Noname manuscript No.
(will be inserted by the editor)

Automated Construction of Predicate Abstractions for Smart Contract Validation

Javier Godoy[1], Eden Torres[1], Juan P. Galeotti[1,5], Diego Garbervetsky[1,2], Sebastian Uchitel[1,3,4]

[1] Departamento de Computación, UBA, ICC/CONICET, Argentina
[2] Department of Mathematics, Guangdong Technion-Israel Institute of Technology, China
[3] Department of Computing, Imperial College London, UK
[4] Departamento de Ingeniería, Universidad de San Andrés, Argentina
[5] School of Economics, Innovation and Technology, Kristiania University College, Norway

Abstract Verification and validation of smart contracts is crucial, as these programs administer valuable assets. Indeed, there is a proliferation of companies, specialized in smart contract audits that has become a key component of the blockchain ecosystem. Current verification and validation practice is heavily reliant on the insight and experience of auditors. However, there is an increasing rich toolbox that supports their work.

In this paper, we show how predicate abstraction can be leveraged to construct models that support smart contract auditing. These models provide auditors with a systematic approach to analyzing and validating the behavior of smart contracts at the function-call level. By proposing predicates, auditors can generate alternative perspectives on a contract's behavior, refine transitions to investigate specific scenarios, and incorporate transient states into the abstraction to uncover reentrancy vulnerabilities. We formally define predicate abstractions for smart contracts and introduce default predicates to support initial exploration. We also present PASCo, a tool that automatically builds an abstract finite state automaton from a set of predicates and a smart contract implementation. We evaluate PASCo on established benchmarks and with an experienced smart contract auditor. Furthermore, we compare its effectiveness in detecting reentrancy vulnerabilities (measured in terms of precision and recall) against 14 state-of-the-art Solidity vulnerability detectors.

Key words smart contracts, predicate abstraction, software validation

1 Introduction

Blockchains support decentralized, transparent and traceable transactions on a network [1]. Although first decentralization was of transaction data only, with the Ethereum blockchain the possibility of decentralized execution of software emerged [2]. These programs, referred to as *smart contracts*, are stored in the blockchain and their execution does not require trusting one particular party as the blockchain's consensus protocol ensures that smart contracts effects are recorded securely by any node in the network. Crucially, the blockchain only guarantees the correct execution of smart contract bytecode. However, as with any software, the code may have defects that stem from an incorrect definition of what the contract was supposed to do or an incorrect implementation of the intended behavior of the contract.

Validation and verification are activities that aim to increase confidence that a software system correctly solves the intended problem. Validation focuses on the question of whether the right system is being built, while verification focuses on whether the system is being built right [3]. Although these are important activities in software engineering in general, in the domain of smart contracts, these activities play a crucial role due to two factors. On one hand smart contracts administer an enormous amount of assets; consequently defects can lead to very significant losses. A notorious defect in the DAO smart contract [4] allowed attackers to steal 60 MUSD in 2016 [5]. On the other hand smart contracts are immutable, meaning that it is not possible to modify them once they are deployed. Immutability increases security and transparency, but also prevents fixing smart contracts defects once deployed on the blockchain. Strictly speaking there are design mechanisms –i.e., proxy patterns – to support some types of changes in a smart contract behavior, but these are limited and can increase code complexity significantly.

Indeed, it is a common industrial practice to have smart contracts reviewed by third-party specialized security audit companies; in many cases more than one independent company is hired. These audits involve understanding and validating the smart contract requirements, with a strong focus on security, and verifying that

the smart contract implementation is correct with respect to these requirements.

Auditing is a human intensive activity that requires engineers with a good understanding of the semantics of smart contract programming languages, the workings of the underlying blockchain (different blockchains have different, subtle, yet important differences) and of the problem domain. The latter is of significance as requirements for smart contracts are rarely described rigorously, and most commonly are only partial and informally communicated. Auditors increasingly rely on tools (e.g., [6,7,8,9,10,11]) to reason about smart contract behavior as part of their validation and verification activities. The result is increased pre-deployment confidence that the smart contract indeed solves the problem at hand and is free of defects, i.e., the domain specific requirements are met and domain independent (e.g., overflow, re-entrancy [12]) properties are satisfied.

The focus of this paper is on validation of smart contracts. In other words, the alignment of the contract's behavior with the mental model that the auditors have (or are attempting to form) of the contract's intended behavior based. Validation efforts largely rely on code inspections and a good understanding of the requirements by the auditor [13]. Recent work [14] concludes that over 80% of exploitable bugs remain undetected by existing tools due the lack of describing and checking domain-specific properties. However, as noted in [13], *"there is likely a large potential payoff in making more effective use of automatic static and dynamic analyses to detect the worst problems in smart contracts"*. In this spirit, we propose an automated abstraction technique that supports validation while not requiring requirements formalization. Instead it relies upon the ability of auditors to spot inconsistencies by contrasting their understanding (i.e., mental model) of the intended contract behavior and an automaton that is formally an abstract representation of the smart contracts behavior. The abstraction can also be used at development time by programmers to find defects and improve documentation.

Concretely, we propose using predicate abstraction to build a finite labelled transition system (LTS) from a smart contract. The LTS exhibits an over-approximation of the valid sequences of function calls that can be applied to the contract. The choice of predicates to use for the abstraction is up to the auditor, different abstraction predicates exhibit different aspects of the contract behavior. We believe that through the inclusion of different abstraction predicates, auditors (or other stakeholders) can explore the behavior of a smart contract, identifying defects and gaining confidence in its correspondence to the intended behavior.

We formally define predicate abstractions for smart contracts, propose default abstraction predicates that can help auditors start exploring a contract's behavior before proposing additional or alternative abstraction predicates. One such default is based on the require clauses typically used by smart contract programming languages such as Solidity [15] and Rust for Near [16] contracts. Such clauses typically capture the precondition of each contract function and allow building compact abstractions that quotient the contract's state space (all the possible states in which a contract implementation may be in) into a small set of states that map well to developer mental models [17]. The other default predicate abstraction is based on the value of enum-type state variables typically found in contracts and that also typically capture relevant abstract problem domain states. We describe a tool that we have developed to automatically build contract abstractions from predicate abstractions and smart contract implementations. We report on the results of using the automated constructed abstractions on existing benchmarks and with an auditor.

This paper is an extension of [18] that introduces two significant contributions. First, the tool presented in this paper (PASCo) automates the construction of predicate abstractions reasoning, directly analyzing smart contract implementations written in Solidity using the VeriSol [8] software model checker. In contrast, [18] describes a prototype that requires the manual modelling of smart contracts in Alloy [19]. Second, this paper shows how to support building predicate abstractions that exhibit behavior resulting from cross-contract calls. This is a common feature used in smart contracts and which is the source of reentrancy exploits, some of which have caused millions of dollars in losses [5,20,21].

Summarizing, the contributions of this paper are: (*i*) A novel validation approach for smart contracts using labelled transition systems that abstract contract behavior at the function call level using user provided abstraction predicates. (*ii*) Default abstraction predicates, based on require clauses and enum-type state variables, extracted from smart contract code that provide rich initial models for validating contract behavior. These initial models can then be further refined or adapted by adding user-defined predicates. (*iii*) Two studies over 16 and 8 subjects each that provide evidence towards the potential of predicate abstractions for supporting smart contract validation. (*iv*) A study of 15 subjects analyzing reentrancy detection by directly examining cross-contract calls within predicate abstractions, and comparing its precision and recall against 14 state-of-the-art Solidity vulnerability detectors. (*v*) A fully automated, publicly available tool that builds predicate abstractions from smart contracts written in Solidity and a combination of default and user provided abstraction predicates, as well as a mode for analyzing reentrancy calls.

The remainder of this paper is organized as follows. In Section 2, we introduce the basics on smart contracts and predicate abstraction. In Section 3, we first define predicate abstractions for smart contracts and introduce abstraction predicates which we consider particularly useful for starting validation efforts, we then exemplify how predicate abstractions can support smart

contract validation. In Section 4, we exemplify how predicate abstractions can support smart contract validation, and extend examples with predicate and transition refinement. In Section 5, we explore how predicate abstractions can be useful to detect reentracy attacks. In Section 6, we report tool details and how we use this to automatically construct the models. In Section 7, we discuss the evaluation of our approach, followed in Section 8 by a discussion of threats to validity. In Section 9, we discuss related work and finally in Section 10 we conclude and present future work.

2 Background

2.1 Solidity Smart Contracts

Succinctly, a blockchain is a distributed ledger, mapping accounts to quantities. These quantities are typically in cryptocurrency, such as Bitcoin or Ether. The blockchain stores the history of transactions between these accounts and is publicly available. The balance of each account can be computed by traversing the blockchain, a sequence of replicated blocks in a peer-to-peer network that implements the ledger.

Ethereum is a blockchain that provides, beyond the standard infrastructure, a distributed program state, participants can modify the program state by executing smart contracts. A *smart contract* is a procedural program that is stored in a blockchain address in the same way a standard account is. This means that smart contracts are also accounts that can have funds transferred to and from. In the same way as transactions between accounts in the blockchain cannot be reversed, a deployed program cannot be modified.

Smart contract functions are called from accounts (addresses in the blockchain) that hold funds. In order to be executed, the smart contract consumes an amount of *gas* (i.e., a quantity of the crypto currency). The amount of *gas* that the smart contract execution costs will depend on the specific actions the smart contract performs.

Ethereum provides a virtual machine, known as the Ethereum Virtual Machine (EVM), to execute the transactions between accounts and update the internal state of the blockchain. Additionally, a programming language called *Solidity* is provided to allow programmers to write how the state of the blockchain has to be updated.

Running Example. Throughout the paper we will use the smart contract for a crowdfunding initiative. The contract is taken from [10] and is listed in Figure 1.

The smart contract is composed of a local state (i.e., variables and their values) defined in the first few lines of the listing, and functions that are called by any participant (i.e., another address) in the blockchain.

A smart contract has state variables which will be stored in the blockchain. The crowdfunding contract consists of a variable `owner` that will be set with the address where the collected funds will be deposited once the crowdfunding effort has ended. Variable `goal` contains the amount that the owner is expecting to raise. The deadline for raising the goal is stored in `max_block`. Note that rather than a timestamp, the deadline is implemented with a block number that refers to the instant in which the blockchain is extended to include block number `max_block`. This is a common mechanism to establish contract expiration because it avoids attacks where the local time is maliciously altered [22]. The contract holds a mapping (`backers`) that stores how much different contributors have committed to the crowdfunding effort. Mappings in Solidity are initialised so every possible key exists and is mapped to a value whose byte-representation is all zeros [23]. Finally, `funded` is a boolean indicating if the initiative reached the funding goal and funds have been withdrawn by the owner.

The Crowdfunding contract has 4 functions: the *constructor()*, *Donate()*, *GetFunds()*, and *Claim()*.

The `constructor` (that is executed only when the smart contract is deployed onto the blockchain) simply stores the arguments into state variables. That is, it initialises the contract with the goal, owner and a deadline for donations. The `payable` modifier for parameter `_owner` declares the address as one to which transfers can be done. Once deployed onto the blockchain, third parties can `Donate` funds. Once the deadline has passed, if the funding goal has been achieved, the owner can `Get(all donated)Funds`. Otherwise, when the deadline has passed donators can `Claim` back their donations.

Note that in Solidity all function calls include two implicit arguments `msg` and `block`. These arguments contain the address from which the contract function call was made (`msg.sender`), the amount of Ether committed by the sender (`msg.value`) and the current block number (`block.number`). Thus, for instance, a call to `Donate()`, that has no explicit parameters, will include the donation address and the donation amount.

Indeed, the `Donate()` function includes two `require` clauses that predicate over the impicit arguments. When the function is invoked, if these clauses are not satisfied, the function fails and a small amount of gas is consumed. The first clause requires that the current blockchain block number (i.e., `block.number`) be less than the value stored in `max_block` –in other words, that the donation deadline has not passed. The second clause requires the caller to have not donated previously. Note that the `payable` modifier is used in the `Donate()` function declaration too. Recall that the smart contract has an address of its own, and the `payable` modifier declares that function `Donate()` will be receiving Ether from the caller. After a successful termination of `Donate()`, the amount committed by the caller (`msg.value`) will be transferred from the caller (`msg.sender`) to the smart contract address (`address(this)`).

```
1 pragma solidity ^0.5.0;
2 contract Crowdfunding {
3  address payable owner;
4  uint max_block;
5  uint goal;
6  mapping(address => uint) backers;
7  bool funded = false;
8
9  constructor(address payable _owner, uint
       _max_block, uint _goal) public {
10    owner = _owner;
11    max_block = _max_block;
12    goal = _goal;
13  }
14  function Donate() public payable {
15    require(max_block > block.number);
16    require(backers[msg.sender] == 0);
17    backers[msg.sender] = msg.value;
18  }
19  function GetFunds() public {
20    require(max_block < block.number);
21    require(msg.sender == owner);
22    uint val = address(this).balance;
23    require(goal <= val);
24    funded = true;
25    owner.call.value(val)("");
26  }
27  function Claim() public {
28    require(max_block < block.number);
29    require(backers[msg.sender] > 0 && !funded);
30    require(goal > address(this).balance);
31    uint val = backers[msg.sender];
32    msg.sender.call.value(val)("");
33    backers[msg.sender] = 0;
34  }
35 }
```

Fig. 1: A smart contract for a crowd-funding initiative

Function `GetFunds()` includes three require clauses. The first requires the funding period to have ended. The second stipulates that only the owner of the contract may call the function, while the third ensures that the funding goal has been achieved. Only under these conditions can the function be fully executed resulting in a transfer of the totality of the funds from the smart contract to the owner. Note the use of contract state variable `address(this).balance` to check the collected funds in the smart contract. Also note the use of `call.value()` to transfer funds.

Finally, function `Claim()` also requires the crowd-funding effort to have ended (see first require clause) and also ended unsuccessfully –less donations than the goal (see third clause). In addition, the call to the function must come from an address that has, according to the `backers` mapping donated funds. Before transferring the funds to the caller of the function, the mapping is updated accordingly.

Note that this smart contract uses a low-level function (line 32) to transfer ether instead of the function `address.transfer(x)`. Although `transfer` function is considered reentrancy-safe, the low-level `call.value()` function is recommended over using transfer/send functions to better support future Ethereum implementation upgrades [24].

2.2 Predicate Abstractions

In program verification, an abstraction is typically used to refer to a mapping of concrete states of a program to an abstract state. Predicate abstraction refers to the fact that the mapping is derived from a set of given predicates: all concrete states that satisfy the same predicates are mapped to the same abstract state [25]. Transitions between states represent state changes due to the execution of program code. Sometimes each transition can represent the execution of one line of code; however, a coarser granularity can be used making transitions represent code blocks or procedures for example.

2.3 Enabledness Preserving Abstractions

In [26], De Caso et al. propose a particular kind of predicate abstraction to support object protocol validation and inspires the work presented herein. Predicates used in [26] correspond to the pre-conditions of all object protocol actions, whilst transition labels correspond to protocol action names. This results in an abstract state machine that effectively quotients the concrete state space of the protocol by the set of actions that states enable. The name enabledness preserving abstraction (EPA) refers to the fact that the characteristic that each abstract state preserved is the set of actions that are enabled (and consequently, the set that are not enabled).

EPAs can be built automatically from code [27,26] or from specifications [28,29] and have been used to support behavior validation [27,26,28] and verification [17].

2.4 Labelled Transition Systems (LTS)

We use labelled transition systems (LTS) to describe the semantics of smart contracts and smart contract abstractions. An LTS is a tuple $\langle \sigma, S, S_0, \delta \rangle$ where σ is the set of transition labels, S a set of states, S_0, a set of initial states where $S_0 \subseteq S$ and $\delta \subseteq S \times \sigma \times S$ is the set of transitions of the LTS.

3 Smart Contract Predicate Abstractions

In this section we first explain predicate abstractions for smart contracts. In the next section we run through an example of how auditors might use these abstractions for validation.

Our approach to abstracting the behavior of smart contracts is at the function call level. That is, we aim for models that provide a black box view in which internal computation of the smart contract functions is abstracted. The focus is on what functions can be successfully called and how the contract state (and consequently the enabled functions) change. In other words, the observable behavior of the contract.

At this level of granularity the semantics of a smart contract can be thought of similarly to that of an object protocol. A smart contract state is defined by the values of the internal variables and transitions are the functions calls that change its state. However, an important difference is that the global state of the blockchain is particularly relevant in a smart contract. For clarity, when we refer to the state of the contract, we will also include the global state. This means, for example, that `balance`, which maps addresses to cryptocurrency amounts, and `block.number`, which holds the current blockchain block number, will be both considered part of the contract state. Another important difference is that smart contract function call includes an implicit argument `msg`, which provides information such as the caller (`msg.sender`) and the amount of cryptocurrency sent with the call (`msg.value`). For clarity, we include `msg` when we refer to the parameters of a smart contract function.

3.1 Concrete LTS for Smart Contracts

We define the semantics of smart contracts, at this black box level of granularity, as an LTS $\langle \Sigma, S, S_0, \delta_c \rangle$. The states in S represent the different values the contract can have along with the current state of the blockchain. Subset $S_0 \subseteq S$ contains all possible initial states at the moment of contract deployment. Labels in Σ are all of the possible function calls F (names plus actual parameter values) and an additional label τ representing changes in the blockchain that are uncontrollable to the contract (e.g., a transfer made outside the contract, the addition of a new block to the blockchain due to the progress of time, etc.). The set of transitions $\delta_c \subseteq S \times \sigma \times S$ models all possible successful terminations of function calls or changes in the blockchain. The existence of an outgoing transition labeled with a specific function name f and parameter values $p_1, \ldots, p_n$ from a state c to a state c' (i.e., $\langle c, f(p_1, \ldots, p_n), c' \rangle \in \delta_c$) represents the fact a call to $f(p_1, \ldots, p_n)$ on the smart contract when in state c can terminate successfully and leave the smart contract in state c'. The existence of an outgoing transition labelled τ from c to c' (i.e., $\langle c, \tau, c' \rangle \in \delta_c$) represents the fact that the blockchain has changed and that the state of the contract has changed from c to c'.

Examples of concrete states for the `CrowdFunding` contract are depicted in Figure 2. The first state indicates that the address of the smart contract is `Addr#2`, the contract owner is `Addr#1`, the donations deadline is 10, the goal to fund 1000 and no donations have been received. The state also shows the balance of three accounts on the blockchain and the current blockchain block number. Of course, this is for illustration purposes only, as the actual state of the blockchain is in practice much larger. The second concrete state is exactly as the first except that a donation of 1 has been received from `Addr#3`.

```
address=Addr#2
owner=Addr#1, max_block=10, goal=1000
backers=[], funded=false
Global.balance=
  [Addr#1->1,Addr#3->2,Addr#4->3]
Global.block.number=5
```

```
address=Addr#2
owner=Addr#1, max_block=10, goal=1000
backers=[Addr#3->1], funded=false
Global.balance=
  [Addr#1->1,Addr#3->2,Addr#4->3]
Global.block.number=5
```

Fig. 2: Two concrete Crowdfunding contract states.

For the first concrete state, a call from any sender to function `Donate()` will pass both require clauses. Indeed, the concrete LTS for the crowdsourcing contract will include a transition from the first state to the second, modelling the execution of a call to `Donate()` with `msg.sender = Addr#3` and `msg.value = 1`

Note that for both concrete states, there are no parameter values such that `Claim()` or `GetFunds()` can be executed successfully. This is due to the fact that `max_block` is greater than expression `block.number`. Consequently, the concrete LTS for the contract will not have outgoing transitions from these states labelled with calls to `Claim()` or `GetFunds()`.

3.2 Abstract LTS for Smart Contracts

To define an abstract LTS for a smart contract with a set of public function names F we require i) a set of boolean predicates P over the state of a smart contract and the blockchain, ii) an LTS $\langle \Sigma, S, S_0, \delta_c \rangle$ describing the behavior of the smart contract, and iii) a subset of relevant unobservable transformations $\delta_\tau \subseteq \{\langle c, \tau, c' \rangle \cdot \langle c, \tau, c' \rangle \in \delta_c\}$. We shall, for convience, sometimes refer to δ_τ as a predicate over transitions in δ_c.

We define an abstract LTS as a tuple $\langle F \cup \{\tau\}, Q, P_0, \delta_a \rangle$. Each state of the abstract LTS is defined as a set of predicates (i.e., $s \subseteq P$). A state s is an abstract state that is mapped to all concrete states of the concrete smart contract LTS that satisfy the predicates in s. We will abuse notation and write that $c \in s$ if the concrete state $c \in S$ satisfies the predicates in s. We denote Q as the superset of all abstract states 2^P. A transition labelled with a function name $f \in F$ exists between two abstract states s and s' if there is a concrete state $c \in s$ and parameters $p_1, \ldots, p_n$ for f such that when $f(p_1, \ldots, p_n)$ is called on c, the call terminates successfully and a concrete state in s' is reached (i.e., $\langle s, f, s' \rangle \in \delta_a \iff \exists c \in s, c' \in s', p_1, \ldots, pn \cdot \langle c, f(p_1, \ldots, p_n), c' \rangle \in \delta_c$). A transition $\langle s, \tau, s' \rangle \in \delta_a$ exists if there are concrete states $c \in s$ and $c' \in s'$ such that $\langle c, \tau, c' \rangle \in \delta_\tau$. The set $P_0 \subseteq P$ is the set of initial states of the abstraction. We

require that if $s \in P_0$ then there is a $c \in s$ that is an initial state of the smart contract (i.e., $c \in S_0$).

To show an example of an abstract LTS for the Crowdfunding contract, we must choose a set of abstraction predicates. We do this in the next subsection.

3.3 Default Predicate Abstractions for Smart Contracts

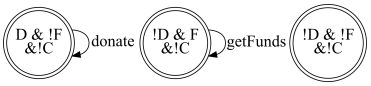
We introduce two predicate abstractions that serve as useful foundations for smart contract validation: *enabledness abstractions* and *enumeration predicate abstractions*. These abstractions are effective starting points because they do not rely on manually provided predicates, simplifying the validation process.

To define enabledness abstractions, let r_f be a boolean predicate over the state of a smart contract such that if r_f does not hold for a concrete state c, then f will not execute successfully from state c. Predicate r_f is essentially a necessary condition for successful termination. Such predicates can be easily extracted (manual or automatically) from the require clauses f may have. An example of a predicate r_f for `Donate()` in Figure 1 is $r_{Donate} \equiv max\_block > block.number$. Note that these predicates are defined solely over state fields, not function parameters, as the state partitioning relies entirely on the contract's internal state.

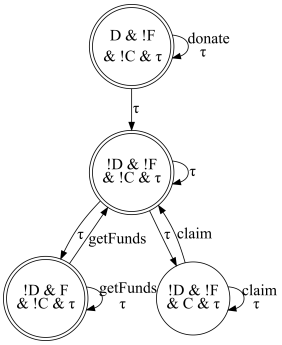
We refer to the *enabledness abstraction* of a smart contract when the predicates include exactly one necessary condition r_f for each contract function f, and when $\delta_\tau = false$ (i.e., no τ transition are considered). This abstraction can help auditors find potential issues in the protocol such as undesirable outgoing transitions (due to a weak require), missing outgoing transitions (due to stronger requires) or other bugs in the code.

An example of an *enabledness abstraction* is shown in Figure 3a. States are labelled with propositional formulas indicating which predicates are true and which are false in that state. We use predicate D for the require clauses of `Donate()`, C for those of `Claim()`, and F for `GetFunds()`. Both concrete states in Figure 2 are abstracted by the state labelled $\{D\&!F\&!C\}$ because, as discussed previously, `Donate()` is enabled while methods `GetFunds()` and `Claim()` are not. The concrete transition discussed previously representing a donation between the two states is a witness to the abstract transition that loops back from state $\{D\&!F\&!C\}$ to itself.

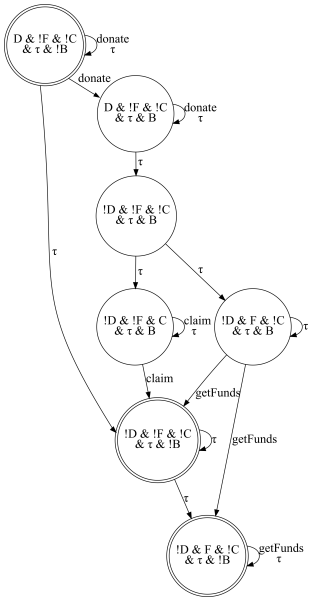
The *enumeration predicate abstraction* can be applied when a smart contract has a state variable of type *enum*. This is a common practice that allows a clear implementation of contract phases (e.g., open, bidding, closed, etc.). For such contracts a predicate for each possible value of the *enum* variable can be defined. As with enabledness abstractions, we also define $\delta_\tau = false$. An example of an *enumeration predicate abstraction* for the Solidity code in [30] is provided in [31]. One example predicate is $r_f \equiv State == StateType.DocumentReview$.



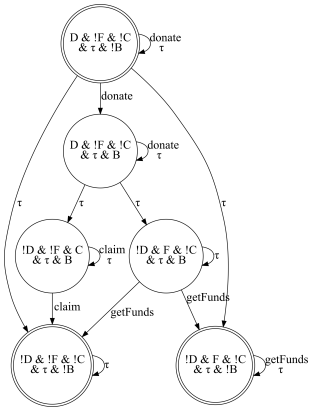
(a) Enabledness abstraction with no time progression (i.e., no δ_τ).



(b) Enabledness abstraction with time progression (i.e., + δ_τ).



(c) Enabledness abstraction + δ_τ + `B>0`.



(d) Enabledness abstraction + δ_τ + B = (`balance>0`). Require clause (line 15) for `Donate()` has been corrected ($\geq$ instead of $>$)

Fig. 3: Predicate Abstractions for `Crowdfunding`.

4 Smart Contract Validation

In this section we provide an overview of how we envision an auditor may approach the validation of the crowdfunding example depicted in Figure 1. We first show how this process might start from the default abstractions presented above, incrementally adding new abstraction predicates as needed. We then discuss the use of transition refinement and avoiding the use of default abstractions.

4.1 Incremental Predicate Refinement

An auditor might use a tool to automatically create an initial abstraction based exclusively on *require* clauses, as shown in Figure 3a.

In this first abstraction the auditor can detect, for example, that function `Claim()` is always disabled. However, the auditor knows that this first abstraction does not accurately represent the progression of time since the abstraction uses $\delta_\tau = false$. Since the contract uses `block.number` in some preconditions (e.g., donation deadline), the auditor decides to include τ transitions. These transitions represent an incremental in the block number by 1 (i.e., $\delta_\tau(c, \tau, c') = \langle c, \tau, c' \rangle \in \delta_c \wedge c'.block.number = c.block.number + 1$). Thus, the auditor specifies in the tool that τ transitions should be considered in δ_c.

```
function tau() public  {
  block.number = block.number + 1;
}
```

Fig. 4: Semantics of the τ function.

In Figure 3b, we show the resulting abstraction where time progression can be observed by τ transitions. It should be noted that these transitions do not modify the contract's state variables directly but rather simulate a change in the blockchain's state. Despite this, it has a significant impact on the contract since several functions use the current block's number in their precondition. Therefore, τ-transitions can either maintain the contract in the same state (e.g., state $\{D\&!F\&!C\&\tau\}$) or transition it to a new state, enabling or disabling calls to specific functions. For instance, starting in state $\{D\&!F\&!C\&\tau\}$, where the `Donate()` function is enabled, simulating the passage of time through τ can lead to a new state $\{!D\&!F\&!C\&\tau\}$ where `Donate()` is no longer possible. This emphasizes the importance of considering such transitions when analyzing the contract's behavior over time.

After manual inspection of the new abstraction, following validation guidelines to identify bad smells [29], the auditor may decide to focus on the state that has no outgoing non-τ-transitions (i.e., $\{!D\&!F\&!C\&\tau\}$). In this state no functions can be called and thus the crowdfunding effort has presumably ended. Yet the auditor may observe that this state is reachable directly through a constructor (i.e., the state is doubly circled). In other words, the auditor observes that, according to the abstract model, it is possible that upon deployment, neither function might be enabled, which is undesirable.

The auditor then may inquire from what blockchain concrete state it is possible to reach a concrete state in $\{!D\&!F\&!C\&\tau\}$ by executing the `constructor`, and which parameters would the constructor need to be called with. A tool may provide such information:

```
// Global state at constructor execution time
Global.balance=[Addr#1->1,Addr#3->2,Addr#4->3]
Global.block.number=1
// constructor arguments
_owner=Addr#1, _max_block=0, _goal=1
msg.sender=Addr#4, msg.value=0
// post-state: !D && !F && !C && τ
address=Addr#2
owner=Addr#1, max_block=0, goal=1
backers=[], funded=false
Global.balance=[Addr#1->1,Addr#3->2,Addr#4->3]
Global.block.number=1
```

From this concrete example, the auditor may discover that the cause for not enabling `Donate()` after construction is that `_max_block` is already less than `block.number` at deployment time. The auditor might recommend adding the *require* clause `_max_block > block.number`, while noting that this will incur a small additional gas cost.

The aforementioned scenario is, however, rather innocuous as no funds are lost by anyone. Auditors focus on scenarios in which funds can be lost or sent to incorrect destinations. Thus, the scenario may prompt the auditor to ask if state $\{!D\&!F\&!C\&\tau\}$ can be reached with a balance greater than 0. This will indicate that it is possible to have funds available in the contract but no action is enabled to extract them. Then, adding predicates `B>0` and `B=0` where `B` is `address(this.balance)`, the tool re-generates the abstraction as shown in Figure 3c. In this refined abstraction, the auditor receives new information. On the one hand, abstract states where $B > 0$ can only be reached by previously calling the `Donate()` function. This behavior is clearly as expected and now can be explicitly validated with the new abstraction. On the other hand, the auditor confirms their previous suspicion: it is indeed possible to reach a state where funds were raised and no function remains enabled (namely, $\{!D\&!F\&!C\&\tau\&B > 0\}$). This is clearly undesirable and an example of how a function call may lead to such state can be requested

```
// pre-state: D && !F && !C
address=Addr#2
owner=Addr#1, max_block=10, goal=2, funded=false
backers=[Addr#1->1]
Global.balance=[...], Global.block.number=9
// Donate arguments
msg.sender=Addr#3, msg.value=1
// post-state: !D && !F && !C
address=Addr#2
owner=Addr#1, max_block=10, goal=2, funded=false
backers=[Addr#1->1, Addr#3->1]
Global.balance=[...], Global.block.number=10
```

By inspecting the concrete instance, the auditor may observe that such transition is indeed possible only when `block.number` gets incremented by one to become equal to `max_block`. This equality impedes all three contract functions from being called successfully at a specific time, although there is an enabled τ transition that leads to a state in which functions are re-enabled. Having a period in which the contract is essentially frozen is clearly undesired. Thus, changing $<$ with $<=$ in the require clauses for `Claim()` and `GetFunds()` (or, similarly, changing $>$ with $>=$ in the require clause for `Donate()`) may be recommended by the auditor.

After changing the require clause for `Donate()`, the auditor may request a new abstraction to ensure the resulting one matches their expected mental model. Figure 3d shows such new abstraction, in which only one abstract state has all contract functions disabled (state $\{!D\&!F\&!C\&\tau\&B = 0\}$). As the balance is zero, it is no longer possible to freeze any funds. This disabled state can be reached by the `Claim()` or `getFunds()` functions. The former scenario occurs when the *goal* has not been reached and every donor has claimed their fund. The latter occurs when the goal has been reached and the owner of the contract collects all funds.

In the refined model the auditor also notices the abstract state $\{!D\&F\&!C\&\tau\&B = 0\}$. This appears suspicious to the auditor since it represents concrete states where the balance is equal to 0, but the `getFunds()` function is still enabled. Upon reviewing this behavior in the code, the auditor concludes no wrongdoing, as this function can only be executed by the owner when the crowdfunding has ended and the *balance* is at least equal to the *goal*. Therefore, when both the goal and the balance are equal to 0, it can be executed despite having no effect on the contract's state.

4.2 Transition Refinement

A frequently used technique for validating contracts is to examine specific scenarios with concrete actors. This is an effective method since defects are often exhibited through a small number of concrete explorations. For instance, in the context of a Crowdfunding contract, while the number of addresses that can make donations is infinite, testing a few selected addresses (e.g., two donors and a crowdfunder) and analyzing the interactions between them can be sufficient for understanding and inspecting corner cases. By constraining the input space in this manner, manual validation becomes more manageable. It is worth noticing that, as the specific input space increases, the resulting abstraction will become more complex, potentially hindering manual validation efforts.

We can achieve this with *transition refinement*. Unlike conditions on state variables when adding abstraction predicates, in transition refinement conditions on the function parameters (or on *msg*) can be specified. For instance, in the Crowdfunding's `Donate()` function, the auditor could decide to produce an abstraction that differentiates between donations made by two distinguished donors A and B. Once this is decided, a tool might produce such an abstraction by (internally) duplicating the original function with new preconditions added to each duplicated function (e.g., $msg.sender == A$ and $msg.sender == B$) as it is shown below:

```
function Donate_A() public payable {
  require(max_block > block.number);
  require(backers[msg.sender] == 0);
  require(msg.sender == AddressA) // new
  backers[msg.sender] = msg.value;
}

function Donate_B() public payable {
  require(max_block > block.number);
  require(backers[msg.sender] == 0);
  require(msg.sender == AddressB) // new
  backers[msg.sender] = msg.value;
}
```

Notice that the generated abstraction will restrict the input space of addresses that can call the `Donate()` function to exclusively addresses A and B. A different address will not be able to donate as it would not satisfy the function's precondition.

In Figure 5a, the abstraction generated by the `Donate` transition refinement can be observed. DA and DB refers to `Donate_A` and `Donate_B` preconditions, respectively. Analyzing this abstraction, it can be observed that the initial state {DA & DB & !F & !C & τ} has both `Donate_A` and `Donate_B` enabled. From this initial state, if `Donate_A` is executed, it can lead to another abstract state where only `Donate_B` is enabled (i.e., in this abstract state `Donate_A` and `getFunds` are disabled). The same occurs when `Donate_B` is executed. This aligns with the expected outcome: according to the function's conditions, each actor can only donate once. However, upon analysis, the auditor could also notice that whenever `Donate_A` or `Donate_B` are enabled, it could lead to the same abstract state. This is a priori an unexpected behavior: once an actor donates, they should not be able to donate again. In such an scenario, the auditor may request for a concrete example of why such behavior can occur. A tool may provide a concrete trace that explains the suspicious behavior:

```
// pre-state: DA &&  DB && !F && !C ! $\tau$
owner=Addr#1, max_block=7, goal=6, funded=false
backers=[]
Global.balance=[...], Global.block.number=6
// Donate arguments
msg.sender=Addr#A, msg.value=0
// post-state: DA &&  DB && !F && !C ! $\tau$
owner=Addr#1, max_block=7, goal=6, funded=false
backers=[Addr#A->0]
Global.balance=[...], Global.block.number=6
```

From this concrete example, the auditor may discover that the suspicious transition is due to the fact that, at the time of donating, the *msg.value* parameter could be equal to zero. The auditor now knows precisely where to

examine the implementation to confirm if this is valid. Indeed, this leads to identifying that the requires clause for `Donate()` should also ensure $msg.value > 0$.

4.3 Removal of Default Predicates

In all the previous examples, the predicates introduced by the default enabledness abstraction provide significant state structure that aids validation without the need for the auditor to know anything a priori about the behavior of the contract. However, with just a little input from the auditor, it can be beneficial in certain scenarios to generate abstractions without these default predicates. The result can be much more compact representations yet still insightful abstractions. For instance, partitioning states solely based on whether the current balance is greater than zero (i.e., B>0) provides a compact yet useful abstraction. Referring to Figure 6, an auditor can validate how different functions impact the total contract balance.

5 Reentrancy

The various strategies for abstracting contract behavior discussed in the previous section are not completely adequate for exhibiting unintended behavior as a result of reentrancy. A reentrancy occurs when an external entity makes another function call during the execution of a contract function. This can lead to problems as the re-entrant function call may find the contract in an intermediate state for which the re-entrant function was not designed for. Reentrancy bugs have been studied extensively (e.g., [32,33,34,21]) as they have led to significant losses (e.g., [4]). A recent work [20] has provided evidence that current tools has a high rate of false negative reentrancy detection (over 99%).

In the previous section, abstractions were constructed only considering those concrete states that resulted from executing a transaction. In other words, they show abstract *final transaction states*. However, to understand the conditions under which reentrancy may occur and the implications of such occurrences it is useful to reason about *transient transaction states*. As there are potentially infinite of such transient states, abstracting them into a small number of abstract transient states may allow validating contract behavior and catching reentrancy bugs. In this section, we demonstrate how abstractions can be constructed to represent transient states using predicate abstraction, and how these abstractions reveal reentrancy bugs.

5.1 Example

Reentrancy is only possible when a contract being executed calls another contract. These scenarios are called cross-contract calls [34]. In Solidity, cross-contract calls are made explicitly by calling directly another known smart contract hardwired in the code or using low-level functions `call` and `delegatecall` on arbitrary addresses. When calling a known smart contract, its code can be inspected to see if a re-entrant call back can be made. However, when the other smart contract is not known in advance, particular care must be taken.

A notoriously pernicious case is `call.value()` that is used for transferring Ether between accounts as in Subsection 2.1. If the callee is a contract, then the call may execute unknown re-entrant code instead of simply transferring Ether. In other words, the instruction `addressA.call.value(amount)` transfers `amount` from the contract balance to the `addressA` account assuming that `addressA` is a "personal" account. However, if this address corresponds to another smart contract account, the transfer can trigger the execution of the receiver contract code. This code may re-call any public function of the caller contract, resulting in a reentrancy.

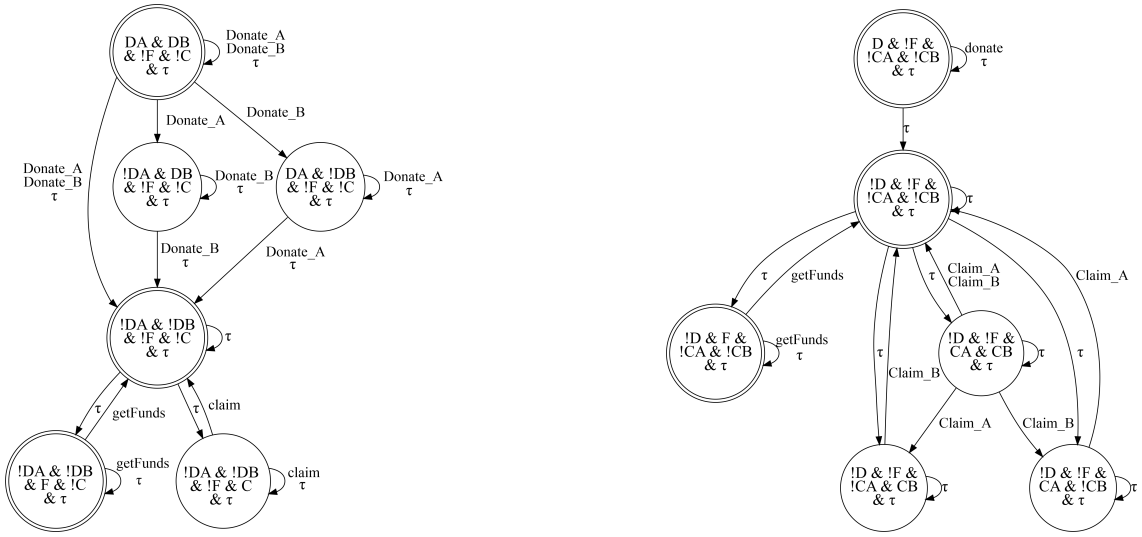
The smart contract shown in Figure 1 has two functions that may result in cross-contract calls, leading to reentrancy: `Claim()` and `GetFund()`. These two functions rely on the `call.value()` primitive to transfer Ether to other accounts. For instance, let us consider the code in `Claim()`. This function checks various conditions including that the caller of the function has funds to claim (Line 29), then stores the amount to be claimed in a local variable and then transfers the assets to the address `msg.sender`. Only after the transfer is the internal state changed to reflect that the address' balance is 0 (line 33). However, if the address `msg.sender` corresponds to a smart contract, then this called contract, in addition to receiving the funds, may perform another call to `Claim()`. Since the `backers` mapping has not been updated yet, more assets will be transferred to the address `msg.sender`.

Note that this behavior cannot be observed in any of the previous abstractions for this contract. In none of them we can observe that a `Claim()` made by a sender can actually trigger a new `Claim()`. This is expected as the abstraction only shows the resulting contract state once the first call to `Claim()` *terminates*, hiding that a second `Claim()` may have occurred within the first as a result of a reentrancy.

5.2 Reentrancy Revealing Abstractions

To reveal reentrancy bugs in predicate abstractions, each function that have cross-contract calls is split into two, precisely at the point where the cross-contract call occurs. Indeed, this makes the transient state at which a reentrancy may occur a final state of the first part of the split function.

There are some subtleties that must be addressed when splitting functions. We explain them by going through



(a) Enabledness abstraction with transition refinement for Donate() method

(b) Enabledness abstraction with method refinement for Claim() method

Fig. 5: Abstraction for *Crowdfunding* with transition refinement.

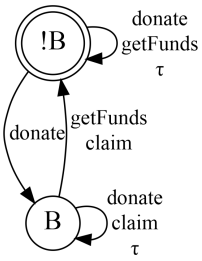


Fig. 6: State Predicate Abstraction example

how the `Claim()` function is split. The two functions below are the result of splitting the `Claim()` function:

```
function Claim_init() public {
  require(max_block < block.number);
  require(backers[msg.sender] > 0 && !funded);
  require(goal > address(this).balance);
  require (countBackers > 0);
  uint val = backers[msg.sender];
  msg.sender.call.value(pr)("");
  backers_stack.push(msg.sender);
}
```

```
function Claim_end() public {
  require(backers_stack.length > 0);
  require(backers_stack[backers_stack.length-1] ==
      msg.sender);
  backers_stack.length--;
  if (backers[msg.sender] > 0) {
    countBackers -= 1;
    backers[msg.sender] = 0;
  }
}
```

The function suffixed with"_init" represents the function call up to the execution of the `call.value()` operation, as beyond this point, another contract may assume control of the execution. However, at some point, the execution control is returned to the original contract, which will continue executing the remaining instructions after the `call.value()` operation. These remaining instructions are represented by the function suffixed with "_end".

Note that `Claim_init()` has an additional line of code that pushes onto a stack the address of the sender. Also note that `Claim_end()` requires the stack to not be empty and for the top of the stack to have the address of the caller. This ensures that both functions preserve the semantics of the original `Claim()` (i.e., the second part of `Claim()` is executed only when it's corresponding first half finished).

Another semantic preserving addition that is required is a require statement in the τ function (code below) to avoid increasing the `block.number` between internal transactions if a `Claim` transaction is being executed. This is consistent with the Ethereum network which uses synchronous cross-contract calling where the entire execution of both contracts goes into the same transaction.

```
function tau() public  {
  require(backers_stack.length == 0);
  block.number = block.number + 1;
}
```

In Figure 7a, we show the abstraction of the crowdsourcing contract that has a split `Claim()` function. Indeed, it is a refinement of Figure 3d where we can now observe re-entrant calls to `Claim()`. Instead of a single `C(claim)` enabledness predicate, this predicate is replaced by `Ci` and `Ce` denoting enabledness for `Claim_init` and `Claim_end` respectively. Note the states S_5 and S_6 are represented with squares. These correspond to *transient states* where control has been potentially relinquished to another contract. These are the states in which a reentrancy may occur. Also note that in these states, the τ transition is not enabled.

Consider the behavior exhibited by the contract upon calling `Claim_init` in S_4. As a result, the call can take the contract to states S_4, S_7 or S_9. The sequence

$$(S_4) - Claim\_init \rightarrow (S_5) - Claim\_end \rightarrow (S_4)$$

could represent the happy sequence where `call.value()` function does not reenter. Similarly, the sequence

$$(S_4) - Claim\_init \rightarrow (S_6) - Claim\_end \rightarrow (S_9)$$

is a non-reentrant call to `call.value()` function in which there is only a single donor. So, after the full `Claim()` transaction terminates, no more claims can be made and the contract balance is zero.

Concrete states in S_7 are *final states* (i.e., non transient) that have the `Claim_init()` operation enabled (i.e., Ci is true in S_7), but the balance is zero (i.e., B=0 in S_7). If `Claim_init()` is enabled, there is at least one donor that has yet to get their funds, but as we already stated, the balance is already zero. In other words, no funds are available for a valid donor's claim. The auditor may find suspicious that the contract might reach such a final state after a sequence of transactions. Observe that, for a reentrancy to be exploited, it requires to actually *complete* a transaction. It is worth noticing that, since `Claim_init` starts an execution of a `Claim`, for a sequence to be considered valid it will require a closing `_end` transition for each `_init`. After a further inspection there is such a valid sequence for reaching state S_7 starting from S_4 as follows:

$$\begin{aligned}(S_4) - Claim\_init &\rightarrow (S_5)\\ (S_5) - Claim\_init &\rightarrow (S_6)\\ (S_6) - Claim\_end &\rightarrow (S_6)\\ (S_6) - Claim\_end &\rightarrow (S_7)\end{aligned} \tag{1}$$

A sequence diagram that shows this trace is provided in Figure 8) and shows that a donor may claim their donation many times, emptying the contract and leaving other donors without their respective funds. Note that the ability for a user to `Claim` funds twice due to a reentrancy bug can only be observed in the presence of transient states.

Such feedback would allow an auditor to confirm the existence of a vulnerability that breaks the contract invariant, where the contract balance should always be equal to the sum of the mapping values. A recommended repair for the reentrancy bug is to update the contract state before making the Ether transfer, swapping lines 7 and 8:

```
function Claim() public {
  require(max_block < block.number);
  require(backers[msg.sender] > 0 && !funded);
  require(goal > address(this).balance);
  require (countBackers > 0);
  uint val = backers[msg.sender];
  backers[msg.sender] = 0;
  msg.sender.call.value(pr)("");
}
```

Recomputing the abstraction for the fixed contract (Figure 7b) shows that it is no longer possible to reach a state where `Claim_init` is enabled and the condition $address(balance) = 0$ holds, as previously observed. Furthermore, the absence of that state is the only difference between with previous abstraction. Yet the new abstraction still includes reentrancy sequences such as:

$$\begin{aligned}(S_4) - Claim\_Init &\rightarrow (S_5)\\ (S_5) - Claim\_Init &\rightarrow (S_5)\\ (S_5) - Claim\_End &\rightarrow (S_5)\\ (S_5) - Claim\_End &\rightarrow (S_4)\end{aligned} \tag{2}$$

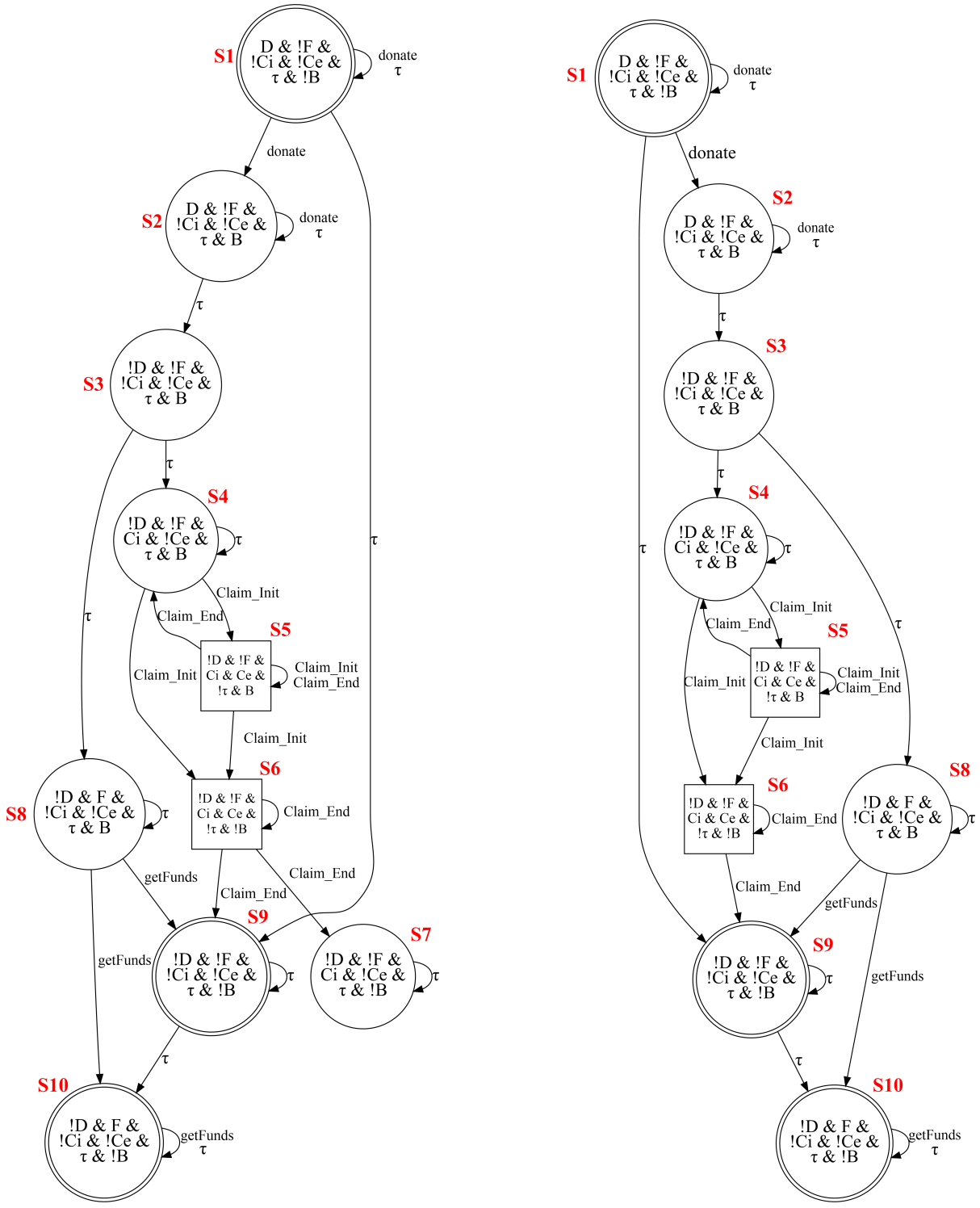
An auditor faced with the question of whether this is a valid or erroneous reentrancy may use tool support to create a sequence of concrete states that match the scenario. The sequence diagram in Figure 9 allows an auditor to observe that the scenario requires two different addresses to perform the claims that exhibit the behavior. The first cross-contract call is to one address that then calls to a second address which then performs a `Claim()`. In other words, both addresses get the funds that they were supposed to get, nobody gets funds twice. An auditor's conclusion would be that this reentrancy is valid and no further changes are needed.

To ensure contract robustness, we assume that the auditor now seeks to eliminate this valid reentrancy by introducing a mutex, thereby preventing any re-entrant calls. This implies adding a `lock` to the `Claim()` function as follows:

```
bool lock = false;
function Claim() public {
  require(max_block < block.number);
  require(backers[msg.sender] > 0 && !funded);
  require(goal > address(this).balance);
  require(!lock); //New precondition
  lock = true;
  uint val = backers[msg.sender];
  msg.sender.call.value(pr)("");
  backers[msg.sender] = 0;
}
```

Figure 10 depicts the updated abstraction after implementing such mutex. The resulting abstraction is similar to Figure 7b, but now it prevents multiple calls to `Claim_init`. When calling to `Claim_init` from state S_4, it may lead to either state S_5 or state S_6. If it ends in state S_6, then the function was called by the last user with funds in the contract. On the other hand, if it ends in state S_5, there are still more users with funds in the contract who can `Claim` their funds.

By introducing this mutex, the contract now ensures that only one instance of `Claim_init` can be executed at a time, preventing any reentrancy scenario. This modification ensures a more secure and predictable behavior for the contract, and it safeguards the integrity of the fund withdrawal process.



(a) Buggy version.

(b) Fixed version.

Fig. 7: Enabledness abstractions + δ_τ + $B(balance > 0)$ for *Crowdfunding* analyzing cross-contract calls in `Claim()` function. Square nodes represents *transient states*.

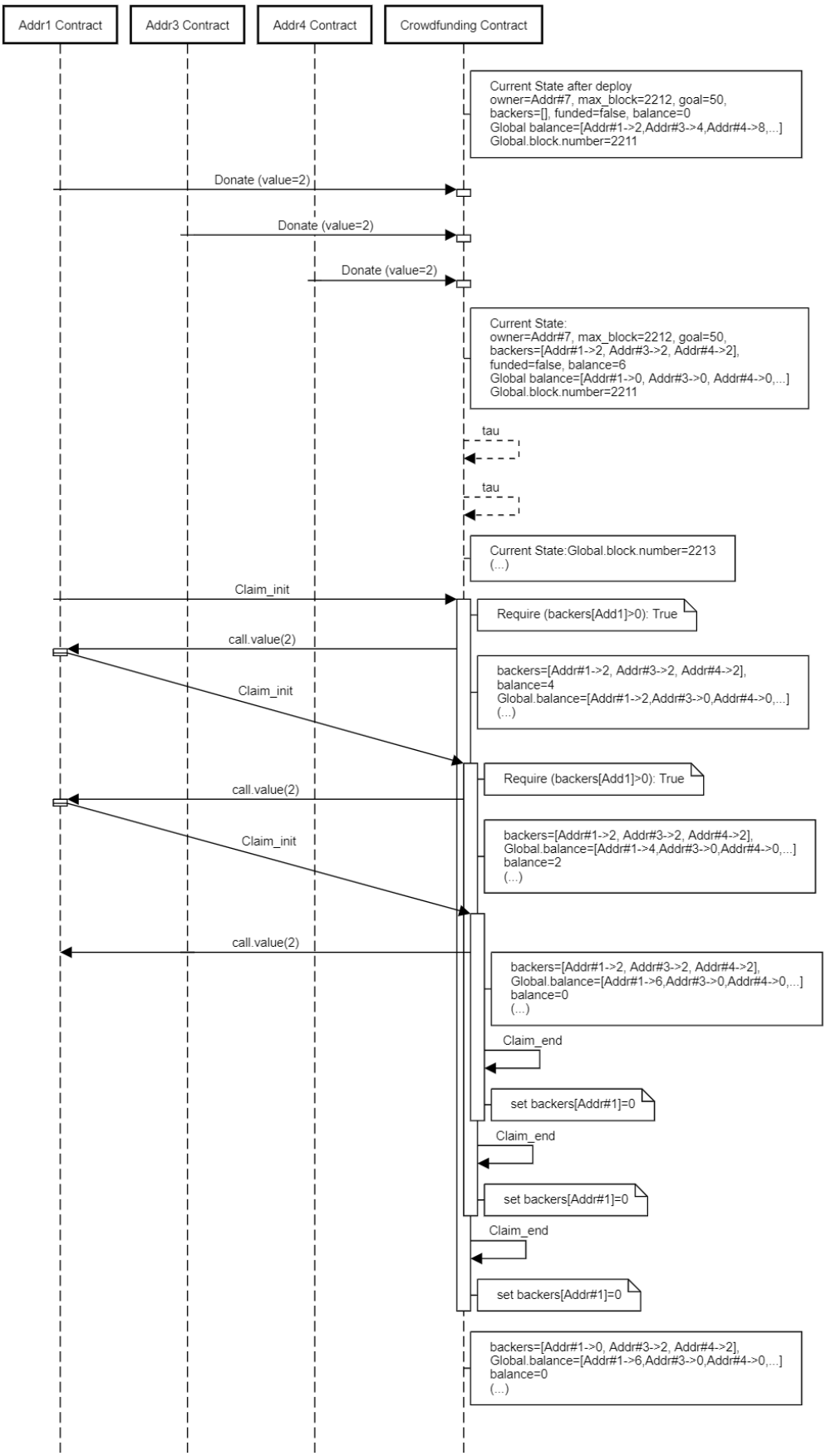


Fig. 8: Sequence diagram example for a concrete sequence that shows a re-entrant buggy scenario.

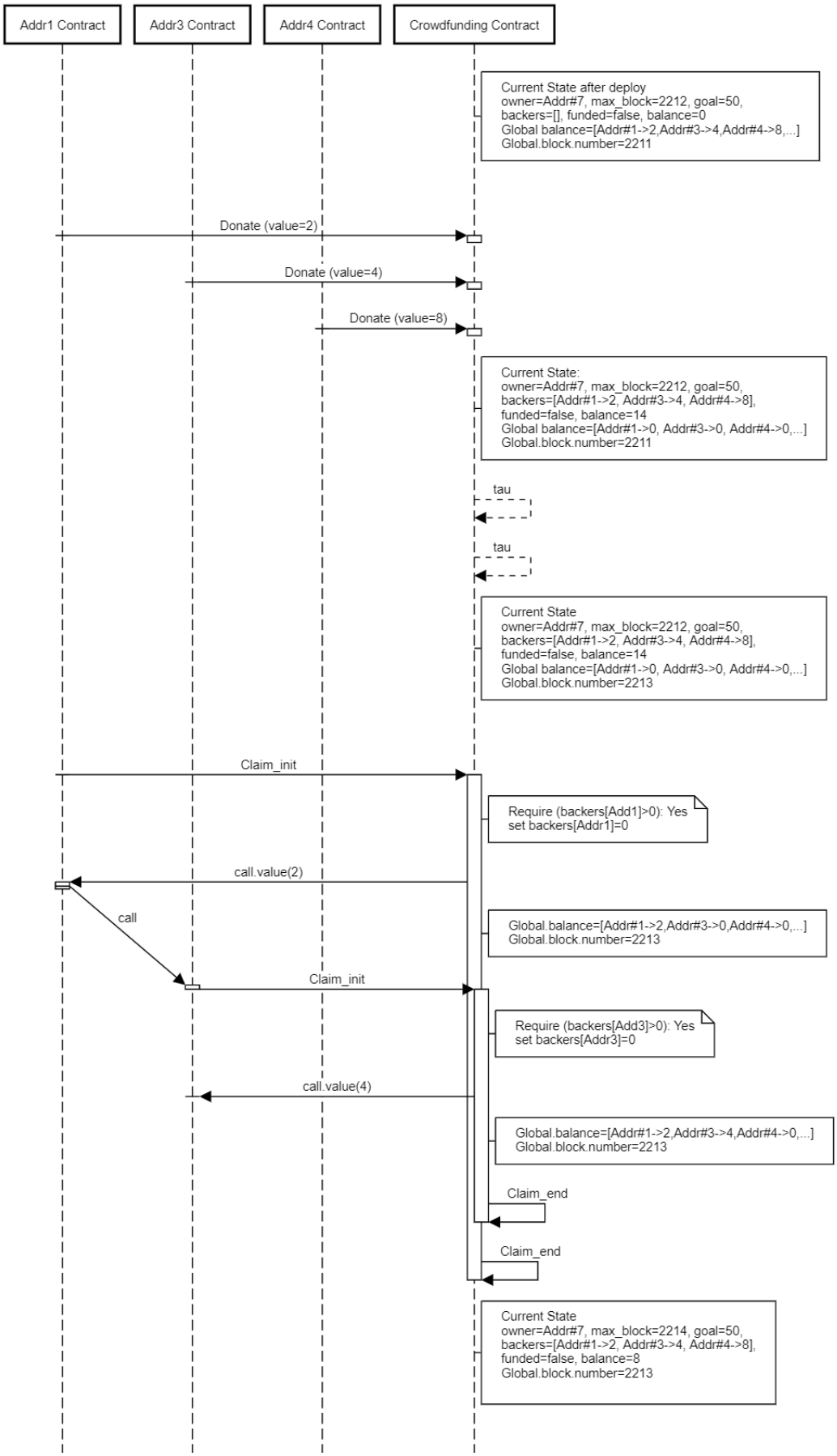


Fig. 9: Sequence diagram example for concrete sequence that show valid re-entrant scenario.

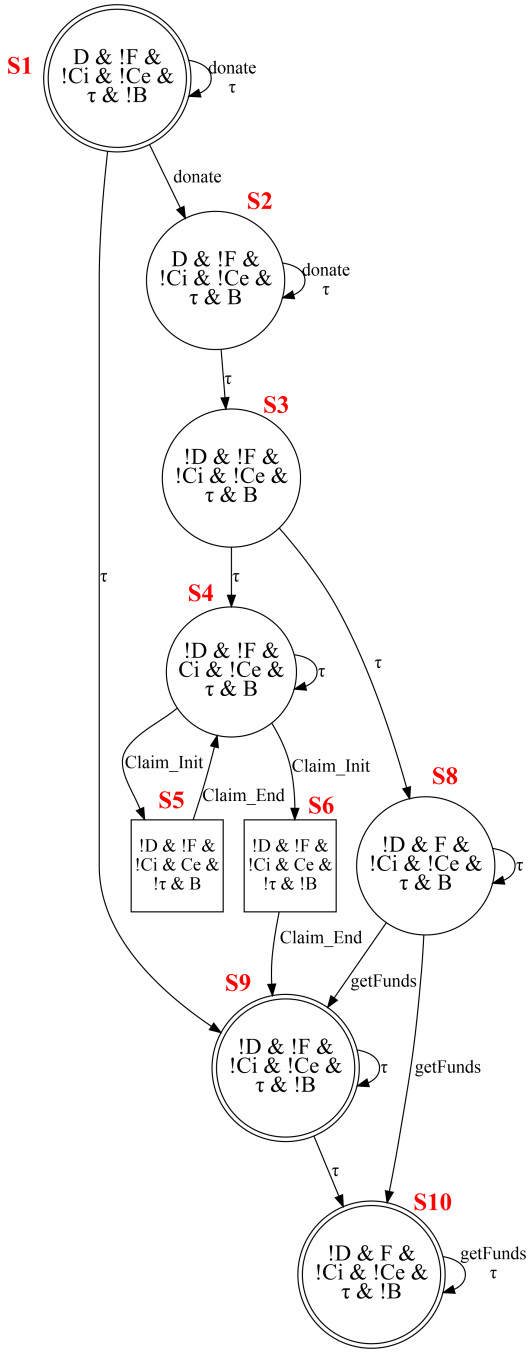


Fig. 10: Enabledness abstraction with mutex fixed Claim function.

6 Automated Smart Contract Abstraction

In this section we report on PASCo [35], a tool designed to automate the construction of smart contract abstractions, inspired by the work presented in [26]. PASCo builds the abstraction incrementally by adding transitions one at a time according to the result of a reachability query made to a smart contract verification engine. A reachability query asks if a user-specified line of code (typically given with an assertion) in the contract is reachable through a sequence of function calls to the contract. PASCo relies on VeriSol [8], a formal verifier for Solidity programs.

We first describe the general approach and then discuss the specific issues resulting from the fact that Verisol uses a bounded software model checker to verify Solidity programs, hence it is only correct and complete up to a given *k_bound* parameter.

6.1 Basic Reachability Queries

To decide if a particular transition between two abstract states must be added to the smart contract abstraction, PASCo writes a new contract function with an assertion such that the possibility of violating the assertion is semantically equivalent to the condition for adding the transition to the abstraction. In other words, to decide whether a transition labelled with function f from an abstract state s_1 to an abstract state s_2 should be included, PASCo automatically creates a function $s_1\_to\_s_2\_by\_f()$ that requires the predicate for being in abstract state s_1 to be satisfied, executes f and then asserts that the predicates for being in s_2 do not hold (see Figure 11). PASCo then uses Verisol to generate a concrete sequence of transaction calls on the contract that violates the assumption. If such a concrete trace exists, the transition $\langle s_1, f, s_2 \rangle$ is added to the abstraction.

```
function s1_to_s2_by_f(params) public {
  require(is_in_s1());
  f(args));
  assert(!is_in_s2);
}
```

Fig. 11: Basic reachability query schema

As a concrete example, consider the reachability query listed in Figure 12. This query is generated by PASCo while building the abstraction shown in Figure 3c. In particular, this query checks if it is possible to loop on the state $\{D\&!F\&!C\&\tau\&B > 0\}$ when executing the *Donate()* function. Internally, this query is generated automatically by a method *reachable(s_1, s_2, f)* that has as parameters the two abstract states s_1 and s_2, and a contract function.

Using VeriSol to check falsifiability of the assertion it provides the following output to PASCo (where line 110 is the line with the failed assert):

```
-----Transaction Sequence for Defect ------
Crowdf.sol(16,1): CrowdF::Constructor(this=addr!0,
msg.sender=addr!2, msg.value=40, _owner=addr!4,
    _max_block=725, _goal=611, _blockNumber=196)
Crowdf.sol(24,1): CrowdF::Donate(this=addr!0, msg.
    sender=addr!31, msg.value = 691)
Crowdf.sol(74,1): D_notF_notC_Tau_Bal_by_Donate_CrowdF
    (this=addr!0,
msg.sender=addr!3, msg.value=52)
Crowdf.sol(110,1): : ASSERTION FAILS!
---------------
```

```
function D_notF_notC_Tau_Bal_by_Donate() public {
  require(pre_donate());          //D
  require(!pre_getFunds());       //!F
  require(!pre_Claim());          //!C
  require(pre_δτ());              //δτ
  require(balance_GTZ());         //B>0

  Donate();

  bool isReach = pre_donate()           //D
  isReach = isReach && !pre_getFunds()  //!F
  isReach = isReach && !pre_Claim()     //!C
  isReach = isReach && pre_δτ()         //δτ
  isReach = isReach && balance_GTZ();   //B>0
  assert(!isReach);
}
```

Fig. 12: A basic reachability query generated by *reachable(s,s,Donate)* with $s = \{D\&!F\&!C\&\tau\&B > 0\}$)

Note that Verisol not only concludes if any assertion has failed, but it also generated the sequence of calls and the actual parameters on each call, including implicit arguments such as the caller address.

6.2 Abstraction Construction Algorithm

Algorithm 1 Abstraction Construction

1: $F \leftarrow$ *public functions* $\cup \{\tau\}$
2: $P \leftarrow$ *user defined predicates*
3: $S \leftarrow 2^P$ // abstract states
4: $S' \leftarrow \{s \in S \cdot reachable(s)\}$
5: $S_0 \leftarrow []$ //initial states
6: $\delta_c \leftarrow [()]$ //initial transitions
7: **for** s_1 in S' **do**
8: **if** $is\_initial\_state(s_1)$ **then**
9: $S_0.append(s_1)$
10: **end if**
11: **for** s_2 in S' **do**
12: **for** f in F **do**
13: **if** $reachable(s_1, s_2, f)$ **then**
14: $\delta_c.append(s_1, f, s_2)$
15: **end if**
16: **end for**
17: **end for**
18: **end for**

In Algorithm 1 we present the pseudocode for building smart contract abstractions. Line 1 defines F to be the set of public functions of the smart contracts and τ functions defined by the auditor (e.g., Figure 4). Line 2 defines variable P to be the set of abstraction predicates given by the auditor (e.g., *balance* > 0). Line 3 defines the set of abstract states S. Each abstract set is represented by a subset of the predicates in P, these are the predicates assumed to be true in the state, the rest are assumed to be false.

Due to implicit invariants in the code, many abstract states in S may not contain reachable concrete contract states. Thus we run an optimization process (line 4) to purge S. To see if an abstract state $s \in S$ is reachable from the initial contract state, we use the basic reachability query schema discussed before. An example checking if state $\{D\&!F\&!C\&\tau\&B > 0\}$ is reachable, generated by the *reachable(s)* is shown in Figure 13. If the assertion is reached, then there exists a concrete reachable state that satisfies being in $\{D\&!F\&!C\&\tau\&B > 0\}$.

```
function D_notF_notC_Tau_Bal() public {
  require(pre_donate());          //D
  require(!pre_getFunds());       //!F
  require(!pre_Claim());          //!C
  require(pre_δτ());              //δτ
  require(balance_GTZ());         //B>0

  assert(!true);
}
```

Fig. 13: A basic reachability query generated by *reachable(s)* with $s = \{D\&!F\&!C\&\tau\&B > 0\}$)

Starting from Line 7, we proceed to identify feasible transitions for the final abstraction. For each abstract state $s_1 \in S'$ we first check if it is an initial state of the abstraction. In other words, we check if the state s_1 is reachable by only calling the `constructor` function. For this, *is_initial_state(s)* checks query *reachable(s)* but restricting the depth of the VeriSol exploration to a single transaction (forcing to explore only the constructor execution since the constructor is always the first operation explored by Verisol in any transaction).

Finally, we check for all possible destinations s_2 and function f, if it is possible to reach, starting from a concrete state in s_1 and calling f with some parameters, a concrete state in s_2. Function *reachable(s_1, s_2, f)* creates a reachability query that if successful, indicates that a new transition should be added to the abstraction.

6.3 On the Bounded Exploration of Verisol

VeriSol attempts to answer to reachability queries by finding a concrete sequence of transaction calls on the contract that violates an assertion. It does so using a bounded software model checker, Corral [36], and as such requires a bound parameter *k_bound* for loop unrolling, unfolding recursive calls, and as a maximum to the number of explored transactions. As smart contracts typically do not have loops or recursion, *k_bound* essentially bounds the number of smart contract calls that are sequenced to find a witness for a reachability query. Thus, when using VeriSol to respond a reachability query one of the following scenarios may arise.

- *reachable*: Verisol has found a concrete sequence of contract calls that violates the assertion.
- *unreachable*: Verisol has exhaustively searched through all possible sequences of contract calls of up to length

k_bound and has not been able to identify one that violates the assertion.
– *timeout*: Verisol has exhausted its time budget (i.e., the *time_bound* parameter) without finding a sequence of contract calls that violates the assertion.

Let us now discuss the verdicts of Verisol in PASCo. Consider the $reachable(s_1, s_2, f)$ query in line 13 of Algorithm 1. If Verisol finds a suitable witness, PASCo can safely add the corresponding transition to the abstraction. In contrast, given the bounded exploration of Verisol, whenever Verisol concludes that a witness was not found, it could be due to the fact that i) such transition is actually infeasible, or ii) the chosen exploration *k_bound* for Verisol is not large enough to find a witness. Since PASCo chooses to discard a transition when Verisol finds no witness, the generated abstraction represents an under-approximation of the formal definition in Subsection 3.2. An *unreachable* verdict when purging S (line 4 of Algorithm 1) also under-approximates the states for which transitions will be computed. This is consistent with the above: if s_2 cannot be reached in k steps, it will not be reachable from any s_1 via any function f in k steps either. Finally, the *unreachable* verdict for identifying initial states (line 8 of Algorithm 1) is correct and complete as a *k_bound* of 1 characterizes states that are initial states.

Similar to *unreachable* verdicts, PASCo also chooses to discard transitions whenever a *timeout* occurs. The *time_bound* and *k_bound* values affect the general performance of the algorithm (both in terms of end-to-end time and in how approximate the result is). A low *k_bound* (or *time_bound*) value will yield increased performance but may miss transitions. On the other hand, a high *k_bound* (or *time_bound*) value to avoid missing transitions may lead to more execution time.

Setting these parameters must be done by an auditor in terms of their prior experience. However, these parameters can and should be adjusted by the auditor as they inspect and reason about abstractions that PASCo produces. In the next section we continue to discuss these parameters as part of the evaluation.

6.4 Tool configuration

PASCo receives a configuration file with a smart contract, a list of functions from the contract, and a list of predicates and produces an abstraction in both DOT and PDF formats. The configuration file allows a user to specify the predicates that would be used for abstractions.

PASCo supports three predicate specification methods: explicit boolean predicates, function preconditions (for default *enabledness abstraction*), or explicit state enumerations (for default *enumeration predicate abstraction*). For example, Figure 3b shows the abstraction generated by the following configuration file using default enabledness abstraction:

```
fileName = 'Crowdfunding.sol'
contractName = 'CrowdFunding'
functions = [
'Donate();',
'GetFunds();',
'Claim();',
't();',
]
statePreconditions = [
'(max_block >= blockNumber)',
'(max_block < blockNumber && goal <= balance)',
'(max_block < blockNumber && !funded && goal >
    balance && countBackers > 0)',
'true',
]
mode = epa
txBound = 8
time_out = 600
```

The `fileName` and `contractName` parameters specify the name of the Solidity file and the specific contract to be analyzed, respectively. Since a Solidity file may contain multiple contracts, `contractName` is required to disambiguate the target contract.

The `functions` parameter lists the function signatures to be included in the analysis, typically encompassing all the public functions. The `statePreconditions` parameter specifies a corresponding list of preconditions, aligned in order with the functions. For example, the first function in the `functions` list is `Donate()`, and the first entry in `statePreconditions` is its associated precondition (i.e., `(max_block >= blockNumber)`). These preconditions are generally straightforward to extract from the Solidity source code, as they commonly appear as `require` clauses. In the current version of PASCo, these preconditions must be provided manually; however, future enhancements may include automated inference directly from the source. It is worth noting that the last function is `t()`. This represents the τ transitions described in Section 4.

The `mode` parameter must be set to `epa` to generate an enabledness abstraction, and the abstract states are calculated through the preconditions. In this example, we use the default values `txBound=8` and `time_out=600`.

6.5 Limitations of Verisol Integration

VeriSol supports Solidity code up to version 0.6 and lacks support for certain language features, such as inheritance and automatic type inference for address variables. To address these limitations, we flattened contracts that used inheritance and introduced explicit type conversions where address-type variables were defined.

Another limitation concerns the use of mappings in Solidity. It is common in smart contracts to use mappings of the form $Address \rightarrow Int$ to track values such as user balances. However, Solidity does not support querying the size of a mapping (e.g., checking whether it is empty). Since several of our case studies required preconditions involving such checks, we modified some con-

tracts by adding an auxiliary integer variable to maintain the count of key-value pairs in the map.

We also removed all uses of *assert* statements in the contracts. This is because VeriSol uses assertions to generate counterexamples, which we leverage in our approach.

Finally, the main inherent limitation of VeriSol is that it requires a user-defined *k_bound*. As previously discussed, this bound limits the depth of transition exploration. A low *k_bound* may cause some transitions to be missed. If the auditor suspects that a transition is missing, it is necessary to re-run the tool with a higher *k_bound*.

7 Evaluation

In this section, we report experimental results supporting our claim that automatically constructed predicate abstractions can aid auditors in the process of validating the correspondence of smart contracts and their requirements. This claim was partially evaluated in [18] using a prototype that requires the manual translation of Solidity code to the Alloy language. Here, we revisit some of the evaluation in [18], but with a fully automated tool, PASCo, compare the output of PASCo with that of the prototype described in [18] and extend the evaluation in [18] with a study on contracts with reentrancy, a central source of problems in smart contracts, unsupported in [18].

Our first research question aims to gain insights into whether automated predicate abstractions automatically built with PASCo can reproduce third-party manually developed ad-hoc abstractions that exist for smart contracts. The ability to do so is an indication that PASCo may be a good fit to support humans when they think about and validate smart contracts.

RQ#1: To what extent can PASCo be used to build predicate abstractions of smart contracts to replicate existing informal and manually produced state-machine documentation of smart contracts. What is the significance of any differences that may arise?

In our second question we analyze if abstractions generated automatically by PASCo are equivalent to the ones generated by the prototype reported in [18] that required manual translation of smart contracts to the Alloy language. This allows gaining insight regarding the ability of PASCo to find all transitions between states despite the fact that it relies on bounded model checking.

RQ#2: To what extent can the abstractions generated by PASCo replicate the abstractions generated by the prototype reported in [18]? What is the significance of any differences that may arise?

RQ1 involves producing predicate abstractions where the expected result is known (i.e., the aim is to replicate a human provided abstraction). In contrast, in our third question we aim to evaluate automated predicate abstraction in an exploration context in which the appropriate abstractions to reveal potential defects are unknown.

RQ#3: To what extent can the successive construction of different predicate abstractions support auditors in validating the behavior of smart contracts against an informal description of their intended behavior. Can the abstractions reveal correspondence defects?

The fourth research question aims to answer if the proposed predicate abstraction augmented with transient states are able to expose reentrancy bugs in smart contracts. It also examines how the approach compares in terms of recall and precision with state-of-the-art Solidity analysis tools for detecting reentrancy bugs.

RQ#4.1: Are the predicate abstractions generated by PASCo able to expose reentrancy bugs?

RQ#4.2: How does the performance of PASCo, in terms of precision and recall, compare with existing state-of-the-art Solidity analysis tools for detecting reentrancy bugs?

7.1 Experimental design

To answer the **RQ#1**, we used the Microsoft Azure Blockchain Workbench [38] and repaired contracts from [7]. The Azure benchmark, used in many previous studies (e.g., [7,8,39,18]), includes 11 smart contract implementations, each accompanied by a detailed description of its intended behavior and a manually produced state transition diagram outlining the expected contract behavior. It is the only benchmark we are aware of in which smart contracts are accompanied by state transition diagrams that abstractly describe their behavior. The contracts in [38] are known to have defects and in [7], modified and verified contracts are provided [40]. We excluded 2 of the 11 contracts because they perform multi-contract calls that create new contracts to interact with, which is not currently supported by our tool.

In summary, we used 9 descriptions of the intended behavior of contracts of which 7 had two contract implementations (the original and the fixed version). Figure 14a summarizes the smart contracts considered for the research question. The column labeled *LOC* indicates the size of the contract measured as the number of lines of Solidity code, $|S|$ is the number of states of the state transition diagram provided in the documentation, and $|M|$ the number of public functions that can modify the smart contract state (i.e., we exclude pure and view functions). Note that for fixed and non-fixed versions of the same contract, $|S|$ coincides as they have the same manually provided state transition diagram, and $|M|$ coincides because fixes do not introduce new public functions. The columns labeled $Diff$ and FN will be discussed in Subsection 7.2.

For each subject we used PASCo with the default enumeration predicate abstraction to build a model and

Subject	LOC	|S|	|M|	Diff		FN	
				Sts	Txs	Sts	Txs
AssetTransfer	181	10	11	0	+3	0	0
AssetTransferFixed	219			0	+1	0	0
BasicProvenance	40	3	3	0	+1	0	0
BasicProvenanceFixed	50			0	0	0	0
DigitalLocker	118	6	10	-1	+38	0	0
DigitalLockerFixed	149			0	+6	0	0
DefectiveCompCount	27	2	1	-1	-1	1	1
DefectiveCompCountFixed	35			-1	-1	-1	-1
FrequentFlyerRC	42	2	1	0	0	0	0
HelloBlockchain	27	2	3	0	+2	0	0
HelloBlockchainFixed	39			0	0	0	0
RefrigeratedTransp.	114	4	3	0	1	0	0
RefrigeratedTranspFixed	130			0	0	0	0
RoomThermostat	38	2	4	0	0	0	0
SimpleMarketplace	58	3	4	0	2	0	0
SimpleMarketplaceFixed	67			0	0	0	0

(a) Benchmark#1. $|S|$ is number of states of the documented state transition diagram. $Diff$ is the number of transitions and states that differ between the output of PASCo and the documented state transition diagram. FN (i.e., False Negatives) is the number of incorrectly missing states and transitions in the output of PASCo.

Subject	LOC	|M|	#abst.
RefundEscrow	122	10	3
EscrowVault	100	10	2
EPXCrowdsale	161	10	3
Crowdfunding	55	6	3
ValidatorAuction	346	17	3
SimpleAuction	50	4	4
Auction	51	4	3
RockPaperScissors	66	4	2

(b) Benchmark#2. Column *#abstr.* is the amount of different abstractions generated for each subject in [18].

Subject	LOC	|M|	Approach	
			non-TS	TS
Etherstore	17	2	×	✓
EtherstoreFixed	17		-	×
EtherstoreFixed_Lock	21		-	×
Reentrancy_dao	18	2	×	✓
Reentrancy_daoFixed	18		-	×
Reentrancy_daoFixed_Lock	22		-	×
Reentrance	19	4	×	✓
ReentranceFixed	19		-	×
ReentranceFixed_Lock	23		-	×
Reentrancy_simple	16	2	×	✓
Reentrancy_simpleFixed	16		-	×
Reentrancy_simpleFixed_Lock	20		-	×
simple_dao	16	3	×	✓
simple_daoFixed	16		-	×
simple_daoFixed_Lock	20		-	×

(c) Benchmark#3. Summary of analyzed contracts with reentrant calls. The column *Subject* represents either the original vulnerable contract or the fixed version that prevent reentrant calls with a mutex (suffix Fixed_lock) or reordering the statements in the function (suffix Fixed). Column *Approach* can be without transient states (*non-TS*) or with transient states (*TS*). If the abstraction detected a reentrancy bug, the cell contains a ✓; otherwise, it contains a ×. For fixed versions, non-TS mode was not executed and it is represented with "-".

Fig. 14: Subjects. LOC refers to lines of code, and $|M|$ denotes the number of public functions, as measured using Solidity Metric [37].

manually compare it with the documented state machine. We documented differences in terms of extra/missing transitions and states, and validated if these differences were due to errors in the documentation or limitations of our implementation.

To answer **RQ#2**, we consider all the subjects used in [18], which include contracts from the Microsoft Azure Blockchain Workbench, along with the fixed versions provided by [7] (Figure 14a). In addition, [18] also uses a benchmark taken from [10]. This benchmark originally includes 9 contracts, each accompanied by an informal description of their intended behavior; however, no state machines or similar representations are provided. One of the 9 subjects, *RefundableCrowdsale*, was excluded because it failed to compile using the EVM compiler. Figure 14b lists the subjects used from this second benchmark in our study. Note that in [18], multiple abstractions are built for variants of the contracts, column *#abstr* indicates how many abstractions for each subject were used in [18].

To answer **RQ#3**, we had an auditor completely unfamiliar with the benchmark subjects in Figure 14b request predicate abstractions and compare them with their understanding of the source code of the contracts. Through manual inspection, the auditor iteratively requested new abstraction variants by adding predicates, concluding the analysis once confident that no further correspondence issues remained. In all cases, the auditor started with an enumeration abstraction if the contract had enum state variable and an enabledness abstraction.

The abstractions inspected by the auditor were actually constructed using the prototype reported in [18], with the help of one of the authors who supplied the alloy models required by the prototype. The experience is reported in [18]. Replicating the study using PASCo required either finding a new auditor (which we did not have access to). Instead, we replicate the report in [18] and analyze what may have been different if the auditor had used a fully automated tool such as PASCo, comparing the output produced by PASCo and the abstractions shown to the auditor.

To answer **RQ#4.1** and **RQ#4.2**, we took smart contracts with reentrancy vulnerabilities from the known curated dataset used in [32]. This dataset [41] provides a collection of vulnerable Solidity smart contracts organized according to the DASP taxonomy[42]. For reentrancy vulnerabilities, the dataset includes seven smart contracts. Of these, two were not fully supported by PASCo: Modifier_reentrancy and Spank_chain_payment. These contracts use Solidity features that are currently unsupported by Verisol, such as "assembly" blocks or instantiating a contract from an address using calls like "Bank(msg.sender)".

In addition to the five vulnerable contracts, we created two modified versions of each: (i) one that introduces a mutex to prevent reentrant calls, and (ii) another that reorders statements to prevent reentrancy by

avoiding outdated state access. This setup allowed for a comparative analysis between the original vulnerable contracts and their corresponding fixed versions. Figure 14c lists the subjects used to address these research questions.

To answer **RQ#4.1**, we applied the enabledness predicates, along with the predicates $B > 0$ and $BM[A] > 0$. Here, $B > 0$ holds if the contract balance is non-zero, while $BM[A] > 0$ holds if the balance mapping for a specific user A is non-zero. For each contract version, we generated two abstractions: one excluding transient states (as defined in Section 5), and another including them. This approach allows us to validate whether transient states are necessary to detect reentrancy bugs. Finally, for each case, we reported if the generated abstraction was able to reveal the reentrancy bug or not.

To answer **RQ#4.2**, we compare the precision and recall of PASCo against all Solidity vulnerability detectors supported by SmartBugs [43,44], specifically for the detection of reentrancy vulnerabilities on the smart contracts included in Benchmark#3. SmartBugs is a framework for analyzing Ethereum smart contracts that provides a standardized output format that enables automated evaluation. At the referenced commit, the default toolset, running it with parameter `-t all`, includes 14 solidity detectors: conkas [45], osiris [46], securify [47], sfuzz [48], slither [6], confuzzius[49], mythril [50], honeybadger [51], maian [52], manticore [53], oyente [54], semgrep [55], smartcheck [56], and solhint [57]. A timeout of 60 minutes was set for each subject, and reentrancy detection outcomes were determined following the criteria defined in [58].

What follows are a number of methodological considerations common to all research questions.

As mentioned in the previous section, the abstraction construction algorithm assumes a public method implementing function τ. We used an implementation as depicted in Figure 4 that increments *block.number*. Additionally, we modify all constructors to allow the variable to be parametrically set. Note that because *block.number* is a reserved variable, we actually replace all its appearances in with *block_number*.

As previously mentioned, PASCo has two important configuration parameters that must be defined: *k_bound* (representing the maximum transaction sequence length) and *time_out* (the maximum time in seconds for each query).

To calibrate *k_bound* we built abstractions for a number of case studies by incrementing *k_bound* until further increments did not find more transitions. More specifically, we started with a default setting of $k\_bound = 4$ and a fixed time out of 600 seconds for each query. We iteratively ran the tool on each case study, doubling the value of *k_bound* each time, and comparing the results to the previous run. For each case study, we stopped increasing *k_bound* when the doubling did not result in finding new transitions. We chose the final *k_bound* to be the biggest bound that introduced a new transition among the case studies that we analyzed. With this procedure we settled on a *k_bound* of 8 for all subjects except those that included transient states. The latter are used to exhibit reentrancy issues, which tend to require longer traces, thus we used $k\_bound = 16$.

7.2 Results

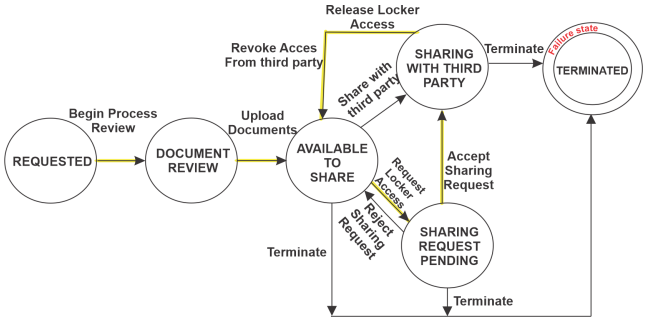
7.2.1 RQ1 To illustrate how we worked with each subject, we first focus on *Digital Locker*. This smart contract allows the sharing of locked documents, with the owner retaining control over access to these documents. In Figure 15a, we present the manually produced state machine as documented for the smart contract.

The enumeration predicate abstraction, constructed using the *enum* values of the contract's variable *state*, is depicted in Figure 15b. Upon initial examination, it becomes apparent that the generated abstraction diverges from the manually provided one. Notably, the generated abstraction features a significantly higher number of transitions than expected. For example, the *Terminated* state exhibits numerous outgoing transitions, whereas the documentation indicates that there are no such transitions. This discrepancy between implementation and documentation suggests weak require clauses within the smart contract, enabling transitions in states that deviate from the intended behavior.

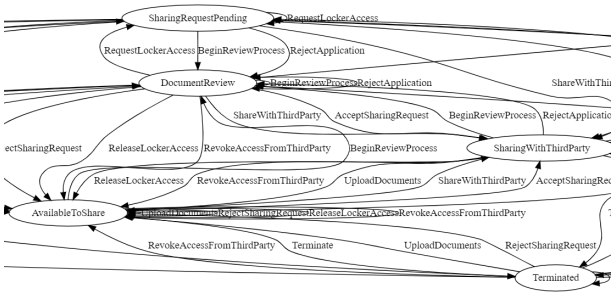
In [7], the corrections applied to the Digital Locker contract primarily involve the addition of require clauses, which intriguingly aligns with the observation made earlier. When using the same predicates employed to construct the abstract model (Figure 15b) to build a predicate abstraction of the code corrected in [7], we notice the emergence of another discrepancy: the presence of a function, the *rejectApplication* function, not documented in the state machine. This discrepancy suggests that the function might not have been intended to be public or could have been inadvertently excluded from the documented diagram. If we disable this function and regenerate the predicate abstraction, we get indeed an isomorphic abstraction to the documented state machine. This is shown in Figure 15c.

Therefore, for the Digital Locker contract, we successfully used the default enumeration abstraction predicates to produce an abstraction comparable to the state machine documented. This process helped us pinpoint correspondence issues between the code and the documentation.

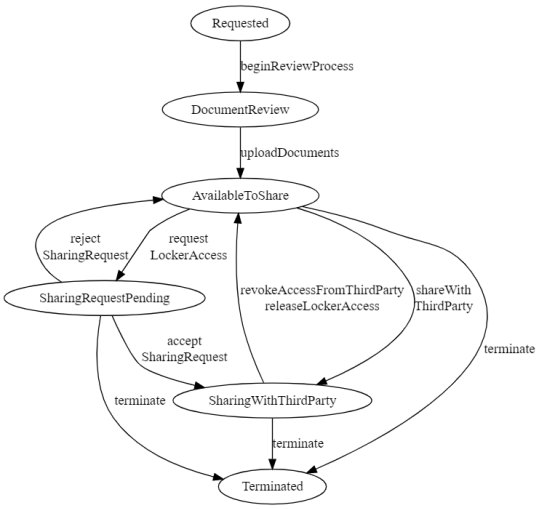
We applied the same procedure to all subjects listed in Figure 14a. In each case, we successfully constructed an enumeration predicate abstraction and were able to manually compare the output of PASCo with the documented state transition diagram. Results are shown in Figure 14a. Column *Diff* indicates the number of states



(a) Digital Locker exactly as documented in [31]. Yellow transitions represent a "*happy path*'.



(b) Enumeration Predicate Abstraction Snippet for the Digital Locker implementation in [38].



(c) Enumeration Predicate Abstraction for Fixed Digital Locker implementation in [7] with *RejectApplication* function disabled.

Fig. 15: Abstractions for *Digital Locker*.

and transitions that differ between the two abstractions. Column *FN* shows the number of states and transitions that the output of PASCo fails to identify but that after manual inspection of the smart contract code, we concluded that they should have been included.

For 7 out of 9 original [38] implementations, we identified mismatches between documentation and implementation. That is, for only two of the original implementations did PASCo not show discrepancies. We also identified in 3 of the 7 fixed and verified contract implementations some discrepancies with the documentation. We discuss these further down.

In only one subject we identified false negative transitions and states. This is the case of *DefectiveComponentCo*, both the fixed and non-fixed versions, where the *k_bound* was insufficient to produce a witness to their existence. We discuss this further below.

Asset Transfer The enumeration predicate abstraction obtained from the fixed version of the contract reveals one transition that is absent in the documented state machine: specifically, an *accept* transition from the *BuyerAccepted* state to the *Accepted* state. Upon code inspection, it was determined that this behavior is indeed feasible and seems to be desirable. In such cases, an auditor should inquire with the contract owner to ascertain whether the implementation contains a bug or if the documented state machine is incorrect.

It is worth noting that the assertions added during the verification process outlined in [7] to guarantee the conformance of the fixed *Asset Transfer* code to the documentation were insufficient in detecting the fact that the code allows for undocumented behavior. This discrepancy was also noted in [8]. This underscores the well-established understanding that verification, as a substitute for validation, is only as effective as the completeness of the properties being verified. As mentioned earlier, it emphasizes the importance of human involvement in validation efforts [13].

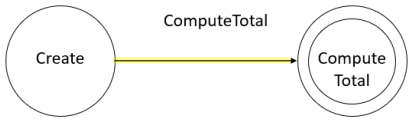
Digital Locker As discussed earlier, the enumeration predicate abstraction generated with PASCo for the fixed version of the Digital Locker subject incorporates transitions for *RejectApplication*. However, the documented diagram does not feature this function, despite its presence in the source code with public visibility. Interestingly, neither [7] nor [8] reported this discrepancy.

DefectiveComponentCounter The documented state machine for this subject is presented in Figure 16a, while the enumeration predicate abstraction obtained from the fixed contract is illustrated in Figure 16b. The generated abstraction omits both the transition *ComputeTotal* and the state *ComputeTotal*. Upon reviewing the code, it was determined that the automatically generated predicate abstraction was incomplete.

The reason behind this discrepancy is that, in order to successfully execute the *computeTotal* method, it is necessary to iterate over an array of length 12 that should have been initialized during deployment. Thus, a minimum of 12 calls to push an element into the array and a call to the constructor are required. In other words, a k_bound lower than 13 would never discover the missed state. We manually adjusted the k_bound parameter to 13, resulting in the addition of the missing transition and state, as depicted in Figure 16c. This highlights

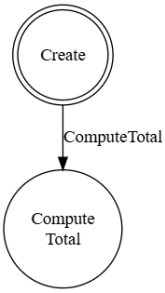
the limitations of the current approach, demonstrating that sub-approximating predicate abstractions may fail to capture certain states or transitions. This underscores the trade-off involved in fully automating the process.

Answer to RQ1: Using the default predicates we were able to produce abstractions comparable to those specified manually for all 16 case studies. That is, the abstractions built by PASCo yielded states that corresponded to those of the documentation. For 10 out 16 subjects we found discrepancies with the documentation and all but one were documentation issues. The exception was related to the use of a low $k - bound$ when building the abstraction. This provides evidence that PASCo results in state machines that describe the behavior of smart contracts at the level of abstraction that users sometimes think of and document contracts.



(a) DefectiveComponentCounter Documented State Machine [59].

(b) Enumeration Predicate Abstraction for Fixed DefectiveComponentCounter [7] with $k\_bound = 8$.



(c) Enumeration Predicate Abstraction for Fixed DefectiveComponentCounter [7] with $k\_bound = 13$.

Fig. 16: Abstractions for *Defective Component Counter*.

7.2.2 RQ2 In total, we analyzed 39 subjects. We compared the abstractions generated with PASCo against those from [18]. All disparities were followed up with a manual inspection. In benchmark#1, only three subjects exhibited disparities: *DefectiveComponentCounter*, *DefectiveComponentCounterFixed*, and *DigitalLocker*. While in benchmark#2, we found 8 discrepancies (out of a total of 23).

DefectiveComponentCounter(original and Fixed versions) A missing transition was identified in the abstraction generated by PASCo. This transition is the same one as mentioned in RQ1 as a false negative. The omission is due to an insufficiently large value of k_bound: due to the array used in the implementation (size 12) a sequence of 13 calls is required to identify the missing transition. In [18], the manually constructed Alloy model assumed an array of length 3. Both reducing the array in the implementation to 3 or increasing the k_bound to 13 makes PASCo yield abstractions identical to that of [18].

Digital Locker For this subject, we identified a missing transition in the abstraction produced by PASCo that was present in the abstraction of [18]. Upon examining the source code, we confirmed that indeed the abstraction from PASCo was correct: the absent transition cannot be called successfully from the abstract state. When inspecting the Alloy model that did exhibit this transition, we found that the invariant modeled in Alloy was too weak, allowing a function call that is actually not possible. The reason this additional transition was not identified as an error is that the abstraction differed significantly from the expected one, making it unnecessary to perform an in-depth analysis to recognize the bug in the code.

EPXCrowdsale In EPXCrowdsale, the three abstractions reported in [18], built with different abstraction predicates, had differences with those built directly from code using PASCo. As with the Digital Locker, we confirmed that the PASCo versions were correct and that the errors were due to a weak invariant in the Alloy model of EPXCrowdsale. The difference exemplifies the problem of having users resolve complex and error-prone tasks such as invariant inference. The subject has many complex conditions leading to an intricate invariant. Because PASCo creates transactions starting from the constructor, all the executions represent feasible traces where the contract invariant holds.

RefundsCrow Out of the three abstractions reported in [18] for this subject, we found a discrepancy with PASCo for only one of them, the abstraction built with the enabledness abstraction predicates. Manual inspection revealed a missing precondition for the *withdraw* method in the Alloy specification. After adding this precondition in the Alloy model, the output was equivalent to the output of PASCo. This case exemplifies the problem of manual specification of solidity code to more abstract and analyzable models.

ValidatorAuction For this subject, two out of three predicate abstraction choices in [18] yield abstractions different from those of PASCo. In both cases, the root causes were simplifications of the contract behavior when modelled in Alloy, primarily due to the complexities involved

in the interaction with another contract. Additionally, the Alloy models had weak preconditions. It is important to note that the enumeration predicate abstraction was the only choice of predicates that yielded equivalent abstractions, despite the inaccuracies in the Alloy model. This is due to the coarse-grained abstraction from enumeration mode. This underscores the importance of generating different abstractions, where broader abstractions are easier for an auditor to comprehend, while more refined abstractions provide more detailed insights and often have a higher likelihood of uncovering errors.

Auction For this subject, two out of three predicate abstractions showed differences. In both cases, the discrepancies were due to a missing invariant condition in the Alloy model related to the state variable *ended*. Only one refined abstraction was equivalent, despite the weak invariant. This equivalence was related to a bug in the contract, specifically in the method *auctionEnd*. Due to this bug, the contract can never reach a state where the auction has ended. As a result, the only refined abstraction that was equivalent in [18] and PASCo was the one that included a predicate on the state variable *ended*. Consequently, both abstractions only represent a slice of the abstraction where *ended=False* holds, and within this slice, the weak invariant does not present an issue.

Answer to RQ2: PASCo successfully automated the construction of all previous abstractions generated by the Alloy prototype, eliminating the need for manual translation from Solidity to Alloy language. This also obviates the requirement for users to input a contract invariant. Out of the 39 different abstractions analyzed, 11 exhibited disparities. Among these 11 differences, most were attributed to issues in the manual translation process from Solidity to Alloy, with the predicate abstractions generated by PASCo proving to be accurate. Only 2 of these differences involved false negative transitions in the abstractions generated by PASCo.

7.2.3 RQ3 We discuss each subject separately.

Crowdfunding This subject and the auditor findings are discussed in detail in Section 2 where the code is explained and Section 4 where the various abstractions presented to and required by the auditor are explained, together with the underlying reasoning that the auditor indeed went through, and recommendations they made. Note that as the contract does not have a state variable with an enumeration, the only initial abstraction provided was the enabledness predicate abstraction of Figure Figure 3a.

The abstractions using PASCo were identical, so we can assume that the conclusions would have remained the same.

ValidatorAuction This contract implements an auction where a limited number of participants can bid only if they were previously whitelisted. Both the enumeration and enabledness predicate abstractions are presented to the auditor.

Observing the enumeration abstraction, the auditor considers suspicious the fact that transitions labelled *closeAuction* and *bid* arrive at the same abstract state (*DepositPending*), in which the auction owner can request that the winning bid be deposited to their account. Indeed, upon code inspection, it can be seen that *bid* changes the state of the contract to *DepositPending* when a maximum number of bids is achieved, which is the expected behavior.

In addition, the auditor identifies that despite the fact that informal specification for the contract states that "*a user withdraws in the DepositPending state, they will eventually be sent the sum of their bids minus the lowest price.*", state DepositPending has no outgoing transitions labelled *withdraw*. The actual contract only allows withdrawals in *failed* and *ended* states. Hence, a mismatch between specification and implementation has been found.

Upon inspection of the enabledness predicate abstraction, the auditor observes a looping transition labeled *withdraw* on one of the states. Participants that did not win the auction get their funds back with this function. To establish if each participant can withdraw only once, the auditor adds a new function to the contract by duplicating withdraw and differentiating the copies with a clause that requires the message sender to be or not be a particular address (defined as a constant *AddrA in the code*). Thus, the contract has two functions *withdrawA* and *withdrawNotA*. The refined abstraction of the contract shows that *withdrawA* does not loop, which lets the auditor establish that withdrawals from the same bidder cannot be done more than once.

Although the auditor was fed abstractions built using the semi manual tool reported in [18], had the auditor been shown the abstractions built with PASCo, this would have led, most likely, to the same conclusions. The enumeration predicate abstraction built by PASCo is identical to the one shown to the auditor. The withdraw loop is not present in the abstraction generated by PASCo, which is correct, and the refined abstraction also shows that *withdrawA* does not loop, leading the auditor to reach the same conclusion.

Auction This auction contract differs from the previous one, among other things, in that there is no enum state. Thus, the auditor is provided only one initial abstraction based on the requires clauses.

Using the abstraction, the auditor identifies a state in which the only function enabled is *auctionEnd* and that its transition loops back to the same state. The possibility of ending the auction multiple times is considered suspicious and upon inspection of the code, it is possible

to see that the function has a statement `ended = true` which presumably should then disable any future call to the same function but this does not seem to be the case.

The auditor requests a refined abstraction with the predicate `ended = true` and obtains a partitioned graph in which from the initial state it is impossible to reach any abstract state in which ended is true. This means that the variable is being incorrectly updated and is reported as a concern by the auditor. As with previous subjects, a looping *withdraw* transition prompts the auditor to split the *withdraw* function into two functions: *withdrawA* and *withdrawNotA* and also a predicate that asserts if the AddrA has any committed funds (i.e., `backer[AddrA] > 0`). The resulting abstraction shows *withdrawA* reaching a state in which `!backer[AddressA] > 0`. This allowed the auditor to conclude that any address that bids will be able to do exactly one withdrawal, which seemed fine for all addresses except the winning one, who should not be allowed to withdraw.

To investigate the latter concern, a similar abstraction is requested but instead of fixing the message sender of the withdraw function to `AddrA` it is set to be equal to the *highestBidder* variable. The resulting abstraction confirms that the highestBidder can withdraw if it has made more than one bid (e.g., if a bidder commits 10 and then 20, the auction accumulates 30 but the highest bid is considered to be 20, thus upon winning, the bidder gets 10 back).

Two out of the three abstractions for this case study differed when using PASCo. The abstraction with the predicate `ended = true` was identical: in both cases, all reachable states showed that *auctionEnd* method is disabled. This behavior was due to a bug in the method that prevented it from being executed successfully. However, in the enabledness abstraction and the abstraction splitting *withdraw* into *withdrawA* and *withdrawNotA*, *auctionEnd* appeared in the abstractions generated by Alloy but not in those generated by PASCo. Upon inspecting the Alloy model, we found that a weak invariant that allowed a state to be instantiated where *auctionEnd* was enabled, even though it was not reachable from the constructor. After fixing the weak invariant, all abstractions became identical. Consequently, we believe that using PASCo would likely lead to the same final conclusions but would have avoided the auditor's concern upon seeing the *auctionEnd* method transitioning back to the same state.

Simple Auction This auction contract does not have an enum state either. Starting from the enabledness predicate abstraction, the auditor finds a state in which *auctionEnd* and *bid* are both enabled. This is unexpected as *auctionEnd* should be enabled only when the auction period has ended while bidding should be enabled when the period has not ended. Inspecting the require clauses for the functions reveals that indeed there is an overlap in the enabling conditions of the two functions: *bid* requires `_now<=_auctionStart+_biddingTime` while *auctionEnd* requires `_now>= _auctionStart+_biddingTime)`. Thus, when `_now` equals to `_auctionStart` plus `_biddingTime` both functions are enabled.

As pointed out by the auditor, this represents a potential vulnerability: An attacker might monitor for *auctionEnd* and once observed it may put in quickly and manipulate the order in which the blockchain registers the transactions to record that their bid happened before the *auctionEnd*. This would allow the attacker to ensure it is the highest bidder.

In addition, from the same state in which both functions are enabled, the occurrence of *auctionEnd* can lead to a state in which *bid* is still enabled. This is because bidding is disabled by the passage of time and not *auctionEnd*. This means that an unlucky bidder that bids just after the auction end, but before time advances will have their funds blocked: Being the highest bidder, they will not be allowed to withdraw. Furthermore, the auction owner having ended the auction before and withdrawn the funds of the previous highest bidder, will not be allowed to end the auction again to withdraw the latest bid.

The abstractions built by PASCo are identical to those shown to the auditor, hence using PASCo would have led the auditor to the same conclusions.

EscrowVault The contract supports an escrow account in which funds are held until the owner signals the goal has been achieved, point at which funds are released. This contract has an enum state variable. However, the enumeration predicate abstraction and the enabledness predicate abstraction are identical. The auditor observed in the initial abstraction that function *withdrawAll* is enabled and can loop in state *GoalReached*. Inspection of the code shows that this is because the first call will empty the balance and the remaining calls are accepted but the amount withdrawn is zero. The auditor recommended adding a clause requiring the contract balance to be greater than zero to avoid unnecessary gas consumption of multiple calls; however, the additional require clause would imply an extra overall cost if the function is called only once. Given that the only address allowed to execute the method is the owner, multiple calls to withdrawal seem unlikely and in any case this is a minor issue.

The abstractions using PASCo were identical, so we can assume that the conclusions would have remained the same.

RefundEscrow This subject is similar to *EscrowVault*. The major difference is that refunds are triggered exclusively by the contract owner. As with the previous contract, the enumeration predicate abstraction and the enabledness predicate abstraction are identical. A looping withdraw function, as before, prompted splitting the function for a fixed message sender. This did not yield

any findings, multiple withdrawals from the same address are not allowed. On the other, the multiple *withdrawBeneficiary* calls on the same abstract state are indeed possible because, as with the previous contract, it is not restricted to when the balance is greater than zero.

The abstractions generated by PASCo differed from those presented to the auditor only in the enabledness abstraction. The discrepancy arose from a missing precondition in the *withdraw* method in the Alloy model, which allowed, for example, a looping *withdraw* transition that raised the auditor's suspicion described above.

We believe the conclusions for this subject would have remained similar since the abstraction generated by PASCo only removed the looping *withdraw* transition that caused the auditor's suspicion.

EPXCrowdsale A crowdsale contract is akin to crowdfunding but where funds are collected in exchange for token. To validate this crowdsale, the auditor started with the enabledness predicate abstraction (there is no enum state variable) and requested a refinement using a predicate over a boolean variable of the contract (*isCrowdSaleClosed*). The resulting abstraction shows that it is possible to call *refund* on a state of the contract in which the crowdsale is closed and to reach a state in which it is open. This is clearly undesirable.

However, upon inspection, it is not clear to the auditor that the closed state from which function *refund* reopens the crowd sale is actually reachable through successive calls to contract functions. This is a characteristic of the predicate abstractions that are shown: they provide a superset of the actual behaviors of the contract. This additional transition could be a consequence of this.

To validate if this transition is a result of an overapproximation of actual behavior of the contract, the auditor proposes an additional predicate taken from function *checkGoalReached*. The function transforms the state of the contract, and in particular that of *isCrowdSaleClosed* based on a number of complex conditions in a nested if statement. From these conditions, the auditor extracts a predicate of the form `(isCrowdSaleClosed => (....) )` which is posed as a potential (and partial) state invariant.

The refined abstraction obtained using the proto-invariant reveals two things. First, the fact that all reachable states of the abstraction correspond to states where the predicate is true indicates that indeed the new predicate is an invariant of the contract. Second, the transition that connected a state in which *isCrowdSaleClosed* is false to one that is true disappeared, confirming that the behavior was spurious.

The abstractions using PASCo differed across all three abstractions compared to those generated by the prototype. However, the refined abstraction obtained with the proto-invariant was nearly identical to the generated by PASCo (although even with the proto-invariant, a transition was missing). Therefore, we assume that using PASCo would provide a better experience for the auditor, as it would avoid false negatives and eliminate the need for the auditor to manually provide an invariant, which is a complex and error prone task.

RockPaperScissors This is a contract for playing the rock, paper and scissors game and awarding prizes for winners. The auditor started from the enabledness predicate abstraction (there were no enum state variables) and concluded that although there were no elements of concern, the abstraction required further refinement to exhibit more interesting behavior regarding the game winning logic. To this end a new abstraction was built partitioning the state space based on which player is the winner. The resulting abstraction showed suspicious behavior: Two different states in which betting is open but that differ in who is the winner were connected by *determineWinner* transitions. This should not be the case as *determineWinner* should establish who is the winner but not change it. A new refinement adding a predicate stating that *player1* and *player2* correspond to different addresses showed that this suspicious behavior is only possible when it is the same player playing against itself. The auditor did observe a defect in the contract through inspection of the code, but this observation did not follow from analysing predicate abstractions. The auditor noted that the contract does not allow receiving funds, as the constructor does not include the modifier `payable`. Thus, winners can never have any funds transferred to them. This can be observed in an abstraction if the predicate `balance>0` is included. The resulting abstraction has no reachable states in which the predicate is true.

The abstractions built by PASCo are identical to those shown to the auditor, hence using PASCo would have led the auditor to the same conclusions.

Answer to RQ3: The experience with these subjects provides evidence that auditors may benefit from reviewing successive predicate abstractions to validate the correspondence of smart contracts behavior against an informal description (or their mental model) of the intended contract behavior. In all cases, exploring the abstractions led to the identification of issues, many of which were actual defects or mismatches with the documentation that were not reported in previous works.

Although the actual experience of the auditor was through an assisted semi-automated tool, for 16 case studies the experience of using PASCo would have been identical. For the rest, the differences between the PASCo output and the models presented to the auditor were minor and we believe would not have affected substantially the process that led to auditor findings. In all cases the differences were due to errors in the modeling of the contract in Alloy and not due to problems of PASCo.

7.2.4 RQ4.1 Figure 14c shows a summary of the abstractions built for contracts with reentrancy. For all subjects, generating the abstraction for the original version without transient states did not reveal any suspicious behavior. When applying the approach introduced in Section 5 that includes transient states, the abstraction generated for the original contract version exhibits a suspicious state in every subject. This suspicious state consistently refers to an abstract state where the predicate $BM[A] > 0$ holds, but predicate $B > 0$ does not. In other words, the abstraction shows an abstract state in which user A has a balance available to withdraw, but the balance of the contract is zero. This suggests that the balance was drained due to the reentrancy bug.

For the two fixed versions, the abstraction did not exhibit any suspicious states in any of the subjects, which aligns with the expected outcome.

It is worth noting that many existing tools suffer from high rates of false negatives and false positives in reentrancy detection [20]. In this regard, the current results are promising, as they highlight the potential of predicate abstractions to accurately detect reentrancy vulnerabilities while minimizing false outcomes.

Answer to RQ4.1: Incorporating *transient states* into predicate abstractions enables a more refined abstraction capable of revealing issues such as reentrancy bugs, which may otherwise go undetected without considering these intermediate states.

7.2.5 RQ4.2 Table 1 presents a comparative analysis of PASCo against the 14 default Solidity security detectors integrated within SmartBugs [44].

As previously discussed, each smart contract in Benchmark#3 has three distinct versions: the `original` version, which is the default contract with a known reentrancy vulnerability; the `Fixed` version, a patched contract where the reentrancy bug is mitigated by reordering statements to prevent access to outdated states (i.e., this version still allows reentrant calls but eliminates the associated vulnerability); and the `Fixed_Lock` version, an alternative patched contract that employs a mutex to completely eliminate the possibility of reentrant calls. This enabled an evaluation of reentrancy detection tools, differentiating between scenarios where reentrancy is semantically safe (i.e., permitted but not exploitable) and those where it is statically prevented (via a mutex).

The tools in Table 1 are sorted in descending order of precision. If the tool reports the *expected* outcome, the corresponding cell is marked with $\checkmark_{TP}$ (True Positive) or $\checkmark_{TN}$ (True Negative). Conversely, if the tool reports an *unexpected* outcome, the cell is marked with $\times_{FP}$ and $\times_{FN}$ for False Positive and False Negative, respectively.

Results show that five tools exhibited 0.0% precision and recall. Among them, two tools (Honeybadger, Main) do not support reentrancy detection, according to their documentation. The remaining three (Manticore, Semgrep and Solhint) claim to support it, but failed to detect any reentrancy bug using the default configuration of SmartBugs.

Most other tools achieved high recall (60% minimum), but low precision (33.3% minimum), indicating a high rate of false positives. For instance, the fixed version of the `Simple_dao` contract was incorrectly flagged by six tools (Conkas, Osiris, Sfuzz, Confuzzius, Mythril, and Smartcheck). This contract includes a `withdraw` function (shown in Figure 17) that allows users to retrieve previously donated funds, constrained by a withdrawal limit parameter. Although the function permits reentrant calls, this behavior does not constitute a bug.

```
function withdraw(uint amount) {
  if (credit[msg.sender]>= amount) {
    credit[msg.sender]-=amount;
    bool res = msg.sender.call.value(amount)();
  }
}
```

Fig. 17: False Positive example that is usually classified as a reentrancy bug by many tools.

The top three best detectors in terms of precision and recall are PASCo, Oyente, and Conkas. All achieved 100% recall, but only PASCo and Oyente reached 100% precision. Conkas showed a lower precision (62.5%) due to false positives in cases with patterns similar to the one in Figure 17. Interestingly, both Conkas and Oyente employ symbolic execution to perform verification tasks and look for specific reentrancy patterns.

Answer to RQ4.2: This RQ strengthens findings from prior studies (e.g., [20]) that report high false-positive rates in reentrancy detection. The proposed tool PASCo achieves both high precision and recall, standing out in comparison to existing state-of-the-art detectors. While many tools exhibit high recall at the expense of precision, PASCo maintains 100% precision without sacrificing recall. This suggests that the abstractions generated by PASCo with *transient states* are both expressive and accurate enough to capture subtle contract behaviors, such as reentrancy bugs.

7.3 Performance

In this section, we provide insights into the time required to generate predicate abstractions using PASCo. The tool incorporates two main optimizations: it discards unreachable states at an early stage, as detailed in Subsection 6.2, and it parallelizes queries to determine the existence of transitions.

All experiments were conducted on a computer with an Intel Core i7-4790 processor (4 cores), 16GB of RAM,

Subject	Status	PASCo	Oyente	Conkas	Osiris	Securify	Sfuzz	Slither-0.10.4	Confuzzius	Mythril-0.24.7	Smartcheck	Honeybadger	Maian	Manticore-0.3.7	Semgrep	Solhint-3.3.8
Etherstore	original	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\times_{FN}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\times_{FN}$	$\times_{FN}$	$\times_{FN}$	$\times_{FN}$	$\times_{FN}$
	Fixed_lock	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\times_{FP}$	$\checkmark_{TN}$	$\times_{FP}$	$\times_{FP}$	$\times_{FP}$	$\times_{FP}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$
	Fixed	$\checkmark_{TN}$	$\checkmark_{TN}$	$\times_{FP}$	$\times_{FP}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\times_{FP}$	$\times_{FP}$	$\times_{FP}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$
Reentrance	original	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\times_{FN}$	$\times_{FN}$	$\times_{FN}$	$\times_{FN}$	$\times_{FN}$
	Fixed_lock	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\times_{FP}$	$\checkmark_{TN}$	$\times_{FP}$	$\times_{FP}$	$\times_{FP}$	$\times_{FP}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$
	Fixed	$\checkmark_{TN}$	$\checkmark_{TN}$	$\times_{FP}$	$\times_{FP}$	$\checkmark_{TN}$	$\times_{FP}$	$\checkmark_{TN}$	$\times_{FP}$	$\times_{FP}$	$\times_{FP}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$
Reentrancy_dao	original	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\times_{FN}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\times_{FN}$	$\times_{FN}$	$\times_{FN}$	$\times_{FN}$	$\times_{FN}$
	Fixed_lock	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\times_{FP}$	$\checkmark_{TN}$	$\times_{FP}$	$\times_{FP}$	$\times_{FP}$	$\times_{FP}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$
	Fixed	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\times_{FP}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\times_{FP}$	$\times_{FP}$	$\times_{FP}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$
Reentrancy_simple	original	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\times_{FN}$	$\times_{FN}$	$\times_{FN}$	$\times_{FN}$	$\times_{FN}$
	Fixed_lock	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\times_{FP}$	$\checkmark_{TN}$	$\times_{FP}$	$\times_{FP}$	$\times_{FP}$	$\times_{FP}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$
	Fixed	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\times_{FP}$	$\checkmark_{TN}$	$\times_{FP}$	$\checkmark_{TN}$	$\times_{FP}$	$\times_{FP}$	$\times_{FP}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$
Simple_dao	original	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\checkmark_{TP}$	$\times_{FN}$	$\times_{FN}$	$\times_{FN}$	$\times_{FN}$	$\times_{FN}$
	Fixed_lock	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\times_{FP}$	$\checkmark_{TN}$	$\times_{FP}$	$\times_{FP}$	$\times_{FP}$	$\times_{FP}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$
	Fixed	$\checkmark_{TN}$	$\checkmark_{TN}$	$\times_{FP}$	$\times_{FP}$	$\checkmark_{TN}$	$\times_{FP}$	$\checkmark_{TN}$	$\times_{FP}$	$\times_{FP}$	$\times_{FP}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$	$\checkmark_{TN}$
Precision		5/5	5/5	5/8	5/10	5/10	3/6	5/10	5/15	5/15	5/15	0/0	0/0	0/0	0/0	0/0
Recall		5/5	5/5	5/5	5/5	5/5	3/5	5/5	5/5	5/5	5/5	0/5	0/5	0/5	0/5	0/5

Table 1: Comparative analysis of reentrancy detection tools. We evaluate **Precision** (TP/(TP+FP)) and **Recall** (TP/(TP+FN)) metrics. Here, $\checkmark_{TP}$ and $\checkmark_{TN}$ denote True Positives and True Negatives, respectively, while $\times_{FP}$ and $\times_{FN}$ denote False Positives and False Negatives, respectively.

running Windows 10. PASCo generated abstractions in under 3 minutes for 55.93% of subjects. For the remaining cases, computation times were distributed as follows: 15.25% took 3–10 minutes, 5.08% required 10–15 minutes, 16.95% needed 15–30 minutes, and 6.78% took 30–60 minutes. Across all runs, the fastest completion time was just 5 seconds, while the longest took nearly 41 minutes. The median time stood at 1.37 minutes, with an average of 8 minutes.

To assess the impact of parallelization on computation times, we analyzed the number of queries and their individual execution times. We separated subjects requiring reentrancy analysis from those that did not. Reentrancy analysis involves a more exhaustive approach, as described in Section 5, and it was conducted using $k\_value = 16$, which differs from the rest of the subjects that use $k\_value = 8$.

Table 2 summarizes the query analysis, including the number of queries for each subject and their execution times (in seconds) for each query. The table shows that the number of queries per subject, in average, is higher in reentrancy mode (188.06) compared to non-reentrancy mode (125.61). Furthermore, queries that were stopped due to timeouts occurred mainly in reentrancy mode, leading to significantly higher average execution times for these queries: 40s in reentrancy mode versus 6s in non-reentrancy mode.

To assess the potential improvements achieved through parallelization, we simulated execution times for a 16-core system, which represents a powerful, but mainstream, hardware setup. To estimate execution times, we sampled 1000 times how long it would take to execute $\frac{1}{16}$th of the queries of each subject. With this simulated setup, PASCo generates abstractions in less than 3 minutes for 71.19% of the subjects, between 3 and 10 minutes for 22.03%, and between 10 and 15 minutes for 6.78%. No subject required more than 15 minutes. The minimum computation time was 1.32 seconds, the average was 2.52 minutes, the median was 16.19 seconds, and the maximum time was 14.52 minutes. While this simulation does not account for potential parallelization overhead, which we believe would be minimal as integration of results from different threads is trivial, it provides insight into the possible performance improvements. Further experiments are necessary, but completing the most complex subject in under 15 minutes with 16 cores, and the potential to incrementally display the abstraction as it is computed, provides some evidence that the tool may be practical to support auditing processes.

8 Threats to Validity

Internal validity Although the preconditions are primarily derived from the require statements from the solidity source code, they may be incorrect. To mitigate this risk, we compare the abstraction generated by PASCo with that produced in [18]. If the extracted preconditions are incorrect, this would lead to discrepancies in the generated abstraction. Moreover, we manually reviewed any differences found during the comparison with prior abstractions. However, there remains the possibility that some differences were not detected. To address this, we have made all newly generated abstractions publicly available for further scrutiny in [60].

Mode	#queries						Query time (secs)				
	total	**avg**	**median**	**min**	**max**	**TO**	**total**	**avg**	**median**	**min**	**max**
no-reentrancy	5527	125.61	54.5	6	1010	10	33,922.13	6.13	3.97	1.36	600
reentrancy	2821	188.06	168	100	340	161	113,707.14	40.1	3.53	1.42	600

Table 2: The table summarizes the number of queries and query times for analyzed subjects, distinguishing between those requiring reentrancy analysis and those that do not. Column under **#queries**: *total* represents the total number of queries across all subjects; *avg* the average number of queries per subject; *median* the median number of queries per subject; *min* and *max* the minimum and maximum number of queries of a subject; **TO** is the total number of timeouts. Under **Query time (secs)**, column *total* refers to the total query execution time across all subjects; *avg* the average query time; *median* the median query time; and *min* and *max* the minimum and maximum query times.

The choice of parameter k is a non-trivial limitation. A lower k value improves efficiency by reducing execution time but increases the likelihood of false negatives. Conversely, a higher k reduces false negatives but may increase execution time and the risk of timeouts. In our experiments, we observed one false negative and found that 1.46% of the total queries resulted in timeouts. While these metrics are encouraging, further research is needed to dynamically balance k based on contract characteristics to optimize performance and accuracy.

The comparison with other Solidity detectors using SmartBugs may be affected by the parameter settings and the execution environment. In particular, we relied on the default configuration provided by SmartBugs for each tool, which may not reflect their optimal performance. Some tools might benefit from fine-tuning or custom inputs to better detect reentrancy vulnerabilities. Additionally, we imposed a fixed timeout of 60 minutes per subject, which may have limited the execution of certain detectors with higher computational demands. While this setup ensures a fair and reproducible comparison under consistent conditions, it may underestimate the full potential of some tools. Moreover, it is worth noting that the analyzed subjects cover only a single, yet very common, type of reentrancy vulnerability (balance-draining reentrancy). Certainly, different types of reentrancy vulnerabilities may require auditor provided domain specific abstraction predicates.

External validity To mitigate the risk of cherry-picking, we selected three well-established benchmarks from the literature [8,10,32]. They are widely used and offer a diverse range of examples. However, the conclusions drawn from analyzing these benchmarks may not fully generalize to real-world industrial use cases.

The scalability of PASCo remains an open question. While potential improvements could involve fully parallelizing the tool, the trade-off between precision and scalability when adjusting the k parameter requires further investigation.

9 Related Work

Our work is inspired by the use of enabledness preserving abstractions (EPAs) for validating software artifacts [29,26]. EPAs were successfully applied to the validation of APIs specified in the form of pre-post conditions [29] and programs [26]. EPAs have also been used for testing program with rich protocols [17]. Our abstractions differ from EPAs in several aspects: EPAs rely on *function preconditions*, while herein any set of predicates can be used to partition the state space (e.g., quotienting with enum state variables produced revealing abstractions). We also consider the global state to model part of the blockchain and incorporate τ operations to model changes to the global state that are unobservable to the contract.

As mentioned, our abstraction approach is a particular instantiation of predicate abstraction [25]. However, predicate abstraction has been mostly used for verification purposes rather than validation, hence the level of abstraction, the size of the resulting model, and the challenge of traceability with the original artifact vary. Handling the abstraction at the level of function calls drastically reduces the state space and facilitates validation at the cost of producing coarser abstractions, less amenable for verification.

The most related work is [18], which uses Alloy to generate predicate abstractions for smart contract validation. However, this approach requires the manual translation of Solidity code into the Alloy language. Additionally, an invariant must be provided by the user. The construction algorithm in [18] is similar to the one presented in this work, with notable differences. The main distinction is that, in the prior approach, each query required checking whether the user-provided invariant was maintained. This ensured that concrete instances could be created without relying on a specific sequence of function calls, as compliance with the invariant was always guaranteed. In contrast, our current approach does not require a contract invariant as input. Instead, it ensures that every counterexample is reachable by first passing through the constructor and then calling functions of the contract. This has the advantage of eliminating the need for users to manually specify invariants, a process that is often complex and error-prone. However, it also introduces a drawback: VeriSol must construct instances solely through sequences of calls, which can be computationally expensive. For example, to analyze a bounded stack with an abstract state $stack.length > 50$, a se-

quence of at least 51 transactions (`constructor` + 50 `push()` calls) is required to reach the desired state. If the user specifies a small *k_bound*, such a state may become unreachable. Despite this limitation, we believe that our approach is more practical for real-world use cases, as requiring users to define invariants has historically been a barrier to achieving fully automated tools. Additionally, we enable detection of reentrancy bugs, which the earlier prototype [18] could not detect.

Our technique is related to approaches that infer typestates [61] or interfaces [62] from a program. Again, they are typically used for verification purposes. The models are less permissive than our abstractions in terms of the possible behavior, allowing only valid traces. There has also been work that aims at producing behavior models from execution traces [63]. These techniques infer specifications which are then used for test case generation and have a dynamic flavor, thus heavily dependent on the quality of the traces used as input.

Some approaches use models [64] or formal specification [65] to construct correct-by-design smart contracts. They try to produce safer contracts by construction. Some works [66,67] use automata-based models to specify properties that are then used for runtime verification and monitoring. Automatically generating smart contracts models as EFSM was proposed in [68]. They generate a model from the abstract syntax tree of the smart contract for non-blocking verification.

Predicate abstractions are related to a state machine related to UML protocol state machines [69] which are used to specify the legal usage of a classifier (such as a class). Protocol transitions correspond to model operations and protocol states are stable situations of the contract that satisfy a particular valuation of the abstraction predicates chosen by the user. A key difference is that we allow exhibiting state changes (τ) that are not the result of an operation over the contract.

In the last years a plethora of tools, languages and techniques has been proposed to facilitate the development, analysis and verification of smart contracts [70]. Most of them are aimed at finding vulnerabilities by looking into specific patterns (e.g., [6,71]), user provided assertions (e.g, Echidna [9], Manticore [72], etc.) or specification (e.g., Celestial [73], Verisol [8], VerX [11]). Other approaches, semi-automated [74] and fully automated [75], based on transforming smart contracts into coloured petri nets (CPN) have been proposed. Unlike our approach, which is aimed at human validation, these two approaches are intended for verifying the smart contract by using a model checker (e.g, LTL properties).

More recently, deep learning approaches for detecting vulnerabilities in smart contracts have gained significant attention. Often framed as a Natural Language Processing (NLP) task, these methods represent smart contract source code as vectors, which are then fed into deep learning models trained on vulnerability datasets. This enables the identification of domain-independent bugs (e.g., [76,77,78,79,80]). However, some works focus on domain-specific vulnerabilities, as we do. For instance, GPTScan [81] utilized GPT to match candidate vulnerable functions based on code-level scenarios (which describe the conditions under which a logic vulnerability might arise) and properties (which characterize the attributes of vulnerable code). GPT is further instructed to recognize key variables and statements, which were then validated through static analysis. While their technique might automate the generation of some scenario and property sentences, they conclude that constructing effective prompts for GPT-based recognition remains a manual process. Additionally, auditors using GPTScan must have prior knowledge of the specific type of vulnerability they are searching for. In contrast, our approach allows auditors to iteratively analyze abstractions, enabling the discovery of potentially unknown behaviors.

10 Conclusions and Further work

In this paper, we propose an approach to support auditors in validating smart contract behavior by contrasting it against intended behavior derived from informal requirements. Our approach leverages predicate abstractions to represent calls to public contract functions, with predicates consisting of default predicates extracted from the contract's source code (e.g., *require* clauses and *enum* state variables) and auditor-provided predicates. We conducted an evaluation, using a fully automatic tool PASCo, demonstrating that default predicate abstractions can produce results comparable to those provided by humans and can identify conformance issues between the code and informal requirements. Additionally, we report on a study with an experienced auditor, showing that default abstractions and auditor observations can guide the creation of more refined abstractions that reveal conformance issues. While this study was conducted with a previous prototype, we analyzed the differences and concluded that the results would likely hold with the current tool. Furthermore, our evaluation provided evidence that the abstraction generated by PASCo using transient states can exhibit reentrancy bugs. Specifically, we evaluated PASCo against 14 state-of-the-art Solidity vulnerability detectors, comparing their performance in terms of precision and recall. PASCo ranked among the top two tools (alongside Oyente) achieving the highest precision and recall across the evaluated subjects.

Future work includes improving the performance of the tool in terms of execution time and exploring the integration or replacement of the Verisol component with complementary tools, such as symbolic execution engines, fuzzing tools, or model checkers. Additionally, incorporating automated predicate suggestion and refinement mechanisms would greatly enhance the tool's usability. Finally, a human study is necessary to further validate the effectiveness of the approach for auditors.

Acknowledgements

This work was partically funded by ANPCYT PICT 2019-1442, 2019-1973, 2021-4862, UBACYT 2020-0233BA, 2023-0134BA, CONICET PIP 1220220100470CO, and Sui Foundation Academic Research Award.

References

1. Kouhizadeh, M. & Sarkis, J. Blockchain practices, potentials, and perspectives in greening supply chains. *Sustainability* **10** (10), 3652 (2018) .
2. Wood, G. *et al.* Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* **151** (2014), 1–32 (2014) .
3. Magazzeni, D., McBurney, P. & Nash, W. Validation and verification of smart contracts: A research agenda. *Computer* **50** (9), 50–57 (2017). https://doi.org/10.1109/MC.2017.3571045 .
4. The dao smart contract (2016). URL http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413.
5. Analysis of the dao exploit (2016). URL http://hackingdistributed.com/2016/06/18/analysis-of-thedao-exploit/.
6. Feist, J., Grieco, G. & Groce, A. *Slither: A static analysis framework for smart contracts*, 8–15 (2019).
7. Antonino, P. & Roscoe, A. W. Hung, C., Hong, J., Bechini, A. & Song, E. (eds) *Solidifier: bounded model checking solidity using lazy contract deployment and precise memory modelling.* (eds Hung, C., Hong, J., Bechini, A. & Song, E.) *SAC '21: The 36th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, Republic of Korea, March 22-26, 2021*, 1788–1797 (ACM, 2021). URL https://doi.org/10.1145/3412841.3442051.
8. Wang, Y. *et al.* Chakraborty, S. & Navas, J. A. (eds) *Formal verification of workflow policies for smart contracts in azure blockchain.* (eds Chakraborty, S. & Navas, J. A.) *Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers*, Vol. 12031 of *Lecture Notes in Computer Science*, 87–106 (Springer, 2019). URL https://doi.org/10.1007/978-3-030-41600-3_7.
9. Grieco, G., Song, W., Cygan, A., Feist, J. & Groce, A. *Echidna: Effective, usable, and fast fuzzing for smart contracts*, ISSTA 2020, 557–560 (Association for Computing Machinery, New York, NY, USA, 2020). URL https://doi.org/10.1145/3395363.3404366.
10. Stephens, J., Ferles, K., Mariano, B., Lahiri, S. K. & Dillig, I. *Smartpulse: Automated checking of temporal properties in smart contracts*, 555–571 (IEEE, 2021). URL https://doi.org/10.1109/SP40001.2021.00085.
11. Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D. & Vechev, M. *Verx: Safety verification of smart contracts*, 1661–1677 (2020).
12. Smart contract weakness classification and test cases (2020). URL https://swcregistry.io/.
13. Groce, A., Feist, J., Grieco, G. & Colburn, M. *What are the actual flaws in important smart contracts (and how can we find them)?* (2020).
14. Zhang, Z., Zhang, B., Xu, W. & Lin, Z. *Demystifying exploitable bugs in smart contracts*, ICSE '23, 615–627 (IEEE Press, 2023). URL https://doi.org/10.1109/ICSE48619.2023.00061.
15. Require in solidity. https://docs.soliditylang.org/en/v0.8.7/control-structures.html (2021).
16. Rust for near. https://docs.rs/near-sdk/latest/near_sdk/macro.require.html (2022).
17. Godoy, J., Galeotti, J. P., Garbervetsky, D. & Uchitel, S. Enabledness-based testing of object protocols. *ACM Trans. Softw. Eng. Methodol.* **30** (2), 12:1–12:36 (2021). URL https://doi.org/10.1145/3415153. https://doi.org/10.1145/3415153 .
18. Godoy, J., Galeotti, J. P., Garbervetsky, D. & Uchitel, S. *Predicate abstractions for smart contract validation*, 289–299 (ACM, 2022). URL https://doi.org/10.1145/3550355.3552462.
19. Jackson, D. Alloy: A language and tool for exploring software designs. *Commun. ACM* **62** (9), 66–76 (2019). URL https://doi.org/10.1145/3338843. https://doi.org/10.1145/3338843 .
20. Zheng, Z. *et al.* *Turn the rudder: A beacon of reentrancy detection for smart contracts on ethereum*, ICSE '23, 295–306 (IEEE Press, 2023). URL https://doi.org/10.1109/ICSE48619.2023.00036.
21. Fatima Samreen, N. & Alalfi, M. H. *Reentrancy vulnerability identification in ethereum smart contracts*, 22–29 (2020).
22. Smart contract best practices revisited: Block number vs. timestamp (2018). URL https://medium.com/@phillipgoldberg/smart-contract-best-practices-revisited-block-\number-vs-timestamp-648905104323.
23. Types in solidity (2020). URL https://docs.soliditylang.org/en/v0.8.7/types.html.
24. Transfer() in solidity: Why you should stop using it? (2021). URL https://www.immunebytes.com/blog/transfer-in-solidity-why-you-should-stop-using-it.
25. Jhala, R., Podelski, A. & Rybalchenko, A. *Predicate Abstraction for Program Verification*, 447–491 (Springer International Publishing, Cham, 2018). URL https://doi.org/10.1007/978-3-319-10575-8_15.
26. de Caso, G., Braberman, V., Garbervetsky, D. & Uchitel, S. Enabledness-based program abstractions for behavior validation. *ACM Trans. Softw. Eng. Methodol.* **22** (3) (2013) .
27. de Caso, G., Braberman, V., Garbervetsky, D. & Uchitel, S. *Program abstractions for behaviour validation*, ICSE '11, 381–390 (Association for Computing Machinery, New York, NY, USA, 2011). URL https://doi.org/10.1145/1985793.1985846.
28. de Caso, G., Braberman, V., Garbervetsky, D. & Uchitel, S. *Validation of contracts using enabledness preserving finite state abstractions*, 452–462 (2009).
29. de Caso, G., Braberman, V., Garbervetsky, D. & Uchitel, S. Automated abstractions for contract validation. *IEEE Transactions on Software Engineering* **38** (1), 141–162 (2012) .
30. Digital locker documentation (2019). URL https://github.com/Azure-Samples/blockchain/blob/master/blockchain-workbench/application-and-smart-contract-samples/digital-locker/ethereum/DigitalLocker.sol.

31. Digital locker documentation (2019). URL https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples/digital-locker.
32. Durieux, T., Ferreira, J. a. F., Abreu, R. & Cruz, P. *Empirical review of automated analysis tools on 47,587 ethereum smart contracts*, ICSE '20, 530–541 (Association for Computing Machinery, New York, NY, USA, 2020). URL https://doi.org/10.1145/3377811.3380364.
33. Xue, Y. *et al. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts*, 1029–1040 (2020).
34. Liu, C. *et al. Reguard: finding reentrancy bugs in smart contracts*, ICSE '18, 65–68 (Association for Computing Machinery, New York, NY, USA, 2018). URL https://doi.org/10.1145/3183440.3183495.
35. Pasco (2024). URL https://github.com/j-godoy/PASCo.
36. Lal, A., Qadeer, S. & Lahiri, S. K. Madhusudan, P. & Seshia, S. A. (eds) *A solver for reachability modulo theories*. (eds Madhusudan, P. & Seshia, S. A.) *Computer Aided Verification*, 427–443 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2012).
37. Solidity metrics (2024). URL https://github.com/Consensys/solidity-metrics/.
38. Azure blockchain workbench (2017). URL https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples.
39. He, X. *Modeling and analyzing smart contracts using predicate transition nets*, 108–115 (2020).
40. Azure fixed code. https://github.com/blockhousetech/research/tree/master/Solidifier/evaluation/examples/azure_fixed (2020).
41. Smartbugs curated dataset. https://github.com/smartbugs/smartbugs-curated (2020).
42. Dasp (2018). URL https://dasp.co/.
43. Ferreira, J. a. F., Cruz, P., Durieux, T. & Abreu, R. *Smartbugs: a framework to analyze solidity smart contracts*, ASE '20, 1349–1352 (Association for Computing Machinery, New York, NY, USA, 2021). URL https://doi.org/10.1145/3324884.3415298.
44. Smartbugs 2.0.10. https://github.com/smartbugs/smartbugs. Commit 9236400.
45. Conkas. https://github.com/nveloso/conkas.
46. Torres, C. F., Schütte, J. & State, R. *Osiris: Hunting for integer bugs in ethereum smart contracts*, ACSAC '18, 664–676 (Association for Computing Machinery, New York, NY, USA, 2018). URL https://doi.org/10.1145/3274694.3274737.
47. Tsankov, P. *et al. Securify: Practical security analysis of smart contracts*, CCS '18, 67–82 (Association for Computing Machinery, New York, NY, USA, 2018). URL https://doi.org/10.1145/3243734.3243780.
48. Nguyen, T. D., Pham, L. H., Sun, J., Lin, Y. & Minh, Q. T. *sfuzz: an efficient adaptive fuzzer for solidity smart contracts*, ICSE '20, 778–788 (Association for Computing Machinery, New York, NY, USA, 2020). URL https://doi.org/10.1145/3377811.3380334.
49. Torres, C. F., Iannillo, A. K., Gervais, A. & State, R. *Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts*, 103–119 (2021).
50. Mythril. https://github.com/ConsenSysDiligence/mythril.
51. Torres, C. F., Steichen, M. *et al. The art of the scam: Demystifying honeypots in ethereum smart contracts*, 1591–1607 (2019).
52. Nikolić, I., Kolluri, A., Sergey, I., Saxena, P. & Hobor, A. *Finding the greedy, prodigal, and suicidal contracts at scale*, ACSAC '18, 653–663 (Association for Computing Machinery, New York, NY, USA, 2018). URL https://doi.org/10.1145/3274694.3274743.
53. Mossberg, M. *et al. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts*, 1186–1189 (2019).
54. Luu, L., Chu, D.-H., Olickel, H., Saxena, P. & Hobor, A. *Making smart contracts smarter*, CCS '16, 254–269 (Association for Computing Machinery, New York, NY, USA, 2016). URL https://doi.org/10.1145/2976749.2978309.
55. Semgrep. https://github.com/semgrep/semgrep.
56. Tikhomirov, S. *et al. Smartcheck: static analysis of ethereum smart contracts*, WETSEB '18, 9–16 (Association for Computing Machinery, New York, NY, USA, 2018). URL https://doi.org/10.1145/3194113.3194115.
57. Solhint. https://github.com/protofire/solhint.
58. Smartbugs: Vulnerabilities mapping. https://github.com/smartbugs/smartbugs/wiki/Vulnerabilities-mapping.
59. Defective component counter documentation (2019). URL https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples/defective-component-counter.
60. Automated construction of predicate abstractions for smart contract validation - generated abstractions (2024). URL https://github.com/j-godoy/pasco-results.
61. DeLine, R. & Fähndrich, M. *Enforcing high-level protocols in low-level software*, 59–69 (2001).
62. Henzinger, T. A., Jhala, R. & Majumdar, R. *Permissive interfaces*, 31–40 (2005).
63. Ghezzi, C., Mocci, A. & Monga, M. *Synthesizing intensional behavior models by graph transformation*, 430–440 (IEEE, 2009).
64. Mavridou, A., Laszka, A., Stachtiari, E. & Dubey, A. Verisolid: Correct-by-design smart contracts for ethereum. *arXiv preprint arXiv:1901.01292* (2019) .
65. Finkbeiner, B., Hofmann, J., Kohn, F. & Passing, N. André, É. & Sun, J. (eds) *Reactive synthesis of smart contract control flows*. (eds André, É. & Sun, J.) *Automated Technology for Verification and Analysis*, 248–269 (Springer Nature Switzerland, Cham, 2023).
66. Ellul, J. & Pace, G. J. *Runtime verification of ethereum smart contracts*, 158–163 (IEEE, 2018).
67. Fekih, R. B., Lahami, M., Jmaiel, M., Ben Ali, A. & Genestier, P. *Towards model checking approach for smart contract validation in the eip-1559 ethereum*, 83–88 (2022).

68. Parekh, N., Ahrendt, W. & Fabian, M. Automatic conversion of smart contracts for non-blocking verification. *IFAC-PapersOnLine* **58** (1), 282–287 (2024). URL https://www.sciencedirect.com/science/article/pii/S2405896324001265. https://doi.org/https://doi.org/10.1016/j.ifacol.2024.07.048, 17th IFAC Workshop on discrete Event Systems WODES 2024 .
69. Omg unified modeling language, version 2.5.1. https://www.omg.org/spec/UML/2.5.1/PDF (2017).
70. Murray, Y. & Anisi, D. A. *Survey of formal verification methods for smart contracts on blockchain*, 1–6 (2019).
71. Eshghie, M., Ahrendt, W., Artho, C., Hildebrandt, T. T. & Schneider, G. Ferreira, C. & Willemse, T. A. C. (eds) *Capturing smart contract design with dcr graphs*. (eds Ferreira, C. & Willemse, T. A. C.) *Software Engineering and Formal Methods*, 106–125 (Springer Nature Switzerland, Cham, 2023).
72. Mossberg, M. *et al. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts*, 1186–1189 (IEEE, 2019).
73. Dharanikota, S., Mukherjee, S., Bhardwaj, C., Rastogi, A. & Lal, A. *Celestial: A smart contracts verification framework*, 133–142 (2021).
74. Duo, W., Xin, H. & Xiaofeng, M. Formal analysis of smart contract based on colored petri nets. *IEEE Intelligent Systems* **35** (3), 19–30 (2020) .
75. Garfatta, I., Klai, K., Graïet, M. & Gaaloul, W. *Model checking of vulnerabilities in smart contracts: a solidity-to-cpn approach*, 316–325 (2022).
76. Tang, X., Du, Y., Lai, A., Zhang, Z. & Shi, L. Deep learning-based solution for smart contract vulnerabilities detection. *Scientific Reports* **13** (2023). https://doi.org/10.1038/s41598-023-47219-0 .
77. Gao, Z. *et al. Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding*, 394–397 (2019).
78. Yu, X., Zhao, H., Hou, B., Ying, Z. & Wu, B. *Deescvhunter: A deep learning-based framework for smart contract vulnerability detection*, 1–8 (2021).
79. Liao, J.-W., Tsai, T.-T., He, C.-K. & Tien, C.-W. *Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing*, 458–465 (2019).
80. Wang, W. *et al.* Contractward: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering* **8** (2), 1133–1144 (2021). https://doi.org/10.1109/TNSE.2020.2968505 .
81. Sun, Y. *et al. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis*, ICSE '24 (Association for Computing Machinery, New York, NY, USA, 2024). URL https://doi.org/10.1145/3597503.3639117.