

# An ASM-based approach for security assessment of Ethereum Smart Contracts<sup>ab</sup>

C. Braghin<sup>1</sup> , E. Riccobene<sup>1</sup>  and Simone Valentini<sup>1</sup> 

<sup>1</sup>*Department of Computer Science, Università degli Studi di Milano, Italy*  
{chiara.braghin, elvinia.riccobene, simone.valentini}@unimi.it

**Keywords:** Blockchain, Smart-Contracts, Exception, Verification, Validation, ASMs, ASMETA

**Abstract:** Blockchain-based smart contracts are gaining widespread adoption due to their potential to automate complex transactions securely and transparently. However, ensuring the correctness and security of smart contracts remains a challenge. This paper proposes a novel approach to modeling and verifying Ethereum smart contracts' exception-related vulnerabilities using Abstract State Machines (ASMs). ASMs provide a formal modeling language that enables the precise representation of system behavior and properties. We developed an ASM model of a Solidity smart contract and demonstrated its use on Unhandled Exception vulnerability identification and check contract correctness. Our approach offers a formal framework for smart contract modeling and verification. It leverages the power of ASM tools to identify vulnerabilities and ensure contract reliability, contributing to more secure and trustworthy blockchain-based applications.

## 1 INTRODUCTION

Blockchain was born as a powerful and innovative solution that implements a distributed ledger to manage people's finances without the need for intermediaries or trusted third parties. In late 2013, Vitalik Buterin introduced Ethereum, a decentralized, open-source blockchain platform designed to facilitate decentralized, trustless, and programmable interactions on the blockchain through smart contracts [1]. The Ethereum Virtual Machine (EVM) serves as the engine behind Ethereum's smart contract execution and transaction processing, ensuring decentralized and deterministic operation across the network. Operating as a distributed state machine, Ethereum maintains a unified state among its decentralized nodes, empowering the execution of smart contracts and decentral-

ized applications with verifiable outcomes [2].

Since smart contracts are also stored on the blockchain, they cannot be modified after deployment. In the past, many attacks on vulnerable smart contracts proved that a stronger verification and validation phase before the contract's deployment is required. For instance, the infamous DAO hack drained \$70 million worth of Ether [3], and also the popular King of the Ether Throne (KotET) contract [4] had a fatal flaw.


In [5], we explored the application of Abstract State Machines (ASMs) for specifying and verifying Ethereum smart contracts written in Solidity. We formalized the EVM and key language primitives, enabling the validation of functional correctness through rigorous proofs. Our approach facilitates verification of intra-contract properties and modeling of inter-contract interactions, allowing assessment of a contract's robustness against potential attacks.


In this paper we go one step further: we extend the work presented in [5] by incorporating the exception-handling mechanisms. This enhancement enables us to identify and evaluate an additional category of potential vulnerabilities due to exceptions and errors within smart contracts (e.g., exception disorder or call-stack overflows), thereby enhancing the completeness of security assessments.


As a proof of concept, we introduce a vulnerable running example, its corresponding ASM model and

<sup>a</sup>This work was supported in part by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the Italian MUR. Neither the European Union nor the Italian MUR can be held responsible for them.

<sup>b</sup>This work was partially supported by the PRIN project SAFEST (G53D23002770006).

<sup>c</sup> <https://orcid.org/0000-0002-9756-4675>

<sup>d</sup> <https://orcid.org/0000-0002-1400-1026>

<sup>e</sup> <https://orcid.org/0009-0005-5956-3945>

a new attacker model exploiting the exception vulnerabilities. Through a comprehensive validation and verification process applied to the contract model, we demonstrate the efficacy of our proposed approach.

The paper is structured as follows. Section 2 introduces ASMs as a formal method for system engineering, explaining the fundamental concepts of ASMs, including states, locations, update rules, and transition rules. In Section 3, we provide a background description of the functioning of smart contracts and exception handling in Ethereum. Section 4 presents the detailed modeling of the EVM and Solidity language primitives using ASMs. We describe the structure of an ASM model and demonstrate its ability to capture the behavior of smart contracts for exception handling. Section 5 introduces the running case study consisting of a Solidity auction contract. Section 6 describes the validation and verification process. We explain how it is possible to define both intra- and inter-contract properties using invariants and CTL formulas. Section 7 reviews existing tools for automated analysis and testing of Ethereum smart contracts. Section 8 concludes the paper.

## 2 ABSTRACT STATE MACHINES

Abstract State Machines [6] (ASMs) is a state-based approach to system engineering. ASMs capture a system's configuration through states and articulate its behavior via a sequential series of state transitions. Compared to Finite Automata, ASMs are more expressive since they replace the concept of unstructured *state* with the concept of state as a mathematical *algebra*, proving a more expressive formal language.

System data and operations on data are represented by *domains* of elements and *functions* defined on them.

State memory units are given as a set of *locations*; they are pairs of the form  $(f(v_1, \dots, v_n), v)$ , meaning that the function  $f$  interpreted on parameters  $v_1, \dots, v_n$  has value  $v$  (assigned in the function codomain).

*State transitions* are given by firing well-formed transition rules, i.e., rules constructed by a given and fixed number of *rule constructors*.

The basic transition rule, which allows for updating locations, is the *update rule* of the form of  $f(v_1, \dots, v_n) := d$ . By firing this rule, in the next state, the value of the function  $f(v_1, \dots, v_n)$  is updated to  $d$ . Update rules are usually embedded in more complex *transition rules*. The rule constructors used for our modeling are those for guarded updates (if-then, switch-case), parallel updates (par), and nondeterministic updates (choose).

```

1
2 import ../../Libraries/CTLLibrary
3 import ../../Libraries/StandardLibrary
4
5 signature:
6   /*DOMAINS DEFINITION*/
7   domain Domain subsetof Integer
8   ...
9   /*LOCATIONS DEFINITION*/
10  controlled location : Domain
11  ...
12 definitions:
13   /*TRANSACTION RULES*/
14   rule r_Rule =
15     ...
16   ...
17   /*INVARIANTS */
18   invariant over locations : term
19   ...
20   /*TEMPORAL LOGIC FORMULAS */
21   CTLSPEC ...
22   ...
23   /*MAIN RULE*/
24   main rule r_Main = ...
25
26 default init s0:
27   /*INITIAL STATE DEFINITION*/
28   function location = value
29   ...

```

Listing 1: ASM model schema

Depending on whether transition rules update locations or not, functions are classified in *static* (never change value) and *dynamic* (value updated by rules). Dynamic functions updated by transition rules are called *controlled*, those updated by the environment are called *monitored*, and those updated by both are called *shared*.

An ASM *run* is a (finite or infinite) sequence  $S_0, S_1, \dots, S_n, \dots$  of states. Starting from the initial state  $S_0$ , in a computation step (*run step*) from  $S_n$  to  $S_{n+1}$ , all enabled transition rules are executed in parallel, leading to simultaneous updates of a number of locations. In case of an inconsistent update (i.e., the same location is updated to two different values) or invariant violations (i.e., some property that must be true in every state is violated), the model execution stops with error.

Besides different ways to structure the control flow of functions update (depending on the use of rule constructors), ASMs can also model different computational paradigms, ranging from a single agent executing the whole transition system to distributed (synchronous or asynchronous) multi-agents working in parallel, each executing its transition system (or agent's *program*).

## 2.1 ASMETA Toolset

*ASMETA* [7] is an Eclipse-based set of integrated tools tailored to the ASM formal method. It offers support across the diverse stages of system development, from specification to analysis.

It provides a textual user-friendly pseudo-code format to write ASM models, with an editor and a syntax checker. An *ASMETA* model has a well-formed structure as reported in Listing 1: a model is composed by an `import` section that allows the import of external libraries and user-defined models that provide predefined domains, functions, and rules; a `signature` section to declare specific domains and functions; a `definitions` section to define specific static domains and functions, transition rules, state invariants, and properties to be verified; it also encompasses the `main rule`, which is the starting point of the machine computation and that may trigger other transition rules; in case of multi-agent models, the `main rule` can be used to orchestrate the execution of different agents; finally, the `init` section defines the values for the model's initial state.

A core part of the toolset is the *AsmetaS* simulator [8], enabling model validation by simulating the execution of ASM models and verifying adherence of the model to the requirements. *AsmetaS* offers a rich set of functionalities, including support for a wide range of validation tasks, from invariant checking, consistent update checking, random simulation, to interactive simulation modes. For more powerful scenario-based validation, the *AsmetaV* [9] tool uses the *AsmetaS* simulator along with the *AVallA* language to express scenarios as sequences of actor actions and expected machine reactions, enabling comprehensive validation of execution scenarios on ASM models.

Formal verification of ASM models is handled by the *AsmetaSMV* [10] tool, which translates ASM models into input for the NuSMV [11] model checker. *AsmetaSMV* supports specifying and checking both linear temporal logic (LTL) and computation tree logic (CTL) properties.

## 3 ETHEREUM AND SMART CONTRACTS

A blockchain is a decentralized, distributed ledger technology that records transactions across a network of computers. It is composed of blocks of data, each containing a set of transactions, a unique cryptographic hash of the previous block and a timestamp, thus creating a chain of blocks. This structure gives the blockchain the characteristic of immutabil-

ity: once recorded, the content of a block cannot be altered without changing all subsequent blocks, making retroactive data tampering virtually impossible.

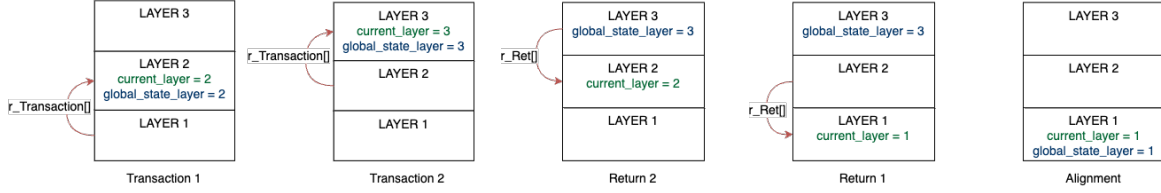
Ethereum [1] is an open-source blockchain platform that enables developers to build and deploy decentralized applications: unlike Bitcoin, which primarily focuses on peer-to-peer electronic cash transactions, Ethereum provides the EVM, a decentralized runtime environment for executing smart contracts, which are self-executing contracts with the terms of the agreement directly written into code. They are written in high-level languages such as Solidity and compiled into bytecode that can be executed by the EVM.

The EVM runs on every node participating in the Ethereum network, ensuring that smart contracts are executed in a decentralized manner. It also maintains the state of the Ethereum blockchain, which includes account balances, smart contract code, and storage. There are two types of accounts that 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 (used as account address) and controlled by the corresponding private key that is used to sign transactions to prove ownership and authorization (by means of elliptic curve digital signature algorithm). Contract accounts are special accounts that have associated code and data storage, a unique address, but not private keys. When a transaction involving a smart contract is initiated, the EVM receives the bytecode associated with the contract and executes it, modifying the state of the blockchain 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.

*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.

In Solidity, exceptions can broadly be categorized into two types: built-in exceptions and custom exceptions. *Out-of-Gas Exceptions* are built-in exceptions that occur when a transaction runs out of gas before completing its execution and cause the transaction to revert, i.e., the transaction is terminated, and any changes made to the blockchain state during the transaction are rolled back, ensuring that the blockchain remains in a consistent state and preventing incomplete or erroneous transactions from being included in the blockchain. Custom exceptions are *user-defined exceptions* that are triggered by `require` or `revert`

Sequence of transactions with no exceptions thrown



Sequence of transactions with an exception thrown during Transaction 3

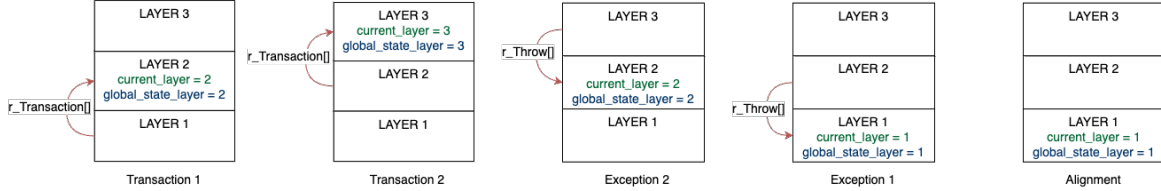


Figure 1: Diagram representing how the two pointers behave in different situations

statements within a smart contract

Solidity provides various functions to transfer Ether between contracts or to external addresses, differing in terms of their behavior, gas management, and error handling mechanisms. For instance, both the `transfer` and `send` functions automatically forward a fixed amount of gas (currently 2300 gas) along with the transfer operation, typically sufficient for basic Ether transfers but potentially insufficient for complex recipient contract operations. In contrast, the `call` function allows for custom gas limits to be specified. Regarding error handling, if a transfer fails due to exceptions thrown by the recipient or an out-of-gas exception, both `call` and `transfer` functions revert the transaction, undoing any state changes made before the call. Instead, in case of failure, the `send` function does not revert the transaction but forwards any remaining gas to the recipient and continues execution, returning *false*, thus enabling the contract to handle the failure gracefully.

## 4 MODELING ETHEREUM

In [5], we formalized in ASM the Ethereum virtual machine and Solidity key primitives. We also developed an approach to systematically translate Solidity smart contracts into ASMETA models: for each smart contract there is a corresponding ASMETA model that includes the EVM library and a transition rule for each Solidity function. The rule is mainly a `switch` case with as many cases as the number of instructions in the function.

The EVM library models the *execution stack* that keeps track of the active smart contract functions

and of the global state of the blockchain. The *stack frame* represents an element of the stack recording the account address that initiated the transaction, the receiver, the amount of ether transferred, the running function, the current instruction being executed and the smart contract the function belongs to. The stack pointer, pointing to the stack frame of the function currently being executed (i.e., at the top of the stack), is modeled by the controlled function `current_layer`.

To manage transactions and the updating of the stack pointer, two different rules have been developed: `r_Transaction`, to perform a transaction, and `r_Ret`, to terminate one. `r_Transaction` rule models the execution of a transaction by increasing `current_layer` and storing all the data on the new execution environment in the stack. It also increments the caller's instruction pointer to continue with the execution left pending once `current_layer` is decreased, i.e., when the function returns. `r_Ret` rule is called when a function ends its execution, the last stack frame is popped and the stack pointer is decremented. Figure 1 visually illustrates the stack structure and the dynamics of the `r_Transaction` and `r_Ret` rules. Specifically, the top row exemplifies a scenario with an initial function call (i.e., Transaction 1) subsequently invoking another function (i.e., Transaction 2). Then, the two functions conclude (i.e., Return 2 followed by Return 1).

In this paper, we improve the approach and model exceptions, enabling us to address all potential attacks stemming from the misuse of exceptions. Indeed, various attacks exploit vulnerabilities related to exceptions, such as *exception disorder*, a common issue resulting from Solidity's inconsistent error handling during function calls, as well as the *call stack overflow*

vulnerability, which arises when a program allocates excessive data on the call stack.

In order to emulate the Ethereum exception-handling mechanism, we needed to implement the rollback feature. As explained in Section 3, when an exception is thrown, a rollback process is performed to revert the blockchain global state to the point before the exception occurred. To this aim, we leveraged the inherent nature of ASMs as a state-based system, encompassing the global state of the blockchain. As a result, we added a new function `global_state_layer` behaving similarly to the `current_layer` function, pointing to a snapshot of the appropriate global state right before potential exception locations. Then, if an exception happens, the global state can be easily reset by referencing `global_state_layer` to the data of a previous global state. The snapshot is performed by the `r_Save` rule, while the `r_Throw` rule mimicks the throwing of an exception during execution, such as exceeding the maximum call stack depth. This rule decrements `global_state_layer` to revert to the previously validated state and then triggers the `r_Ret` rule to update also the `current_layer`.

Figure 1 provides a visual representation of how the `current_layer` and `global_state_layer` functions behave when a transaction ends normally versus when it throws an exception. The scenario depicted in the top row illustrates the case when Transaction 2 naturally concludes with a return statement and no exception is thrown. In this case, the `current_layer` function is decreased to resume the execution environment of the previous layer, but the `global_state_layer` remains unchanged to record the changes in the global state performed by Transaction 2. No rollback occurs because the data generated in the third layer remains valid.

The scenario in the bottom row of Figure 1 illustrates the case of an exception raised during the execution of Transaction 2. This abnormal termination triggers a rollback, resetting the blockchain global state to the previously validated data (i.e., the global state reached after the execution of Transaction 1). As a result, both `current_layer` and `global_state_layer` functions are decremented: `current_layer` moves back to the second layer to continue the execution of Transaction 1 since the execution of Transaction 2 has failed, while the global state reverts to the state before the execution of Transaction 2. In the end, when all transactions conclude either successfully or raise an exception, `current_layer` is equal to 1. At this point, the values of the two functions are aligned.

Figure 1 also clarifies the general behavior of the two functions: during a sequence of nested func-

tion calls, whenever a function concludes successfully with a `r_Ret` rule, only `current_layer` is decremented, while `global_state_layer` remains pointing to the new global state of the blockchain. However, in the event of an exception, both are decremented.

## 5 AUCTION CASE STUDY

In this Section, we introduce a running case study, serving as a proof of concept for our validation and verification approach concerning vulnerable smart contracts. The case study is a Solidity *Auction contract* [12], commonly encountered in real-world blockchain applications.

```

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

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 requires that (1) the submitted bid value (`msg.value`) is greater than the current highest bid, and (2) that if there is an existing highest bidder (`currentFrontrunner != 0`), the contract must refund their bid amount before assigning the new bid.

This contract can reach a Denial of Service (DoS) state due to an *exception disorder* vulnerability. Indeed, a malicious user can design a malicious contract that (i) proposes the highest bid to become the current front-runner, and (ii) contains a fallback function (i.e., the function that is triggered when the address receiving a transfer does not have a `receive` function) containing operations that require more than the fixed amount of gas of the `send` function. Then, when another contract makes a bid higher than the attacker's one, the Auction contract tries to refund the bid to the attacker and an *out-of-gas* exception is raised, making the `send` function returns *false* (see Section 3). In this way, also the Auction contract raises an exception



when the `require` statement receives the *false* return value. In this case, the exception triggers a rollback of the entire `bid` function execution. As a consequence, the attacker is able to keep its front-runner position until the auction is finished since the `Auction` contract will not be able to accept any new higher bid.

From the contract code, it is possible to specify the corresponding model<sup>1</sup> by following the schema described in [5]. Listing 2 reports the rule `r_Bid`, which is triggered when `bid` is the currently executing function in the current layer. The `switch` statement of the rule controls the execution flow based on the value of the instruction pointer in the current layer. Firstly, (case 0) the rule enforces a requirement that the amount specified in the `current_layer` must be greater than the current bid stored in the `global_state_layer`; then, (case 1) if the current front runner for the bid is defined, the instruction pointer is incremented to proceed 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 and calling the fallback function. After the transaction is performed, case 3 requires that the response from the previous transaction is successful. The last cases aim to set the current front runner with the transaction sender and to update the current bid with the transaction amount.

## 6 CONTRACT VERIFICATION

Vulnerabilities in smart contracts can lead to security breaches, financial losses, and other undesirable consequences. It is possible to identify two kinds of vulnerabilities, *intra-contract* and *inter-contract*. Intra-contract vulnerabilities exist within a single smart contract and can arise from coding errors, design flaws, or misunderstandings of the underlying blockchain platform’s functionality. Inter-contract vulnerabilities are due to unsafe interaction of smart contracts. They occur when a smart contract interacts with another specific contract containing functions designed to exploit a vulnerability.

In the following, we discuss how intra/inter-contract vulnerabilities can be detected in ASM models of smart contracts by using the ASMETA toolset for validation and verification.

Model validation is possible through the use of AValLa (ASM Validation Language) *scenarios*, i.e., interaction sequences consisting of actions by external actors (users) that can set monitored function values and check the internal machine state, combined

<sup>1</sup>The model is available at <https://github.com/smart-contract-verification/ethereum-via-asm>

with actions of the machine that performs steps by executing the model transition rules.

Model verification is possible by using AsmetaSMV, which translates an ASM model into a NuSMV model and invokes the symbolic model checker on specified temporal logic properties. In case a property is false, a full counterexample, i.e., a possible run leading to a state where the property is false, is provided. Such a run can then be translated into an AValLa scenario (by exploiting other features of the ASMETA toolset [13]), which can reproduce the same faulty model execution.

### 6.1 Checking intra-contract vulnerabilities

Examples of intra-contract vulnerabilities include integer overflows/underflows, where arithmetic operations can produce unexpected results due to the limited range of integer data types, and access control issues, where unauthorized parties can manipulate sensitive contract data or functions. One specific type of intra-contract vulnerability is the *unprotected function vulnerability*. This occurs when a contract exposes sensitive functions without proper access control mechanisms, allowing unauthorized parties to execute them.

In Solidity, a known example of an unprotected function is the powerful `selfdestruct` function that terminates the contract and sends any remaining Ether in the contract’s balance to a specified address. Usually, this function is protected so that only the contract owner can disable the contract. If this function is not properly protected, an attacker can potentially call it and force the contract to self-destruct, leading to a denial of service (DoS) state.

Consider the scenario in which the following `destroy` function, which uses the `selfdestruct` function in an unprotected way, would be added to the `Auction` smart contract:

```
1 function destroy() {
2     selfdestruct(msg.sender);
3 }
```

When translating the `Auction` contract extended with the `destroy` function into an ASMETA model, the latter is mapped into the `r_Destroy` rule reported in Listing 3.

The rule checks if the `executing_function` is `destroy`, then, if `instruction_pointer` is 0, the `r_Selfdestruct` rule is called. The latter rule (not reported here) behaves like the Solidity `selfdestruct` function: it sets to true a predefined boolean function `disabled`, defined on the `Users` do-

```

1 rule r_Save ($n in StackLayer) =
2   par
3     currentFrontrunner($n) := currentFrontrunner(global_state_layer)
4     currentBid($n) := currentBid(global_state_layer)
5   endpar
6
7 rule r_Bid =
8   if executing_function(current_layer) = bid then
9     switch instruction_pointer(current_layer)
10      case 0 :
11        r_Require[amount(current_layer) > currentBid(global_state_layer)]
12      case 1 :
13        if isDef(currentFrontrunner(global_state_layer)) then
14          instruction_pointer(current_layer) := instruction_pointer(current_layer) + 1
15        else
16          instruction_pointer(current_layer) := 4
17        endif
18      case 2 :
19        r_Transaction[auction, currentFrontrunner(global_state_layer), currentBid(
20          global_state_layer), fallback]
21      case 3 :
22        r_Require[call_response(current_layer+1)]
23      case 4 :
24        par
25          currentFrontrunner(global_state_layer) := sender(current_layer)
26          instruction_pointer(current_layer) := instruction_pointer(current_layer) + 1
27        endpar
28      case 5 :
29        par
30          currentBid(global_state_layer) := amount(current_layer)
31          instruction_pointer(current_layer) := instruction_pointer(current_layer) + 1
32        endpar
33      case 6 :
34        r_Ret[]
35    endswitch
36  endif

```

Listing 2: Model translated from the Auction solidity contract

```

1 rule r_Destroy =
2   if executing_function(current_layer) =
3     destroy then
4     switch instruction_pointer(
5       current_layer)
6       case 0 :
7         r_Selfdestruct[owner]
8       case 1 :
9         r_Ret[]
10    endswitch
11  endif

```

Listing 3: The r\_Destroy rule

main, which specifies whether the provided contract's user has been disabled or not.

To ensure that the selfdestruct function is used securely (i.e., only the contract owner can terminate the contract), we add the following state invariant into the contract model that includes the r\_Destroy rule.

It is specifically designed to ensure the protected invocation of the r\_Selfdestruct subrule.

```

1 invariant over sender : executing_function(
  current_layer) = destroy implies sender(
  current_layer) = owner(global_state_layer)

```

It establishes that, if the executing\_function value is destroy, then the transaction sender must be the contract's owner. Therefore, the invariant would be violated if the function's caller is someone different from the contract owner.

The simple model simulation is not enough to reveal an invariant violation, and we need to use the model checker to guarantee that the protected function property is not violated by the behavior of the r\_Selfdestruct rule in any possible computation.

During the translation of an ASMETA model into a NuSMV model, a state invariant is automatically translated into a CTL AG (i.e., *always along all paths*)

property to guarantee that the property holds in all possible runs. In the case of a false property, the AsmetaSMV model checker returns a counterexample, which can be rented as an Avalla scenario.

The following scenario is, for example, obtained by the proof of the faulty AG property corresponding to the invariant.

```

1 load ../Auction.asm
2
3 set random_user := auction;
4 set random_amount := 0;
5 set random_function := destroy;
6
7 step
8 set random_user := auction;
9 set random_amount := 0;
10 set random_function := destroy;
11
12 step
13 step

```

Executing this scenario leads to an illegal state where the invariant is broken, and an Invariant violation message is reported. This feature allows the designer to have concrete evidence of the vulnerability.

## 6.2 Checking inter-contract vulnerabilities

Inter-contract vulnerabilities are more complex and involve interacting smart contracts. Indeed, to discover these vulnerabilities, we need an attacker contract able to interact with the vulnerable contract and to exploit the vulnerability.

In [5], we presented how to model attackers, i.e., specific smart contracts endowed with functions tailored to exploit a vulnerability, and how to execute a given model of a smart contract in combination with an attacker. We also started to build a library of attackers, differing in the capabilities they have and based on well-known vulnerabilities. However, these attacker models do not exploit exception disorder vulnerabilities. Instead, the approach presented here covers cases of attacks dealing with exceptions.

The contract presented in Section 5 encompasses an *exception disorder* vulnerability that allows an attacker to bring the auction contract to a denial of service (DoS) state, preventing an honest user from becoming the new front-runner.

The attacker model specified to exploit the exception disorder vulnerability contains two different rules (see the listing below): the `r_Attack` rule, which allows the attacker to perform a transaction and interact with the contract, and the `r_Fallback_attacker` rule, which is executed when a transaction without

valid information is received, in particular, it only raises an exception.

```

1 rule r_Attack =
2   let ($cl = current_layer) in
3     let ($scl = sender($cl)) in
4       if executing_function($cl) = attack
5         then
6           switch instruction_pointer($cl)
7             case 0 :
8               r_Transaction[attacker,
9                 random_user,
10                  random_amount,
11                  random_function]
12             case 1 :
13               r_Ret[]
14           endswitch
15         endif
16       endlet
17     endlet
18
19 rule r_Fallback_attacker =
20   let ($cl = current_layer) in
21     if executing_function($cl) != attack
22       then
23         switch instruction_pointer($cl)
24           case 0 :
25             r_Throw[]
26           case 1 :
27             r_Ret[]
28         endswitch
29       endif
30     endlet

```

Once the model of the Auction smart contract is combined with the attacker's model, the following CTL formula has to be added to the resulting model to guarantee the absence of the exception disorder vulnerability:

```

1 CTLSPEC AG((sender(current_layer) = user and
2   executing_function(current_layer) = bid
3   and instruction_pointer(current_layer) =
4   1) implies AF(currentFrontrunner(
5   global_state_layer) = user))

```

In particular, the property checks, for all the possible model states (AG), that, if the transaction sender is user, the executing function is bid, and the instruction pointer is 1, then, in some future state in all the paths (AF), the user will certainly become the new front-runner.

The CTL property is proved to be false. By exploiting the ASMETA feature of building a scenario from a model checker counterexample, we can reconstruct the execution leading to the attack.

The following listing reports (some of) the location values of the last state returned by the scenario execution.



```

1 <State 20 (controlled)>
2 ...
3 call_response(1)=false
4 call_response(2)=false
5 currentFrontrunner(1)=attacker
6 currentFrontrunner(2)=attacker
7 executing_contract(1)=auction
8 executing_contract(2)=attacker
9 executing_function(1)=bid
10 executing_function(2)=fallback
11 global_state_layer=0
12 ...
13 </State 20 (controlled)>

```

Since each model state stores the whole call stack, analyzing all the nested transactions' data is possible. In particular, in lines 3 and 4, the responses for the two nested calls are `false`, which means that an exception has been raised during the second transaction and, since the `bid` function propagates the exception, the first transaction returns a negative value too. Moreover, in lines 5 and 6, the value for the `currentFrontrunner` function does not change during any of the performed transactions. Lines 7, 8, 9, and 10 report values of the executing contract and executing function of the transaction. Finally, line 11, reports the value of `global_state_layer`, which is 0 because of the raised exceptions; if no exception had been raised, the value for the `global_state_layer` would have been 2.

A V&V analysis similar to the one performed on the running example has been carried out on various models of smart contracts we developed, in combination with the different models of attackers available. So far, we modeled a number of vulnerable/correct smart contracts taken from the literature. As for the attackers, we also modeled an inoffensive attacker and an attacker that exploits the reentrancy attack.

## 7 RELATED WORK

Several tools have been developed for automated analysis, testing, and debugging of Ethereum smart contracts. However, as highlighted in the study by [14], a large portion of exploitable bugs in real-world smart contracts (around 80%) are not effectively identified by current tools. These bugs are classified as “machine unauditable bugs” (MUBs) due to the lack of appropriate test oracles and the need for domain-specific knowledge to detect them. Different surveys and reviews [14][15][16] have been viewed and analyzed to extract some of the most popular and effective tools for analyzing Ethereum smart contracts. In particular, most of the literature categorizes the tools depending on the methodology. Among the most

popular methodologies, model checking and theorem proving are mostly used to check contract correctness, while fuzzing and symbolic execution are mostly used for vulnerability identification.

**Correctness.** Tools facing smart contract's correctness or at least used to ensure that a smart contract adheres to the expected behavior, involve formal methods, theorem proving, model checking, and runtime verification techniques.

Tools following the approach presented in [17], e.g., Solidity\* and EVM\*, typically translate Solidity contracts into F\* programs, and call an F\* runtime library for all Ethereum operations. Specifically, Solidity\* compiles Solidity contracts into F\* for source-level correctness verification; EVM\* decompiles EVM bytecode into F\* code to analyze low-level properties.

The work in [18] exploits the K-Framework [19], which enables verification of smart contracts through runtime verification of bytecode instead of Solidity, as bytecode language can be used for any high-level contract language (used to specify the contract model).

Finally, the work in [20] uses Isabelle/HOL to verify the EVM bytecode of smart contracts. This is done by splitting the contracts into basic blocks, and the program correctness of each block is proved by using Hoare triples. The entire verification procedure has been successfully applied to a case study.

In general, all the above-mentioned approaches require a strong mathematical and logical background and use complex mathematical notations; this often prevents practitioners from using them. Instead, the ASM models can be read as high-level programs, and have a simple notation and very basic control flow constructors. Moreover, since they are executable, ASM models are suitable for simulation and other validation techniques that are much more *lighter* than complex verification approaches, at least to have immediate feedback regarding model reliability before checking for correctness. In addition, whereas other approaches cover only few vulnerability classes, we provide both intra- and inter-contract analysis of security properties. In particular, having one or more attacker models running in parallel with the smart contract to be validated allows us to detect many more unsafe interactions than those detected by the approaches that only search a specific instructions pattern in the code.

**Vulnerability detection.** These types of tools aim to improve the security of smart contracts by detecting vulnerabilities and verifying contracts against known vulnerability patterns.

SolCMC and Certora are two leading formal verification tools for Solidity contracts. SolCMC [21]

is a symbolic model checker integrated into the Solidity compiler since 2019. Developers specify properties as assert statements within the contract code, which are treated as verification targets. SolCMC transforms the instrumented contract into Constrained Horn Clauses (CHCs) and uses CHC satisfiability solvers like Spacer (integrated into Z3) or Eldarica to check if any assert can fail, producing a trace witnessing the violation.

Certora [22], on the other hand, decouples property specifications from the contract code. Properties are written in the Certora Verification Language (CVL), an extension of Solidity with metaprogramming primitives. Certora compiles the contract and associated properties into a logical formula and sends it to an SMT solver. Unlike SolCMC, verification in Certora is done remotely on a cloud service.

Among the other tools, Mythril [23] employs concolic analysis, taint analysis, and control flow checking of the Ethereum Virtual Machine (EVM) bytecode to identify vulnerabilities. Its approach involves pruning the search space and identifying values that allow exploiting vulnerabilities in the smart contract.

Another promising tool is Slither [24], developed by TrailOfBits. Slither is a static analysis framework that converts Solidity smart contracts into an intermediate representation called SlithIR and applies program analysis techniques like dataflow and taint tracking to extract and refine vulnerability information. Notably, the combination of Mythril and Slither was found to be notably effective [25].

Securify [26], developed by ETH Zurich, also demonstrated good performance. It statically analyzes EVM bytecode to infer semantic information about the contract using the Souffle Datalog solver. It then checks compliance and violation patterns that capture sufficient conditions for proving if a property holds or not.

While the above-mentioned tools effectively detect common vulnerabilities, they exhibit several limitations. Most do not support identifying attacks involving exception handling or complex interactions between contracts. The majority of tools do not even facilitate behavioral analysis [15][14]. Furthermore, attack prevention is not the sole consideration during smart contract design and development; contract correctness is equally crucial to ensure appropriate behavior. Many tools focus solely on attack identification and do not verify contract correctness.

Our proposed approach aims to address both design correctness and vulnerability detection. It can deal with complete and complex Solidity contracts, also using exceptions. It allows for providing the user with a library of malicious contracts that can be used

to check the robustness of a smart contract against given types of vulnerability.

## 8 CONCLUSIONS

This paper presents a novel approach to modeling and verifying Solidity smart contracts using the Abstract State Machine formal method. It exploits the specification and analysis capabilities, as well as the wide range of tools for model validation and verification, of the ASMs to improve the security and reliability of smart contracts. Since we model the exception-handling mechanism, the proposed method can deal with all features and characteristics of complex Solidity smart contracts.

We use a vulnerable version of the Solidity auction contract, commonly used in blockchain applications, to show our approach's modeling and vulnerability-checking mechanism. In particular, we face the verification of intra- and inter-contract properties, allowing us to identify vulnerabilities and ensure contract reliability.

Our approach offers several advantages:

- Formal foundation: ASMs provide a formal foundation for modeling and verifying smart contracts, ensuring mathematical rigor and precision.
- Modular design: The ASM model is modular, allowing the analysis of individual contracts and the investigation of interactions between contracts.
- Tool support: ASMETA provides a suite of tools for model development, simulation, validation, and verification, and they can represent efficient support to practitioners in the tool-drive analysis of smart contracts.

In future work, we plan to extend our modeling capabilities to address more complex smart contract scenarios, developing automated tools for model generation from Solidity code, and investigating the application of machine learning techniques to enhance vulnerability detection.

We believe that our approach can contribute to the development of more effective and complete model-checking methods to improve smart contracts' security and reliability, fostering wider adoption and innovation in blockchain-based applications.

## REFERENCES

- [1] G. Wood, et al., Ethereum: A secure decentralised generalised transaction ledger, Ethereum project yellow paper 151 (2014) (2014) 1–32.

- [2] Ethereum, Ethereum Virtual Machine (EVM), <https://ethereum.org/en/developers/docs/evm/> (Apr 2024).
- [3] D. Siegel, Understanding The DAO Attack, <https://www.coindesk.com/learn/understanding-the-dao-attack/> (2016).
- [4] King of the Ether Throne - Post-Mortem Investigation, <https://www.kingoftheether.com/postmortem.html> (2016).
- [5] C. Braghin, E. Riccobene, S. Valentini, Modeling and verification of smart contracts, with Abstract State Machines., in: Proceedings of ACM SAC Conference (SAC'24). Accepted to 39th ACM/SIGAPP Symposium on Applied Computing, 2024.
- [6] E. Borger, R. Stark, E. Borger, Abstract state machines: a method for high-level system design and analysis / Egon Borger, Robert Stark, Springer, Berlin, 2003.
- [7] A. Gargantini, E. Riccobene, P. Scandurra, Model-driven language engineering: The ASMETA case study, in: 2008 The Third International Conference on Software Engineering Advances, IEEE, 2008, pp. 373–378.
- [8] A. Gargantini, E. Riccobene, P. Scandurra, et al., A metamodel-based simulator for ASMs, in: Proc. of the 14th Intl. Abstract State Machines Workshop, 2007.
- [9] A. Carioni, A. Gargantini, E. Riccobene, P. Scandurra, A scenario-based validation language for ASMs, in: Abstract State Machines, B and Z: First International Conference, ABZ 2008, London, UK, September 16–18, 2008. Proceedings 1, Springer, 2008, pp. 71–84.
- [10] P. Arcaini, A. Gargantini, E. Riccobene, AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications, in: Abstract State Machines, Alloy, B and Z: Second International Conference, ABZ 2010, Orford, QC, Canada, February 22–25, 2010. Proceedings 2, Springer, 2010, pp. 61–74.
- [11] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri, NuSMV: A new symbolic model verifier, in: Computer Aided Verification: 11th International Conference, CAV'99 Trento, Italy, July 6–10, 1999 Proceedings 11, Springer, 1999, pp. 495–499.
- [12] C. Braghin, S. Cimato, E. Damiani, M. Baronchelli, Designing smart-contract based auctions, in: C.-N. Yang, S.-L. Peng, L. C. Jain (Eds.), Security with Intelligent Computing and Big-data Services, Springer International Publishing, Cham, 2020, pp. 54–64.
- [13] P. Arcaini, E. Riccobene, Automatic Refinement of ASM Abstract Test Cases, in: 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2019, pp. 1–10. doi:10.1109/ICSTW.2019.00025.
- [14] Z. Zhang, B. Zhang, W. Xu, Z. Lin, Demystifying exploitable bugs in smart contracts, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, 2023, pp. 615–627.
- [15] M. Almakhour, L. Sliman, A. E. Samhat, A. Mellouk, Verification of smart contracts: A survey, Pervasive and Mobile Computing 67 (2020) 101227.
- [16] M. Bartoletti, F. Fioravanti, G. Matricardi, R. Pettinau, F. Sainas, Towards benchmarking of Solidity verification tools, arXiv preprint arXiv:2402.10750 (2024).
- [17] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, et al., Formal verification of smart contracts: Short paper, in: Proceedings of the 2016 ACM workshop on programming languages and analysis for security, 2016, pp. 91–96.
- [18] M. Sotnichek, Formal verification of smart contracts with the K framework, <https://www.apriorit.com/dev-blog/592-formal-verification-with-k-framework> (2019).
- [19] A. Ștefănescu, D. Park, S. Yuwen, Y. Li, G. Roșu, Semantics-based program verifiers for all languages, in: Proceedings of the 31th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16), ACM, 2016, pp. 74–91.
- [20] S. Amani, M. Bégel, M. Bortin, M. Staples, Towards verifying Ethereum smart contract bytecode in Isabelle/HOL, in: Proceedings of the 7th ACM SIGPLAN international conference on certified programs and proofs, 2018, pp. 66–77.
- [21] L. Alt, M. Blicha, A. E. Hyvärinen, N. Sharygina, SolCMC: Solidity compiler's model checker, in: International Conference on Computer Aided Verification, Springer, 2022, pp. 325–338.
- [22] D. Jackson, C. Nandi, M. Sagiv, Certora technology white paper, <https://docs.certora.com/en/latest/docs/whitepaper/index.html>.
- [23] Smashing Ethereum Smart Contracts for Fun and ACTUAL Profit, Smashing Ethereum Smart Contracts for Fun and ACTUAL Profit, HITBSecConf, <https://github.com/Consensys/mythril> (2018).
- [24] J. Feist, G. Greico, A. Groce, Slither: a static analysis framework for smart contracts, in: Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB '19, IEEE Press, 2019, p. 8–15.
- [25] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, H.-N. Lee, Ethereum smart contract analysis tools: A systematic review, IEEE Access 10 (2022) 57037–57062.
- [26] P. Tsankov, A. Dan, D. Drachler-Cohen, A. Gervais, F. Bünzli, M. Vechev, Securify: Practical security analysis of smart contracts, in: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 67–82.