# Verifying Smart Contract Security against Re-entrancy Attacks through Relational Value Analysis

DIVYA RATHORE, IIT Madras, India
KARTIK NAGAR, IIT Madras, India

Reentrancy vulnerabilities are a critical security risk in smart contracts, posing a significant threat to the entire blockchain ecosystem. These vulnerabilities arise when a malicious attacker exploits the design of a smart contract to re-enter a function within the execution of another function, thus breaking atomicity and manipulating the smart contract state in unintended ways. While multiple countermeasures have been proposed to fortify smart contracts against re-entrancy based attacks, automatically verifying their effectiveness remains a difficult problem due to the inherent complexity of smart contracts and evolving attack techniques. In this work, we propose RAVEN, a sound and precise approach to verify smart contract safety against re-entrancy attacks automatically. At its core, RAVEN performs a content-sensitive semantic relational value analysis using the polyhedral abstract domain to establish hyper-properties such as absorption and commutativity of different program segments, which are sufficient to ensure safety against re-entrancy. Notably, unlike many prior approaches, we also prove the soundness of RAVEN, thus guaranteeing that contracts deemed as safe by RAVEN would not suffer from classical re-entrancy attacks. We have implemented our approach and evaluated RAVEN against nine state-of-the-art tools using four comprehensive test suites of Solidity smart contracts labeled for re-entrancy. The results demonstrate that RAVEN attains higher precision than existing approaches in detecting both re-entrancy-safe and vulnerable contracts. In particular, RAVEN produced 0/781 false positives on two test suites and 444/21,355 false positives on the remaining two, representing an approximate 77.3% reduction in false positives over prior tools. Moreover, this improvement was achieved with a comparable average analysis time of 141.9 seconds, versus 128.8 seconds for prior tools.

CCS Concepts: • **Theory of computation** → **Program analysis**; • **Software and its engineering** → **Automated static analysis**.

Additional Key Words and Phrases: Blockchain, Smart Contracts, Solidity

**ACM Reference Format:**
Divya Rathore and Kartik Nagar. 2026. Verifying Smart Contract Security against Re-entrancy Attacks through Relational Value Analysis. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE065 (July 2026), 21 pages. https://doi.org/10.1145/3797093

## 1 Introduction

Programming on the blockchain via smart contracts [14] has seen widespread adoption across various domains such as decentralized finance, gaming, crowdfunding, etc. Smart contracts, coded in the Solidity language on the Ethereum blockchain, take advantage of the blockchain infrastructure to enable transparency and establish trust between parties. However, the blockchain's immutability and the Solidity language's intricate semantics expose smart contracts to vulnerabilities exploitable by malicious actors [34].

One prominent vulnerability smart contracts have faced in recent years is re-entrancy, which gained significant attention following the infamous hack of "TheDAO" [13], resulting in a loss of

Authors' Contact Information: Divya Rathore, IIT Madras, Chennai, India, cs21d011@cse.iitm.ac.in; Kartik Nagar, IIT Madras, Chennai, India, nagark@cse.iitm.ac.in.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/7-ARTFSE065
https://doi.org/10.1145/3797093

```
1 contract Synthetix {
2   mapping(uint => uint) deposits;
3   uint initIndex, count = 0;
4   function exchange(uint remaining) {
5     count += 1;
6     uint lc = count;
7     uint deposit = deposits[initIndex];
8     if (deposit == 0) initIndex++;
9     else if (deposit > remaining) {
10        uint newAmount =
              deposit-remaining;
11        msg.sender.call(remaining);
12        deposits[initIndex] = newAmount;
13    } else {
14        deposits[initIndex] = 0;
15        msg.sender.call(deposit);
16        initIndex++;
17    }
18    require(lc == count);
19    }
20 }
```

(a) Simplified Synthetix Contract

```
1 contract Attacker {
2
3     address public syn;
4     uint amt = 50;
5
6     constructor() {
7         syn = 0x...; // chain address
8         syn.call(abi.encodeWithSignature
9         ("exchange(uint256)", amt));
10    }
11
12    function () external payable {
13        while (syn.balance > 0)
14        syn.call(abi.encodeWithSignature
15        ("exchange(uint256)", amt));
16    }
17 }
```

(b) Attacker Contract

Fig. 1. Synthetix Contract (left) and Attacker (right).

$60 million. Ensuring safety against re-entrancy is paramount, especially as smart contracts, once deployed on the blockchain, become immutable; hence, auditing the contract's source code before deployment becomes even more critical than traditional software. We illustrate re-entrancy attacks using an example of a real-world contract as shown in Figure 1a.

The *Synthetix* contract maintains a mapping `deposits` containing ether (unit of currency in Ethereum blockchain) stored at various indices along with two more state variables `initIndex` and `count`. `exchange` is a public procedure that any arbitrary contract can call. The objective of `exchange` function is to transfer `remaining` amount of ether to its caller only if the `deposits` map at `initIndex` contains the required amount, otherwise increment `initIndex` to the subsequent index after exhausting the `deposit` present at `deposits[initIndex]`. In addition to the available state variables, the Ethereum Virtual Machine (EVM) also maintains a balance variable, `this.balance`, accounting for the amount of ether a particular contract holds.

To illustrate a re-entrancy attack on *Synthetix*, consider the attacker contract in Fig 1b. The attacker initiates the attack by invoking the `exchange` function in the constructor at line 9. Suppose initially `deposits[initIndex] = 100`, then the attacker gains 50 ether through the external call `msg.sender.call(remaining)` at Line-11 in Fig. 1a (the increment to `this.balance` of `Attacker` happens implicitly). This also transfers the control to attacker's fallback function `Attacker.()`, within which it initiates a new call at line 15 to gain 50 ether again. Notice that this new instance of the `exchange` function has started without completing the older instance; such instances are known as re-entrant calls. The attacker continues this process, repeatedly invoking new instances of `exchange` from its fallback function to increase its own balance, exhausting the *Synthetix* contract balance.

However, notice that Line-12, which updates the `deposits` mapping, has not yet executed in any instance, and hence the read of `deposits[initIndex]` at Line-7 in every instance will be the same (i.e., 100). Eventually, all these instances that were halted in the middle will execute Line-12, updating the `deposits` mapping to 50. Still, the attacker has gained access to a much larger quantity

of ether. This phenomenon is known as *double spending* and is clearly an undesirable outcome resulting from re-entrancy.

To tackle the re-entrancy problem, Ethereum foundation recommended various syntactic patterns [31] such as Check-Effect-Interaction (CEI) to ensure safety. The CEI pattern requires deferring external calls (i.e. `msg.sender.call` at Line-11) towards the end of function bodies. However, these syntactic patterns are often too restrictive since assuming every external call can lead to a re-entrancy attack is very pessimistic, which can hamper the language's expressiveness. Indeed, we find that many smart contracts do not follow such syntactic patterns but instead use semantical countermeasures, which detects and blocks attackers from mounting a re-entrancy attack without restricting normal re-entrancy-free executions.

For example, the *Synthetix* contract does not follow the CEI pattern, but actually prevents the double spending attack with the `require` check on Line 18. Notice that each call to `exchange` increments the `count` variable (Line-5) and also keeps track of the current value of `count` in the local variable `lc`. Now, for the innermost re-entrant call, the relationship `lc==count` certainly holds, however for any outer re-entrant call (including the original call), it fails as the value of state variable `count` has been updated to the total number of calls, but the variable `lc` is local to each instance, and hence contains the older value of `count` when the instance started. Failure of any `require` statement during execution results in the rollback of entire execution, reverting the entire series of re-entrant calls (and the original call), thereby preventing double-spending. The *Synthetix* contract is in fact safe against any kind of re-entrancy attack.

Hence, checking for syntactic patterns–which is the modus operandi of a large number of smart contract verification tools [6, 7, 9, 16, 23–26, 33, 38, 39, 41, 44]–may not be very effective as developers get smarter in fortifying their smart contracts against re-entrancy attacks. At the same time, formally verifying the absence of any re-entrancy-based vulnerability is paramount, especially as smart contracts, once deployed on the blockchain, become immutable; hence, auditing the contract's source code before deployment becomes even more critical than traditional software.

Semantic approaches [2, 17] to re-entrancy safety verification go beyound pattern matching, and they typically use symbolic execution to track state changes through a possible re-entrancy attack to show its (in-)feasibility. However, re-entrant executions are cyclic in nature, as an attacker can repeatedly enter functions of the smart contract an arbitrary number of times. Showing the absence of such executions often requires a call-back invariant, analogous to loop invariants used in traditional sequential programming, which guarantees safety against re-entrancy despite an arbitrary number of callbacks. Existing symbolic approaches only reason about finite portions of re-entrant executions [6, 24, 26, 41] and hence cannot infer such call-back invariants, leading to failure in verifying the absence of re-entrancy attacks. In fact, all of the existing approaches would fail to show that the Synthetix contract is safe from re-entrancy.

Proving the safety of the Synthetix contract requires the callback invariant $count' > count$, where $count'$ is the value of the `count` state variable after an arbitrary number of re-entrant calls to `exchange` starting from the initial value of $count$. This, coupled with the fact that $lc = count$ would ensure failure of the `require` clause. In traditional sequential programming, abstract interpretation [11]-based approaches have been immensely successful in inferring complex loop invariants, so a promising solution could be to deploy such an approach to inferring call-back invariants in smart contracts.

In this work, we investigate the feasibility of such an approach and develop a methodology to use abstract interpretation based value analysis for verifying smart contracts against re-entrancy-based attacks. Smart contracts are typically used for financial tasks and hence feature many arithmetic computations, making them ideal targets for numerical abstract domains such as the polyhedral domain. However, using abstract interpretation for smart contracts is not straightforward, as it

typically operates over the control-flow graph of a program, but in the case of smart contracts, the control flow itself is disrupted by the attacker through re-entrancy. We show how to carefully orchestrate the fixpoint computation used by abstract interpretation over callback-free segments of the program to infer the callback invariant, which can then be used to show safety against re-entrancy. To summarize, we propose a novel context-sensitive *semantic relational analysis* over smart contracts to prove their safety against re-entrancy.

We have implemented our approach in a tool named RAVEN and evaluated it against nine state-of-the-art analyzers using a comprehensive labeled dataset of real-world smart contracts. The results demonstrate that RAVEN significantly outperforms existing tools by accurately detecting contracts vulnerable to re-entrancy while reliably proving the safety of non-vulnerable contracts. Furthermore, owing to its carefully chosen design decisions, RAVEN scales effectively to large contracts even for the computationally expensive polyhedral abstract domain, maintaining an average analysis time of 141.93 seconds, which is comparable to other tools (128.87 seconds). We also present the first systematic software engineering study that categorizes and evaluates the adoption of semantic countermeasures against re-entrancy by developers, offering fresh insights into practical defense strategies.

The rest of the paper is organized as follows: §2 presents an overview of RAVEN. §3 gives a formal foundation for our approach, while §4 contains the analysis algorithms. Details of our experimental evaluation are given in §5, followed by comparison with related work and conclusion in §6.

## 2 Overview

Before diving into an overview of our approach, we give a brief primer on abstract interpretation [11], which is a program analysis technique used for statically inferring properties of programs. It is parameterized by an abstract domain $\mathcal{D}$, and computes values from the domain at every program point such that the abstract value $d$ at a program point $p$ encapsulates all the concrete program states that are reachable at $p$ in any execution. To compute these abstract values, each program statement is associated with an abstract transfer function, which encodes the effect of the program statement on elements of the domain. In addition, the abstract domain is a lattice, with associated operators such as join ($\sqcup$), meet ($\sqcap$), comparison ($\leq$) which are used in the fixed-point computation procedure for computing the abstract value at each program point. Polyhedral domain [12] is a powerful abstract domain typically used for precisely inferring linear relations among variables of a program. Each element of the polyhedral domain is a collection of linear inequations of the form $\sum \alpha_i X_i \geq \beta$ where $X_i$ are program variables and $\alpha_i, \beta$ are integer/rational/floating constants.

We now present how RAVEN uses abstract interpretation to determine whether a smart contract is safe from re-entrancy attacks. We will gradually build up to how RAVEN successfully determines the `Synthetix` contract to be safe. To simplify the exposition, we only present a basic overview of RAVEN's analyses, without going into too many details, which will be presented in the later sections. Like previous symbolic approaches, RAVEN tries to show that a re-entrant execution is either infeasible or is equivalent to another re-entrancy-free execution. We use the notion of final state equivalence [2, 17], which requires the existence of another re-entrancy-free execution reaching the same values of state variables. This notion is more precise than conflict serializability based approaches such as SliSE [41] or Sailfish [6], which relies on detecting conflicting accesses to state variables. To check final state equivalence, RAVEN checks properties such as commutativity and absorption of different external call-site free code segments, as we will now explain in detail.
**Absorption:** Absorption relies on determining whether there are any state changes at all during the execution of a callback in a re-entrant execution. If there are none, then the callback can be effectively absorbed, i.e. removed from the execution without any change in the final state.

More formally, consider a public function $f$ of the smart contract under analysis and another function $f'$ (could be the same as $f$), and suppose $f$ makes an external call at call-site $s$. Starting from some arbitrary state $\sigma$, suppose we execute $f$ until it just reaches the call-site $s$ with the state $\sigma'$ (we call this call-site free portion of the function as the pre-call segment). Following this, through re-entrancy, suppose an attacker executes a call-back to $f'$, leading to the state $\sigma_c$. We say that $f'$ is absorbed at call-site $s$ in $f$ if $\sigma' = \sigma_c$.

To illustrate the absorption property, consider the `Defi` contract (a simplified version of a real-world contract) in Fig. 2. The `init` function is susceptible to a potential re-entrancy attack through the `call` statement at line-9. However, it uses the variable `te` to protect itself from re-entrancy, since it is set to `true` before the `call` statement, thus making a re-entry possible only if `te` is `false`. Hence, any re-entrant invocation to `init` will leave the contract state unaltered.

We now consider designing a value analysis using the polyhedral abstract domain to infer the absorption property. In Fig. 3, we depict the control flow graphs used by the analysis, along with the abstract values computed at various program points. We begin the analysis from the start of `init` function with an arbitrary state $\sigma$, abstracted by the value $\hat{\sigma} = \top$ from the polyhedral domain. We pause the analysis at the call-site, (thus analyzing only the pre-call segment), establishing the fact `te = 1` $\wedge$ `me = 1` (an abstract value in the polyhedral domain with true modelled as 1 and false as 0). In the context of this abstract state, we now perform another value analysis of `init` (which corresponds to the callback to `init`), by first introducing a primed version V′ of every state variable, and beginning the analysis with an abstract value that captures the equality of the primed and unprimed variables (given by $\hat{\sigma'}$ beside the Entry block on the right side in Fig. 3). The abstract transfer functions of statements in the callback invocation operate over the primed versions of the state variables. Since the analysis is performed under the context of the state reached at the call-site, the `if` check will succeed, directly exiting the callback with the state $\hat{\sigma_c}$.

Upon the end of the analysis, the absorption check is performed over the abstract state $\hat{\sigma_c}$, by checking that the unprimed and primed version of every state variable should have the same value. Notice that the unprimed version captures the value before the re-entrant call, while the primed version captures the value after the call. The equality check is done using the meet operator (denoted by $\sqcap$) of the polyhedral domain (we only show the check for `te`, the same is done for all state variables):

$$(\hat{\sigma_c} \sqcap \mathtt{te}' - \mathtt{te} > 0 == \bot) \wedge (\hat{\sigma_c} \sqcap \mathtt{te'} - \mathtt{te} < 0 == \bot) \tag{1}$$

This effectively checks whether the abstract state $\hat{\sigma_c}$ does not encapsulate any concrete state where the variables `te` and `te'` have different values. The *Absorption* property can help in verifying re-entrancy safety of smart contracts which use such lock/mutex mechanisms. While Absorption considers those executions where the callback doesn't have any effect on the state, there could be callbacks which rely on algebraic properties such as commutativity of arithmetic operations, thereby ensuring the existence of another callback free execution leading to the same final state. Such behaviors can be identified by using the *commutativity* property, which constructs an equivalent callback free execution by moving the callback either before pre-call segment or after post-call segment.

**Pre-Commutativity:** Consider a public function $f$ containing an external call-site $s$, and another function $f'$. We say that $f$ pre-commutes with $f'$ at $s$ if executing the pre-call segment of $f$ and the callback $f'$ in either order leads to the same state, for all possible start states. Formally, starting from arbitrary state $\sigma$, if executing the pre-call segment followed by the callback leads to state $\sigma'$, and executing them in the other order leads to state $\sigma''$, then $\sigma' = \sigma''$. To illustrate pre-commutativity, consider the `Defi` contract in Fig. 2 again, and the functions `init` and `minting`. Suppose `minting` is performed as a callback by the attacker in the `call` statement at line-9 from `init`. Notice that

```
1 contract Defi {
2   uint ct, mt;
3   bool te, me;
4   function init(uint amt){
5     if (te) return;
6     require(me);
7     te = true;
8     mt += 1;
9     msg.sender.call.value(amt)();
10    ct += 1;
11    me = false;}
12  function minting(){
13    require(me);
14    mt += 1;
15  }
16  function transfer(){
17    require(te);
18    ct += 1;
19  }
20 }
```

Fig. 2. Defi Contract

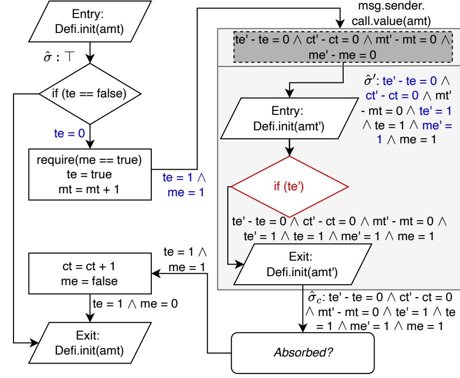


Fig. 3. Absorption

unlike the re-entry of the `init` function which was prohibited by the boolean `te`, `minting` can be freely re-entered. However, executing `minting` and the pre-call segment of `init` in any order would lead to the same final state due to the commutativity of integer addition, since these code segments only contain an increment of the variable `mt`.

We now consider designing a value analysis using the polyhedral domain to check the pre-commutativity property. As shown in Fig. 4, we perform two different value analyses capturing the relational summary corresponding to the two different orders of executing the pre-call segment and the callback, and then check the values of the state variables at the end. To ensure that both the flows start from the same (but arbitrary) initial state, we introduce three different versions of all the state variables: unprimed V, primed V′ and double-primed V″ as depicted in grey boxes. The unprimed V variables correspond to the initial state (on which we do not make any assumptions). For the flow corresponding to execution of pre-call segment of `init` followed by `minting`, we introduce a primed V′ version of the variables to track the state changes. For the other flow, we use double-primed V″ variables. Crucially, the initial abstract state of both analyses equates both the primed V′ and double-primed versions V″ with the unprimed version V. In Fig. 4, we have highlighted the interesting bits of the abstract state at every program point in blue.

Finally, if $\hat{\sigma'}$ and $\hat{\sigma''}$ are the abstract states obtained at the end of the both the analyses, we use both the join operator (⊔) and the meet operator (⊓) of the polyhedral domain to check pre-commutativity (we show the check only for the variable `mt`, the check is similar for the rest of the variables):

$$((\hat{\sigma'} \sqcup \hat{\sigma''}) \sqcap \mathtt{mt}' - \mathtt{mt}'' > 0 == \bot) \land ((\hat{\sigma'} \sqcup \hat{\sigma''}) \sqcap \mathtt{mt}'' - \mathtt{mt}' < 0 == \bot) \quad (2)$$

**Post-Commutativity:** Similar to pre-commutativity, this property considers showing equivalence of the final state if the callback in a re-entrant execution is moved after the post-call segment (i.e. the portion of the function after the external call). Consider functions $f, f'$ where $f$ is a public function with call-site $s$, we say that $f$ post-commutes with $f'$ at $s$ if executing the post-call segment of $f$ and the call-back $f'$ in either order leads to the same state. Formally, beginning from an arbitrary state $\sigma$, we first execute the pre-call segment leading to the state $\sigma_1$, following which the call-back and post-call segment are executed in either order beginning from the state $\sigma_1$, leading to states $\sigma'$ and $\sigma''$. Post-commutativity then requires that $\sigma' = \sigma''$.

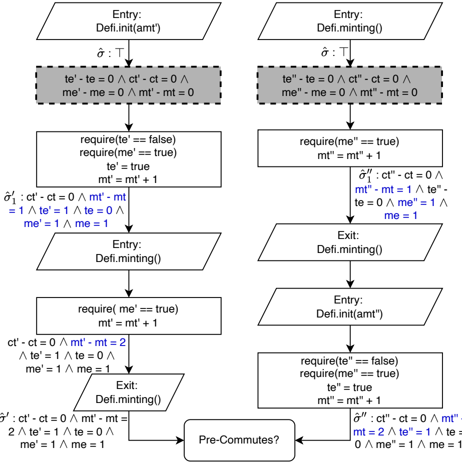

Fig. 4. Pre-Commutativity

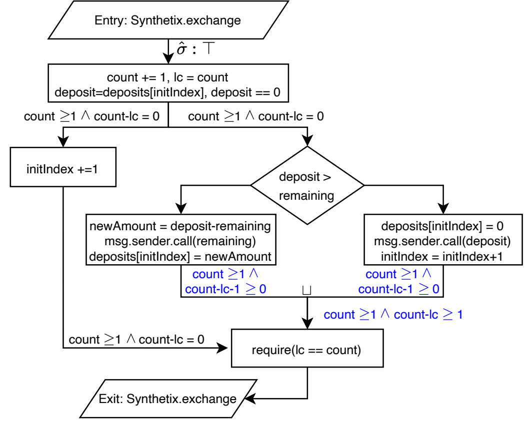

Fig. 5. Synthetix Contract Control Flow

To demonstrate post-commutativity, consider the `Defi` contract again, but this time suppose the callback in the `init` function is made to `transfer` function as shown in Fig. 2. Notice that the pre-call segment of `init` sets the `te` flag required by `transfer`, and hence, it will neither get absorbed, nor will it pre-commute. However, we can establish commutativity with the post-call segment, again due to the commutativity of integer addition, as both the post-call segment of `init` and the function `te` contain increment operations to the variable `ct`. Our value analysis for determining post-commutativity follows the same strategy used for pre-commutativity: we perform two different analyses over the two different ordering of the post-call segment and the callback, using different sets of variables. A detailed explanation of how we detect post-commutativity for the `Defi` contract can be found in the supplementary material.

Finally, we now return back to the `Synthetix` contract (Fig. 1a) which uses the rollback strategy to protect itself from re-entrancy attacks, and show how our analysis is able to prove its safety. Notice that a callback to `exchange` from the call-site inside `exchange` will neither get absorbed, nor will it pre-commute or post-commute due to the read and update to the `deposits` mapping. In such cases, RAVEN performs a fix-point value analysis to compute reachable states after an arbitrary number of call-backs originating from the call-site, and uses this fix-point result to continue the analysis of the post-call segment to check whether the `require` clause at the end is guaranteed to fail. Fig. 5 shows RAVEN's analysis of the `exchange` function of `Synthetix`. `exchange` has two external call-sites, and the abstract states (depicted in blue) at the end of the basic blocks containing the call-sites are obtained through a fixpoint value analysis of all possible callbacks to any public function of `Synthetix`. Next, we perform a join of the two abstract states along the branches with callsites to conclude that `count - lc` $\geq 1$, thus causing the `require` clause to fail. Interestingly, there is one more control path which does not contain any call-site, along which `count = lc`. We do not consider this abstract state, instead only determining reachable states at the start of the `require` statement which are obtained along paths which contain call-sites. Since the `require` statement is guaranteed to fail for all such reachable states, we conclude that the function is safe from re-entrancy.

In the exposition so far, we have not considered how to establish absorption, pre- and post-commutativity for executions with multiple call-backs or nested call-backs, functions with multiple call-sites, and the intricacies of how the various analyses for the different properties interact with

each other. All these factors pertain to various design choices, and in the next section, we give a formal framework to ground the soundness arguments for the design of RAVEN.

## 3 Preliminaries

A smart contract is expressed as a tuple $C = \langle \text{StateVars}, \mathcal{G} \rangle$, where StateVars is a set of state variables which are maintained by the contract as part of the blockchain, and $\mathcal{G} = \{G_1, G_2, \ldots, G_n\}$ is a set of Control-Flow Graphs (CFG) corresponding to every function in the contract. A CFG $G_i = \langle V_i, E_i, Params_i, LocalVars_i \rangle$ consists a set of vertices $V_i$, edges $E_i$, a set of parameter variables $Params_i$ and a set of local variables $LocalVars_i$. Let $LocalVars$ denote the union of $LocalVars_i$ for all functions in the contract. We associate each vertex in every CFG with a program statement which obeys the grammar given below:

$$\text{Statement} ::= \quad \text{x} = e \mid \text{assume}(b) \mid y = \text{extcall} \mid \text{require}(b) \mid \text{return} \mid \text{skip}$$

Here, x, y $\in LocalVars \cup \text{StateVars}$ while $e, b$ are expressions containing local, state and parameter variables with $b$ being a boolean expression. The edges represent control flow among the statements. We simulate conditionals and looping constructs through the assume statement. extcall refers to any of the Solidity commands `call`, `staticcall`, `delegatecall`, `callcode` or `ERC20` call [28] which allow calls to other smart contracts, and thus could become a source of re-entrancy. Finally, `require` will evaluate its input boolean expression $b$, and if $b$ is not satisfied, then it will rollback the execution, i.e. reinstate the state variables to their original values before the execution began. For every CFG $G$, there exist two special vertices: $G.entry$ with no incoming edge and $G.exit$ with no outgoing edge.

An execution of a smart contract is initiated by a call to any of its functions along with supplying values for all the parameters of the function. An **execution** is represented formally using a trace of states $\sigma_1 \ldots \sigma_n$ where each **state** $\sigma_i$ stores information about the current vertex in a CFG of the contract (given by $\text{CFG}(\sigma_i), \text{loc}(\sigma_i)$) and values of state variables and local variables (given by $\gamma(\sigma_i), \psi(\sigma_i)$ resp.). For every consecutive pair of states $\sigma_i, \sigma_{i+1}$ in an execution, there must be an edge between $\text{loc}(\sigma_i)$ and $\text{loc}(\sigma_{i+1})$, and the values of the local and state variables at $\sigma_i$ and $\sigma_{i+1}$ must obey the semantics of the statement at $\text{loc}(\sigma_i)$. An execution $\sigma_1 \ldots \sigma_n$ where $\text{loc}(\sigma_1)$ is $G.entry$ for some CFG $G$, with parameter variables of $G$ initialized appropriately at $\sigma_1$, and $\text{loc}(\sigma_n)$ is $G.exit$ is known as a **transaction**. Henceforth, we will use the terms execution and transaction interchangeably. The semantics of Solidity program statements are similar to those of a normal imperative language, with state variables acting as global variables whose values persist across different function instances, while local variables are freshly allocated for every function instance. A smart contract can call other smart contracts using the `extcall` statement. For our purpose, we are interested in modeling re-entrant execution of a single smart contract. Hence, we model the semantics of `extcall` slightly differently: upon reaching this statement, execution can non-deterministically proceed to any *entry* vertex of any function of the contract, or directly to the successor statement of `extcall`. Further, upon reaching the *exit* vertex of the callee, the execution would continue with the successor statement of `extcall`. Note that this is equivalent to launching a fresh instance of any function of the contract from an `extcall` statement, or simply treating it as a NOP and directly moving to the next statement.

An execution is called **re-entrant** if it contains a transition from an `extcall` statement to the *entry* vertex of a function. The new function instance launched from an `extcall` statement is also known as a **call-back** instance. As demonstrated through the example in §2, such callbacks can happen because of malicious attacker contracts which are called by the original contract. Re-entrant executions with call backs subvert the usual control-flow of Solidity where each function instance of a smart contract is expected to execute atomically. Note that there could also be nested

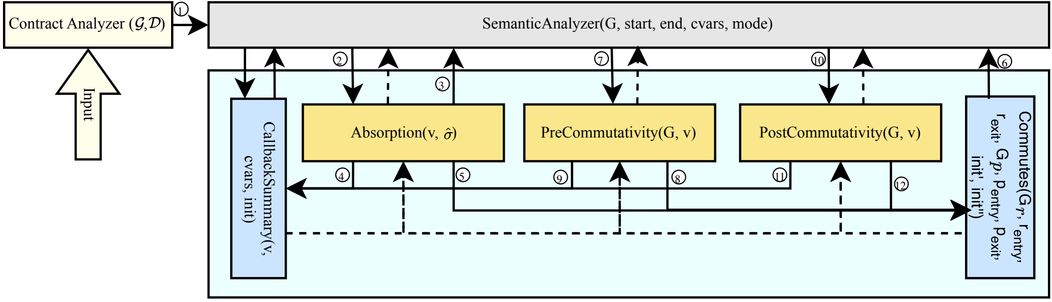


Fig. 6. RAVEN Framework (□:EntryPoint, □:Helpers, □:HyperProperties, □:ProgramStatement handler, Sequence number on the arrows establishes the order of execution of different modules)

re-entrancy, where a call-back instance itself may launch another call-back. Conversely, **non-re-entrant** or callback-free executions do not have any callbacks, so that each function instance executes atomically. In such executions, `extcall` statements would be treated as NOPs. For a more formal definition of re-entrant executions in smart contracts, we refer to earlier works such as Grossman et. al. [17].

A re-entrant execution $\tau = \sigma_1 \dots \sigma_n$ is called **effectively call-back free** (ECF) [17] if there exists a sequence of non-re-entrant executions $\tau_1, \dots, \tau_m$ such that the values of the state variables in the final state $\sigma_n$ are identical to the values in the final state of $\tau_m$, and for every consecutive pair of executions $\tau_i$ and $\tau_{i+1}$, the values of state variables in the final state of $\tau_i$ and the initial state of $\tau_{i+1}$ are identical. Intuitively, the final state reachable through a re-entrant ECF execution can be simulated through multiple non-re-entrant executions. A contract is said to be ECF if all of its executions are ECF.

In this work, we aim to show that a contract is ECF by proving properties such as absorption and commutativity of call-back instances in re-entrant executions. As explained in §2, a call-back instance gets absorbed if it does not cause any change to the state variables. Commutativity of a call-back instance relies on showing that the instance can be moved in the execution before the caller or after the caller without any change in the final state. For proving these properties, we rely on an important result established by Grossman et. al. [17]: for statically proving that a contract is ECF, it is enough to only consider executions with no nested call-backs. Similar to earlier works [2, 3, 17], we assume that call-back instances themselves will not cause any call-backs, which means that the `extcall` statements in call-backs will always behave as NOPs.

## 4 Analysis Algorithms

In this section, we go through the RAVEN architecture in detail. Given a smart contract with public functions containing external call-sites, we would like to establish that each call-site is ECF, i.e. for every execution which contains a call-back originating from a call-site, there exists another non-re-entrant execution without call-backs leading to the same final state. For this purpose, we leverage properties such as absorption and commutativity, which we propose to check using static value analysis. Since this requires multiple value analyses on different code segments with context sensitivity, we organize the entire framework into five major components: *Contract Analyzer, Semantic Analyzer*, *Absorption* Checker, *Commutativity* Checker, and *CallBackSummary Generator*, as shown in Fig. 6. Next, we give a brief summary of what each component does, before diving into the details.

The framework takes as input the CFGs ($\mathcal{G}$) of all the functions of the smart contract, and is parametric on the choice of the underlying abstract domain ($\mathcal{D}$) for performing the value analysis.

The *Contract Analyzer* initiates the analysis by calling the *Semantic Analyzer* (① in Fig. 6) on each public function and using its output to determine whether the contract is ECF or not. The *Semantic Analyzer* drives the overall analysis of a function and determines whether each external call-site in the function is ECF or not. For a function $f$ in the smart contract, it performs a value analysis using the abstract domain $\mathcal{D}$. The value analysis proceeds normally until it hits a call-site. Upon reaching a call-site $v_c$ with incoming abstract state $\hat{\sigma}$ which encapsulates all actual values of program variables reachable at $v_c$, it invokes the *Absorption* and *Commutativity* Checker to determine whether callbacks originating from $v_c$ follow the absorption or commutativity properties under the state $\hat{\sigma'}$.

The *Absorption* Checker (②) determines whether a callback to a function $f_c$ originating from $v_c$ can be absorbed or is guaranteed to cause a rollback, by again invoking the Semantic Analyzer (③) on $f_c$. Since multiple callback instances can originate from $v_c$ and it is possible that all of them may not get absorbed (i.e. they can cause non-zero state changes), hence to maintain soundness, the *Absorption* Checker determines all possible states that can arise after an arbitrary number of callbacks from the non-absorbed callbacks, and under such a context, checks absorption of $f_c$. To obtain such a context, it invokes the *CallbackSummary* Generator (④). As an optimization, the absorption checker also tries to establish commutativity (⑤,⑥) between the absorbed and non-absorbed callbacks (more details on this later in the section). If the absorption check fails, then the Semantic Analyzer invokes the *Commutativity* Checker, which first attempts to establish pre-commutativity (⑦). The *pre-commutativity* Checker calls the Commutes module (⑧) to perform the actual commutativity check, and also invokes the *CallbackSummary* Generator (⑨) to establish a sound context under which the pre-commutativity check should be carried out.

If the pre-commutativity check fails, then the *Commutativity* Checker module attempts to establish post-commutativity (⑩). For soundness, the post-commutativity check is first performed for the last call-site ($v_c^{last}$) in the function. The *CallbackSummary* Generator is again invoked (⑪) to obtain the appropriate context under which post-commutativity of $v_c^{last}$ is checked. Following this, the *Commutativity* Checker attempts to establish post-commutativity of every call-site in between $v_c$ and $v_c^{last}$ through Commutes module (⑫).

### 4.1 Setup

The value analysis operates on different versions of the program variables, namely `UNP`, `PRIME` and `DPRIME` (analogous to V, V′ and V″), given as input to the *Semantic Analyzer* as the argument `cvars`. The treatment of call-sites, (i.e. its abstract transfer function) changes under three modes of the analysis, namely `CHK`, `RE` and `NRE`. In the `CHK` mode, the semantic analyzer will check whether callbacks originating from a call-site get absorbed or commute. In the `NRE` mode, call-sites are assumed to be non re-entrant, and hence their transfer function is assumed to be the identity function. In the `RE` mode, call-sites are assumed to be re-entrant, and hence the *CallbackSummaryGenerator* is invoked to find the transfer function of call-sites.

**Algorithm 1** Contract Analyzer

```
Require: 𝒢 := {G_1 : (V_1, E_1), ... G_n : (V_n, E_n)}
Ensure: isECF – flags the contract 𝒢 as (non)ECF
 1: procedure IsContractECF(𝒢)
 2:     isECF ← True;
 3:     for all G ∈ 𝒢 do
 4:         Initialize: IN, OUT;
 5:         IN[G.entry] ← ⊤
 6:         for all cs ∈ CallSites_G do
 7:             ReM[cs] ← 𝒢
 8:         SAnalysis(G, IN, OUT, G.entry, G.exit, UNP, CHK)
 9:         if OUT[G.exit] is ⊥ then
10:             for all v ∈ CallSites_G do
11:                 ReM[v] ← φ
12:         for all v ∈ CallSites_G do
13:             if ReM[v] ≠ φ then
14:                 isECF ← False
15:     return isECF
```

To perform the standard fixpoint computation required by Abstract Interpretation, the analysis maintains the IN and OUT abstract states for each program statement. However, since there are

multiple analyses which would operate over the same program statements, we maintain multiple IN and OUT maps which are each local to their analysis. Finally, for each call-site, every public function of the smart contract which can be invoked through a callback from the call-site is classified into one of the maps `ReM`, `AbsM`, `PreM`, `PostM`, `RbM`. Functions in `AbsM`, `PreM`, `PostM`, `RbM` get absorbed, pre-commute, post-commute and roll-backed respectively, while the remaining functions are put in `ReM`.

Alg. 1 initiates the analysis for a contract $C = \langle \mathsf{Var}, \mathcal{G} \rangle$ composed of functions with CFGs $\mathcal{G} = \{G_1, \ldots, G_n\}$. In the following, we will often use a function and its CFG interchangeably to mean the same thing. IsContractECF iterates through all the functions present in $C$. First, we initialize the $IN, OUT$ maps for every statement in $G$, with $IN[G.entry]$ initialized to $\top$, which encapsulates every possible state, while $IN, OUT$ of every other statement is initialized to $\bot$ (by the (*Initialize*) call at Line-4), which indicates un-reachability. Next, for each function $G = (V, E)$, we initially assume each call-site $cs$ in $G$ to be re-entrant by initializing $\mathtt{ReM}[cs]$ to contain every function in $\mathcal{G}$, meaning that any of these functions can be invoked through a callback originating from $cs$. Next, The *Semantic Analyzer* (SAnalysis) is then invoked for $G$, with the initialized $IN, OUT$ being passed-by-reference. The `UNP` option indicates that the analysis is to be performed over the unprimed version of the program variables, while the `CHK` option means that every call-site needs to be checked for re-entrancy , with the result being updated in the global map `ReM`. After the call to SAnalysis returns, we next check reachability of the exit node $G.exit$, which allows us to determine whether the function is guaranteed to roll-back in the presence of callbacks due to non-ECF call-sites. In such a case, we modify `ReM` for each call-site to $\emptyset$, to indicate that no call-backs are feasible. Finally, if the `ReM` map is non-empty for some call-site, then the contract is flagged to not be ECF.

**Algorithm 2** Semantic Analyzer

```
1: procedure SAnalysis(G, IN, OUT, start, end, cvars, mode)
2:   worklist ← {start}
3:   while worklist ≠ ∅ do
4:     v ← ♦ worklist
5:     if v ∈ CallSites_G then
6:       if ReM[v] = φ or mode = NRE then
7:         OUT[v] ← IN[v]
8:       else if mode = CHK and ∀p_c ∈ GetPredecessor-
         Callsites(v), ReM[p_c] = φ and (Abs(v, IN[v]) or
         PreC(G, v) or PostC(G, v)) then
9:         OUT[v] ← IN[v]
10:      else
11:        OUT[v] ← CallBackSummary(v, cvars, IN[v])
12:    else
13:      Analyze(IN, OUT, v, cvars)
14:    for all s ∈ Successors(v) do
15:      if ¬(OUT[v] ≤ IN[s]) then
16:        IN[s] ← IN[s] ⊔ OUT[v]
17:        worklist ← worklist ∪{s}
18: procedure CallBackSummary(v, cvars, init)
19:   oldInv ← ⊥; inv ← init
20:   while oldInv ≠ inv do
21:     initialize IN′, OUT′; sum_g ← ⊥
22:     for all G ∈ ReM[v] do
23:       IN′[G.entry] ← inv
24:       SAnalysis(G, IN′, OUT′, G.start, G.end, cvars, NRE)
25:       sum_g ← sum_g ⊔ OUT′[G.exit]
26:     oldInv ← inv
27:     inv ← inv ⊔ sum_g
28:   return inv
```

Alg. 2 contains the SAnalysis procedure, which analyzes the function with CFG $G$ from nodes $start$ to $end$, assuming that $IN[start]$ has been appropriately initialized. It performs a typical fixed-point computation, visiting the nodes of $G$ in topological order, beginning from $start$. For this, it maintains a *worklist* of nodes, initialized to $\{start\}$, with every iteration removing a node $v$ from the worklist in topological order (denoted by $\blacklozenge worklist$), processing its abstract transfer function on $IN[v]$ to compute $OUT[v]$ (Lines 5-13), and propagating the $OUT[v]$ value to its successors (Lines 14-17). The analysis operates on the *cvars* version of variables (recall that Alg. 1 calls SAnalysis with $cvars$ = `UNP`). For non-call-site nodes (i.e., assignments and assumes), the sub-routine *Analyze* on Line-13 is called, which applies the abstract transfer function of the corresponding program statement. The *Analyze* procedure uses the standard transfer functions of the chosen abstract domain $\mathcal{D}$ for assignments and assume statements, and we omit it here for brevity. Note that since $\mathcal{D}$ could be a lattice with unbounded height, to ensure termination of the fixpoint process while analyzing smart contract functions with loops, we use the widening operator while performing the

join operation when a loop header is revisited in the `while` loop. We have omitted this detail from the algorithm description for brevity.

If the node $v$ is a call-site, then we consider the *mode* parameter to determine its transfer function. In particular, if the call-site has already been determined as ECF (which can happen since SAnalysis is also called from other places) or if the mode is NRE, then the transfer function is assumed to be the identity function. Otherwise, if the mode is CHK, then we try to establish the ECF property for all callbacks that can originate from $v$, by checking absorption (Abs), pre-commutativity (PreC) or post-commutativity (PostC) (these procedures are given in Algorithms 3-5, explained in the next sub-section). If any of these properties hold for all possible callbacks to any function of the contract, then the abstract transfer function of the call-site can be soundly considered to be the identity function. The *GetPredecessorCallSites* function on Line-8 returns all the call-sites which occur before $v$ in $G$ according to topographical order. The procedures Abs, PreC and PostC are called only if every call-site before $v$ has already been shown to be ECF (i.e. its ReM mapping should be empty). This check is crucial to maintain soundness as it ensures that the pre-call segment is callback free, which is assumed while checking absorption and commutativity.

If a call-back originating from $v$ cannot be proven to satisfy absorption or commutativity, then we conservatively consider all possible re-entrant calls (to functions which are neither absorbed nor commute and which will be in `ReM[v]`), and calculate the abstract state encapsulating all concrete states that are reachable after any number of callbacks. This is carried out through another fixed point computation by the procedure CallBackSummary, which is also given in Alg. 2. CallBackSummary takes as input a call-site $v$, a version of variables *cvars*, and an initial abstract state *init*. The goal of this procedure is to compute an abstract state which encapsulates all concrete states reachable after any execution of the form $f_1 f_2 \dots f_n$ from any state in *init* where $n \geq 0$ and each of the functions $f_i$ belong to ReM$[v]$. For this purpose, it uses two nested loops, where the inner loop performs a value analysis (by calling SAnalysis) on each function in ReM$[v]$ and calculates the join of all the resultant abstract states. The outer loop repeats this computation until no new states are discovered. Note that CallBackSummary procedure uses the global ReM map to find out which functions have not been shown to get absorbed or commute, and computes the fixpoint summary only using such functions. Abs, PreC and PostC procedures will modify the ReM map by removing such functions.

## 4.2 Absorption and Commutativity

Alg. 3 presents the Abs$(v, \hat{\sigma})$ procedure which identifies those callback functions entered from call-site $v$ which get absorbed under the context of the abstract state $\hat{\sigma}$. This essentially means that executing a callback function $f_r$ from $v$ beginning from a concrete state encapsulated by $\hat{\sigma}$ does not result in any state change. To check the absorption property, Abs considers every function $G_r$ in ReM$[v]$, and calls SAnalysis on $G_r$ to perform value analysis to determine whether execution of $G_r$ can cause a state change (Lines 3-6). For this purpose, it introduces a `primed` version of all the state variables, which are initialized to the `unprimed` version as encoded by the abstract state assigned to $IN'[G_r.entry]$ on Line-5. Further, this abstract state is also joined (using the join operator $\sqcup$ of $\mathcal{D}$) with $\hat{\sigma}$, thus essentially checking for absorption of $G_r$ in the context of $\hat{\sigma}$. Notice that the call to SAnalysis on

**Algorithm 3** Absorption

```
1:  procedure Abs(v, σ̂)
2:      absorbed_f ← {}; ret ← false
3:      for all G_r ∈ ReM[v] do
4:          Initialize: IN′, OUT′
5:          IN′[G_r.entry] ← σ̂ ⊔ (∀s ∈ StateVars : s′ − s = 0)
6:          SAnalysis(G_r, IN′, OUT′, G_r.entry, G_r.exit, PRIME,
            NRE)
7:          if ∀s ∈ StateVars: OUT′[G_r.exit] ⊓ (s′ − s > 0) = ⊥
            and OUT′[G_r.exit] ⊓ (s′ − s < 0) = ⊥ then
8:              absorbed_f ← absorbed_f ∪ {G_r}
9:      if ReM[v] − absorbed_f = ∅ then
10:         ReM[v] ← ∅
11:         AbsM[v] ← absorbed_f
12:         ret ← true
13:     return ret
```

Line-6 is performed using the mode PRIME which indicates that the abstract transfer functions of statements in $G_r$ should be over the primed versions of variables. Also, the mode NRE indicates that all call-sites in $G_r$ are assumed to be non-re-entrant (i.e. their transfer function should be identity), since we do not need to consider nested call-backs.

After the call to SAnalysis, $OUT'[G_r.exit]$ would encapsulate all reachable states at the end of the callback to $G_r$, and hence we check whether there are any non-zero changes to the primed state variables in $OUT'[G_r.exit]$. This check is performed using the meet operator ($\sqcap$) of $\mathcal{D}$, as also demonstrated by the example in Fig. 3 and Eqn. 1 in §2. If there are guaranteed to be no state changes to any of the state variables, then $G_r$ is added to the $absorbed_f$ set.

Finally, if all functions in ReM$[v]$ get absorbed, ReM$[v]$ is set to $\emptyset$, and the procedure returns *true*. Recall that Abs would be called by SAnalysis while analyzing a call-site for re-entrancy in CHK mode. If Abs returns true, then the call-site is ECF, meaning that all call-backs do not have any effect on the state, and the abstract transfer function of the call-site will be taken to be the identity function by SAnalysis.

If all functions do not get absorbed, (i.e. ReM$[v]$ $- absorbed_f \neq \emptyset$), then the Abs procedure would return `false`. However, as an optimization we still take advantage of knowing which functions do get absorbed, and not attempt to establish pre or post commutativity of these functions. Non-absorption of some functions need not invalidate absorption of another function. This can happen due to commutativity of the absorbed and non-absorbed functions: given a sequence of call-backs $f_1 f_2 \dots f_n f_a$, where the call-backs $f_1$ to $f_n$ do not get absorbed, but $f_a$ does get absorbed (i.e. $f_1$ to $f_n$ do not belong to $absorbed_f$, but $f_a$ belongs to $absorbed_f$), if $f_a$ commutes with all of $f_1$ to $f_n$, then the sequence can be re-arranged $f_a f_1 \dots f_n$ (without changing the final state), and the absorption of $f_a$ can be used to conclude that the two sequences $f_1 \dots f_n f_a$ and $f_1 \dots f_n$ would lead to the same final state. Also, functions which get absorbed are generally protected by lock or mutex-like variables (as the `init` function of `Defi` contract in Fig. 2) and these functions will neither pre-commute nor post-commute, thus making it critical for the overall analysis to utilize their absorption properties. To identify such functions, we check the commutativity of every function $f$ which is in $absorbed_f$ with every function $f'$ that is not in $absorbed_f$, and if $f$ commutes with every such function $f'$, we remove it from ReM$[v]$. For brevity, we have elided this step from Alg. 3, but all details about it can be found in section C of the supplementary material.

**Algorithm 4** Pre Commutativity

```
1:  procedure PreC(G, v)
2:      pre_f ← {}; ret ← false
3:      for all G_r ∈ ReM[v] do
4:          init' ← ∀s ∈ StateVars : s' − s = 0 ⊔
                    ∀p ∈ Params_G, ∀r ∈ Params_{G_r} : r' − r'' = 0∧
                    p' − p'' = 0
5:          init'' ← ∀s ∈ StateVars : s'' − s = 0 ⊔
                    ∀p ∈ Params_G, ∀r ∈ Params_{G_r} : r' − r'' = 0∧
                    p' − p'' = 0
6:          if Commutes(G_r, G_r.entry, G_r.exit, G, G.entry, v, init',
            init'') then
7:              pre_f ← pre_f ∪ {G_r}
8:      if ReM[v] − pre_f = ∅ then
9:          PreM[v] ← pre_f
10:         ReM[v] ← ∅
11:         ret ← true
12:     return ret
```

After finding the reentrant functions which get absorbed at call-site v in the function $G$, SAnalysis then calls the procedure PreC to determine pre-commutativity for the functions remaining in ReM$[v]$. Intuitively, if $G_r$ is the re-entrant call, then $G_r$ would pre-commute if executing $G_r$ and the pre-call segment of $G$ before the call-site $v$ in any order would lead to the same final state. We perform this check in the procedure PreC given in Alg. 4.

For every function $G_r$ in ReM$[v]$, we perform the pre-commutativity check separately. We introduce two versions of state variables to correspond to executions of $G_r$ and the pre-call segment of $G$ in two different orders. On Lines 4,5, we create abstract states $init'$ (resp. $init''$) which assert the primed (resp. double-primed) version of every state variable to be equal to the unprimed version. We also constrain the parameters of $G_r$ (and $G$) to remain the same for executions in the two orders,

thus equating the primed and double primed version. Next, we call the Commutes procedure, which will perform a value analysis on $G_r$ and the pre-call segment of $G$ in two different orders, and return *true* if the primed and double-primed versions of all state variables are identical. We omit the Commutes procedure here for brevity, and it can be found in section B.2 supplementary material.

If Commutes returns *true*, then $G_r$ is added to the set $pre_f$. After analyzing all the functions in ReM$[v]$, if we find that all of them pre-commute, then ReM$[v]$ is set to $\emptyset$ and PreC returns true. If there are functions which do not pre-commute, then we return $false$. As an optimization, similar to the Abs procedure, we again check whether the presence of such functions invalidates the pre-commutativity of functions which are in $pre_f$. For this, we again perform commutativity check between functions in ReM$[v] - pre_f$ and $pre_f$. Again, we have elided this detail from Alg. 4, and the complete algorithm can be found in the supplementary material.

Finally, if there are functions which neither get absorbed nor pre-commute at call site $v$ (i.e. both Abs and PreC return $false$), then SAnalysis will check for post-commutativity of these functions using the procedure PostC in Alg. 5. Here, the goal is to check whether executing such a function and the post-call segment of $G$ after the call-site $v$ in either order leads to the same state. However, the post-call segment after $v$ may have other call-sites, and we have not yet established ECF property for these later call-sites. Notice that we did not face this issue while showing pre-commutativity for the pre-call segment, because SAnalysis proceeds in topological order, and hence would have already shown call-sites in the pre-call segment to be ECF.

**Algorithm 5** Post Commutativity

```
1: procedure PostC(G, v)
2:   repeat
3:     last_cs ← GetNonECFLastCallSite(G)
4:     Initialize: IN1, OUT1
5:     ret ← true, post_f ← {}
6:     for all G_r ∈ ReM[last_cs] do
7:       IN1[G.entry] ← ⊤
8:       SAnalysis(G, IN1, OUT1, G.entry, last_cs, UNP, RE)
9:       σ̂_re ← CallBackSummary(last_cs, UNP,
         IN1[last_cs])
10:      init′ ← σ̂_re ⊔ ∀s ∈ StateVars : s′ − s = 0 ⊔
                ∀r ∈ Params_{G_r} : r′ − r″ = 0
11:      init″ ← σ̂_re ⊔ ∀s ∈ StateVars : s″ − s = 0 ⊔
                ∀r ∈ Params_{G_r} : r′ − r″ = 0
12:      if Commutes(G_r, G_r.entry, G_r.exit, G, last_cs,
         G.exit, init′, init″) then
13:        post_f ← post_f ∪ {G_r}
14:    if ReM[v] − post_f ≠ ∅ then
15:      ret ← false
16:      break
17:    else
18:      ReM[last_cs] ← ∅
19:      PostM[last_cs] ← post_f
20:      ret ← true
21:  until last_cs = v
22:  return ret
```

Hence, for showing post-commutativity, we first proceed to the last call-site in topological order in $G$ (on Line-3), and attempt to show post-commutativity for this call-site. This guarantees that the post-call segment will not contain any call-sites. Before showing commutativity of a call back to $G_r$ at $last_{cs}$ and the post-call segment, we first need to establish a sound context. For this, we perform a value analysis of $G$ from the beginning (i.e. $G.entry$) to $last_{cs}$ by calling SAnalysis in the RE mode (Line-8). This mode ensures that for call sites which have already been shown to be ECF (i.e. if their ReM mapping is empty), their transfer functions will be taken to be identity, while for other call-sites, SAnalysis would perform a fixpoint computation over all possible call-backs through a call to the procedure CallBackSummary (see Line-11 in Alg. 2). Thus, after the call to SAnalysis on Line-8, $IN1[last_{cs}]$ would contain an abstract state which encapsulates all reachable states at $last_{cs}$ in executions with arbitrary callbacks at call-sites which are not yet shown as ECF.

Next, we consider the effect of an arbitrary number of callbacks before $G_r$ at $last_{cs}$ itself, for which we call CallBackSummary with state $IN1[last_{cs}]$ (Line-9). We now perform commutativity check between $G_r$ and the post-call segment of $G$ after $last_{cs}$ under the context of the state $\hat{\sigma}_{re}$. For this, we again set up the appropriate abstract states $init'$ and $init''$ and then call the Commutes procedure. If Commutes returns *true*, then $G_r$ is added to $post_f$. After processing all the functions in ReM$[last_{cs}]$, if we find a non post-commuting function, then the return value is set to $false$ and

we break out of the `repeat ... until` loop. Otherwise, we have been successfully able to show that the call-site $last_{cs}$ is ECF, after which the loop continues processing all the call-sites (including $v$) in reverse topological order.

If all the call-sites (including $v$) are shown to be ECF, then PostC returns *true* (assigned at Line-5), which will be returned to the caller SAnalysis procedure instance. Note that this means that while SAnalysis is checking the ECF property for call-sites in the `CHK` mode, once it hits a call-site $v$ for which it cannot show neither absorption nor pre-commutativity for all call-backs from $v$, PostC will be called which will only check post-commutativity for all call-sites occurring topographically after $v$ in $G$. We formally establish the soundness of our analysis algorithms, as stated in the following theorem[1].

Theorem 4.1. *If for all callsites $v$ in $G$,* `ReM[v]` $= \phi$*, then for any state $\sigma_s$, for all executions $\tau$ such that $\tau$ begins in $G$ with state $\sigma_s$ and finishes in $\sigma_e$, there exists a call-back free execution $\tau_{cbf}$ which begins in $G$ with some start state $\sigma'_s$ and finishes in $\sigma_e$.*

The theorem states that if at the end of Algorithm 1, the `ReM` map of each call-site is empty, then for any execution with call-backs, there exists another execution without call-backs ending in the same state, thereby showing that the contract is ECF. We prove the above theorem using a modular verification approach, where we identify pre-conditions and post-conditions for each of Algorithms 1-5, which together establish the soundness proof.

## 5 Evaluation

We have implemented the proposed algorithms in a prototype tool named RAVEN and applied the tool on an extensive benchmark set of real-world smart contracts. RAVEN is written in Python, and takes as input a contract written in the Solidity programming language. It internally uses Slither [16] to parse and obtain CFGs of every function in the contract. For performing value analysis, RAVEN uses Apron [19] library (specifically apronpy, the python interface), choosing the polyhedral abstract domain.

We evaluate RAVEN over five labelled testsuites summarized in Table 1: **(i) FSC: Fabricated Smart Contracts**, developed by identifying both syntactic and semantic re-entrancy avoidance patterns gathered through an extensive case-study of real-world smart contracts, **(ii) MTC: Most Transacted Contracts** on Ethereum Blockchain as of October 3, 2023, collected by crawling through Etherscan, **(iii) SBS: Sailfish Benchmark Suite**, curated by Bose et al. [6] consisting of 750 real-world contracts, **(iv) RSD: Reentrancy Study Data**, the benchmark suite collected by Zheng et al. [46] containing 21,212 real-world contracts, and **(v) SBTS: SmartBugs TestSuite**, a benchmark suite collected from SmartBugs [15] containing 143 real-world contracts. Table 1 also contains the number of unsafe and safe contracts in each test-suite. Across all datasets, the lines of code (LoC) range from 6 to 4841 for safe contracts and from 14 to 658 for unsafe contracts, highlighting the diversity in contract sizes considered in our evaluation.

In SBS, we found a slight discrepancy with the reported numbers from the Sailfish paper [6]: instead of 26 unsafe contracts, we identified 24. Specifically, two contracts originally labeled as unsafe were actually reported as safe when analyzed using both the Sailfish tool and RAVEN. Similarly, in the RSD dataset, we discovered a variation in the reported ground truth. Instead of 41 unsafe contracts as reported in [46], we found 98 contracts to be unsafe. Our supplementary material (section E.1) provides evidence for this claim: among the 57 disputed contracts, four were actually annotated as reentrant in their source code but not included in the unsafe set by [46], while each of the remaining 53 contracts were exactly identical to some contract in the unsafe

[1]The complete proof is provided in the supplemental material, Section B

dataset (we provide evidence for this in the form of similarity scores computed by MOSS in the supplementary material[2]).

Table 1. Statistics for number of safe and unsafe contracts

| TestSuite | Safe (#P) | Unsafe(#N) |
|---|---|---|
| **FSC** | 0 | 12 |
| **MTC** | 6 | 25 |
| **SBS** | 24 | 726 |
| **RSD** | 98 | 21114 |
| **SBTS** | 31 | 112 |
| **Total** | 159 | 21989 |

Table 2. Comparison across Polyhedral, Octagon, and Interval domains

| | | MTC | SBS | SBTS | Avg. Time |
|---|---|---|---|---|---|
| **Interval** | **Safe** | 25 | 726 | 109 | 141.93 |
| | **Unsafe** | 6 | 24 | 32 | |
| **Octagon** | **Safe** | 23 | 683 | 99 | 72.38 |
| | **Unsafe** | 8 | 67 | 42 | |
| **Polyhedral** | **Safe** | 13 | 597 | 75 | 2.17 |
| | **Unsafe** | 18 | 153 | 66 | |

We compared RAVEN with 9 state-of-the-art tools as shown in Table 3. All our experiments were performed on a 64-bit CPU @ 2.30GHz with 64 GB RAM. For comparison with the existing tools, we have used their Docker images whenever available. We focus on four research questions:

**RQ1: Is RAVEN effective in detecting semantically embedded reentrancy avoidance patterns?** We address RQ1 over **FSC**, which is a collection of 13 synthetic, hand-crafted smart contracts that demonstrate various techniques that are commonly used for protecting against re-entrancy, such as implicit/explicit mutex, forced rollbacks, write-only post-call segments to make re-entrant calls harmless, etc. Over this synthetic test-suite, we found that Conkas, Oyente, and Osiris have relatively high accuracy (≈**84.62%**) compared to SliSE (≈**61.54%**), Sailfish (≈**53.84%**) and remaining tools (≈**23.08%**), while RAVEN was able to prove all the contracts as safe. This experiment demonstrates that many of the recently proposed state-of-the-art tools are actually not capable of reasoning about all semantically embedded re-entrancy avoidance patterns used in practice.

**RQ2: Is RAVEN effective in distinguishing between safe and unsafe real-world smart contracts?** We address RQ2, using the datasets **MTC**, **SBS**, **RSD**, and **SBTS** collected from different sources. Table 3 contains the results of applying RAVEN and the 9 other tools on these datasets. RAVEN consistently outperforms all state-of-the-art tools in detecting reentrancy vulnerabilities across all datasets, reflected in lower numbers of both false positives and false negatives. In particular, RAVEN detects the highest number of true negatives (21533/21977 ≈ 97.98%) and true positives (157/159 ≈ 98.74%) with minimal false positives (444/21977 ≈ 2.02%) and 0 false negatives across all the datasets. We also highlight the comparison with Sailfish and SliSe, two of the most recent state-of-the-art tools in re-entrancy detection. On the **SBS** dataset, which was curated by the Sailfish paper, RAVEN reports 0 false positives as opposed to 21 false positives by Sailfish. On the **RSD** dataset which was used in the SliSe paper, RAVEN reports a significantly lower number of false positives (441 vs. 1867) as compared to SliSe. Some of the older tools such as Mythril, Oyente, and Osiris further illustrate scalability limitations, with many unsupported contracts and frequent timeouts, in contrast to RAVEN's stable performance. Overall, RAVEN demonstrates clear advantages in accuracy, scalability, and reliability, marking it as the most effective and practical tool for reentrancy detection in smart contracts. We have also analyzed the false positives of RAVEN in the RSD dataset, and found that they mainly occurred due to over-approximations caused by the polyhedral model due to the presense of non-linear arithmetic in the smart contracts, use of widening operator, etc. More details regarding this can be found in Supplementary material section D.5 and D.6.

**RQ3: Which hyperproperties are more prevalent for avoiding re-entrancy in smart contracts?**:The effectiveness of RAVEN in terms of low false positives as compared to other tools

[2]Data available in Section E of the Supplementary material.

Table 3. RQ2: Effectiveness in detecting reentrancy (TP: True Positives, FP: False Positives, TN: True Negatives, FN: False Negatives, US: Unsupported/Timeout (7200 secs), Avg.: Average Time in secs, Worst: Worst Time in secs)

| Tool/TS | MTC | | | | | SBS | | | | | RSD | | | | | SBTS | | | | | Avg. | Worst |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | TP | FP | TN | FN | US | TP | FP | TN | FN | US | TP | FP | TN | FN | US | TP | FP | TN | FN | US | | |
| **Conkas**[40] | 3 | 15 | 10 | 3 | 0 | 24 | 598 | 123 | 2 | 3 | 95 | 15246 | 5815 | 3 | 53 | 31 | 78 | 34 | 0 | 0 | 25.90 | 6131.01 |
| **Oyente**[24] | 1 | 1 | 18 | 3 | 0 | 2 | 6 | 551 | 19 | 172 | 41 | 127 | 12864 | 53 | 8127 | 13 | 3 | 107 | 17 | 3 | 16.13 | 454.59 |
| **Slither**[16] | 3 | 8 | 6 | 0 | 14 | 25 | 288 | 435 | 1 | 1 | 98 | 5873 | 11032 | 0 | 4209 | 29 | 28 | 83 | 0 | 3 | 2.15 | 119.26 |
| **Osiris**[37] | 0 | 0 | 1 | 0 | 30 | 0 | 25 | 2 | 1 | 748 | 22 | 265 | 11016 | 48 | 9861 | 14 | 25 | 81 | 16 | 7 | 15.38 | 2862.34 |
| **Mythril**[26] | 2 | 7 | 9 | 2 | 11 | 11 | 348 | 127 | 9 | 255 | 201 | 10888 | 709 | 20 | 9392 | 10 | 78 | 14 | 2 | 39 | 3263.51 | 7187.11 |
| **Securify**[7] | 0 | 0 | 25 | 6 | 0 | 0 | 0 | 724 | 26 | 0 | 0 | 10 | 260 | 0 | 20942 | 27 | 15 | 80 | 3 | 18 | 119.97 | 7164.31 |
| **SmartCheck**[35] | 5 | 7 | 18 | 1 | 0 | 11 | 248 | 476 | 15 | 0 | 97 | 4544 | 16570 | 1 | 0 | 29 | 27 | 85 | 2 | 0 | 10.86 | 84.84 |
| **Sailfish**[6] | 3 | 5 | 15 | 2 | 6 | 24 | 21 | 697 | 2 | 6 | 98 | 1084 | 13651 | 0 | 6379 | 28 | 2 | 106 | 1 | 6 | 65.68 | 6915.29 |
| **SliSE**[41] | 4 | 7 | 18 | 2 | 0 | 13 | 86 | 633 | 13 | 5 | 98 | 1867 | 19247 | 0 | 0 | 31 | 5 | 107 | 0 | 0 | 128.87 | 6875.06 |
| **RAVEN** | 6 | 0 | 25 | 0 | 0 | 24 | 0 | 726 | 0 | 0 | 98 | 441 | 20673 | 0 | 0 | 29 | 3 | 109 | 0 | 2 | 141.93 | 7101.67 |

can be explained by the semantic hyper-properties that it uses for establishing the ECF property as opposed to other tools. For every contract that is proven to be safe by RAVEN, we find out which and how many hyper-properties are used for showing its call-sites to be ECF. This directly corresponds to contents of the maps AbsM, PreM and PostM as maintained by Algorithms 3, 4, 5 in Section 4. For example, for every safe contract, we ask whether there is a callsite v for which AbsM[v] was non-empty, which corresponds to the fact that a call-back was absorbed at v. In Fig. 7, for every individual hyper-property and its combinations, we show the percentage of smart-contracts which are proven to be safe only using that property or the combination across all our datasets. Note that we club together pre and post commutativity into a single property. As far as we are aware, this is the first such software engineering study which shows the distribution and prevalence of different kinds of re-entrancy countermeasures employed real world smart contracts.

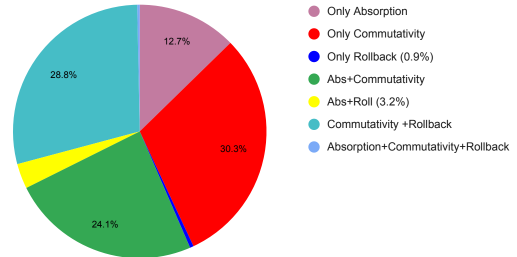


Fig. 7. RQ3: Distribution of Hyperproperties

We find that majority of safe smart contracts actually use multiple hyper-properties, with many of them also employing all three properties. A classic example of this would be the simplified `Defi` Contract in Fig. 2 discussed in §2, which employs both absorption and commutativity. This demonstrates that while contracts use lock/mutex variables (such as the variable `te` in the `Defi` contract) to prevent re-entry during callbacks (which would be identified by the absorption property in our analysis), contracts also take advantage of commutativity of arithmetic operations to allow re-entry during call-backs. We also find that the rollback property employed for example, by the Synthetix contract, is not very prevalent, with a very small fraction of contracts employing this property. This study also highlights the importance of reasoning about multiple hyper-properties to verify safety against re-entrancy, as using pattern matching techniques or even single property detection would fail to capture the nuanced interplay of behaviours required for guaranteeing safety against re-entrancy.

**RQ4: Does the choice of abstract domain affect RAVEN's performance?** To answer this question, we instantiate RAVEN with the interval and octagon domain for performing value analysis. The interval domain cannot express relational information between variables, while the Octagon domain can only express simpler relations involving at most 2 variables ($\pm X_i \pm Y_i \leq \beta$). Table 2 shows a detailed comparison of the performance of RAVEN using the Polyhedral, Octagon, and Interval domains on different datasets (we exclude the RSD data-set for this experiment). For each

dataset, Table 2 shows that number of contracts reported as safe/unsafe using the three different abstract domains. From the results, it is evident that the Polyhedral domain consistently classifies a higher number of contracts as safe compared to both the Octagon and Interval domains across all datasets. Theoretically, the polyhedral domain can express more complex relational information, and empirically, this is also reflected by its precise results. We do note that even with octagon and interval domain, RAVEN is able to correctly classify many contracts as safe. Morever, the average analysis time is significantly lower when using the interval domain as compared to the polyhedral domain, thus illustrating the precision-analysis time tradeoff.

**Limitations:** We note that RAVEN can only be applied when all call-sites are located outside of loops. While loops themselves are generally allowed, any call-sites within loops prevent processing in a topological order, which is essential for maintaining soundness of our analysis algorithms. Since topological ordering is not possible when call-sites appear inside loops, such cases fall outside the scope of this work.

Nonetheless, in our evaluation we identified 124 contracts containing call-sites within loops. Rather than excluding them, we applied bounded unrolling and found that two iterations were sufficient to expose reentrancy vulnerabilities in all cases. This unrolling strategy can only demonstrate the presence of a vulnerability, not guarantee its absence. Moreover, external calls inside loops are generally discouraged in practice due to their high gas cost.

Further, our definition of a re-entrancy vulnerability is based on the classical notion of effectively callback free (ECF) executions, which is a per-contract guarantee. As such RAVEN would miss out on the newer class of vulnerabilities such as Read-only Reentrancy (ROR) [4] which is an attack involving multiple contracts, where each contract considered in isolation would be ECF, but because there is some common shared state (in say a token contract) which is read and written by different contracts, and because the re-entrancy protection logic is specific to each contract, an attacker can orchestrate a re-entrancy attack involving callbacks to a different contract than the entry contract, manipulating the shared contract's state in an inconsistent manner. While it is possible to merge together different inter-dependent contracts into a single contract and apply RAVEN on this mega-contract to detect ROR attacks, further investigation is needed to assess the scalability of such an approach, which we leave as future work [45].

## 6 Related Work and Conclusion

Re-entrancy in smart contracts is a well-studied problem, with multiple approaches proposed by both academia and industry for detection and mitigation. In response to the infamous DAO attack [13], a long line of works [7, 9, 24, 25, 33, 38, 39, 41, 44] looked at detecting re-entrancy attacks in smart contracts using known attack patterns, with the later works introducing more sophisticated patterns involving multiple contracts and chains of transactions. These works rely on either symbolic execution using SMT solvers or static analysis with simple abstract domains, such as the constant domain, to determine the feasibility of attacks. Since they rely on known patterns rather than a more general definition of re-entrancy, they can both miss actual bugs and also report a significant number of false positives due to over-approximation caused by the usage of SMT solvers or simple static analysis. Moreover, Liu et. al [23] identify commutativity by checking for a feasible initial state that can lead to a common end state; however, this approach does not consider other properties such as absorption or transaction rollback, and also does not use contextual reasoning for the commutativity check.

Fuzzing-based approaches [10, 20, 22, 27, 36, 42, 43, 45] have also been successfully applied for the detection of re-entrancy attacks, with many of these approaches using sophisticated static and dynamic analysis to increase coverage. Although fuzzing techniques would detect true vulnerabilities with a very low number of false positives, they are not sound and cannot prove the absence of

vulnerabilities, which is the primary focus of our approach. There are also dynamic analysis-based approaches to both detect and mitigate re-entrancy attacks, either based on analyzing the actual execution traces [1, 17] or augmenting the Ethereum runtime [32] to detect and prevent attacks.

Verifying the functional correctness of smart contracts has also received considerable attention. Typically these approaches [5, 8, 17, 18, 21, 29] require expressive specifications from the programmers, proof artifacts such as loop invariants, pre and post-conditions, or even call-back invariants [17], and hence are not fully automated. Further, many of these approaches actually assume the contract to be non-reentrant, and prove more complex properties. Hence, these approaches can be applied after proving non-re-entrancy using our technique. Further, as we demonstrate in this paper, complex relational invariants can be effectively inferred by performing value analysis using an expressive domain such as the polyhedral domain, which may be of further use for verifying functional correctness.

To conclude, in this work, we propose a sound and precise static analysis based approach named RAVEN for detecting and proving re-entrancy safety of smart contracts. RAVEN utilizes the expressive polyhedral domain to automatically infer precise relational invariants linking program variables, which it then uses to show program hyper-properties such as absorption and commutativity. It overcomes the limitations of previous approaches which rely heavily on SMT solvers to precisely encode semantics of programs, but lack the necessary contextual information which leads to solvers assigning infeasible values, thus resulting in higher number of false positives. Our extensive experimental evaluation demonstrates the feasibility of our approach, with notably lower rate of false positives compared to previous state-of-the-art approaches, while maintaining similar analysis times. Since smart contracts are primarily used for financial tasks and hence use a lot of arithmetic operations, numerical abstract domains such as the polyhedral domain are ideally suited for expressing and inferring complex invariants, and we have leveraged this observation to significantly improve the precision of static re-entrancy verification.

## 7 Data Availability Statement

The supplementary material contains details of the algorithms missing from the paper, a proof of soundness, and supporting experimental data. The complete tool, along with the dataset, is available in our artifact repository Figshare [30] as a Docker image.

## References


[1] Sefa Akca, Ajitha Rajan, and Chao Peng. 2019. SolAnalyser: A framework for analysing and testing smart contracts. In *2019 26th Asia-Pacific software engineering conference (APSEC)*. IEEE, 482–489. doi:10.1109/APSEC48747.2019.00071

[2] Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. 2020. Taming callbacks for smart contract modularity. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 209 (Nov. 2020), 30 pages. doi:10.1145/3428277

[3] Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. 2022. Relaxed effective callback freedom: a parametric correctness condition for sequential modules with callbacks. *IEEE Transactions on Dependable and Secure Computing* 20, 3 (2022), 2256–2273. doi:10.1109/TDSC.2022.3178836

[4] Rob Behnke. 2023. Blockchai Security. https://www.halborn.com/blog/post/what-is-read-only-reentrancy.

[5] Sidi Mohamed Beillahi, Gabriela Ciocarlie, Michael Emmi, and Constantin Enea. 2020. Behavioral simulation for smart contracts. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 470–486. doi:10.1145/3385412.3386022

[6] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2022. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 161–178. doi:10.1109/SP46214.2022.9833721

[7] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on*

*Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 454–469. doi:10.1145/3385412.3385990

[8] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981* (2018).

[9] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2021. Defectchecker: Automated smart contract defect detection by analyzing evm bytecode. *IEEE Transactions on Software Engineering* 48, 7 (2021), 2189–2207. doi:10.1109/TSE.2021.3054928

[10] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 227–239. doi:10.1109/ASE51524.2021.9678888

[11] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 238–252.

[12] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 84–96. doi:10.1145/512950.512973

[13] DAO Hack 2016. CRITICALUPDATERe:DAOVulnerability

[14] Defi 2019-2024. Chains DefiLlama Chains. https://defillama.com

[15] Monika di Angelo, Thomas Durieux, João F. Ferreira, and Gernot Salzer. 2023. SmartBugs 2.0: An Execution Framework for Weakness Detection in Ethereum Smart Contracts. In *38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2102–2105. doi:10.1109/ASE56229.2023.00060

[16] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15. doi:10.1109/WETSEB.2019.00008

[17] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2017. Online detection of effectively callback free objects with applications to smart contracts. *Proc. ACM Program. Lang.* 2, POPL, Article 48 (Dec. 2017), 28 pages. doi:10.1145/3158136

[18] Ákos Hajdu and Dejan Jovanović. 2020. solc-verify: A modular verifier for solidity smart contracts. In *Verified Software. Theories, Tools, and Experiments: 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13–14, 2019, Revised Selected Papers 11*. Springer, 161–179. doi:10.1007/978-3-030-41600-3_11

[19] Bertrand Jeannet and Antoine Miné. 2009. Apron: A library of numerical abstract domains for static analysis. In *International Conference on Computer Aided Verification*. Springer, 661–667. doi:10.1007/978-3-642-02658-4_52

[20] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 259–269. doi:10.1145/3238147.3238177

[21] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts. In *Ndss*. 1–12.

[22] Bixin Li, Zhenyu Pan, and Tianyuan Hu. 2022. Redefender: detecting reentrancy vulnerabilities in smart contracts automatically. *IEEE Transactions on Reliability* 71, 2 (2022), 984–999. doi:10.1109/TR.2022.3161634

[23] Yinxi Liu, Wei Meng, and Yinqian Zhang. 2025. Detecting Smart Contract State-Inconsistency Bugs via Flow Divergence and Multiplex Symbolic Execution. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 22–43. doi:10.1145/3715712

[24] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. doi:10.1145/2976749.2978309

[25] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1186–1189. doi:10.1109/ASE.2019.00133

[26] Mythril 2021. https://github.com/Consensys/mythril

[27] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sFuzz: an efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 778–788. doi:10.1145/3377811.3380334

[28] OpenZeppelin. 2021. OpenZeppelin Contracts. https://docs.openzeppelin.com/contracts/4.x/erc20. Accessed: September 9, 2025.

[29] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*. IEEE, 1661–1677. doi:10.1109/SP40000.2020.00024

[30] RAVEN. 2026. Artifact for "Verifying Smart Contract Security against Re-entrancy Attacks through Relational Value Analysis ". doi:10.6084/m9.figshare.30095356.v5

[31] Reentrancy Avoidance 2016. https://scsfg.io/hackers/reentrancy/

[32] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2018. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv preprint arXiv:1812.05934* (2018).

[33] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 2020. eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 621–640. doi:10.1145/3372297.3417250

[34] SWC Registry [n. d.]. Smart Contract Weakness Classification. https://swcregistry.io/

[35] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*. 9–16. doi:10.1145/3194113.3194115

[36] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 103–119. doi:10.1109/EuroSP51992.2021.00018

[37] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th annual computer security applications conference*. 664–676. doi:10.1145/3274694.3274737

[38] Christof Ferreira Torres, Mathis Steichen, and Radu State. 2019. The art of the scam: demystifying honeypots in ethereum smart contracts. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 1591–1607.

[39] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 67–82. doi:10.1145/3243734.3243780

[40] Veloso, N. 2021. https://fenix.tecnico.ulisboa.pt/downloadFile/1689244997262417/94080-Nuno-Veloso_resumo.pdf

[41] Zexu Wang, Jiachi Chen, Yanlin Wang, Yu Zhang, Weizhe Zhang, and Zibin Zheng. 2024. Efficiently Detecting Reentrancy Vulnerabilities in Complex Smart Contracts. *Proc. ACM Softw. Eng.* 1, FSE, Article 8 (July 2024), 21 pages. doi:10.1145/3643734

[42] Valentin Wüstholz and Maria Christakis. 2020. Harvey: a greybox fuzzer for smart contracts *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1398–1409. doi:10.1145/3368089.3417064

[43] Valentin Wüstholz and Maria Christakis. 2020. Targeted greybox fuzzing with static lookahead analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 789–800. doi:10.1145/3377811.3380388

[44] Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. 2020. 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*. 1029–1040. doi:10.1145/3324884.3416553

[45] Jingwen Zhang, Zibin Zheng, Yuhong Nan, Mingxi Ye, Kaiwen Ning, Yu Zhang, and Weizhe Zhang. 2024. SmartReco: Detecting Read-Only Reentrancy via Fine-Grained Cross-DApp Analysis. *arXiv preprint arXiv:2409.18468* (2024). doi:10.48550/arXiv.2409.18468

[46] Zibin Zheng, Neng Zhang, Jianzhong Su, Zhijie Zhong, Mingxi Ye, and Jiachi Chen. 2023. Turn the rudder: A beacon of reentrancy detection for smart contracts on ethereum. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 295–306. doi:10.1109/ICSE48619.2023.00036

Received 2025-09-12; accepted 2025-12-22