Using Symbolic Model Execution to Detect Vulnerabilities of Smart Contracts *

Chiara Braghin[1][0000−0002−9756−4675], Giuseppe Del Castillo[2][0009−0005−7020−6607], Elvinia Riccobene[1][0000−0002−1400−1026], and Simone Valentini[1][0009−0005−5956−3945]

[1] Computer Science Department,
Università degli Studi di Milano, via Celoria 18, Milan, Italy
{chiara.braghin,elvinia.riccobene,simone.valentini}@unimi.it
[2] Munich, Germany
giuseppedelcastillo@acm.org

Abstract. Smart contracts are programs that automate agreements between parties without the need for intermediaries. Embedded in a blockchain, they ensure transparency, immutability, and trustworthiness. While efficient, their immutable nature and reliance on internet-connected nodes make them susceptible to attacks. Identifying vulnerabilities before deployment is critical to mitigate risks, prevent catastrophic events, and avoid significant financial losses. This paper introduces a method for detecting vulnerabilities in smart contracts written in Solidity and deployed on the Ethereum blockchain. The approach models a smart contract as an Abstract State Machine (ASM), where the absence of specific vulnerabilities is encoded as invariants. An existing symbolic execution technique for ASM models was extended and improved to enable the processing of the ASM models of the smart contracts. By symbolically executing the ASM, the method identifies faulty execution paths that violate these invariants, exposing potential vulnerabilities in the contract's behavior. Vulnerable execution scenarios of the smart contract can be generated using the symbolic execution results.
As a proof of concept, we show the approach on a running case study, the *Auction* smart contract. Furthermore, we discuss the results of applying the technique to a number of Solidity smart contracts.

Keywords: Blockchain · Ethereum · Solidity · Smart contract vulnerabilities · Abstract State Machines · Symbolic execution.

1 Introduction

Blockchain technology has emerged as a fundamental component in applications ranging from finance to supply chain management and beyond, enabling secure, transparent, and decentralized transaction management without the need

* This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

for trusted intermediaries. Through cryptographic techniques and distributed consensus mechanisms, it ensures data integrity and trust. In addition, the introduction of smart contracts has made it possible to automate agreements between parties without intermediaries, enhancing efficiency and reducing costs. However, despite their benefits, they remain vulnerable to attacks, as they are publicly accessible to all network nodes and often handle significant amounts of cryptocurrency. For instance, the notorious DAO hack [27] on the Ethereum blockchain [31] resulted in the loss of $70 million worth of Ether due to a vulnerable smart contract that lacked proper verification [25]. Addressing vulnerabilities before (permanent) deployment is essential for mitigating risks, preventing catastrophic incidents, and avoiding substantial financial losses. Moreover, high-cost vulnerabilities and exploits can severely impact the trust and acceptance of the blockchain ecosystem.

Formal verification can play a crucial role in advancing the maturity and adoption of blockchain technology by providing a rigorous approach to ensuring the accuracy and reliability of smart contracts. However, while formal verification for smart contracts has made significant advancements [19,29], several challenges remain. Existing tools have notable limitations: (a) bytecode-based tools generate results that are challenging to trace back to code-level vulnerabilities; (b) tools operating at a higher level, rely on complex notations that require a strong mathematical background, making them less accessible to many developers, and (c) some tools produce vulnerability reports that are difficult to interpret, often failing to accurately locate the exact code segment containing the issue or clearly identify the type of vulnerability detected [24].

In [10,11], we explored the potential of using the Abstract State Machines (ASMs) [7,8] to model Ethereum smart contracts written in Solidity. Thanks to the pseudo-code format of an ASM model, which allows for an easy comprehension by practitioners and not-experts, and the tool support of this formal method [6], our long-term vision [30] is to develop a practical verification framework, leveraging ASMs as the foundational formalism for specifying and analyzing smart contract vulnerabilities. Thus, we formalized the Ethereum Virtual Machine (EVM) and introduced key language primitives that enable a straightforward translation of Solidity smart contracts into ASMETA models. ASMETA provides a toolset for model editing, validation, and verification, including a translator to the NuSMV model checker for formal verification. However, as model size increases, we encountered the state space explosion problem, a common challenge in model checking. To address this, in this paper, we explore the potential of a recently developed symbolic execution tool [14] for ASMs. Unlike model checking, symbolic execution does not require exhaustive state searching, mitigating state explosion and allowing it to manage infinite domains more effectively. While model checking is better suited for exhaustively proving correctness properties, symbolic execution excels in detecting concrete bugs and vulnerabilities. On the other hand, compared to traditional testing approaches, symbolic execution offers significant advantages in vulnerability detection by systematically exploring execution paths and achieving higher coverage than random or

manually designed test cases. Additionally, it identifies exact inputs that trigger bugs, whereas fuzzing often detects crashes without providing precise failure conditions.

In our approach, we express the absence of specific vulnerabilities as model invariants within an ASMETA smart contract model. The symbolic execution engine is then employed to uncover faulty execution paths that lead to invariant violations. To achieve this, we significantly enhanced the symbolic executor, adapting it to ASMETA models and improving its ability to detect invariant violations. The insights provided by the symbolic execution tool can also be used to generate execution scenarios, helping to identify and analyze vulnerabilities within the smart contract. We demonstrate our approach using a case study, the *Auction* smart contract, and discuss the results of applying our vulnerability detection method to a good broad set of Solidity smart contracts.

The rest of the paper is organized as follows: in Sect. 2, we provide a background description of Ethereum smart contracts, and we introduce the Auction smart contract. In Sect. 3, we briefly present the ASM-based modeling of the Ethereum virtual machine and of smart contracts. In Sect. 4, we present the tool for symbolic execution of ASMETA models, its improvement and features. In Sect. 5, we show the application of our vulnerability detection strategy to the running example, while in Sect. 6 we report results to evince the effectiveness of the approach and discuss the potential limitations and threats of our contribution. In Sect. 7, we compare our results with existing approaches. Sect. 8 concludes the paper and outlines future research directions.

2 Blockchain and Solidity Smart Contracts

A blockchain is a decentralized and distributed ledger that records transactions across a network of computers. It comprises data blocks, each containing a set of transactions, a unique cryptographic hash of the previous block, and a timestamp. These blocks are linked together, forming an immutable chain: once a block is added, its contents cannot be altered without modifying all subsequent blocks, making retroactive tampering virtually impossible.

Ethereum [31] is an open-source blockchain platform designed for developers to create and deploy decentralized applications. It offers the Ethereum Virtual Machine (EVM) as a decentralized runtime environment for executing smart contracts, self-executing programs that automate agreements with the terms directly coded into them. They are written in high-level languages like Solidity and compiled into bytecode that the EVM can run.

The EVM operates on every node within the Ethereum network, guaranteeing decentralized execution of smart contracts. Additionally, it upholds the state of the Ethereum blockchain, encompassing account balances, smart contract code, and storage. Two types of accounts have an Ether (ETH) balance and can interact with the blockchain by sending transactions on the chain: *externally owned accounts* (EOA) and *smart contract accounts*. EOA are humans-managed accounts identified by a public key, which serves as the account address, and

controlled by a corresponding private key. These accounts use the private key to sign transactions, proving ownership and authorization through an elliptic curve digital signature algorithm. Contract accounts, on the other hand, are special accounts that contain associated code and data storage. They have unique addresses but do not have private keys. When a smart contract transaction is initiated, the EVM processes the bytecode linked to the contract and executes it. This execution modifies the blockchain's state by updating account balances or storage values. A special global variable called `msg` is employed to capture transaction-related information, including `msg.sender` representing the address that initiated the function call, and `msg.value` yielding the liquidity sent with the message.

2.1 Running case study

The code in Listing 2.1 reports the *Auction* smart contract written in Solidity [9]. This contract is commonly encountered in real-world blockchain applications to guarantee greater transparency and avoid cheating auctioneers.

```
1  contract Auction {
2    address currentFrontrunner;
3    address owner;
4    uint currentBid;
5
6    function destroy() {
7      selfdestruct(owner);
8    }
9    function bid() payable {
10     require(msg.value > currentBid);
11     if (currentFrontrunner != 0) {
12       require(currentFrontrunner.send(currentBid));
13       }
14     currentFrontrunner = msg.sender;
15     currentBid         = msg.value;
16   }
17 }
```

Listing 2.1: Solidity code of the Auction smart contract

The contract keeps track of the current highest bidder (`currentFrontrunner`) and the highest bid amount (`currentBid`). Participants can submit bids through the `bid()` function, which takes payment in Ether. The function `require`s that (1) the submitted bid value (`msg.value`) is greater than the current highest bid, and that (2) if there is an existing highest bidder (`currentFrontrunner != 0`), the contract must refund his/her bid amount before assigning the new bid. The `payable` keyword is required for a function to receive Ether. In Solidity, the `send` function, used as `address.send(amount)`, returns true if the transfer to the specified `address` succeeds; otherwise, it returns false.

The contract also defines a `destroy` function, which invokes Solidity's built-in `selfdestruct(recipientAddress)` function to terminate the contract and transfer any remaining Ether to a specified recipient. This function reveals a vulnerability that falls under the category of *unprotected function vulnerability*,

where a smart contract exposes a critical functionality without enforcing access control, thus allowing unauthorized users to execute sensitive operations. In this case, without a restriction ensuring that only the contract owner can disable the contract and withdraw its remaining balance, an attacker could exploit the function to force the contract to self-destruct, resulting in a denial-of-service (DoS) attack.

3 Modeling Solidity Smart Contracts

In [11,30], we demonstrated how to model the EVM and Solidity smart contracts in ASMETA [4]. Our approach supports all the core features of smart contracts, except for the *gas* mechanism.[3] In the sequel, regarding the used formalism and notation, we refer the non-expert reader to [6,8,7] for a very short introduction.

An ASMETA library, `EVMlibrary`, provides functions and rules to model both the EVM behaviour and Ethereum accounts. Specifically, the library defines the EVM stack structure, representing the execution stack through the `StackLayer` domain and the `current_layer` function, which tracks the stack frame of the method currently being executed (i.e., the top of the stack). To manage execution flow, the library includes functions such as `executing_function` and `instruction_pointer`, which track the currently running function and the instruction being executed. Each stack layer is enriched with additional functions - `sender`, `receiver`, `amount` - that store essential transaction details, including the initiating account's address, the recipient, and the amount of ether transferred. Two library rules, `r_Transaction` and `r_Ret` are defined to move along the stack. The rule `r_Transaction` simulates a transaction execution with an Ether transfer (due to a call to either a `send` or `transfer` function); it increases the stack size and stores relevant execution details within the stack layer, such as the executing contract and instruction pointer; it also updates the caller's instruction pointer accordingly. The rule `r_Ret` stops the execution by decreasing the stack size and restoring the previous execution frame. Additionally, the library provides rules corresponding to predefined Solidity functions. Specifically, `require` is modeled by the r_Require rule, which increments the `instruction_pointer` on the true value of a given condition, or stops the execution otherwise; `selfdestruct` is modeled by the r_Selfdestruct rule, which stops the execution and sends the whole contract balance to the User (passed as parameter). Table 1 presents a structured mapping of Solidity smart contracts to ASMETA models, written in `AsmetaL`, the model editing textual notation of the ASMETA toolset.

In addition to the special *global* variables msg.sender and msg.value, which are defined in the `EVMlibrary`, a Solidity contract includes two main types of variables: *state* variables, which are permanently stored in the contract's storage, and *local* variables, which exist only during the execution of a function.

[3] *Gas* is the unit used to measure computational effort in Ethereum: each operation in the EVM consumes a certain amount of gas, and users must pay gas fees to execute transactions and smart contracts.

Solidity contract	**ASMETA model**
contract `SCName`	create a model asm SCName and add `scname` to domain User to identify the contract
for each *state* variable `type p`	add a function p: Type
for each *local* variable `type v`	add a function v: StackLayer → Type
for each mapping *state* variable `mapping(keyType => valueType)`	add a function p: KeyType → ValueType
for each mapping *local* variable `mapping(keyType => valueType)`	add a function p: StackLayer x KeyType → ValueType
for each function `f()`	add static f: Function and a rule r_F[]
for the body of function `f()`	add the body of the rule r_F[]
if the `fallback` function is present	add a corresponding rule r_fallback
if the `fallback` function is absent	add the *default* rule r_fallback
if the contract is executed in isolation	add the rule r_main calling in parallel all rules r_F[] and set the initial state
if two or more contracts are co-executed	add the rule r_main orchestrating the rules r_contractName$_i$ of all the contracts and set the initial state

Table 1: Schema for mapping a Solidity contract to an ASMETA model

Furthermore, Solidity provides the `mapping` keyword, a reference type used to store data as key-value pairs.

In the mapping schema, particular attention deserves the translation of a Solidity function into an ASMETA rule. The latter mainly consists of a `case` rule on the value of the instruction_pointer(current_layer), which refers to the Solidity instruction within the function body, and allows the instruction pointer to jump to a specific instruction when needed.

A special handling applies to the Solidity `fallback` function, which is executed when a contract receives Ether with a call to a function that does not exist in the contract, or when no data is provided in the transaction. While including a `fallback` function is generally recommended, it is not mandatory. If absent, the contract raises an exception and halts execution. To model this behavior, we introduce a *default version* of the r_Fallback rule. If no contract function is invoked, this rule calls r_Require with a false value to raise an exception, followed by r_Ret to handle termination. A version of this r_Fallback rule for the Auction contract is shown in Listing 3.2.

3.1 Model of the Auction contract

Based on the Solidity contract code in Listing 2.1, we define the corresponding ASMETA model Auction.asm following the mapping schema outlined in Table 1. The complete model is available at [12].

Listing 3.1 presents the functions that model the contract's state variables. Additionally, constants are added to the predefined library domains `User` and `Function` to include the contract-specific functions and users.

```
signature:
/* contract's arguments */
  dynamic controlled owner : User
  dynamic controlled currentFrontrunner : User
  dynamic controlled currentBid : MoneyAmount

/* initializing library domains for the specific contract*/
  static auction : User
  static bid : Function
  static destroy : Function
```

Listing 3.1: Signature of the ASMETA Auction Model

Listing 3.2 reports the specification of the three rules corresponding to the three functions of the contract. The rule `r_Bid` is triggered when `bid()` is the currently executing function in the current layer. The `switch` statement in the rule manages the execution flow based on the instruction pointer's value in the current layer. Initially, in case 0, the rule ensures that the amount specified in current_layer exceeds the currentBid. Next, in case 1, if the current front-runner for the bid is defined, the instruction pointer increments to the next case; otherwise, it jumps to case 4. In case 2, a transaction is initiated, sending the current bid to the current front-runner.[4] After the transaction is performed, case 3 requires that the response from the previous transaction is successful. Cases 4 and 5 update the current front-runner with the transaction sender and the current bid with the transaction amount. The last case invokes the rule r_Ret since the execution of function `bid`() terminates.

Listing 3.2 also includes the rule for the Auction function `destroy`(), which calls the `r_Selfdestruct` rule from the library (not shown here) on the contract owner. This rule mirrors the behavior of Solidity `selfdestruct` function by setting a predefined boolean variable, `disabled` to true. Defined within the `Users` domain, this variable indicates whether the user associated with the given contract has been disabled.

4 Symbolic execution of ASMETA Models

The ASE Tool To symbolically execute the smart contract models, the prototype ASM symbolic execution tool ("ASE tool") presented in [14] and available at [16] was used. The main feature of the ASE tool is the ability to transform a sequential composition of ASM rules into a semantically equivalent parallel ASM rule that has a simpler structure and is easier to reason about. Such a rule is essentially a decision tree, i.e. a rule consisting of nested conditionals (inner nodes of the decision tree) and blocks of parallel updates of individual, unambiguously identified locations (leaves of the decision tree). The running example throughout the other sections shows examples of these rules and how the

[4] The argument `none` in the r_Transaction rule is because the function `bid`() implies money transfer and no nested function calls.

```
1  rule r_Bid =
2      if executing_function(current_layer) = bid then
3        switch instruction_pointer(current_layer)
4          case 0 :
5            r_Require[amount(current_layer) > currentBid]
6          case 1 :
7            if currentFrontrunner != undef then
8              instruction_pointer(current_layer) := instruction_pointer(current_layer) + 1
9            else
10             instruction_pointer(current_layer) := 4
11           endif
12         case 2 :
13           r_Transaction[auction, currentFrontrunner, currentBid, none]
14         case 3 :
15           r_Require[exception]
16         case 4 :
17           par
18             currentFrontrunner := sender(current_layer)
19             instruction_pointer(current_layer) := instruction_pointer(current_layer) + 1
20           endpar
21         case 5 :
22           par
23             currentBid := amount(current_layer)
24             instruction_pointer(current_layer) := instruction_pointer(current_layer) + 1
25           endpar
26         case 6 :
27           r_Ret[]
28       endswitch
29     endif
30
31 rule r_Destroy =
32     if executing_function(current_layer) = destroy then
33       switch instruction_pointer(current_layer)
34         case 0 :
35           r_Selfdestruct[owner]
36       endswitch
37     endif
38
39 rule r_Fallback =
40     if executing_function(current_layer) != bid and executing_function(current_layer) != destroy then
41         r_Require[false]
42     endif
```

Listing 3.2: Model of the `bid()`, `destroy()` and `fallback()` functions

tool is used. For details on the formal definition of the transformation and its implementation, see [14,15].

To meet the needs of the vulnerability analysis of the models presented in this paper, the tool had to be enhanced in various ways. The following subsections describe these enhancements.

AsmetaL Support The ASE tool had to be made compatible with `AsmetaL`, the source language of the ASMETA toolset [4] that was used to specify the smart contract ASM models (including the Auction example). For this purpose,

an `AsmetaL` parser was added to the ASE tool and support for symbolic execution of the required subset of `AsmetaL` was implemented, including ASM constructs such as `forall` that were not implemented in the initial version of the ASE tool. In particular, the tool has been extended to support enumerated and (non-dynamic) abstract domains, function definitions and initializations, macro rule definitions, `forall` rules and quantifiers over finite domains.

The implementation of these constructs is rather straightforward, mostly consisting of some form of syntactic expansion followed by symbolic execution. For example, a universally quantified Boolean term "`forall` x `in` A `with` t" (where A is a finite set $\{a_1, \ldots, a_n\}$) is expanded into a term "$t[x/a_1] \wedge \ldots \wedge t[x/a_n]$", which is then symbolically evaluated. Similarly, a `forall` rule over the same set A is expanded into a `par` rule "$R[x/a_1]$ `par` ... `par` $R[x/a_n]$", which is then symbolically executed according to the rules given in [14] for `par`.[5]

Symbolic Execution of Basic ASMs The symbolic execution scheme, which was introduced in [14] to transform ASM rules including the sequential composition rules `seq` and `iterate` into basic ASM rules, has been adapted to symbolically execute regular sequential ASMs.

The state S_n of a sequential run $S_0, S_1, \ldots$ starting in the initial state S_0 can be seen as $S_0 \oplus P^n$, where P^n is the main rule (*program*) P of the ASM iterated n times. As such iteration can be defined as $P^0 :=$ `skip`, $P^{i+1} := P^i$ `seq` P $(i \geq 0)$, the transformation rules of [14] for `seq` can be applied repeatedly to obtain S_n.

The ASE tool now provides a `-steps` n option that constructs an ASM rule equivalent to P^n using symbolic execution. This rule can be inspected to understand the system behavior and the root causes of any identified invariant violation, if it is found that an invariant is violated in state S_n (see "Invariant Checking" below). In Sect. 6, we show a concrete use of this option for the Auction contract vulnerability analysis.

Nested Non-Static Functions A limitation of the symbolic execution method presented in [14] is its inability to process rules containing terms of the form $f(t_1, \ldots, g(s_1, \ldots, s_m), \ldots, t_n)$, $m \geq 0$, where f is a non-static function and g is an uninitialized non-static function[6], when no value has yet been assigned to the relevant location $(g, (x_1, \ldots, x_m))$ (where $x_1, \ldots, x_m$ are the values of terms

[5] A `forall` Boolean term is used for example in the definition of invariant E3 in the KotET model. A `forall` rule is used for example in the `r_Main` rule of the Auction model, see the GitHub repository [12].

[6] Non-static functions are all functions that are not static, i.e. all functions that can have a different interpretation in different states of an ASM run (such as controlled and monitored functions). In the Auction model, unary functions executing_contract and balance are examples of (uninitialized) non-static functions; nullary function current_layer, instead, is an initialized non-static function. Even though the only non-static functions supported by the ASE tool at the time of writing are controlled functions (no monitored functions), the remarks of this subsection apply in principle to all nested non-static functions. The essential difference is that static functions

$s_1, \ldots, s_m$). The problem is that, as the location $(g, (x_1, \ldots, x_m))$ does not hold a concrete value, the term $f(t_1, \ldots, g(s_1, \ldots, s_m), \ldots, t_n)$ cannot be mapped to an unambiguously identified location $(f, (y_1, \ldots, y_n))$.

Due to this limitation, none of the smart contract models presented here could be symbolically executed using the initial ASE tool introduced in [14]. This problem can be illustrated using the Auction example. For example, the `r_Selfdestruct` rule contains an update rule

```
balance(executing_contract(current_layer)) := 0
```

While the initialized non-static function current_layer always evaluates to a concrete value, so that the subterm executing_contract(current_layer) unambiguously identifies a location of the executing_contract function, this is not the case for the whole term. Indeed, the uninitialized (i.e., uninterpreted) non-static function executing_contract occurs as an argument of non-static function balance, so that it is not clear which location of balance is to be updated.

However, it was observed that, in all the smart contract models presented in this paper, the range of the problematic "inner terms" (such as the $g(s_1, \ldots, s_m)$ above, or the executing_contract(current_layer) in the example) is always finite. Accordingly, terms can be transformed by making a case distinction over the elements of the range of g:

$$\frac{[\![g(s_1, \ldots, s_m)]\!]_{S,C} \neq \langle \text{val } x \rangle \qquad \text{range}(g) = \{x_1, \ldots, x_p\} \qquad f, g \text{ not static}}{\begin{array}{l}[\![\, f(t_1, \ldots, g(s_1, \ldots, s_m), \ldots, t_n) \,]\!]_{S,C} = \\ \quad [\![\texttt{if } g(s_1, \ldots, s_m) = x_1 \texttt{ then } f(t_1, \ldots, x_1, \ldots, t_n) \\ \quad \ldots \\ \quad \texttt{else if } g(s_1, \ldots, s_m) = x_{p-1} \texttt{ then } f(t_1, \ldots, x_{p-1}, \ldots, t_n) \\ \quad \texttt{else } f(t_1, \ldots, x_p, \ldots, t_n) \,]\!]_{S,C}\end{array}}$$

The above rule is applied for all subterms t_i of $f(t_1, \ldots, t_n)$ that cannot be fully evaluated to a value $\langle \text{val } x \rangle$. Further transformation rules similar to those defined in [14] are then applied to the terms and rules that include $f(t_1, \ldots, t_n)$ to "move out" the introduced conditional terms until the main ASM rule has been transformed into the "decision tree" form mentioned above.[7] A complete specification of the transformation rules and algorithm is outside the scope of this paper and may be provided elsewhere, but the basic technique should be clear from the above explanation in combination with [14].

---

have a fixed interpretation, while non-static functions can be (at least partially) uninterpreted.

[7] This may appear to be a prohibitively expensive transformation, but in practice it turned out that in many cases, with the help of the SMT solver and taking into account the path condition that holds in the given context, the conditionals generated by the case distinction are simplified (even to a single branch, resulting in the elimination of the conditional) and/or lead to the generation of further path conditions for their subrules that cause considerable simplifications in the subrules.

Invariant Checking The last addition to the ASE tool is a feature to check whether the invariants specified in the ASMETA model are met within the first n steps of the symbolic ASM run (command line option `-invcheck` n, which is shown in action in Sect. 6 on the running example). The invariant checking is carried out by symbolically evaluating each Boolean term inv_j defining the j-th invariant ($0 \leq j \leq m$) in the appropriate state S_i ($0 \leq i \leq n$) as follows:

1. each invariant j is checked in the initial state by evaluating $[\![inv_j]\!]_{S_0,\emptyset}$, where S_0 is the initial state and $\emptyset$ is the empty path condition;
2. for each $i \in \{1, \ldots, n\}$:
   (a) a decision tree rule equivalent to P^i is built using symbolic execution, as explained in "Symbolic Execution of Basic ASMs" above;
   (b) the decision tree rule for P^i is traversed and, at each leaf of the tree, the invariants are checked by evaluating $[\![inv_j]\!]_{S_0 \oplus U,C}$, where U is the symbolic update set found at that leaf (see "The ASE Tool" above) and C is the path condition corresponding to the path from the root to the leaf (C is constructed starting with $\emptyset$ at the root and adding, at each inner conditional node with guard G, either G or $\neg G$ depending on whether the "then" or the "else" branch is taken);
   (c) note that there are three possible outcomes for $[\![inv_j]\!]_{S_0 \oplus U,C}$:
      i. **true**: the invariant inv_j is *met* on the given path;
      ii. **false**: the invariant inv_j is *definitely violated* on the given path (by this terminology we mean that, based on the specification of the initial state S_0, it can be established that the invariant is violated);
      iii. a Boolean term ϕ_j, which is more or less simplified in comparison to inv_j, but neither **true** nor **false**, and depends on one or more non-static functions that are uninterpreted in S_0: in this case, we say that inv_j is *possibly violated* on the given path, i.e. it is violated when the relevant locations of those uninterpreted functions have certain values, specifically the values for which $\neg\phi_j \wedge C$ is satisfied.[8]

5 Vulnerabilities detection of Solidity Smart Contracts

Here, we explain our strategy for detecting vulnerabilities in smart contracts through the symbolic execution of ASMETA models.

Given an ASMETA model of a contract such as the Auction case study, we define model invariants to represent the absence of vulnerabilities as safety properties. These invariants are based on either the intended contract behavior or, as explained in Sect. 6, on known unsafe operations. The vulnerabilities we consider here relate to contract-intrinsic issues, rather than those arising from interactions with malicious smart contracts designed to exploit weaknesses.

[8] By inspecting the simplified invariant $[\![inv_j]\!]_{S_0 \oplus U,C} = \phi_j$ and the path condition C, both of which are displayed by the tool, it is possible to identify which values lead to the invariant violation. A possible future improvement is the automatic generation of a counterexample with concrete values that lead to the violation of the invariant.

```
1 // A_1 - The destroy function can only be called by the owner of the contract
2 invariant over sender : (current_layer = 0 and executing_contract(1) = auction and executing_function(1) =
      destroy and not exception and destroyed(auction)) implies (sender(1) = owner)
3 // A_2 - If a call is made to the bid function and a current_frontrunner already exists,
      the previously deposited money is returned to it
4 invariant over balance : (current_layer = 1 and instruction_pointer(1) = 6 and executing_contract(1) =
      auction and executing_function(1) = bid and old_frontrunner != undef_user and not exception and
      old_frontrunner = user and sender(1) = user) implies (old_balance(user) + old_bid = balance(user)
5 // A_3 - If the bid function is called with a msg.value greater than current_bid then the
      caller become the new current_frontrunner
6 invariant over balance : (current_layer = 0 and executing_contract(1) = auction and executing_function(1) =
      bid and amount(1) > old_bid and not exception) implies (currentFrontrunner = sender(1))
7 // A_4 - If the destroy function is called, all the money in the contract go to the owner
8 invariant over balance : (current_layer = 0 and executing_contract(1) = auction and executing_function(1) =
      destroy and not exception) implies (old_balance(user2) + old_balance(auction) = balance(user2))
```

Listing 5.1: Invariant specification for the Auction model

They can result from coding errors, design flaws, or misunderstandings of the blockchain platform's functionality. In the models we consider for the purposes of this paper, we model a slightly simplified exception-handling mechanism with no full rollback, without affecting the analysis results.

For the Auction contract, we defined four invariants, with their informal descriptions and their ASMETA specification in Listing 5.1. We expect that two of them, A_1 and A_4, would be violated, while the other two should hold as true safety properties. Specifically, A_1 checks for the absence of an unprotected function vulnerability by stating that if the destroy function is executed and the auction contract is *destroyed*, then the transaction sender must be the contract owner. However, this invariant can be violated because the `r_Selfdestruct` rule is not protected since there is no restriction on the sender(current_layer) value. A_4 states that if the destroy function executes successfully (without raising an exception), all the contract's funds are transferred to the owner. However, since the r_Selfdestruct rule uses the transaction sender as the recipient instead of the contract owner, this invariant is likely to be violated. We included them to show that ASE does not generate false positives within the explored state space (see Sect. 6).

Note that some invariants compare the correct value of a location with its value in the previous state; this requires adding, for that specific location, an auxiliary function declaration recording such previous value. Listing 5.2 presents the additional auxiliary functions used in the Auction model to evaluate almost all the invariants in Listing 5.1. Moreover, before executing the ASE tool on the contract's model, a slight modification of the model is needed to deal with:

- *Monitored functions*: ASE does not deal with monitored values, so the transformation monitored foo: $\mathsf{D}_1 \rightarrow \mathsf{D}_2$ in controlled foo: Integer $\times \mathsf{D}_1 \rightarrow \mathsf{D}_2$ is required for each monitored function foo of the model signature, and foo(i,d) yields the monitored value of foo(d) at state i.
- *Number of steps*: a new function controlled stage: Integer is added to the model to index the current execution state during the symbolic execution (it

```
1  signature:
2  .....
3  /* functions to save previos values of contract's arguments */
4    controlled old_frontrunner : User
5    controlled old_bid : MoneyAmount
6    controlled old_balance : User −> MoneyAmount
```

Listing 5.2: Signature of the ASMETA Auction Model

```
1  − this path is taken when the following conditions hold in the initial state:
2  not ((random_sender (0) = auction))
3  not ((random_sender (0) = undef_user))
4  (random_receiver (0) = auction)
5  not ((random_function (0) = bid))
6  ((random_amount (0) >= 0) and ((3 >= random_amount (0)) and not ((random_amount (0) > 0))))
7  (random_sender (0) = user)
8  (random_function (0) = destroy)
9  ...
10 −−− S_2 summary:
11 '_inv_1': met on 62 paths / definitely violated on 1 paths / possibly violated on 0 paths
12 '_inv_2': met on 63 paths / definitely violated on 0 paths / possibly violated on 0 paths
13 '_inv_3': met on 63 paths / definitely violated on 0 paths / possibly violated on 0 paths
14 '_inv_4': met on 61 paths / definitely violated on 2 paths / possibly violated on 0 paths
```

Listing 5.3: Faulty path and invariants violation summary for the Auction model

also yields the current depth of the symbolic execution tree). Initialized to 0, it is incremented by 1 in parallel with all the other rules by r_main rule.

– *Undef value*: the current version of ASE does not support the predefined ASM *undef* value (while ASMETA does). This requires replacing each occurrence of undef with a suitable undef_value upon adding to the signature the declaration static undef_value: D, being D the domain of the model function that could take value *undef*. For example, the guard in line 7 of Listing 3.2 is replaced with currentFrontrunner != undef_user and the declaration static undef_user : User is added to the signature.

These model transformations are currently performed manually but can be automated, and future tool improvements will incorporate this automation. The modified version of the Auction model is available at [12]. When the model, augmented with the four invariants in Listing 5.1, is symbolically executed, ASE identifies violation of invariants A_1 and A_4, as shown in Listing 5.3: invariants are violated in two execution steps (see stage S_2 in line 10), on one out of 62 paths for A_1 and two out of 63 paths for A_4. At this stage S_2, invariants A_2 and A_3 are true and are never violated within the state space ASE constructs. This, of course, does not guarantee their truth at subsequent stages.

Examining ASE's detailed reports on the violated invariants, consider `inv_1`, which corresponds to A_1. The report provides information on the *initial state conditions* required to trigger the faulty execution path (lines 2-8 of Listing 5.3). These initial values for the monitored functions can be used to generate an AS-

META scenario, shown in Listing 5.4.[9] The scenario begins by assigning values to the monitored functions (lines 4-7) ensuring these values adhere to the conditions outlined in Listing 5.3: the value `user` is assigned to `random_sender`, as stated by the condition at line 7; `auction` is assigned to `random_receiver`, following the condition at line 4; `random_amount` is set to 0, as at line 6; `random_function` is set to `destroy`, as in the condition at line 8. From this initial configuration, after two model steps, both invariants A_1 and A_4 are violated.

```
1 scenario inv_1
2 load ../Auction.asm
3
4 set random_sender := user;
5 set random_receiver := auction;
6 set random_amount := 0;
7 set random_function := destroy;
8 step
9 step
```

Listing 5.4: Generated scenario violating `inv_1` in Auction contract

Based on the model analysis results, we refined the contract model by introducing appropriate rule guards to prevent the violation of invariants A_1 and A_4. The symbolic execution of this revised version, *Auction v2* in Table 2 and available at [12], does not violate any invariant, at least within the state space ASE constructs for the given input stage value.

6 Analysis Results

To assess the effectiveness of our vulnerability detection strategy, we applied the approach described in Sect. 3 to model a set of smart contracts. Some of these contracts were inspired by existing repositories [5,20], while others were either widely used or specifically developed for this evaluation.

For each modeled contract, we defined invariants asserting the absence of vulnerabilities, either based on the original contract's repositories or crafted specifically for the given contract. To rigorously test the tool's ability to detect invariant violations and assess false positives or false negatives, all invariants in this dataset were intentionally designed to be false. A detailed list of the smart contracts and their corresponding invariants, described in natural language, is provided in Table 2.

Each smart contract was then slightly modified to enable symbolic execution, following the method described in Sect. 5 for the Auction contract (all models are available at [12]). Multiple versions of each contract were produced, with the same invariants applied across versions. The purpose of these variations was to stress the tool's ability to detect invariant violations, particularly as the number of execution paths and monitored functions increased. This process involved progressively increasing the contract's complexity (such as adding more users or imposing additional conditions on state variables), thereby making symbolic execution increasingly difficult.

Symbolic execution was conducted on each model, exploring a state space of up to 30 stages, with a maximum runtime of 3600 seconds. These limits were set

[9] `set` is the command to provide monitored values; `step` induces a model step.

Contracts	Invariants	
Auction	A_1	The `destroy` function can only be called by the owner of the contract
	A_2	If a call is made to the `bid` function and a `current_frontrunner` already exists, the previously deposited money is returned to it
	A_3	If a call is made to the `bid` function with a `msg.value` greater than `current_bid` then the caller becomes the new `current_frontrunner`
	A_4	If a call is made to the `destroy` function, all the money in the contract goes to the owner
StateDao	B_1	If there was no exception and the contract is not running, the contract's state is INITIALSTATE
	B_2	If a call to `deposit` is made with a `msg.sender` value greater than 0 then it does not raise an exception
	B_3	An exception is not raised even if a call to `deposit` is made and the balance of `state_dao` is greater or equal than 12
	B_4	There is always at least one balance that is greater than the corresponding `customer_balance`
	B_5	If there was no exception and the contract is not running, the balance of `state_dao` is less than 12
Airdrop	C_1	Even if a call to `receive_airdrop` is made and no exceptions are raised, the value for `msg.sender` of `received_airdrop` remains false
	C_2	If a call to `receive_airdrop` is made from an account with `received_airdrop` set to 0, an exception is not raised
	C_3	Not all users received the airdrop
Crowdfund	D_1	If a call to `donate` is made, and no exceptions have been raised, then `donors(msg.sender)` is greater than 0
	D_2	Even if a call to donate is made and the donation phase is over, an exception is not raised
	D_3	If a call to `withdraw` completes without any exceptions being raised, then the sender was the owner of the contract
	D_4	After a call to `reclaim`, if no exceptions are raised, then the value of donors for the sender is 0
KotET	E_1	Every time a user becomes king it must be a different user from the previous king
	E_2	It is not possible for the balance of the contract to reach 0
	E_3	`claim_price` cannot be greater than all user balances
	E_4	If a call to the Kotet `fallback` is made with an amount greater than or equal to `claim_price` an exception is not raised
Baz	F_1	Not all the states are set to true

Table 2: The list of proposed invariants for each smart contract

using the tool's `-invcheck` n parameter and `gtimeout` command. The results are summarized in Table 3, which allows for an effective comparison of the tool performance across different contracts and their versions.

Table 3 is structured into several horizontal sub-tables, each corresponding to a specific contract and its versions. The first column, labeled *Contracts*, lists the

Contracts	EV	FV	Invariant Results														
			A_1	s	t	A_2	s	t	A_3	s	t	A_4	s	t			
Auction v1	2	2	✓	2	1.5	×			×			✓	2	1.5			
Auction v2	0	0	×			×			×			×					
			B_1	s	t	B_2	s	t	B_3	s	t	B_4	s	t	B_5	s	t
StateDao v1	5	5	✓	13	1.0	∽	8	0.6	✓	4	0.2	∽	1	0.1	✓	13	1.0
StateDao v2	5	4	×			∽	8	0.6	✓	4	0.2	∽	1	0.1	∽	7	0.3
			C_1	s	t	C_2	s	t	C_3	s	t						
Airdrop v1	3	3	✓	5	0.3	✓	3	0.2	✓	4	0.2						
Airdrop v2	3	3	✓	5	21	✓	3	3.4	✓	9	1214						
Airdrop v3	3	2	×			✓	5	25	✓	9	1396						
			D_1	s	t	D_2	s	t	D_3	s	t	D_4	s	t			
Crowdfund v1	4	4	∽	5	2.9	✓	4	2.0	✓	7	6.8	✓	10	20			
Crowdfund v2	4	4	∽	4	1.5	✓	4	1.5	✓	7	5.6	∽	13	62			
			E_1	s	t	E_2	s	t	E_3	s	t	E_4	s	t			
Kotet v1	4	4	✓	12	9.2	∽	0	0	∽	0	0	✓	3	0.3			
Kotet v2	4	3	×			∽	0	0	∽	0	0	✓	3	1.9			
			F_1	s	t												
Baz	1	1	✓	29	515												

Table 3: Symbolic Execution results

contract names and versions. The second column, (EV), represents the number of *expected violations*, i.e., the number of invariant violations the tool ideally should find. The third column, FV, indicates the number of invariant violations actually found by the tool. Ideally, the FV value should closely match the EV value, reflecting the tool's accuracy. The final section of Table 3 presents the aggregated execution results. Each row in this section corresponds to an individual invariant and includes three key evaluation metrics: the *invariant ID* (e.g., A_1, ...,A_n), the stage s at which the invariant was violated (representing the depth of the state in the execution), and the time t in seconds taken to reach that stage.

While the values for s and t are straightforward, the *invariant ID* column may contain one of three different symbols:

- ✓: the tool confirmed that the corresponding invariant is always violated on at least one path at stage s ("*definitely violated*", see Sect. 4, item 2c);
- ∽: the tool found that the corresponding invariant is violated in some cases on at least one path at stage s ("*possibly violated*", see Sect. 4, item 2c);
- ×: the tool did not detect any violation within the defined state space, which is bounded by the time and stage limits.

Note that, in our context, even if a property is possibly violated, it can be counted as a violation, since a dangerous system configuration exists. Therefore, observing the results in Table 3, we note that ASE was able to identify all invariant violations in the initial versions of each contract (i.e., those corresponding to

Solidity deployed contracts), as we expected. Indeed, for these versions, the number of find violations FV is the same as the number of the expected violations EV. However, performance tends to decline for later versions, with increased execution times or failure to detect violations. In these cases, the value of FV is less than that of EV.

Despite the promising results, the approach suffers from the well-known limitation of symbolic execution path explosion. In the ASE tool, this is exacerbated by the sequential execution logic of an ASM. This limitation also affects the symbolic execution of multi-agent ASMETA models (not used here, but in [11,10]) useful to model *good* contracts operating in combination with *bad* contracts, namely those that try to exploit the vulnerability to make an attack. Ideas on how to overcome this limitation are to be addressed in future work.

However, compared to the model checking approach employed in [11,10,30] to guarantee the safety properties of Solidity contracts, by using the symbolic execution-based strategy we can deal with infinite domains, and the encountered path explosion problem is very limited with respect to the state explosion problem of the model checker.

7 Related Work

Numerous automated tools exist for analyzing, testing, and debugging Ethereum smart contracts. However, as demonstrated by [32], current tools fail to detect a significant majority (approximately 80%) of exploitable bugs found in real-world smart contracts.

An analysis of several surveys and reviews, including [32], [1], and [5], was conducted to identify some of the most widely used and effective smart contract analysis tools.

Several tools address the challenge of identifying vulnerable behavior in smart contracts, employing techniques such as formal methods, theorem proving, model checking, runtime verification, and fuzzing to specify and verify properties and invariants. Certora [22] is a commercial tool offering a proprietary cloud-based platform for verification, where property specifications are separated from the contract code. Properties are expressed in the Certora Verification Language (CVL), an extension of Solidity incorporating metaprogramming primitives. While the verification process is effective, using Certora requires learning CVL and the tool works as a black box. Similarly, Halmos [3] offers a Certora like workflow, by exploiting a combination of symbolic execution and testing. It allows to execute Solidity tests providing all possible inputs.

The work presented in [28] utilizes the K-Framework [13], enabling smart contract analysis through runtime verification of bytecode, rather than Solidity, subsequently, many other tools try to improve K-Framework usability and effectiveness, like KEVM [21] or Kontrol [26]. Isabelle/HOL is employed to verify the EVM bytecode of smart contracts. This process involves partitioning contracts into basic blocks, with the properties of each block proven using Hoare triples. Similarly TLA+ [23] is used in [17] to analyze different security-critical smart

contracts. They have been able to detect bugs, including reentrancy. These aforementioned approaches often demand a strong mathematical and logical background, employing complex mathematical notations.

SolCMC [2] is a symbolic model checker integrated into the Solidity compiler since 2019. Developers specify properties using assert statements within the contract code. Similarly, HEVM [18] is a symbolic EVM written in Haskell language. However, these approaches are limited to the verification of assertions placed at specific positions in the code.

Finally, Echidna [20] is a Haskell program for fuzzing and property-based testing of Ethereum smart contracts. It automatically generates inputs and verifies user-defined invariants. However, fuzzing necessitates substantial computational power and resources.

ASM/ASMETA offers a more accessible alternative for smart contract verification. Models appear as high-level programs, use a simple notation and basic control flow constructs, are executable and supported by other lightweight validation techniques. This allows for immediate feedback on model reliability, providing a preliminary assessment before resorting to more complex verification methods. In [10,11,30], we employed the *code to model* translation schema outlined in Sect. 3 and used the NuSMV model checker to verify CTL properties, facing the classical limitations of state explosion and domain size.

8 Conclusion

In this paper, we presented a new strategy for vulnerability detection in Solidity smart contracts through the symbolic execution of ASMETA models. This approach required enhancements to the ASE tool, mainly to adapt it to the ASMETA model notation and to deal with model invariant checking. The preliminary results shown here and obtained on (different versions of) various contracts, confirm that the method is promising for design-time detection of vulnerabilities. However, the current focus is on contract-intrinsic issues and does not yet address interactions with other smart contracts. Some work is planned in the future: (1) optimizing the ASE tool, improving its consistency with ASMETA (e.g., handling *undef* values and monitored functions) and developing a more user-friendly interface; (2) exploring partial order reduction techniques to mitigate the path explosion problem in multi-agent models; (3) automating some key steps, such as mapping Solidity code to ASMETA models, preparing models for ASE execution, and generating execution scenarios from the initial states identified in invariant violations; (4) expanding the approach to a broader range of concrete smart contracts, particularly those with unknown vulnerabilities; (5) evaluating the method on contract models with a fully specified exception-handling mechanism.

References

1. Almakhour, M., Sliman, L., Samhat, A.E., Mellouk, A.: Verification of smart contracts: A survey. Pervasive and Mobile Computing **67**, 101227 (2020).

https://doi.org/https://doi.org/10.1016/j.pmcj.2020.101227
2. Alt, L., Blicha, M., Hyvärinen, A.E., Sharygina, N.: SolCMC: Solidity Compiler's Model Checker. In: 34th Int. Conf. on Computer Aided Verification, CAV 2022. pp. 325–338. Springer-Verlag (2022). https://doi.org/10.1007/978-3-031-13185-1_16
3. Andreessen Horowitz VC: Halmos. `https://github.com/a16z/halmos` (2025)
4. Arcaini, P., Gargantini, A., Riccobene, E., Scandurra, P.: A model-driven process for engineering a toolset for a formal method. Software: Practice and Experience **41**(2), 155–166 (2011). https://doi.org/10.1002/spe.1019
5. Bartoletti, M., Fioravanti, F., Matricardi, G., Pettinau, R., Sainas, F.: Towards Benchmarking of Solidity Verification Tools. In: 5th Int. Workshop on Formal Methods for Blockchains (FMBC 2024). Open Access Series in Informatics (OASIcs), vol. 118, pp. 6:1–6:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2024). https://doi.org/10.4230/OASIcs.FMBC.2024.6
6. Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: ASMETA Tool Set for Rigorous System Design. In: Formal Methods. vol. 14934, pp. 492–517. Springer, Cham (2025). https://doi.org/10.1007/978-3-031-71177-0_28
7. Börger, E., Raschke, A.: Modeling Companion for Software Practitioners. Springer (2018). https://doi.org/10.1007/978-3-662-56641-1
8. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer (2003). https://doi.org/10.1007/978-3-642-18216-7
9. Braghin, C., Cimato, S., Damiani, E., Baronchelli, M.: Designing Smart-Contract Based Auctions. In: Security with Intelligent Computing and Big-data Services, SICBS 2018. pp. 54–64. Springer International Publishing, Cham (2020). https://doi.org/0.1007/978-3-030-16946-6_5
10. Braghin, C., Riccobene, E., Valentini, S.: An ASM-Based Approach for Security Assessment of Ethereum Smart Contracts. In: Proc. of the 21st Int. Conf. on Security and Cryptography, SECRYPT 2024. pp. 334–344. SCITEPRESS (2024). https://doi.org/10.5220/0012858000003767
11. Braghin, C., Riccobene, E., Valentini, S.: Modeling and verification of smart contracts with Abstract State Machines. In: Proc. of the 39th ACM/SIGAPP Symposium on Applied Computing, SAC 2024. pp. 1425–1432. Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3605098.3636040
12. Braghin, C., Riccobene, E., Valentini, S.: Ethereum Via ASM. `https://github.com/smart-contract-verification/ABZ2025` (2025), version used in this paper: `https://github.com/smart-contract-verification/ABZ2025`.
13. Ştefănescu, A., Park, D., Yuwen, S., Li, Y., Roşu, G.: Semantics-Based Program Verifiers for All Languages. In: Proc. of the 31th Conf. on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016. pp. 74–91. ACM (2016). https://doi.org/10.1145/2983990.2984027
14. Del Castillo, G.: Using Symbolic Execution to Transform Turbo Abstract State Machines into Basic Abstract State Machines. In: 10th Int. Conf. on Rigorous State-Based Methods, ABZ 2024. Lecture Notes in Computer Science, vol. 14759, pp. 215–222. Springer (2024). https://doi.org/10.1007/978-3-031-63790-2_15
15. Del Castillo, G.: Using symbolic execution to transform turbo Abstract State Machines into basic Abstract State Machines (extended version) (2024), `https://github.com/constructum/asm-symbolic-execution/blob/main/doc/2024--Del-Castillo--extended-version-of-ABZ-2024-paper.pdf`
16. Del Castillo, G.: ASM Symbolic Execution. `https://github.com/constructum/asm-symbolic-execution` (2025), version used for the experiments presented in this paper: `https://github.com/constructum/asm-symbolic-execution/tree/32251c45d43b41f39cdbff061ddd64914976c244`.

17. Dfinity: Eliminating smart contract bugs with TLA+ (2023), https://medium.com/dfinity/eliminating-smart-contract-bugs-with-tla-e986aeb6da24
18. Dxo, Soos, M., Paraskevopoulou, Z., Lundfall, M., Brockman, M.: Hevm, a Fast Symbolic Execution Framework for EVM Bytecode. In: International Conference on Computer Aided Verification. pp. 453–465. Springer (2024)
19. Fekih, R.B., Lahami, M., Jmaiel, M., Bradai, S.: Formal Verification of Smart Contracts Based on Model Checking: An Overview. In: IEEE Int. Conf. on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2023. pp. 1–6 (2023). https://doi.org/10.1109/WETICE57085.2023.10477834
20. Grieco, G., Song, W., Cygan, A., Feist, J., Groce, A.: Echidna: effective, usable, and fast fuzzing for smart contracts. In: Proc. of the 29th ACM SIGSOFT Int. symposium on software testing and analysis. pp. 557–560 (2020). https://doi.org/10.1145/3395363.3404366
21. Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A., et al.: KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF). pp. 204–217. IEEE (2018)
22. Jackson, D., Nandi, C., Sagiv, M.: Certora technology white paper, https://docs.certora.com/en/latest/docs/whitepaper/index.html
23. Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems (TOPLAS) **16**(3), 872–923 (1994)
24. Marmsoler, D., Ahmed, A., Brucker, A.D.: Secure Smart Contracts with Isabelle/-Solidity. In: Madeira, A., Knapp, A. (eds.) Software Engineering and Formal Methods. pp. 162–181. Springer Nature Switzerland, Cham (2025)
25. New Alchemy: A short history of Smart Contract hacks on Ethereum: A.k.a. why you need a smart contract security audit. https://medium.com/new-alchemy/a-short-history-of-smart-contract-hacks-on-ethereum-1a30020b5fd (2018)
26. Runtime Verification Inc.: Kontrol. https://github.com/runtimeverification/kontrol (2025)
27. Siegel, D.: Understanding The DAO Attack (2016), https://www.coindesk.com/learn/understanding-the-dao-attack/
28. Sotnichek, M.: Formal verification of smart contracts with the K framework (2019), https://www.apriorit.com/dev-blog/592-formal-verification-with-k-framework
29. Tolmach, P., Li, Y., Lin, S.W., Liu, Y., Li, Z.: A survey of smart contract formal specification and verification. ACM Comput. Surv. **54**(7) (Jul 2021). https://doi.org/10.1145/3464421
30. Valentini, S., Braghin, C., Riccobene, E.: A modeling and verification framework for ethereum smart contracts. In: 10th Int. Conf. on Rigorous State-Based Methods, ABZ 2024. Lecture Notes in Computer Science, vol. 14759, pp. 201–207. Springer (2024). https://doi.org/10.1007/978-3-031-63790-2_13
31. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)
32. Zhang, Z., Zhang, B., Xu, W., Lin, Z.: Demystifying exploitable bugs in smart contracts. In: 2023 IEEE/ACM 45th Int. Conf. on Software Engineering, ICSE. pp. 615–627. IEEE (2023). https://doi.org/10.1109/ICSE48619.2023.00061