{R}$ and $\dot{\div}$ be two binary relations over $\mathbb{P} \times \Sigma$. We define the *unwinding class* $\mathcal{W}(\doteq, \mathcal{R}, \dot{\div})$ by:

$$\begin{aligned} \mathcal{W}(\doteq, \mathcal{R}, \dot{\div}) \overset{\text{def}}{=} \{\langle P, \sigma \rangle \in \mathbf{Prog} \times \Sigma \mid \forall\, \langle F, \psi \rangle \in \mathcal{R}(\langle P, \sigma \rangle) \\ \text{if } \langle F, \psi \rangle \overset{\texttt{high}}{\to} \langle G, \varphi \rangle \text{ then} \\ \forall\, \pi \in \Sigma \text{ such that } \pi \doteq \psi,\ \ \exists\, \langle R, \rho \rangle : \\ \langle F, \pi \rangle \to \langle R, \rho \rangle \text{ and } \langle G, \varphi \rangle \dot{\div} \langle R, \rho \rangle\} \end{aligned}$$

We will now apply the concept of generalized unwinding in one of its simplest forms, which will serve as a sufficient foundation for our analysis in the subsequent section focusing on our case study. As far as the relation $\doteq$ is concerned, it is quite natural to consider two states to be equivalent when they assign the same values to the low-level variables, i.e., we consider $\doteq$ to be the relation $=_l$. The relation $\mathcal{R}$ is the notion of reachability we rely on, i.e., $\leadsto$ is defined by the operational semantics. In [27], we introduced the concept of low-level bisimulation to define what the low-level user can observe. Specifically, we utilized low-level bisimulation to instantiate the equivalence relation denoted by $\dot{\div}$. Bisimulation is the appropriate notion to employ when it is assumed that the low-level user can observe the values of low-level variables at any point during the execution. Also other more involved forms of approximating equivalences such as the one presented in [32] could be used for our aims. However, for the sake of simplicity in our current presentation, we make the assumption that the low-level user can only observe the values of variables at the end of the execution. We are aware of the fact that this is not a good choice when non-terminating programs are considered. Formally this means that we instantiate $\dot{\div}$ as $\approx_l$ defined as follows.

Definition 2 ($\approx_l$)**.** Let $\langle G, \varphi \rangle$ and $\langle R, \rho \rangle$ be two pairs. It holds that $\langle G, \varphi \rangle \approx_l \langle R, \rho \rangle$ if and only if whenever $\langle G, \varphi \rangle \leadsto \langle \texttt{end}, \varphi' \rangle$, there exists a pair $\langle \texttt{end}, \rho' \rangle$ such that $\langle R, \rho \rangle \leadsto \langle \texttt{end}, \rho' \rangle$ with $\varphi' =_l \rho'$, and vice-versa (i.e., $\approx_l$ is symmetric).

Now that we have gathered all the necessary components, our objective is to demonstrate that a program P does not exhibit interference. In other words, for all possible states σ, the pair $\langle P, \sigma \rangle$ belongs to $\mathcal{W}(=_l, \leadsto, \approx_l)$.

As shown in our previous work [27], when a program does not belong to a given unwinding class, it is possible to construct a malicious environment in which information can flow from high to low. This observation implies that even by analyzing the program in isolation, we can anticipate and identify potential security vulnerabilities. Conversely, if a program does belong to an unwinding class, then no information leaks can occur—regardless of how the environment behaves.

One of the key strengths of the unwinding approach lies in its ability to pinpoint the exact points in the program where information flow might arise. As illustrated in Figure 1, such flows are triggered when transitions involving high-level data are executed.

Example 1. In order to provide some more intuition on the meaning of the unwinding condition, let us consider the followning toy example.

$$P \equiv H := 0; \texttt{if}(H > 0)\{L := H\}\ \texttt{else}\ \{\texttt{skip}\}$$

Let H be high-level and L be low-level. Apparently, when we reach the if test the value of the high-level variable is 0, so the else branch is always taken and the value of the high-level variable is not revealed.

$$\begin{tikzcd}
\langle F, \psi \rangle \arrow[r, "\text{high}"] \arrow[d, leftrightarrow] & \langle G, \varphi \rangle \arrow[r, rightsquigarrow] \arrow[d, leftrightarrow] & \langle \text{end}, \varphi' \rangle \arrow[d, leftrightarrow] \\
\pi =_l \psi \arrow[d, leftrightarrow] & \approx_l \arrow[d, leftrightarrow] & \varphi' =_l \rho' \arrow[d, leftrightarrow] \\
\langle F, \pi \rangle \arrow[r] & \langle R, \rho \rangle \arrow[r, rightsquigarrow] & \langle \text{end}, \rho' \rangle
\end{tikzcd}$$

Figure 1: A pictorial representation of the unwinding condition $\mathcal{W}(=_l, \rightsquigarrow, \approx_l)$.

However, this program does not satisfy our unwinding condition as one can observe taking for instance ψ which assigns value 0 to H and π that is $\psi[H/1]$. As a matter of fact, when the if test is reached we have to consider the possibility that another program running in parallel with P has modified the value of H, thus allowing to take the if branch and reveal the value of the variable H to the low-level user.

3.2. Downgrading

As observed by numerous researchers, the notion of noninterference, while foundational, can be overly restrictive in many practical scenarios. By definition, noninterference enforces a strict absence of any information flow from high to low levels. However, in real-world applications, programs often require controlled release, or downgrading, of sensitive information. Common examples include password validation, access-controlled data retrieval, and computations in spreadsheets involving both public and confidential inputs [22, 33].

The intuitive concept of downgrading conceals several subtle challenges at the implementation level. It naturally leads to critical questions such as: Who is authorized to perform downgrading? What information can be downgraded? Where and when is downgrading permissible? In our previous work [27], we addressed these concerns by introducing a set of high-level expressions and proposing a delimited notion of noninterference. This refined model allows for the controlled downgrading of designated expressions at any point during program execution.

In this paper, we focus on a notion of downgrading that is tied to the specific point during execution at which the downgrading takes place. This choice is consistent with our use of unwinding conditions to identify precise execution points in a smart contract where potential reentrancy vulnerabilities may occur. To support this approach, we introduce a function, `downgrade`, which maps any arithmetic or boolean expression to itself while explicitly lowering its confidentiality level to `low`. As a result, program instructions involving `downgrade` are treated as declassified in the operational semantics and are not considered dangerous within the unwinding test.

Example 2. We consider again the program P of Example 1. If we know that revealing to the low-level user the value of H is not dangerous, and it is necessary (e.g., the system administrator needs to send to the user a message), then we can modify the program P as follows:

$$P \equiv H := 0; D := \texttt{downgrade}(H); \texttt{if}(D > 0)\{L := D\} \ \texttt{else} \ \{\texttt{skip}\}$$

If D is low-level, the only point where we should check the unwinding condition, is the second assignment instruction. However, since the `downgrade` operator is used, when the assignment is executed a low-level transition is performed and the unwinding condition is satisfied.

4. Case Study: An Auction Contract

We are now ready to model a smart contract implementing an auction by using our language. The original contract written in Solidity is depicted in Table 2 and is taken from [34].

```
pragma solidity ^0.8.28;

contract ReentrantAuction {

    address payable public seller;
    uint public endTime;
    address public highestBidder;
    uint public highestBid;

    mapping(address => uint) public bids;

    constructor(uint _startingBid, uint _duration) {
        seller = payable(msg.sender);
        highestBid = _startingBid;
        endTime = block.timestamp + (_duration * 1 seconds);
    }

    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.");
    }

    function end() external {
        require(msg.sender == seller, "Only the seller");
        require(block.timestamp >= endTime, "Auction not ended");
        (bool success, ) = seller.call{value: highestBid}("");
        require(success, "Transfer failed.");
    }
}
```

Table 2
Auction sample contract in Solidity. The `withdraw()` function exhibits a reentrancy vulnerability due to the assignment to zero taking place *after* the `call` rather than before. This allows attackers to call again the `withdraw()` function without ever reaching that line and performing unwanted payments.

In our model, a contract is a tuple of programs, thus we define:

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

We will show that the code of this contract is subject to malicious reentrancy attacks due to the way it is implemented. The contract contains two main procedures: Bid and Withdraw, defined as follows:

```
1: **Program** Bid
2:     PrevBid := Bids[Sender];
3:     **if** (CallValue $\leq$ HighestBid $\vee$ PrevBid $\neq 0$) **then**
4:         skip
5:     **else**
6:         HighestBidder := Sender;
```

```
7:        HighestBid := CallValue;
8:        Bids[Sender] := CallValue
1: Program WITHDRAW
2:     Amt := Bids[Sender];
3:     call(RECEIVE, Amt);        ▷ $B_{caller}$ -= Amt,  $B_{callee}$ += Amt
4:     Bids[Sender] := 0
```

We denote variables using capitalized identifiers. In all examples, we omit the caller program in the subscript of the `call` primitive, as it is implicitly understood to be the program from which the call originates. Thus, any call of the form $\texttt{call}(P', a)$ should be interpreted as being issued by the currently executing program, typically the one enclosing the call. The identifiers highlighted in red represent *high* locations, from which unwanted information flows may originate; conversely, those highlighted in green represent *low* locations that may be affected by such flows. A few identifiers have a special meaning: `Sender` refers to the identifier of the user invoking the current smart contract, while `CallValue` denotes the amount of money transferred by the caller via the `call` primitive.

Each bid is accepted only if the `Sender` has not already placed a bid, and the offered amount (`CallValue`) exceeds the current `HighestBid`. If both conditions hold, the bid is recorded and the highest bidder is updated accordingly. Our model faithfully reproduces the fund transfer mechanism used in Solidity on Ethereum, where transferring money to another contract is done by invoking a `call` method that implicitly dispatches a RECEIVE() function defined in the target contract. The same happens in our language: the WITHDRAW procedure allows participants to reclaim their funds by calling a RECEIVE function with the amount previously bid.

The call to RECEIVE within the WITHDRAW procedure is crucial to understand the mechanisms undergoing reentrancy. It serves two purposes: triggering the execution of the target program (RECEIVE) and transferring funds from the balance of the caller contract to the balance of the callee contract. Notably, balances are associated with contracts, not programs, whereas the `call` primitive manipulates programs, not contracts. Being contracts just tuples of programs, though, it is easy to reconstruct which contract a given program belongs to.

In the comment, the B_{caller} and B_{callee} are two placeholders indicating, respectively, the balance of the caller contract and the callee contract, whatever they are upon execution. The examples in the sections below will show how those two placeholders become contract names when a contract is run.

4.1. A Harmless Interaction

We now introduce another contract, which implements the code of a user participating to the auction. Such contract does not exploit the reentrancy vulnerability and is an example of a harmless interaction consisting of the following two programs:

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

where the implementation are:

```
1: Program MAIN
2:     call(BID, 100);        ▷ $B_{HARMLESS}$ -= 100,  $B_{REENTRANTAUCTION}$ += 100
3:     call(WITHDRAW, 0)
1: Program RECEIVE
2:     skip
```

In the MAIN program, a user wishing to participate in the auction places a bid with an amount of 100. This amount is transferred from the balance of HARMLESS to that of REENTRANTAUCTION, since the former contains the calling context and the latter contains the BID program. In other words, that call could be rewritten in our formal language as:

$$\texttt{call}_{\textsc{Main}}(\textsc{Bid}, 100)\ \texttt{where}\ \textsc{Main} \in \textsc{Harmless} \wedge \textsc{Bid} \in \textsc{ReentrantAuction}$$

hence, the two balances involved are $B_{\textsc{Harmless}}$ and $B_{\textsc{ReentrantAuction}}$.

The following line performs a call with zero amount, which reduces to a simple program invocation and produces no effect on balances. When invoking WITHDRAW, the user expects to receive money via the execution of the RECEIVE program supplied. The dummy implementation of RECEIVE is key to understanding why the contract is harmless and does not allow reentrancy attacks. Since our example mimicks the same callback mechanism undergoing in Solidity, which allows the RECEIVE function to perform any arbitrary logic, implementing it as a `skip` ensures that it solely receives the funds without triggering any additional operation.

When running this code, once the WITHDRAW program is invoked, the caller's balance appearing in the comment of the WITHDRAW function in the previous section refers to $B_{\textsc{ReentrantAuction}}$, and the callee's balance to $B_{\textsc{Harmless}}$. Hence, the caller's balance is decreased by the transferred amount, while the callee's balance is increased by the same amount.

The execution of the HARMLESS contract starts from the MAIN program:

$$\begin{array}{cl}
\langle \textsc{Main},\ \sigma[B_{\textsc{ReentrantAuction}}/200,\ B_{\textsc{Harmless}}/100]\rangle & \\
\downarrow & \\
\dots & \textit{execution of } \textsc{Main} \\
\downarrow & \\
\langle \textsc{Bid},\ \sigma[B_{\textsc{ReentrantAuction}}/200,\ B_{\textsc{Harmless}}/100]\rangle & \\
\downarrow & \\
\dots & \textit{execution of } \textsc{Bid} \\
\downarrow & \\
\langle \textsc{Withdraw},\ \sigma[B_{\textsc{ReentrantAuction}}/300,\ B_{\textsc{Harmless}}/0,\ \mathrm{Bids[Sender]}/100]\rangle & \\
\downarrow & \\
\dots & \textit{execution of } \textsc{Withdraw} \\
\downarrow & \\
\langle \textsc{Receive},\ \sigma[B_{\textsc{ReentrantAuction}}/200,\ B_{\textsc{Harmless}}/100,\ \mathrm{Bids[Sender]}/100]\rangle & \\
\downarrow & \\
\dots & \textit{execution of } \textsc{Receive} \\
\downarrow & \\
\langle \textsc{Withdraw},\ \sigma[B_{\textsc{ReentrantAuction}}/200,\ B_{\textsc{Harmless}}/100,\ \mathrm{Bids[Sender]}/100]\rangle & \\
\downarrow & \\
\dots & \textit{return to } \textsc{Withdraw} \\
\downarrow & \\
\langle \textsc{skip},\ \sigma[B_{\textsc{ReentrantAuction}}/200,\ B_{\textsc{Harmless}}/100,\ \mathrm{Bids[Sender]}/0]\rangle &
\end{array}$$

4.2. A Reentrancy Attack

Consider now a malicious setup where the RECEIVE programs are written by an attacker and embedded in the following contract:

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

Whereas the MAIN program is unchanged from the previous section, the RECEIVE implementation differs, as it exploits a reentrant call to WITHDRAW to steal money.

```
Program RECEIVE
    call(WITHDRAW, 0)
```

Reentrancy takes place via repeated, mutually recursive calls between WITHDRAW and RECEIVE, which gradually deplete the caller's balance until it reaches zero, at which point execution halts. The following trace shows this process in action. We focus on a state σ in which the variables have the following values:

$$\sigma(B_{\textsc{ReentrantAuction}}) = 200;\ \sigma(B_{\textsc{Attacker}}) = 100;$$

$$\langle \textsc{Main}, \sigma[B_{\textsc{ReentrantAuction}}/200,\ B_{\textsc{Attacker}}/100]\rangle$$

$$\downarrow$$

$$\ldots \qquad \textit{execution of } \textsc{Main}$$

$$\downarrow$$

$$\langle \textsc{Bid},\ \sigma[B_{\textsc{ReentrantAuction}}/200,\ B_{\textsc{Attacker}}/100]\rangle$$

$$\downarrow$$

$$\ldots \qquad \textit{execution of } \textsc{Bid}$$

$$\downarrow$$

$$\langle \textsc{Withdraw},\ \sigma[B_{\textsc{ReentrantAuction}}/300,\ B_{\textsc{Attacker}}/0,\ \text{Bids[Sender]}/100]\rangle$$

$$\downarrow$$

$$\ldots \qquad \textit{execution of } \textsc{Withdraw}\ \#1$$

$$\downarrow$$

$$\langle \textsc{Receive},\ \sigma[B_{\textsc{ReentrantAuction}}/200,\ B_{\textsc{Attacker}}/100,\ \text{Bids[Sender]}/100]\rangle$$

$$\downarrow$$

$$\ldots \qquad \textit{execution of } \textsc{Receive}\ \#1$$

$$\downarrow$$

$$\langle \textsc{Withdraw+Receive},\ \sigma[B_{\textsc{ReentrantAuction}}/200,\ B_{\textsc{Attacker}}/100],\ \text{Bids[Sender]}/100\rangle$$

$$\downarrow$$

$$\ldots \qquad \textit{execution of } \textsc{Withdraw}\ \#1 \text{ and } \textsc{Receive}\ \#1$$

$$\downarrow$$

$$\langle \textsc{Withdraw+Receive},\ \sigma[B_{\textsc{ReentrantAuction}}/100,\ B_{\textsc{Attacker}}/200,\ \text{Bids[Sender]}/100]\rangle$$

$$\downarrow$$

$$\ldots \qquad \textit{execution of } \textsc{Withdraw}\ \#2 \text{ and } \textsc{Receive}\ \#2$$

$$\downarrow$$

$$\langle \textsc{Withdraw+Withdraw},\ \sigma[B_{\textsc{ReentrantAuction}}/0,\ B_{\textsc{Attacker}}/300,\ \text{Bids[Sender]}/100]\rangle$$

$$\downarrow$$

$$\ldots \qquad \textit{return to } \textsc{Withdraw}\ \#3,\ \textsc{Withdraw}\ \#2,\ \textsc{Withdraw}\ \#1$$

$$\downarrow$$

$$\langle \textsc{skip},\ \sigma[B_{\textsc{ReentrantAuction}}/0,\ B_{\textsc{Attacker}}/300,\ \text{Bids[Sender]}/0]\rangle$$

4.3. A Non-Reentrant Auction

In order to avoid the attacker from being able to leverage a reentrancy attack, the Withdraw program should be edited as follows:

```
Program Withdraw
    Amt := Bids[Sender];
    Bids[Sender] := 0;
    call(Receive, Amt)                ▷ $B_{SafeAuction}$ -= Amt,  $B_{Attacker}$ += Amt
```

The only change lies in the position of the line that resets the bid to zero: it has now been moved *before* the call to Receive.

With reference to the Solidity code in Table 2, this means to apply the same simple fix:

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

This reordering is sufficient to prevent the reentrancy attack, as an attacker invoking the modified Withdraw procedure recursively would find the `Bids[Sender]` location already cleared, and thus be unable to extract further funds. As a result, any subsequent calls to Receive would transfer zero value only.

Let us update the auction contract embedding the new, safer version of the Withdraw as well as the original Bid shown in the previous section:

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

And let the contract invoking the auction be the same malicious Attacker contract defined above. Its execution produces the following trace, in which no information flow compromises the caller's balance. Reentrancy attacks are rendered harmless by the assignment to zero occurring *before* the call, effectively neutralizing any attempt to extract additional funds.

$$\begin{array}{cl}
\langle \textsc{Main}, \sigma[B_{\textsc{SafeAuction}}/200,\ B_{\textsc{Attacker}}/100]\rangle & \\
\downarrow & \\
\dots & \textit{execution of } \textsc{Main} \\
\downarrow & \\
\langle \textsc{Bid},\ \sigma[B_{\textsc{SafeAuction}}/200,\ B_{\textsc{Attacker}}/100]\rangle & \\
\downarrow & \\
\dots & \textit{execution of } \textsc{Bid} \\
\downarrow & \\
\langle \textsc{Withdraw},\ \sigma[B_{\textsc{SafeAuction}}/300,\ B_{\textsc{Attacker}}/0,\ \text{Bids[Sender]}/100]\rangle & \\
\downarrow & \\
\dots & \textit{execution of } \textsc{Withdraw}\ \#1 \\
\downarrow & \\
\langle \textsc{Receive},\ \sigma[B_{\textsc{SafeAuction}}/200,\ B_{\textsc{Attacker}}/100,\ \text{Bids[Sender]}/0]\rangle & \\
\downarrow & \\
\dots & \textit{execution of } \textsc{Receive}\ \#1 \\
\downarrow & \\
\langle \textsc{Withdraw+Receive},\ \sigma[B_{\textsc{SafeAuction}}/200,\ B_{\textsc{Attacker}}/100,\ \text{Bids[Sender]}/0]\rangle & \\
\downarrow & \\
\dots & \textit{execution of } \textsc{Withdraw}\ \#2 \\
 & \text{and } \textsc{Receive}\ \#2 \\
\downarrow & \\
\langle \textsc{Withdraw+Receive},\ \sigma[B_{\textsc{SafeAuction}}200,\ B_{\textsc{Attacker}}/100,\ \text{Bids[Sender]}/0]\rangle & \\
\downarrow & \\
\dots & \textit{execution of } \textsc{Withdraw}\ \#\text{n} \\
 & \text{and } \textsc{Receive}\ \#\text{n} \\
\downarrow & \\
\langle \textsc{skip},\ \sigma[B_{\textsc{SafeAuction}}/200,\ B_{\textsc{Attacker}}/100,\ \text{Bids[Sender]}/0]\rangle &
\end{array}$$

The last state transition proves that the balance associated to the malicious contract Attacker has not increased, and the original amount of money stored in the balance associated to the SafeAuction contract is unchanged.

4.4. A case of Downgrading

We now present a simplified version of a crowdfunding contract written in Solidity. At first glance, the `withdraw` function appears vulnerable to a reentrancy attack, as it invokes the `call` function to transfer Ether to an external address `receiver` before updating the contract's internal state, a well-known red flag in smart contract security. However, in this case, the `receiver` address is hardcoded and not dynamically computed, ensuring that no malicious contract can exploit the callback. Furthermore, the 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. This example highlights the limitations of pure noninterference and underscores the importance of taking into account the execution context when evaluating smart contract security.

```
pragma solidity ^0.8.28;

contract Crowdfund {

    bool open;     // flag that closes the Crowdfund
    address receiver;   // receiver of the donated funds
    address owner;

    constructor (address payable receiver_) {
        receiver = receiver_;
        owner = msg.sender;
        open = true;
    }

    function donate() public payable {
        require (open);
    }

    function withdraw() public {
        require (open);
        require (msg.sender == owner);
        (bool succ,) = receiver.call{value: address(this).balance}("");
        open = false;
        require(succ);
    }
}
```

In our model, we define the `CrowdFund` contract as follows:

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

and we consider the $\textsc{Main}$ and $\textsc{Receive}$ programs of the $\textsc{Owner}$ and the $\textsc{Receiver}$:

$$\textsc{Owner} \equiv \langle \textsc{Main} \rangle$$
$$\textsc{Receiver} \equiv \langle \textsc{Receive} \rangle \,.$$

The execution starts from the $\textsc{Main}$ program belonging to the `Owner` contract, which then invokes the $\textsc{Withdraw}$ of the `Crowdfund` contract and finally the $\textsc{Receive}$ of the `Receiver`.

```
1: **Program** Main
2:     call(Withdraw, 0)
1: **Program** Receive
2:     skip
```

We provide only the implementation of `Withdraw`, as the `Donate` function is straightforward.

```
Program WITHDRAW
    if (open = $false \lor$ Sender $\neq$ Owner) then
        skip
    else
        Balance := $B_{\text{CrowdFund}}$;
        call(RECEIVE, downgrade(Balance));
        open := $false$;                    ▷ $B_{\text{CrowdFund}}$ -= $Balance$,   $B_{\text{Receiver}}$ += $Balance$
```

In this case, we use of the downgrade operator that lowers the confidentiality level of Balance, so when the call is executed, it is treated as declassified and is therefore not considered dangerous in the unwinding test, i.e., the unwinding condition is satisfied.

5. Conclusion

The case study presented in this article demonstrates that the non-interference framework is particularly well-suited for capturing the vulnerability known as reentrancy in smart contracts. By applying the unwinding conditions, we were able to systematically identify and understand potential reentrancy vulnerabilities in an auction contract, highlighting the effectiveness of this formal approach.

As future work, we propose to develop a proof system in the style of [35] for the analysis of smart contracts, which will enable formal verification of the absence of reentrancy vulnerabilities. Additionally, we aim to implement a tool based on the non-interference framework as in [36] to facilitate automated analysis of smart contracts, thereby enhancing security and reliability in blockchain applications.

Acknowledgements

This study was carried out within the PE0000014 - Security and Rights in the CyberSpace (SERICS) and received funding from the European Union Next-GenerationEU - National Recovery and Resilience Plan (NRRP) – MISSION 4 COMPONENT 2, INVESTIMENT 1.3 – CUP N. H73C22000890001. This work has been also partially supported by the Research Project INDAM GNCS 2025 - CUP E53C24001950001 - "Modelli e Analisi per sistemi Reversibili e Quantistici (MARQ)" and by the Project PRIN 2020 - CUP N. 20202FCJMH "NiRvAna - Noninterference and Reversibility Analysis in Private Blockchains". This manuscript reflects only the authors' views and opinions, neither the European Union nor the European Commission can be considered responsible for them.

Declaration on Generative AI

The authors used AI-based tools (ChatGPT, Grammarly) exclusively for language editing purposes (grammar, spelling, and style). The scientific content, analysis, and conclusions are entirely the work of the authors, who take full responsibility for the integrity and accuracy of the manuscript.

References

[1] H. Rameder, M. Di Angelo, G. Salzer, Review of automated vulnerability analysis of smart contracts on ethereum, Frontiers in Blockchain 5 (2022) 814977.

[2] D. Ressi, A. Spanò, L. Benetollo, C. Piazza, M. Bugliesi, S. Rossi, Vulnerability detection in ethereum smart contracts via machine learning: A qualitative analysis, arXiv preprint arXiv:2407.18639 (2024).

[3] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, A. Hobor, Making smart contracts smarter, in: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, 2016, pp. 254–269.

[4] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, M. Vechev, Securify: Practical security analysis of smart contracts, in: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security, 2018, pp. 67–82.

[5] B. Mueller, Smashing smart contracts, in: 9th HITB Security Conference, 2018.

[6] B. Jiang, Y. Liu, W. K. Chan, Contractfuzzer: Fuzzing smart contracts for vulnerability detection, in: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, 2018, pp. 259–269.

[7] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, Q. T. Minh, sfuzz: An efficient adaptive fuzzer for solidity smart contracts, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 778–788.

[8] C. Schneidewind, I. Grishchenko, M. Scherer, M. Maffei, ethor: Practical and provably sound static analysis of ethereum smart contracts, in: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, 2020, pp. 621–640.

[9] Y. Xue, M. Ma, Y. Lin, Y. Sui, J. Ye, T. Peng, Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts, in: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020, pp. 1029–1040.

[10] Z. Liao, Z. Zheng, X. Chen, Y. Nan, Smartdagger: a bytecode-based static analysis approach for detecting cross-contract vulnerability, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2022, pp. 752–764.

[11] Y. Wu, X. Xie, C. Peng, D. Liu, H. Wu, M. Fan, T. Liu, H. Wang, Advscanner: Generating adversarial smart contracts to exploit reentrancy vulnerabilities using llm and static analysis, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 1019–1031.

[12] B. Boi, C. Esposito, S. Lee, Smart contract vulnerability detection: The role of large language model (llm), ACM SIGAPP Applied Computing Review 24 (2024) 19–29.

[13] C. Laneve, A. Spanò, D. Ressi, S. Rossi, M. Bugliesi, Assessing code understanding in llms, in: C. Ferreira, C. A. Mezzina (Eds.), Formal Techniques for Distributed Objects, Components, and Systems, Springer Nature Switzerland, Cham, 2025, pp. 202–210.

[14] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, X. Wang, Combining graph neural networks with expert knowledge for smart contract vulnerability detection, IEEE Transactions on Knowledge and Data Engineering 35 (2021) 1296–1310.

[15] L. Guo, H. Huang, L. Zhao, P. Wang, S. Jiang, C. Su, Reentrancy vulnerability detection based on graph convolutional networks and expert patterns under subspace mapping, Computers & Security 142 (2024) 103894.

[16] M. Rizzo, D. Ressi, A. Gasparetto, S. Rossi, A comparison of machine learning techniques for ethereum smart contract vulnerability detection, in: D. Porello, C. Vinci, M. Zavatteri (Eds.), Proceedings of the 6th International Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis, OVERLAY 2024, Bolzano, Italy, November 28-29, 2024, volume 3904 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2024, pp. 119–126.

[17] J. Yu, X. Yu, J. Li, H. Sun, M. Sun, Smart contract vulnerability detection based on multimodal feature fusion, in: International Conference on Intelligent Computing, Springer, 2024, pp. 344–355.

[18] M. Rizzo, A. Spanò, L. Benetollo, D. Ressi, A. Gasparetto, S. Rossi, Advanced large language models prompting strategies for reentrancy classification and explanation in smart contracts, in: D. Ressi, S. Rossi, F. Tiezzi, W. Knottenbelt (Eds.), Proceedings of the 4th EAI International Conference on Blockchain Technology and Emerging Applications BLOCKTEA 2025, Venice, Italy, September 18-19, 2025, Springer – LNICST series, 2025.

[19] D. Perez, B. Livshits, Smart contract vulnerabilities: Vulnerable does not imply exploited, in: 30th USENIX Security Symposium (USENIX Security 21), 2021, pp. 1325–1341.

[20] J. A. Goguen, J. Meseguer, Security policies and security models, in: 1982 IEEE Symposium on Security and Privacy, 1982, IEEE Computer Society, 1982, pp. 11–20. doi:10.1109/SP.1982.10014.

[21] R. Focardi, S. Rossi, Information flow security in dynamic contexts, J. Comput. Secur. 14 (2006)

65–110. doi:10.3233/JCS-2006-14103.

[22] S. Crafa, S. Rossi, Controlling information release in the pi-calculus, Inf. Comput. 205 (2007) 1235–1273. doi:10.1016/J.IC.2007.01.001.

[23] J. Hillston, A. Marin, C. Piazza, S. Rossi, Persistent stochastic non-interference, Fundam. Informaticae 181 (2021) 1–35. doi:10.3233/FI-2021-2049.

[24] A. Bossi, R. Focardi, C. Piazza, S. Rossi, Bisimulation and unwinding for verifying possibilistic security properties, in: L. D. Zuck, P. C. Attie, A. Cortesi, S. Mukhopadhyay (Eds.), Verification, Model Checking, and Abstract Interpretation, 4th International Conference, VMCAI 2003, New York, NY, USA, January 9-11, 2002, Proceedings, volume 2575 of *Lecture Notes in Computer Science*, Springer, 2003, pp. 223–237.

[25] A. Bossi, C. Piazza, S. Rossi, Unwinding conditions for security in imperative languages, in: Logic Based Program Synthesis and Transformation, 14th International Symposium, LOPSTR 2004, volume 3573 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 85–100. doi:10.1007/11506676_6.

[26] A. Bossi, R. Focardi, C. Piazza, S. Rossi, Verifying persistent security properties, Comput. Lang. Syst. Struct. 30 (2004) 231–258. doi:10.1016/J.CL.2004.02.005.

[27] A. Bossi, C. Piazza, S. Rossi, Compositional information flow security for concurrent programs, J. Comput. Secur. 15 (2007) 373–416. doi:10.3233/JCS-2007-15303.

[28] S. Guesmi, C. Piazza, S. Rossi, Noninterference analysis for smart contracts: Would you bet on it?, in: M. Bartoletti, C. Schifanella, A. Vitaletti (Eds.), Proceedings of the Sixth Distributed Ledger Technology Workshop (DLT 2024), Turin, Italy, May 14-15, 2024, volume 3791 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2024.

[29] M. Bartoletti, J. H.-y. Chiang, A. Lluch Lafuente, Maximizing extractable value from automated market makers, in: International Conference on Financial Cryptography and Data Security, Springer, 2022, pp. 3–19.

[30] B. Weintraub, C. F. Torres, C. Nita-Rotaru, R. State, A flash (bot) in the pan: measuring maximal extractable value in private pools, in: Proceedings of the 22nd ACM Internet Measurement Conference, 2022, pp. 458–471.

[31] G. R. Andrews, Foundations of Multithreaded, Parallel, and Distributed Programming, Addison-Wesley, 2000.

[32] A. Marin, C. Piazza, S. Rossi, Proportional lumpability, in: É. André, M. Stoelinga (Eds.), Formal Modeling and Analysis of Timed Systems - 17th International Conference, FORMATS 2019, Amsterdam, The Netherlands, August 27-29, 2019, Proceedings, volume 11750 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 265–281. doi:10.1007/978-3-030-29662-9_16.

[33] A. Marin, C. Piazza, S. Rossi, *D_PSNI*: Delimited persistent stochastic non-interference, Theor. Comput. Sci. 884 (2021) 116–135. doi:10.1016/J.TCS.2021.08.007.

[34] M. Bartoletti, L. Benetollo, M. Bugliesi, S. Crafa, G. Dal Sasso, R. Pettinau, A. Pinna, M. Piras, S. Rossi, S. Salis, et al., Smart contract languages: A comparative analysis, Future Generation Computer Systems 164 (2025) 107563.

[35] A. Bossi, R. Focardi, C. Piazza, S. Rossi, A proof system for information flow security, in: M. Leuschel (Ed.), Logic Based Program Synthesis and Tranformation, 12th International Workshop, LOPSTR 2002, Madrid, Spain, September 17-20,2002, Revised Selected Papers, volume 2664 of *Lecture Notes in Computer Science*, Springer, 2002, pp. 199–218.

[36] C. Piazza, E. Pivato, S. Rossi, Cops - checker of persistent security, in: K. Jensen, A. Podelski (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings, volume 2988 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 144–152.