Security Assessment of interacting Ethereum Smart Contracts***

Chiara Braghin[1][0000−0002−9756−4675], Elvinia Riccobene[1][0000−0002−1400−1026], and Simone Valentini[1][0009−0005−5956−3945]

Department of Computer Science, Università degli Studi di Milano, Italy
{chiara.braghin, elvinia.riccobene, simone.valentini}@unimi.it

Abstract. Smart contracts on blockchain platforms like Ethereum are increasingly used to automate complex transactions in a secure and transparent manner. However, ensuring their correctness and resistance to attacks remains a significant challenge.
This paper presents a formal approach for modeling and assessing the security of Ethereum smart contracts using Abstract State Machines (ASMs).
ASMs provide a rigorous framework for precisely describing smart contract behavior, including interactions with potentially malicious contracts. By modeling attackers as contracts designed to exploit known vulnerabilities, our method enables the systematic evaluation of a contract's robustness in adversarial scenarios, thereby providing a quantifiable assessment of its security. We demonstrate this approach on a Solidity smart contract, assessing its resilience against three well-known vulnerabilities to illustrate the effectiveness of our security analysis.

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

1 Introduction

Blockchain emerged as a novel and promising technology, introducing a distributed ledger system for managing financial transactions without relying on intermediaries or trusted third parties. Subsequently, in late 2013, Vitalik Buterin conceptualized Ethereum, an open-source, decentralized blockchain platform engineered to enable programmable, trustless interactions directly on the blockchain by leveraging smart contracts [28]. The Ethereum Virtual Machine (EVM) is central to Ethereum's functionality, which acts as the core engine

* 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.
** This work was partially supported by the PRIN project SAFEST (G53D23002770006).

for executing smart contracts and processing transactions, guaranteeing deterministic and decentralized operations throughout the network. As a distributed state machine, Ethereum preserves a consistent, unified state across its network of decentralized nodes, thus facilitating the execution of smart contracts and decentralized applications that yield reliably verifiable outcomes [16].

Due to their storage on the blockchain, smart contracts are inherently immutable, meaning they cannot be altered once deployed. This immutability makes thorough verification and validation essential prior to deployment. Several high-profile incidents have highlighted the risks of deploying vulnerable contracts, emphasizing the need for robust security measures. Notable examples include the infamous DAO attack, which led to the theft of approximately $70 million worth of Ether [24], and the King of the Ether Throne (KotET) contract, which was found to contain a critical vulnerability [1].

The work in [11] explores the use of Abstract State Machines (ASMs) for the specification and verification of Ethereum smart contracts written in Solidity. The EVM and key Solidity language primitives have been specified, allowing smart contract modeling and the functional correctness of a Solidity smart contract to be verified through rigorous proofs. In [10], this work has been extended to include the exception handling mechanisms. This enhancement enabled us to identify and evaluate an additional category of potential vulnerabilities due to exceptions. Moreover, we tackled the verification of intra-contract properties and inter-contract interactions, allowing the assessment of the robustness of a contract against potential attacks (e.g., exception disorder or call-stack overflows) due to internal code bugs or unsafe interactions with other contracts.

This paper extends the work in [10] by presenting a systematic approach to assess the robustness of a smart contract against specific cross-contract threats. Starting with a vulnerable example contract and its corresponding ASM model, we present different attacker models, such as *exception disorder*, *reentrancy* and *force to receive ether*. These attacker models are then used to determine whether the contract is susceptible to the associated vulnerabilities. By expanding the set of attacker models, the approach can be iteratively applied to enhance the thoroughness of the contract's security assessment.

This paper is organized as follows. Section 2 introduces the ASM formal method and its supporting toolset for model specification and verification. In Section 3, we provide essential background information concerning the smart contracts operating on the Ethereum blockchain, and the mechanisms for exception handling. Section 4 details modeling of the EVM and Solidity language primitives using the ASMs. We describe the structure of an ASM model and how to specify the behavior of smart contracts, particularly concerning exception handling. Section 5 introduces the running case study: the Solidity *Auction* smart contract. The verification process to evaluate the contract robustness against specific attacks is presented in Section 6, where different models of attackers are presented. This section reports the application of the verification process to the primary case study and its slightly vulnerable modified versions. Section 7

reviews existing approaches designed for the automated analysis and testing of Ethereum smart contracts. Finally, Section 8 concludes the paper.

2 Theoretical and Tooling Basis: ASMs and ASMETA

We base our verification approach on the Abstract State Machine (ASM) formal method [8] and its ASMETA [19] toolset[1]. This section details them.

2.1 Abstract State Machines

ASMs are an extension of Finite State Machines. They model system configurations by *states* and system behavior by *state transitions*. Compared to Finite Automata, ASMs are a more expressive formal language since they replace the concept of unstructured *state* with the concept of state as a mathematical *algebra*, thus enabling specification of system data by *domains* of elements and operations on data by mathematical *functions* defined over domains.

State memory units are given as a set of *location*s; they are pairs of the form $(f(v_1, \ldots, v_n), v)$, meaning that the function f interpreted on parameters $v_1, ..., v_n$ has value v (assigned in the function codomain). State *transitions* are expressed by transition rules describing how the data (function values saved into locations) change from one state to the next.

An ASM *run* is a finite or infinite sequence S_0, S_1, ..., S_n, ... of states: starting from an initial state S_0, a *run step* from S_n to S_{n+1} consists in firing, in parallel, all transition rules and 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 an error.

ASMs can 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*).

Code 1.1 shows the schema of an ASM model (written in the ASMETA notation), which is composed of:

- The `import` section to import external libraries and user-defined models that can provide predefined domains, functions, and rules.
- The `signature` section, where the model domains and functions are declared. Functions that are not updated by rule transitions are *static*. Those updated are *dynamic*, and distinguished in *monitored* (read by the machine and modified by the environment), *controlled* (read and written by the machine), *shared* (read and written by the machine and its environment); *derived* functions are those defined in terms of other (dynamic) functions.
- The `definitions` section, where all model transition rules and possible invariants and properties are specified. The *update* rule having form $f(t_1, \ldots, t_n) :=$

[1] https://asmeta.github.io/

```
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    /*INVARIANTS */
17    invariant over locations : term
18    ...
19    /*TEMPORAL LOGIC FORMULAS */
20    CTLSPEC ...
21    ...
22    /*MAIN RULE*/
23    main rule r_Main = ...
24
25 default init s0:
26    /*INITIAL STATE DEFINITION*/
27    function location = value
28    ...
```

Code 1.1: ASM model schema

v is the basic unit of rules construction, being f an n-ary function, t_i terms, and v the value of $f(t_1, \ldots, t_n)$ in the next state.
Transition rules are built by using *rule constructors* depending on the update structure they express, e.g., guarded updates (`if-then`, `switch-case`), simultaneous parallel updates (`par`), non-deterministic updates (`choose`), etc.
State *invariants* are first-order formulas that must be true in all states. Temporal *properties* of ASM models can be specified, enabling the formal verification of dynamic system aspects such as safety, liveness, and correctness over time via model checking.

- The `main rule` serves as the entry point for model computation at each state and is responsible for invoking all other transition rules, defined as *macro call rules*. In multi-agent models, it coordinates the execution of individual agents, effectively orchestrating their interactions within the system.
An ASM without a main rule is referred to as a *module*, which includes declarations and definitions of domains, functions, invariants, and macro call rules.

The keywords `asm` and `module` are used to define a complete machine and a module, respectively. Modules can be imported into an ASM model to promote modularity and reuse.
- The `default init` section where initial values for the *controlled* model functions are defined.

2.2 The ASMETA Toolset

ASMETA [19] is a suite of Eclipse-based tools designed to support the ASM formal method. It facilitates the entire modeling process, from writing models to performing various types of analysis.

ASMETA allows users to write ASM models in a user-friendly textual pseudo-code format, supported by a dedicated editor and syntax checker. The well-formed structure of an ASMETA model is outlined in Sect. 2.1. In this work, we utilize two key tools from the ASMETA suite: the model simulator and the model verifier.

The *AsmetaS* simulator [18] enables model validation by executing 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, and random simulation, to interactive simulation modes. For more powerful scenario-based validation, the *AsmetaV* [12] 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* [5] tool, which translates ASM models into input for the NuSMV [14] model checker and invokes the symbolic model checker on specified temporal logic properties. AsmetaSMV supports specifying and checking both linear temporal logic (LTL) and computation tree logic (CTL) 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. This trace can be further converted into an AValLa scenario using other features of the ASMETA toolset [6], enabling reproduction and detailed analysis of the faulty execution path.

3 Ethereum Ecosystem: Blockchain, EVM, and Solidity

The Ethereum blockchain is a decentralized and distributed ledger technology that maintains a secure and transparent record of *transactions* across a peer-to-peer network. Transactions are grouped into data units called *blocks*, each containing a set of transactions, a timestamp, and a cryptographic hash of the preceding block, thus forming a chronological and tamper-resistant chain of blocks. This design ensures the immutability of the blockchain: once data is committed to a block, altering it would require re-computing all subsequent blocks, which is computationally impractical in a well-secured network.

Ethereum [28] differs from earlier blockchain systems like Bitcoin by providing not only a digital currency (known as *Ether*, ETH), but also a decentralized computing platform via the Ethereum Virtual Machine (EVM). The EVM enables the deployment and execution of *smart contracts*, self-executing programs whose logic is encoded directly within the contract's code. Developers write these contracts in high-level languages such as Solidity, which are then compiled into EVM bytecode for execution across the network.

The EVM operates on each node in the Ethereum network, ensuring that smart contracts execute deterministically and without centralized control. The EVM maintains the global state of the Ethereum system, including account balances, contract code, and persistent storage. Ethereum supports two types of accounts: *externally owned accounts* (EOAs) and *smart contract accounts*. EOAs are controlled by users through cryptographic key pairs (public/private keys), whereas contract accounts are governed by deployed code and lack private keys. Transactions initiated by EOAs can trigger the execution of smart contracts, which in turn update the blockchain state by modifying balances, storing values, or invoking other contracts. The EVM provides a global `msg` variable that contains contextual information about the transaction, including `msg.sender`, which identifies the caller's address.

To manage computational effort and prevent abuse, Ethereum employs a metering system known as *gas*. Every operation executed by the EVM consumes a predefined amount of gas, and users must pay transaction fees in proportion to the gas consumed. If a transaction exceeds its gas limit, it results in an *Out-of-Gas exception*, a built-in mechanism that causes the transaction to revert, thereby rolling back any state changes and preserving system consistency. Solidity also supports *custom exceptions*, which are user-defined error-handling constructs invoked through statements like `require` or `revert`. These mechanisms are essential for enforcing preconditions and managing contract behavior under invalid input or failure scenarios.

Transferring Ether in Solidity is handled through different functions, each with distinct behaviors regarding gas forwarding and error handling. Functions such as `transfer` and send `forward` a fixed gas amount (currently 2300 gas), which suffices for basic operations but may be inadequate for complex recipient logic. In contrast, the `call` function allows explicit control over the gas limit and is thus preferred in advanced use cases.

In the event of a failure during a transfer, the behavior differs by function: `transfer` and `call` revert the transaction and undo all state changes if execution fails or gas is insufficient. Conversely, `send` returns false on failure without reverting the state, enabling the sending contract to handle the error manually. Moreover, only functions marked with the `payable` keyword can receive Ether. If Ether is sent to a non-payable function, the transaction will automatically fail. These design decisions play a critical role in the reliability and security of smart contract-based systems.

4 ASM-Based Modeling of Ethereum and Solidity

In our previous work [11], we formally specified the main components of the EVM and Solidity language primitives using ASMs. We also defined a method to map, consistently, Solidity smart contracts into ASMETA models. This process results in an ASMETA model for every smart contract. We here briefly recall these concepts and we discuss how they have been extended to deal with the EVM's exceptions mechanism. This extension enables us to consider further potential security attacks that result from the incorrect use of exceptions. Examples include *exception disorder*, a frequent problem caused by how Solidity sometimes inconsistently handles errors during function calls, or the *call stack overflow* vulnerability, which occurs when a program uses too much memory space on the call stack.

The EVM library includes a model of the *execution stack*. This stack monitors which smart contract functions are currently running and also holds the overall state of the blockchain. A *stack frame* is one item on this stack. It stores important details like the address of the account that started the transaction, the address receiving the transaction, the amount of Ether sent, the function currently being executed, the specific instruction being processed, and the smart contract containing that function. There is a stack pointer that indicates the *stack frame* for the function running at the moment (this frame is at the top of the stack). This pointer is represented in the model by a controlled function named `current_layer`.

We defined two specific rules to manage transactions and update the stack pointer: `r_Transaction`, which executes a transaction, and `r_Ret`, which terminates the execution.

The `r_Transaction` rule models the execution of a transaction by incrementing the stack pointer (`current_layer`). It also updates the instruction pointer of the calling function to point to the next instruction, ensuring that execution can resume correctly once the called function completes. This mechanism prepares the system to return to the caller's context after the callee finishes and `current_layer` is decremented. The `r_Ret` rule is triggered when a function concludes its execution; it removes the topmost stack frame and decrements the stack pointer accordingly.

The top part of Fig. 1 provides a diagram showing the stack's structure and how the `r_Transaction` and `r_Ret` rules operate. More specifically, it shows a scanario 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).

To model Ethereum's exception handling mechanism, we incorporated a rollback feature into our model. As discussed in Section 3, when an exception occurs in Ethereum, the system reverts to the blockchain state that existed immediately before the exception, effectively discarding any intermediate changes. The rollback capability relies on a new function, `global_state_layer`, which operates similarly to `current_layer`, but refers to a saved snapshot of the global state taken just before points in execution where exceptions may occur. In the event of

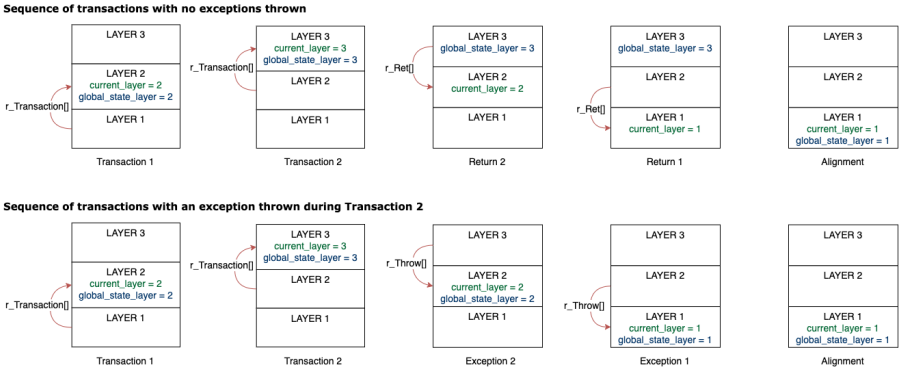

Fig. 1: Diagram representing how the two pointers behave in different situations

an exception, the system can restore the correct global state simply by reverting `global_state_layer` to the corresponding snapshot.

The `r_Save` rule is responsible for creating these snapshots. It copies all values from the current `global_state_layer` to a distinct layer, preserving the global state prior to potentially unsafe operations. During transaction execution, a snapshot is created by invoking the `r_Save` rule on the next available `global_state_layer`, after which the layer counter is incremented. Conversely, the `r_Throw` rule simulates exception handling (for example, when the call stack exceeds its limit) by decrementing `global_state_layer` to roll back to the last valid snapshot. It then invokes the `r_Ret` rule to unwind the call stack by decrementing `current_layer`.

Fig. 1 illustrates the interaction between `current_layer` and `global_state_layer` under normal and exceptional execution paths. In the top part of the figure, Transaction 2 completes successfully with a return statement. As a result, `current_layer` is decremented to resume the caller's execution, while `global_state_layer` remains unchanged, preserving the state modifications made by Transaction 2. In contrast, the bottom part of the figure shows a case where Transaction 2 ends due to an exception, triggering a rollback. This restores the blockchain state to its value before Transaction 2 began by decrementing both `global_state_layer` and `current_layer`. Execution resumes from the context of Transaction 1, while discarding all changes made by Transaction 2. Finally, when all transactions are completed, whether normally or via exception, `current_layer` returns to 1. At that point, the `r_Save` rule is triggered to snapshot the current global state into layer 1, and `global_state_layer` is updated accordingly.

Additionally, we extended the EVM model to include the `payable` mechanism, which determines whether a method is permitted to receive Ether. This is formalized through a `controlled` function, `payable : Function -> Boolean`, defined within the EVM library. It allows any function (i.e., an element of the predefined domain `Function`) to be explicitly marked as `payable`.

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: StackLayer → 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: StackLayer × KeyType → ValueType
for each mapping *local* variable `mapping(keyType => valueType)`	add a function p: StackLayer × 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

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

In [10], we also defined a mapping from Solidity smart contracts to ASMETA models. Table 1 summarizes these mapping rules, which have been refined here to incorporate the extensions introduced for modeling Ethereum's exception handling mechanism.

The table outlines how both state and local variables are represented. State variables, denoted by `p`, are mapped to controlled functions of the form of `p: StackLayer → Type`, as shown in rows 2 and 3. This mapping enables retrieval of the value of a state variable `p` at a specific execution context by evaluating `p(global_state_layer)`. Local variables follow a similar pattern but are indexed using `current_layer`, indicating their association with the current function call rather than the global state in the `StackLayer` domain.

Rows 4 and 5 demonstrate how more complex data structures, such as mappings, are also translated into controlled functions. Like simple variables, mappings can be either state or local. State-level mappings are associated with the `global_state_layer`, whereas local mappings are bound to the `current_layer`.

Each function defined in a Solidity contract is mapped to a `static` function `f: Function` and an associated rule, as illustrated in rows 6 and 7. The rule body is typically specified in terms of a `switch` statement, with separate `case` branches handling each instruction of the method.

`fallback` functions are treated similarly, as shown in rows 8 and 9. A Solidity contract may or may not define `fallback` function. If it does, it is modeled like any other contract method. Otherwise, a default `fallback` rule is used (not shown here), which simulates an exception by invoking the `r_Throw` rule, effectively modeling the absence of a valid `fallback` handler.

5 Auction Case Study: contract and model

In this section, we present an ongoing case study, which acts as a practical example to show how our method for evaluating robustness of possible vulnerable smart contracts.

5.1 The Auction Smart Contract

The case study is the Solidity *Auction contract* described in [9]. This type of contract is frequently found in real-world blockchain applications. The code for this contract is shown in Code 1.2.

```
contract Auction {
  address currentFrontrunner;
  uint currentBid;

  function bid() payable {
    require(msg.value > currentBid);

    if (currentFrontrunner != 0) {
      require(currentFrontrunner.send(currentBid));
    }

    currentFrontrunner = msg.sender;
    currentBid         = msg.value;
  }
}
```

Code 1.2: Auction Case Study

This contract records who the current highest bidder is (using the variable `currentFrontrunner`) and what the highest bid amount is (using `currentBid`). People participating in the auction can place their bids by calling the `bid()` function. This function requires payment in Ether. This function has two main conditions that must be met: (1) The value of the submitted bid (found in `msg.value`) must be higher than the current highest bid (`currentBid`). (2) If someone is already the highest bidder (meaning `currentFrontrunner != 0`), the contract has to send back their previous bid amount before accepting the new, higher bid.

This contract can enter a state where it stops working correctly, known as Denial of Service (DoS) condition. This happens because of a weakness related to *exception disorder*. Specifically, an attacker can create a harmful contract with two features: (*i*) It places the highest bid, making the attacker the leading bidder. (*ii*) It includes a special `fallback` function. (This function automatically runs if Ether is sent to the contract address and there is no specific `receive` function). This `fallback` function is designed to use more gas than the standard amount provided by the `send` function. Later, when a legitimate user submits a bid higher than the attacker's bid, the `Auction` contract attempts to return the attacker's

previous bid using the `send` function. However, this attempt fails because the attacker's `fallback` function runs out of gas, causing an *out-of-gas* exception. As explained in Section 3, this failure results in the `send` function returning the value *false*.

Because `send` returned *false*, a `require` statement inside the `Auction` contract's code also fails, which causes the `Auction` contract itself to generate an exception. This exception then causes the system to undo (rollback) everything that happened during the execution of the `bid` function. The final result is that the attacker remains the highest bidder until the auction ends. This is because the `Auction` contract is now unable to process any new, legitimate bids that are higher than the attacker's bid.

5.2 The ASM model of the Auction Contract

Starting from the contract's source code, the corresponding ASMETA model[2] can be straightforwardly got by applying a mapping method presented in [11]. We refer the reader to this work for the general mapping steps.

Here, we focus on a specific part of the model (the complete model can be found in Appendix A). The rule `r_Bid`, which is reported in Code 1.3, models the `bid()` function of the contract. This rule becomes active when the `bid` function is the one currently running (in the current execution layer). Inside this rule, a `switch` statement manages the order of execution. It decides what to do next based on the current value of the instruction pointer for this layer. First (in case 0), the rule checks a condition: the bid amount provided in the current transaction (represented in the `current_layer`) must be higher than the existing highest bid (stored in the contract's state, part of the `global_state_layer`). Next (in case 1), it checks if there is already the highest bidder (if `currentFrontrunner` is not `undef`). If yes, the instruction pointer increases to move to the next step (case 2). If no, the execution jumps directly to case 4. Case 2 starts a transaction to send the previous highest bid amount back to the person who was the highest bidder. This action may trigger the fallback function of the previous bidder's address. After the refund transaction from case 2 is attempted, case 3 checks if this transaction was completed successfully (e.g., if the refund function returned true). The final steps (case 4 and case 5) are intended to update the contract's state: they set the new highest bidder to be the address that sent the current transaction, and they update the highest bid amount to the value of this new transaction.

6 Contract Robustness Assessment

Vulnerabilities in smart contracts can lead to security breaches, financial losses, and other undesirable consequences. *Robustness assessment* of a smart contract,

[2] The model is available at https://github.com/smart-contract-verification/ethereum-via-asm/tree/with_exception

```
rule r_Save ($n in StackLayer) =
    par
        currentFrontrunner($n) := currentFrontrunner(global_state_layer)
        currentBid($n) := currentBid(global_state_layer)
    endpar

rule r_Bid =
    if executing_function(current_layer) = bid then
        switch instruction_pointer(current_layer)
            case 0 :
                r_Require[amount(current_layer) > currentBid(
                    global_state_layer)]
            case 1 :
                if isDef(currentFrontrunner(global_state_layer)) then
                    instruction_pointer(current_layer) :=
                        instruction_pointer(current_layer) + 1
                else
                    instruction_pointer(current_layer) := 4
                endif
            case 2 :
                r_Transaction[auction, currentFrontrunner(
                    global_state_layer), currentBid(global_state_layer),
                    fallback]
            case 3 :
                r_Require[call_response(current_layer+1)]
            case 4 :
                par
                    currentFrontrunner(global_state_layer) := sender(
                        current_layer)
                    instruction_pointer(current_layer) :=
                        instruction_pointer(current_layer) + 1
                endpar
            case 5 :
                par
                    currentBid(global_state_layer) := amount(current_layer)
                    instruction_pointer(current_layer) :=
                        instruction_pointer(current_layer) + 1
                endpar
            case 6 :
                r_Ret[]
        endswitch
    endif
```

Code 1.3: Model transated from the Auction solidity contract

namely the capability of a smart contract to be robust against (given) vulnerabilities, can be addressed by exploiting the verification approach presented in [10] to detect *inter-contract* vulnerabilities, which are due to unsafe interaction of

smart contracts. They occur when a smart contract interacts with another specific contract, denoted as *attacker*, endowed with functions tailored to exploit a vulnerability.

In [11], we presented how to model attackers, and how to execute a given model of a smart contract in combination with an attacker. More specifically, the ASM model of an attacker always includes a `r_Attack` rule that launches the attack. To execute a given model of a smart contract in combination with an attacker, we follow the schema of Code 1.4.

```
main rule r_Main =
  if executing_contract(current_layer) = contractName then
       r_contractName[]
  else r_Attacker[]
  endif
```

Code 1.4: The `r_Main` rule schema for co-execution of a contract and an attacker

In [11], we also mention a library of attackers we have developed over time, differing in the capabilities they have and based on well-known vulnerabilities. However, these attacker models do not exploit exception disorder vulnerabilities. In this work, we improve this attacker modeling mechanism by adding cases of attacks dealing with exceptions. Specifically, we introduce three models of attackers: *exception disorder*, *reentrancy* and *force to receive ether*. We then co-execute these attackers with the model of our case study (or a slightly modified version of it) to check for the possible presence of such vulnerabilities; in the case of their occurrence, we can generate a corresponding faulty execution of the contract by exploiting the counterexample obtained by the verification.

6.1 Exception Disorder Vulnerability

The version of the Auction 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 (see Code 1.5) specified to exploit the exception disorder vulnerability contains two different rules: 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.

```
rule r_Attack =
    let ($cl = current_layer) in
        let ($scl = sender($cl)) in
            if executing_function($cl) = attack then
                switch instruction_pointer($cl)
                    case 0 :
                        r_Transaction[attacker, random_user, random_amount,
                             random_function]
                    case 1 :
```

```
                        r_Ret[]
                endswitch
            endif
        endlet
    endlet

rule r_Fallback_attacker =
    let ($cl = current_layer) in
        if executing_function($cl) != attack then
            switch instruction_pointer($cl)
                case 0 :
                    r_Throw[]
                case 1 :
                    r_Ret[]
            endswitch
        endif
    endlet

rule r_Attacker =
  par
    r_Attack[]
    r_Fallback_attacker[]
  endpar
```

Code 1.5: The exception disorder attacker model

Once the model of the Auction smart contract is combined with the attacker's model, the CTL formula in Code 1.6 has to be added to the resulting model to guarantee the absence of the exception disorder vulnerability:

```
CTLSPEC AG((sender(current_layer) = user and executing_function(
    current_layer) = bid and instruction_pointer(current_layer) = 1)
    implies AF(currentFrontrunner(global_state_layer) = user))
```

Code 1.6: CTL formula for Exception Disorder

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.

Code 1.7 reports (some of) the location values of the last state returned by the scenario execution.

```
<State 20 (controlled)>
...
call_response(1)=false
call_response(2)=false
currentFrontrunner(1)=attacker
```

```
currentFrontrunner(2)=attacker
executing_contract(1)=auction
executing_contract(2)=attacker
executing_function(1)=bid
executing_function(2)=fallback
global_state_layer=0
...
</State 20 (controlled)>
```

Code 1.7: Exceprion Disorder Counterexample

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.

6.2 Force to Receive Ether Vulnerability

The *force receive ether* vulnerability refers to the possibility of sending Ether to a contract even if the contract does not have a `fallback` function marked as payable. However, in some contracts, the execution flow depends on the contract balance value, in these cases it is possible to manipulate the execution by forcing the contract to receive ether.

Let us slightly modify the Auction contract presented in Section 5 to make it vulnerable to the force receive ether vulnerability. In particular, let us change the `currentBid = msg.value` in Code 1.2, line 13, with `currentBid += 1`. In this version of the Auction contract, the `currentBid` attribute is increased by 1 each time a new bid is performed. Moreover, let us modify the condition in Code 1.2, line 6, by replacing `require(msg.value > currentBid)` with `require(msg.value = currentBid + 1)`, forcing `msg.value` to be equal to `currentBid`. Finally let us add a non-payable method, named `withdraw()`, which enables the contract owner, represented by a contract state variable, to withdraw the whole contract's balance and terminate the bid. The rule modeling this newly introduced `withdraw()` method is reported in Code 1.8.

```
rule r_Withdraw =
    if executing_function(current_layer) = withdraw then
        switch instruction_pointer(current_layer)
            case 0 :
                r_Require[balance(global_state_layer, auction) >= 10]
            case 1 :
```

```
7               par
8                   r_Transaction[auction, owner, balance(
                        global_state_layer, auction), fallback]
9                   currentFrontrunner(global_state_layer) := owner
10                  currentBid(global_state_layer) := 0
11                  instruction_pointer(current_layer) :=
                        instruction_pointer(current_layer) + 1
12              endpar
13          case 1 :
14              r_Ret[]
15      endswitch
16  endif
```

Code 1.8: The `r_Withdraw` rule modeling the `withdraw()` method

The code begins with an `r_Require` (line 5) rule, which checks if the contract's balance is greater than or equal to 10. Afterward (case 1), the rule performs a transaction to the contract's owner transferring the whole contract's balance and setting the bid related functions to the initial values.

Since `bid` is the only `payable` method in the contract, the only way for the contract to receive Ether is by receiving a new bid. As a consequence, a bid reset would occur only when the `currentBid` value reaches 10. However, it is possible to force the contract to receive Ether and manipulate its behavior through a `selfdestruct` instruction, this allows a hypothetical attacker to become the `currentFrontrunner` and immediately terminate the bid.

To exploit this vulnerability, an attacker has been specified. The attacker model is reported in Code 1.9.

```
1  rule r_Attack =
2      if executing_function(current_layer) = attack then
3          switch instruction_pointer(current_layer)
4              case 0 :
5                  par
6                      input_user(current_layer) := random_user
7                      instruction_pointer(current_layer) :=
                           instruction_pointer(current_layer) + 1
8                  endpar
9              case 1 :
10                 r_Selfdestruct[input_user(current_layer)]
11         endswitch
12     endif
13
14 rule r_Fallback_attacker =
15     if executing_function(current_layer) != attack then
16         switch instruction_pointer(current_layer)
17             case 0 :
18                 r_Throw[]
19         endswitch
20     endif
21
```

```
22 rule r_Attacker =
23     par
24         r_Attack[]
25         r_Fallback_attacker[]
26     endpar
```

Code 1.9: Selfdestruct attacker model

The attacker model receives a user as input through the `random_user` monitored function and stores it inside the `input_user` function; afterward, the rule invokes the `r_Selfdestruct` rule, which disables the contract and sends the whole contract balance to the target user passed as input.

The CTL formula in Code 1.10 can verify this behavior.

```
1 CTLSPEC AG((executing_contract(current_layer) = auction and
      executing_function(current_layer) = withdraw and instruction_pointer(
      current_layer) = 1) implies (currentBid(global_state_layer) = 10) )
```

Code 1.10: CTL formula for Force to Receive Ether

It states that for all the model states (AG), if the executing contract is `acution`, the executing function is `withdraw` and the instruction pointer value is 1, then the `currentBid` value should be 10. The model checker violates this property and a counterexample is returned; it is partially shown in Code 1.11: at State 15 the auction's balance is bigger than 10 (line 9) and the attacker contract has been destroyed (line 16); however, in State 14 the `currentBid` value was 1 (line 3), violating the specification.

```
1  ...
2  <State 14 (controlled)>
3  currentBid(1)=1
4  </State 14 (controlled)>
5  <State 15 (controlled)>
6  ...
7  amount(1)=0
8  balance(attacker,1)=0
9  balance(auction,1)=11
10 balance(user,1)=9
11 balance(user_owner,1)=10
12 call_response(1)=true
13 currentBid(1)=0
14 currentFrontrunner(1)=user_owner
15 current_layer=1
16 disabled(attacker,1)=true
17 executing_contract(1)=auction
18 executing_function(1)=withdraw
19 global_state_layer=1
20 owner(0)=user_owner
21 instruction_pointer(1)=1
22 receiver(1)=auction
23 sender(1)=user
```

```
...
</State 15 (controlled)>
```

Code 1.11: Force to Receive Ether Counterexample

6.3 Reentrancy Vulnerability

A *reentrancy* attack is a type of vulnerability where a malicious contract can repeatedly call a function of a vulnerable contract before the initial execution of the function is completed allowing the attacker to drain funds or manipulate the state of the vulnerable contract in unintended ways.

To check this vulnerability, let us adopt the model of the original case study presented in Code 5 and incorporating the contract modifications explained in Section 6.2. Furthermore, the `send` transaction in Code 1.2, line 9, is replaced with a `call` transaction to enable code execution.

Through this vulnerability, a malicious contract could invoke the `bid` method, creating a reentrancy loop. This loop allows the bidding process to be repeated multiple times for the same user within a single transaction. Upon termination, the `currentBid` value increases with each iteration. Exploiting this, an attacker could perform many subsequent bids with the aim of inflating the `currentBid` value. The goal is either to discourage other users from placing new bids or to reach the limit required to invoke the `withdraw` method, as introduced in Section 6.2.

To exploit this vulnerability we present an attacker model (see Code 1.12), which implements a reentrancy mechanism. In particular, the model is composed of two main rules, the `r_Attack` rule aims to initialize the attack by performing a transaction (case 0) to the vulnerable method (i.e. in this case, the `bid` method). Subsequently, it terminate the execution through the `r_Ret` rule (case 1).

```
rule r_Attack =
    let ($cl = current_layer) in
        let ($scl = sender($cl)) in
            if executing_function($cl) = attack then
                switch instruction_pointer($cl)
                    case 0 :
                        r_Transaction[attacker, random_user, random_amount,
                             random_function]
                    case 1 :
                        r_Ret[]
                endswitch
            endif
        endlet
    endlet

rule r_Fallback_attacker =
    let ($cl = current_layer) in
        if executing_function($cl) != attack then
            switch instruction_pointer($cl)
```

```
                case 0 :
                    r_Transaction[attacker, sender($cl), random_amount,
                        random_function]
                case 1 :
                    r_Ret[]
            endswitch
        endif
    endlet

rule r_Attacker =
    par
        r_Attack[]
        r_Fallback_attacker[]
    endpar
```

Code 1.12: Reentrancy attacker model

The `r_Fallback_attacker`, which models the attacker's fallback method, aims to react to an incoming transaction by performing a new transaction (case 0) to the sender and invoking a random method. Subsequently, it ends the execution on case 1.

A CTL formula can be specified to check for a reentrancy vulnerability. It asserts that for all global states (AG), there not exists two distinct values (`sl1`,`sl2`) in the `StackLayer` domain such that both `sl1` and `sl2` are less than the `current_layer` value, and their corresponding `executing_function` values are identical (`executing_function(sl1) = executing_function(sl2)`). The formula is reported in Code 1.13.

```
CTLSPEC ag(not (exist $sl1 in StackLayer, $sl2 with $sl1 != $sl2 and $sl1
     < current_layer and $sl2 < current_layer and executing_function($sl1
    ) = executing_function($sl2)))
```

Code 1.13: CTL formula for Reentrancy

The model checker fails to prove the property and generates the counterexample shown in Code 1.14. Specifically, at State 36, the `current_layer` value reaches 9. Notably, two previous states (State 29 and State 35) have lower `current_layer` values (6 and 8, respectively), yet both indicate the same `executing_function` value (`bid`) violating the specification.

```
...
<State 29 (controlled)>
current_layer=6
executing_contract(6)=auction
executing_function(6)=bid
</State 29 (controlled)>
<State 35 (controlled)>
current_layer=8
executing_function(8)=bid
</State 35 (controlled)>
<State 36 (controlled)>
```

```
current_layer=9
</State 36 (controlled)>
...
```

Code 1.14: Reentrancy counterexample

7 Related Work

Many tools have been created for the automatic analysis, testing, and debugging of Ethereum smart contracts. To identify common and effective analysis tools, several surveys and reviews [29, 2, 7] were reviewed. Typically, these tools are grouped based on the method they use. Model checking and theorem proving are mostly used for checking if contracts are correct, while fuzzing and symbolic execution are used for finding vulnerabilities. Furthermore, techniques using Large Language Models (LLMs) and Machine Learning (ML) are becoming increasingly popular in the research literature.

Formal analysis. These tools check the correctness of smart contracts (i.e., ensuring they behave as expected) or automatically find common and well-known vulnerabilities. In particular, these tools involve techniques like formal methods, theorem proving, model checking, and runtime verification.

The work described in [26] uses the K-Framework [15]. This framework allows smart contract verification through runtime verification of bytecode instead of the original Solidity code. This is advantageous because bytecode can be generated from any high-level language used to write the contract model.

Finally, the research in [4] uses Isabelle/HOL to verify the EVM bytecode of smart contracts. This process involves splitting the contracts into basic blocks. The correctness of each block is then proved using Hoare triples. This entire verification method has been successfully applied in a case study.

SolCMC and Certora are two main tools for the formal verification of Solidity contracts. SolCMC [3] is a symbolic model checker that has been included in the Solidity compiler since 2019. Developers define properties using `assert` statements directly in the contract code, and the tool uses these as targets for verification. SolCMC converts the contract (with these assertions) into Constrained Horn Clauses (CHCs). It then employs CHC satisfiability solvers like Spacer (part of Z3) or Eldarica to check if any assertion can fail. If a potential failure is found, it provides a trace showing how the violation occurs.

Certora [21], in contrast, keeps the property specifications separate from the actual contract code. Users write these properties in the Certora Verification Language (CVL), which is an extension of Solidity that includes special programming features. Certora compiles the contract and the related properties into a logical formula. This formula is then sent to an SMT solver for checking. Unlike SolCMC, the verification process with Certora happens remotely on a cloud-based service.

Another tool, Mythril [25], uses concolic analysis, taint analysis, and control flow checking on the EVM bytecode to find vulnerabilities. Its method involves

reducing the possible areas to check and identifying specific values that can trigger weaknesses in the smart contract.

Generally, many of the approaches mentioned before require a strong understanding of mathematics and logic. They often use complex mathematical notations, which can make it difficult for practitioners (like developers) to adopt and use them easily.

Static Analysis. This type of tool involves examining the code of a smart contract without actually running it. The aim is to understand the contract's behavior and identify potential vulnerabilities. Generally, these tools use data-flow analysis to track how data moves through the contract, and control-flow analysis to understand the possible order in which instructions are executed.

Another useful tool is Slither [17], developed by TrailOfBits. Slither is a static analysis framework that changes Solidity smart contracts into an intermediate format called SlithIR. It then applies program analysis techniques, like dataflow analysis and taint tracking, to this format to find and get more details about vulnerabilities. Notably, research has shown that combining Mythril and Slither is particularly effective [22].

Securify [27], developed by ETH Zurich, has also shown good performance. It statically analyzes EVM bytecode to understand the contract's meaning (semantic information), using the Souffle Datalog solver. It then checks for compliance and violation patterns. These patterns provide strong indications for proving whether a certain property is met by the contract or not.

However, these static analysis tools often identify potential bugs using general rules (heuristics) but may not provide proof that these bugs can actually be exploited. They might report potential issues without confirming if they represent a real security risk. Furthermore, the warnings generated by these tools are often unclear (ambiguous), which makes it harder for developers to effectively fix the vulnerable smart contracts.

Fuzzing. Fuzzing is a dynamic testing technique where a smart contract is automatically given large amounts of random, invalid, or unexpected inputs. The main goal is to trigger unexpected behaviors, find issues in unusual situations (edge cases), and discover potential vulnerabilities that traditional testing methods might overlook.

Echidna [20] is a fuzzer designed for smart contracts compatible with the EVM. It operates by generating sequences of transactions that call the smart contract. The basic process is quite simple. First, it uses the Slither [17] static analysis tool to scan the contract being tested and collect necessary information. Next, it looks at any custom configuration options that the user may have provided (usually in a YAML file). After this initial setup, it performs a first run of Echidna using either the default settings or the user's configurations.

EF4CF [23] is described as a high-performance fuzzing tool for Ethereum smart contracts. It is capable of modeling complex interactions, including reentrancy and interactions between different contracts, by simulating the actions of multiple attacker-controlled contracts. It uses a technique that converts smart

contract bytecode into native C++ code. This conversion allows developers to reuse existing and efficient fuzzing tools available for C++.

SMARTIAN [13] is a tool designed to find bugs in smart contracts using a mix of static and dynamic analysis methods. First, it statically analyzes the contract's code to understand its structure and identify parts that may need closer testing. The information gathered from this static analysis is then used to guide the dynamic fuzzing process more effectively. This fuzzing involves generating various transaction sequences to test the smart contract's behavior.

However, fuzzing has limitations. It cannot explore every possible execution path or state within a smart contract, particularly in complex ones. This is because fuzzing relies on generating and testing only a selection of all possible inputs. Consequently, the tool might miss vulnerabilities that need specific sequences of interactions or inputs that the fuzzer does not happen to generate.

In contrast to the approaches mentioned previously, ASM models are similar to high-level programs and use simplified notation and basic control flow structures. Furthermore, because ASM models can be executed, they allow for simulation and other validation techniques. These techniques generally require much less computational power than complex verification methods. Our proposed approach intends to handle both the correctness of the design and the detection of vulnerabilities.

8 Conclusions

This paper introduces a method for security assessment of Solidity smart contracts. This approach is based on the ASM/ASMETA formal framework. We here extend the results in [10] by introducing additional attacker's models capable of exploiting further vulnerabilities such as *Exception disorder*, *Force to receive Ether*, and *Reentrancy*. Verifying a given smart contract interacting with these malicious smart contracts enables us to evaluate the contract's robustness against vulnerabilities exploitable by these attackers. Our verification process specifically considers temporal properties on interactions between contracts and exploits temporal logic verification tools available for ASMs.

We apply our approach to the Auction Solidity smart contract and show its weakness against these three well-known vulnerabilities. However, extending the proposed set of attacker models, the same approach can be repeated to improve the robustness evaluation of the contract. Moreover, the approach is not tailored to the specific case study contract, but is is applicable to any Solidity contract.

Our approach offers several advantages:

- *Formalization*: ASMs offer a formal method for rigor modeling and formal verification of smart contracts.
- *Modularity*: The ASM modeling approach is modular and allows model analysis at different composite model level; indeed, it is possible to analyze the same contract model just by compositing it with different attacker models.

– *Tooling*: ASMETA provides a set of tools for model editing, simulation, validation, and verification, and they provide efficient support to practitioners in a tool-drive analysis of smart contracts.

For our future work, we have several plans. We intend to improve our modeling abilities to handle more complex smart contracts. We also plan to develop tools that can automatically generate models directly from Solidity source code. Additionally, we will explore how machine learning methods, combined with formal methods, could be used to better identify security weaknesses.

References

1. King of the Ether Throne - Post-Mortem Investigation (2016), https://www.kingoftheether.com/postmortem.html
2. Almakhour, M., Sliman, L., Samhat, A.E., Mellouk, A.: Verification of smart contracts: A survey. Pervasive and Mobile Computing **67**, 101227 (2020)
3. Alt, L., Blicha, M., Hyvärinen, A.E., Sharygina, N.: SolCMC: Solidity compiler's model checker. In: International Conference on Computer Aided Verification. pp. 325–338. Springer (2022)
4. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In: Proceedings of the 7th ACM SIGPLAN international conference on certified programs and proofs. pp. 66–77 (2018)
5. Arcaini, P., Gargantini, A., Riccobene, E.: 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. pp. 61–74. Springer (2010)
6. Arcaini, P., Riccobene, E.: Automatic Refinement of ASM Abstract Test Cases. In: 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 1–10 (2019). https://doi.org/10.1109/ICSTW.2019.00025
7. Bartoletti, M., Fioravanti, F., Matricardi, G., Pettinau, R., Sainas, F.: Towards benchmarking of Solidity verification tools. arXiv preprint arXiv:2402.10750 (2024)
8. Borger, E., Stark, R.: Abstract state machines: a method for high-level system design and analysis. Springer, Berlin (2003)
9. Braghin, C., Cimato, S., Damiani, E., Baronchelli, M.: Designing smart-contract based auctions. In: Yang, C.N., Peng, S.L., Jain, L.C. (eds.) Security with Intelligent Computing and Big-data Services. pp. 54–64. Springer International Publishing, Cham (2020)
10. Braghin, C., Riccobene, E., Valentini, S.: An ASM-Based Approach for Security Assessment of Ethereum Smart Contracts. In: di Vimercati, S.D.C., Samarati, P. (eds.) Proceedings of the 21st International Conference on Security and Cryptography, SECRYPT 2024, Dijon, France, July 8-10, 2024. pp. 334–344. SCITEPRESS (2024). https://doi.org/10.5220/0012858000003767, https://doi.org/10.5220/0012858000003767
11. Braghin, C., Riccobene, E., Valentini, S.: Modeling and verification of smart contracts with Abstract State Machines. In: Hong, J., Park, J.W. (eds.) Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing, SAC 2024, Avila, Spain, April 8-12, 2024. pp. 1425–1432. ACM (2024). https://doi.org/10.1145/3605098.3636040, https://doi.org/10.1145/3605098.3636040

12. Carioni, A., Gargantini, A., Riccobene, E., Scandurra, P.: 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. pp. 71–84. Springer (2008)
13. Choi, J., Kim, D., Kim, S., Grieco, G., Groce, A., Cha, S.K.: Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In: Proceedings of the International Conference on Automated Software Engineering (2021)
14. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: A new symbolic model verifier. In: Computer Aided Verification: 11th International Conference, CAV'99 Trento, Italy, July 6–10, 1999 Proceedings 11. pp. 495–499. Springer (1999)
15. Ştefănescu, A., Park, D., Yuwen, S., Li, Y., Roşu, G.: Semantics-based program verifiers for all languages. In: Proceedings of the 31th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16). pp. 74–91. ACM (Nov 2016)
16. Ethereum: Ethereum Virtual Machine (EVM) (Apr 2024), https://ethereum.org/en/developers/docs/evm/
17. Feist, J., Greico, G., Groce, A.: Slither: a static analysis framework for smart contracts. In: Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain. p. 8–15. WETSEB '19, IEEE Press (2019)
18. Gargantini, A., Riccobene, E., Scandurra, P.: A Metamodel-based Language and a Simulation Engine for Abstract State Machines. J. Univers. Comput. Sci. **14**(12), 1949–1983 (2008). https://doi.org/10.3217/JUCS-014-12-1949, https://doi.org/10.3217/jucs-014-12-1949
19. Gargantini, A., Riccobene, E., Scandurra, P.: Model-driven language engineering: The ASMETA case study. In: 2008 The Third International Conference on Software Engineering Advances. pp. 373–378. IEEE (2008)
20. Grieco, G., Song, W., Cygan, A., Feist, J., Groce, A.: Echidna: effective, usable, and fast fuzzing for smart contracts. In: Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis. pp. 557–560 (2020)
21. Jackson, D., Nandi, C., Sagiv, M.: Certora technology white paper, https://docs.certora.com/en/latest/docs/whitepaper/index.html
22. Kushwaha, S.S., Joshi, S., Singh, D., Kaur, M., Lee, H.N.: Ethereum smart contract analysis tools: A systematic review. IEEE Access **10**, 57037–57062 (2022)
23. Rodler, M., Paaßen, D., Li, W., Bernhard, L., Holz, T., Karame, G., Davi, L.: Ef/cf: High performance smart contract fuzzing for exploit generation. In: IEEE European Symposium on Security and Privacy (EuroS&P). IEEE (2023)
24. Siegel, D.: Understanding The DAO Attack (2016), https://www.coindesk.com/learn/understanding-the-dao-attack/
25. Smashing Ethereum Smart Contracts for Fun and ACTUAL Profit: Smashing Ethereum Smart Contracts for Fun and ACTUAL Profit. HITBSecConf (2018), https://github.com/Consensys/mythril
26. Sotnichek, M.: Formal verification of smart contracts with the K framework (2019), https://www.apriorit.com/dev-blog/592-formal-verification-with-k-framework
27. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.: Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. p. 67–82. CCS '18, Association for Computing Machinery, New York, NY, USA (2018)
28. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)

29. Zhang, Z., Zhang, B., Xu, W., Lin, Z.: Demystifying exploitable bugs in smart contracts. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). pp. 615–627. IEEE (2023)

A Auction Model

```
1
2  asm Auction
3
4  import ../../lib/asmeta/StandardLibrary
5  import ../../lib/asmeta/CTLlibrary
6  import ../../lib/solidity/EVMlibrary
7  import ../../lib/attackers/SelfdestructAttacker
8
9  signature:
10
11   dynamic controlled currentFrontrunner : StackLayer -> User
12   dynamic controlled currentBid : StackLayer -> Integer
13
14   dynamic controlled owner : StackLayer -> User
15
16
17   /* USER and METHODS */
18   static auction : User
19   static user_owner : User
20
21   static bid : Function
22   static destroy : Function
23
24 definitions:
25
26   rule r_Save ($n in StackLayer) =
27     par
28       currentFrontrunner($n) := currentFrontrunner(global_state_layer)
29       currentBid($n) := currentBid(global_state_layer)
30       owner($n) := owner(global_state_layer)
31     endpar
32
33
34   rule r_Destroy =
35     if executing_function(current_layer) = destroy then
36       switch instruction_pointer(current_layer)
37         case 0 :
38           r_Selfdestruct[user_owner]
39         case 1 :
40           r_Ret[]
41       endswitch
42     endif
```

```
43
44  rule r_Bid =
45    if executing_function(current_layer) = bid then
46      switch instruction_pointer(current_layer)
47        case 0 :
48          r_Require[amount(current_layer) > currentBid(global_state_layer)
                ]
49        case 1 :
50          if isDef(currentFrontrunner(global_state_layer)) then
51            instruction_pointer(current_layer) := instruction_pointer(
                  current_layer) + 1
52          else
53            instruction_pointer(current_layer) := 4
54          endif
55        case 2 :
56          r_Transaction[auction, currentFrontrunner(global_state_layer),
                currentBid(global_state_layer), fallback]
57        case 3 :
58          r_Require[call_response(current_layer+1)]
59        case 4 :
60          par
61            currentFrontrunner(global_state_layer) := sender(current_layer
                  )
62            instruction_pointer(current_layer) := instruction_pointer(
                  current_layer) + 1
63          endpar
64        case 5 :
65          par
66            currentBid(global_state_layer) := amount(current_layer)
67            instruction_pointer(current_layer) := instruction_pointer(
                  current_layer) + 1
68          endpar
69        case 6 :
70          r_Ret[]
71      endswitch
72    endif
73
74  rule r_Fallback_Auction =
75    if executing_function(current_layer) = fallback then
76      switch instruction_pointer(current_layer)
77        case 0 :
78          r_Ret[]
79      endswitch
80    endif
81
82
83  CTLSPEC ag((sender(current_layer) = user and
84    executing_function(current_layer) = bid and
85    instruction_pointer(current_layer) = 0)
86    implies af(currentFrontrunner(global_state_layer) = user)
```

```
87  )
88
89
90  invariant over sender : executing_function(current_layer) = destroy
        implies
91        sender(current_layer) = owner(global_state_layer)
92
93
94  main rule r_Main =
95    if transaction then
96      seq
97        r_Save[global_state_layer + 1]
98        r_Save_Env[global_state_layer + 1]
99        r_Save_Att[global_state_layer + 1]
100       global_state_layer := global_state_layer + 1
101       r_Transaction_Env[]
102     endseq
103   else
104     if current_layer = 0 then
105       seq
106         par
107           r_Save[0]
108           r_Save_Env[0]
109           global_state_layer := 0
110         endpar
111         r_Transaction[user, random_user, random_amount, random_function]
112       endseq
113     else
114       switch executing_contract(current_layer)
115         case auction :
116           par
117             r_Bid[]
118             r_Destroy[]
119             r_Fallback_Auction[]
120           endpar
121         case attacker :
122           r_Attacker[]
123         otherwise
124           r_Ret[]
125       endswitch
126     endif
127   endif
128
129
130 default init s0:
131
132  /*
133   * LIBRARY FUNCTION INITIZLIZATIONS
134   */
```

```
135 function executing_function ($sl in StackLayer) = if $sl = 0 then none
        endif
136 function executing_contract ($cl in StackLayer) = if $cl = 0 then user
        endif
137 function instruction_pointer ($sl in StackLayer) = if $sl = 0 then 0
        endif
138 function current_layer = 0
139 function transaction = false
140 function balance($c in User, $n in StackLayer) = if $n = 0 then 10
        endif
141 function global_state_layer = 0
142 function call_response($n in StackLayer) = false
143 function disabled($u in User, $sl in StackLayer) = false
144 function owner($sl in StackLayer) = user_owner
145
146 /*
147  * MODEL FUNCTION INITIALIZATION
148  */
149 function currentBid($n in StackLayer) = if $n = 0 then 0 endif
```