Modeling and verification of smart contracts with Abstract State Machines*

Chiara Braghin, Elvinia Riccobene, Simone Valentini
Computer Science Department, Università degli Studi di Milano, Italy
{chiara.braghin,elvinia.riccobene,simone.valentini}@unimi.it

ABSTRACT

Blockchain is a decentralized and distributed ledger system that records and verifies transactions across a network of computers and ensures transparency, immutability, and trustworthiness. Smart contracts are programs or protocols embedded into the distributed ledgers and are used to automate agreements between parties of a blockchain. Smart contracts are vulnerable to attacks due to their immutable and public nature, thus, it is important to guarantee the correctness of contracts already at design-time in order to avoid catastrophic events and huge loss of money.

In this paper, we investigate the usage of the Abstract State Machine (ASM) formal method for the specification, validation and verification of Ethereum smart contracts. We present the ASM model of the Ethereum virtual machine, and we define a set of ASM libraries that simplify smart contracts modeling. We also provide models of malicious contracts in terms of ASM libraries that can be used to check if a given smart contract is vulnerable to a certain attack. As a proof of concept, we show our approach by exploiting the DAO smart contract and its vulnerability to a reentrancy attack.

CCS CONCEPTS

• **Security and privacy** → **Formal security models**; • **Software and its engineering** → **Software verification and validation**;

KEYWORDS

Smart contracts, Ethereum Virtual Machine, Modeling, Validation & Verification

ACM Reference Format:
Chiara Braghin, Elvinia Riccobene, Simone Valentini. 2024. Modeling and verification of smart contracts, with Abstract State Machines. In *Proceedings of ACM SAC Conference (SAC'24).* ACM, New York, NY, USA, 8 pages. https://doi.org/https://doi.org/10.1145/3605098.3636040

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

SAC '24, April 8–12, 2024, Avila, Spain

ACM ISBN 979-8-4007-0243-3/24/04…$15.00
https://doi.org/https://doi.org/10.1145/3605098.3636040

1 INTRODUCTION

Blockchain is a decentralized and distributed ledger system that securely records and verifies transactions across a network of computers. It relies on cryptographic primitives and consensus mechanisms to ensure transparency, immutability, and trustworthiness. While the first-generation blockchains were focused mainly on cryptocurrencies, a new generation emerged later by an envisioned idea of Nick Szabo in 1996 [17], which embeds the distributed ledgers by so-called *smart contracts* that enable them to function as distributed computing platforms.

Ethereum [9] was the first blockchain implementing the idea of smart contracts, and this combination of a distributed ledger with the ability to execute immutable code without the need for a trusted third party when predetermined conditions are met opened to a lot of potential new use cases. Nowadays, smart contracts are used to automate and enforce contractual terms between two or more parties entering the agreement, in scenarios ranging from financial trades and services, to insurance, credit authorization, legal processes, and even crowdfunding agreements.

As a consequence, both the number of blockchains supporting smart contracts and, unfortunately, the number of attacks have grown with expensive consequences. For example, the infamous DAO exploit [1] resulted in the loss of almost $60 million worth of Ether. Logic errors tend to be one of the most common types of blockchain smart contract vulnerabilities. They may include typographical errors, misinterpretation of specifications, or more serious programming errors that decrease the security of smart contracts. Since smart contracts cannot be deleted by default, and interactions with them are irreversible, the only remedy for recovering from errors is to hard-fork the blockchain and revert one of the forks back to the state before the incorrect smart contract was executed. However, this remedy itself is devastating as it defeats the core values of blockchain, as immutability and decentralized trust. Thus, it is of paramount importance to guarantee the correctness of contracts at design-time in order to avoid catastrophic events and huge loss of money.

Formal verification, which uses formal methods for specifying, designing, and verifying programs, has been used for years to ensure correctness of critical hardware and software systems. When used for smart contracts, formal verification can prove that a contract's business logic meets a predefined specification. With respect to other techniques such as testing or code inspection, formal verification gives stronger guarantees that a smart contract is functionally correct. This comes at the expense of other challenges such as (i) a difficult modeling language, making the writing of the model error-prone as well; (ii) a verification process that might require

user interaction and knowledge of the tool's internals; (iii) the verification results being difficult to interpret and to bind to the original protocol; (iv) scalability.

Several studies have been performed to identify current topics and challenges of smart contracts and different approaches have been proposed, especially towards the formal verification of Solidity smart contracts [10]. However, the proposed approaches either target a limited number of vulnerabilities, or use complex notations that may lead to incorrect formal specifications [14, 15].

In this paper, we present the first results of ongoing long-term research that investigates the usage of the Abstract State Machine formal method [7, 8] for the specification and validation of Ethereum smart contracts. ASMs offer several advantages as a formal method: (1) due to their *pseudo-code format*, they can be easily understood by practitioners and can be used for *high-level programming*; (2) they allow for system specification at any desired *level of abstraction* and with different computational paradigms, from a *single* agent to distributed *multiple* agents; (3) they are *executable models*, so they are suitable also for lighter forms of model analysis such as simple simulation to check model consistency w.r.t. system requirements; (4) the state-based computational paradigm of the ASMs well captures the execution model of the Ethereum state machine; (5) the ASMETA framework allows for an integrated usage of tools for different forms of model analysis (e.g., ASM models can be validated in terms of model simulation, animation, and scenarios execution. It is also possible to verify properties expressed in temporal logic by means of model checking).

Specifically, we here present the ASM model of the Ethereum virtual machine, which is on the base of a smart contract execution. This formalization has led us to the definition of a set of ASM libraries (i.e., a set of predefined domains and functions, as well as rules for invoking the execution of smart contract functions, fallback function included) that help and simplify smart contracts modeling. We also show how to model malicious contracts, and we provide an ASM library of possible attackers in order to check whether a given smart contract is vulnerable to a specific type of attack. We show how to exploit the model checking and invariant violation approaches of the ASMETA framework to detect contract vulnerabilities and potential attacks.

The paper is organized as follows: in Section 2 we provide a background description of the Ethereum smart contracts and the formal approach we use. Section 3 presents the ASM-based modeling of the Ethereum virtual machine, of smart contracts, and of malicious contracts working as attackers. In Section 4 we show how to exploit validation and verification techniques supported by ASMETA to detect smart contracts vulnerabilities. In Section 5 we compare our results with existing approaches. Section 6 concludes the paper and outlines future research directions.

2 BACKGROUND

2.1 Ethereum and Smart Contracts

In Ethereum, 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 such as a Metamask or Coinbase wallet. They are 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.

Smart contract's code is executed on the Ethereum blockchain by the Ethereum Virtual Machine (EVM). The Ethereum blockchain can then be considered as a distributed state machine with a global state that is updated when transactions are executed, rather than just a distributed ledger. There are three types *transactions* supported on Ethereum: (i) *regular transactions*, transferring ether from one party to another; (ii) *contract deployment transactions*, creating a smart contract; and (iii) *execution of a contract*, a transaction that interacts with a deployed smart contract. Only EOAs can initiate transactions, contract accounts can only send transactions in response to receiving a transaction. Transactions are cryptographically signed data messages that contain a set of information such as the sending address, the receiving address (in case of an externally-owned account, the transaction will transfer value; if a contract account, the transaction will execute the contract code), the amount of ETH to transfer from sender to recipient (denominated in wei, where 1 ETH equals 1e+18 wei), and a data field that can contain either code or a message to the recipient. These information is recorded in the special *global variable* called `msg` (for example, `msg.sender` specifies the address that originated the function call, and `msg.value` the number of wei sent with the message).

Transactions are broadcasted to the network. Then, a block producer will execute the transaction in its local EVM and propagate the resulting state change to the rest of the Ethereum network. Transactions require a fee to the sender and must be mined to become valid. The Ethereum VM is the runtime environment for smart contracts in Ethereum, with a persistent memory system to record the blockchain state, and both a transient memory and a program stack with a depth of 1024 items to execute smart contracts. Compiled smart contract bytecode is executed as a number of EVM opcodes, however programmers can write smart contracts in Solidity, a high-level programming language. Smart contracts have a limited execution context - they can only access their own state, the context of the calling transaction, and some information about recent blocks.

2.2 Abstract State Machines in a Nutshell

The Abstract State Machine (ASM) formal method [7, 8] is a rigorous approach to software and system design, analysis, and verification. It provides a mathematical framework for modeling complex systems, capturing their behavior in a precise and abstract manner.

ASMs are based on the concept of state machines, where each state represents a possible configuration of the system in terms of an algebraic structure (i.e., domains of objects with functions defined on them), and *transitions* between states are given by firing well-defined rules, i.e., rules constructed by a given and fixed number of *rule constructors*. At each computation step, all transition rules are executed in parallel by leading to simultaneous (consistent) updates of a number of locations – i.e., memory units defined as pairs (*function-name*, *list-of-parameter-values*)–, and therefore changing

```
asm example
import ../../Libraries/CTLlibrary
import ../../SolidityLibraries/SimpleReentrancyAttack
import ../../SolidityLibraries/EVMlibraryimport
import ../../Libraries/StandardLibrary

signature:
/* CONTRACT ATTRIBUTES */
dynamic controlled customer_balance : Prod(User, StackLayer) -> MoneyAmount
/* FUNCTIONS PARAMETERS */
dynamic controlled value_deposit : StackLayer -> MoneyAmount
dynamic controlled value_withdraw : StackLayer -> MoneyAmount
/* METHODS DEFINITIONS AND USER DEFINITIONS */
static dao : User
static deposit : Function
static withdraw : Function

definitions:
/* FUNCTION RULES for CONTRACT SPEC */
rule r_Deposit = ...
rule r_Withdraw = ...
rule r_Fallback_Contract = ...
/* INVARIANT */
Invariant over customer_balance : customer_balance(attacker) >= 0
/* TEMPORAL LOGIC FORMULAS */
CTLSPEC ef(customer_balance(attacker) < 0)
/* MAIN */
main rule r_Main = ...

default init s0:
/* LIBRARY FUNCTION INITIZLIZATIONS */
function current_layer = 0
function balance($c in User, $n in StackLayer) = if $n = 0 then 10 endif
...
/* MODEL FUNCTION INITIALIZATION */
function customer_balance($c in User, $n in StackLayer) = if $n = 0 then 1 endif
...
```

Listing 1: Skeleton of an ASM model.

functions interpretation from one state to the next one. Location *updates* are given as assignments of the form $loc := v$, where loc is a location and v its new value. The rule constructors used for our modeling are those for guarded updates (`if-then`, `switch-case`), parallel updates (`par`), nondeterministic updates (`choose`).

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), and *controlled* (read and written by the machine).

Despite their strong mathematical basis, due to their *pseudo-code format*, ASM models can be easily understood by practitioners and can be used for *high-level programming*. Indeed, an ASM model has a predefined structure consisting of (see Listing 1 for a schema written in the `AsmetaL` notation): an *import* section to include parts of already defined models; a *signature*, which contains declarations of domains and functions; a block of *definitions* of static domains and functions, transition rules, state invariants and properties to verify; a *main rule*, which is the starting point of a machine computation; a set of *initial states*, one of which is elected as *default* and defines an initial value for the (controlled) machine locations.

An ASM *computation* (or *run*) is defined as a finite or infinite sequence $S_0, S_1, \ldots, S_n, \ldots$ of states of the machine, where S_0 is an initial state and each S_{n+1} is obtained from S_n by firing the unique *main rule* which in turn could fire other transitions rules.

Tools and Validation/Verification Techniques. ASMETA [4] is a tool-set around the ASMs. It allows for an integrated usage of tools for model editing (with `AsmetaL` notation) and different forms of model analysis, it is well-maintained and under continuous features improvement.

Model validation exploits the characteristic of an ASM model of being *executable*. ASMs can be validated in terms of model simulation (by using `AsmetaS`) –interactively or randomly –, animation (by `AsmetaA`), and scenarios execution (by `AsmetaV`). In the latter case, each scenario contains a description of the expected system behaviour and the tool checks whether the model behaves as expected. Scenarios are written in the `Avalla` language, providing special commands to: **set** the values of monitored functions, perform one **step** of simulation, **exec** rules or function updates, **check** that some properties hold. For our purposes of detecting contract vulnerabilities and the possibility of expected attacks, we also exploit the mechanism of *invariant violation* during execution. Invariants are logic formulas expressing safety properties that need to be guaranteed; they can be expressed either in relation to a model and must hold for any possible execution of the model, or in relation to a specific scenario and must hold only for the particular execution. Invariants violation reveals unsatisfaction of such properties and therefore an incorrect behavior.

Model verification is possible by verifying properties expressed in temporal logic; the model checking `AsmetaSMV` maps `AsmetaL` models to the model checker `NuSMV`: the tool will check if the property holds during all possible model executions.

3 ASM MODELING OF ETHEREUM

In order to ensure the correctness of a smart contract both in terms of security properties such as safety, liveness, reachability, deadlocks, and of safe interaction with other contracts, all the main components described in Section 2.1 need to be modeled. To this aim, we defined a library of ASM functions and domains in the `AsmetaL` notation modeling both types of accounts and the Ethereum VM functioning (Section 3.1), we specified a template to help translating a smart contract into an ASM model (Section 3.2), and we defined an initial set of possible attackers, i.e., ASM models of smart contracts trying to exploit known vulnerabilities (Section 3.3).

When designing and validating a specific smart contract, all that is needed is to import the EVM library and the attacker model, then library domains and functions have to be instantiated to deal with the specific smart contract, while library rules can be directly used within the model, and new rules need to be defined in order to model the smart contract functions.

In the following, we show only the relevant parts of the ASM models. The complete code is available on GitHub at https://github.com/smart-contract-verification/ethereum-via-asm.

3.1 Modeling of the Ethereum Virtual Machine

The Ethereum VM orchestrates the interaction between accounts and the blockchain by means of transactions, which are the basic elements to be modeled.

Accounts are modeled by means of the EVM library domains and functions (see Listing 2).

```
abstract domain User
domain MoneyAmount subsetof Integer

/* USER ATTRIBUTES */
dynamic controlled balance : User -> MoneyAmount
derived is_contract : User -> Boolean
```

Listing 2: EVM library: Definition of Ethereum accounts.

The domain `User` identifies *Ethereum accounts*, whereas the function `is_contract` distinguishes between the two possible types of account, i.e., externally-owned account and smart contract account. The function `balance` associates the ether *balance* to the account.

As described in Section 2.1, the Ethereum VM is a stack-based, big-endian VM with a word size of 256-bits and is used to run the smart contracts on the Ethereum blockchain. As in any stack-based VM, the memory is an *execution stack* that keeps track of the active functions and the *stack frame* represents an element of the stack recording a collection of local variables, function parameters and return address. In our case, in the ASM model of the Ethereum VM (Listing 3), the domain `StackLayer` models the stack recording the active smart contract functions, and the stack frame is populated by functions `sender`, `receiver`, `amount`, recording the account address that initiated the transaction, the receiver, and the amount of ether transferred, respectively. In addition, functions `executing_function` and `executing_contract` keep track of the running function and of the smart contract it belongs to, while `instruction_pointer` is a function that indicates the current instruction being executed. Then, there is the *stack pointer* (function `current_layer`) that points to the stack frame of the method currently being executed (i.e., at the top of the stack).

```
abstract domain Function
domain StackLayer subsetof Integer
domain InstructionPointer subsetof Integer

/* STACK MANAGEMENT */
dynamic controlled current_layer : StackLayer

/* STACK FRAME CONTENT 1 – TRANSACTION INFO */
dynamic controlled sender : StackLayer –> User
dynamic controlled receiver : StackLayer –> User
dynamic controlled amount : StackLayer –> MoneyAmount

/* STACK FRAME CONTENT 2 – SMART CONTRACT EXECUTION */
dynamic controlled executing_contract : StackLayer –> User
dynamic controlled executing_function : StackLayer –> Function
dynamic controlled instruction_pointer : StackLayer –> InstructionPointer
```

Listing 3: EVM library: the EVM Execution Stack.

Listing 3 reports the fragment of the `EVMLibrary` where the EVM execution stack is defined, whereas Fig. 1 graphically exemplifies the structure of the stack.

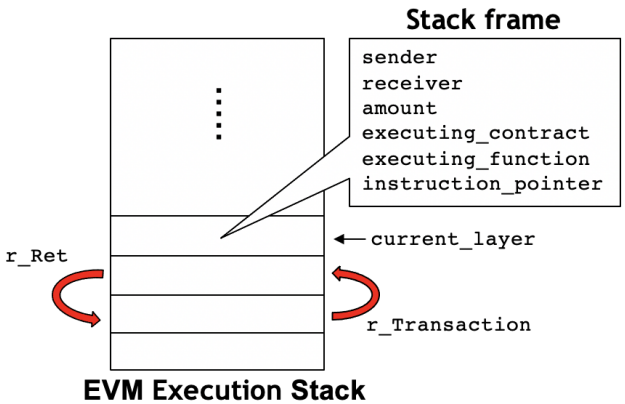

Figure 1: Ethereum VM Modeling.

Whenever a new method is called, a new stack frame is created and is pushed into the program stack. Whenever a method ends its execution, the frame is popped from the stack. To manage transactions and the updating of the stack pointer `current_layer`, we designed two different rules: `r_Transaction`, to perform a transaction, and `r_Ret`, to terminate one.

`r_Transaction` rule models a transaction that can be a regular one, or a call to a smart contract's method (see Listing 4). When the rule is called, a first condition checks if the sender owns enough ether to perform the transaction (line 5). If so, in case of a regular transaction, the balances of sender and receiver are updated (lines 8-9). If the receiver is a smart contract (the check of line 10), it is either the case of the call to a method, or the continuation of the execution of a method (lines 20-21). In the first case, a new stack frame is pushed in the stack (i.e., the `current_layer` is increased), with metadata copied from the transaction parameters (e.g., sender and contract address, cost of the transaction, name of the function, etc.), and `instruction_pointer` set to 0, i.e., pointing to the first instruction of the function. In the case of the continuation of the execution of a function, only the instruction pointer needs to be incremented.

```
1  /* TRANSACTION RULE */
2  macro rule r_Transaction($s in User, $r in User, $n in MoneyAmount, $f in Function) =
3    if balance($s) >= $n and $n >= 0 then
4    let ($cl = current_layer) in
5      par
6        balance($s) := balance($s) – $n // subtracts the amount from the sender user balance
7        balance($r) := balance($r) + $n // adds the amount to the receiver user balance
8        if is_contract($r) then
9          par
10           sender($cl + 1) := $s // set the transition attribute to the sender user
11           amount($cl + 1) := $n // set the transaction attribute to the amount of coin to transfer
12           current_layer := $cl + 1
13           executing_contract($cl + 1) := $r
14           executing_function($cl + 1) := $f
15           instruction_pointer($cl + 1) := 0
16         endpar
17       endif
18       if is_contract($s) then
19         instruction_pointer($cl) := instruction_pointer($cl) + 1
20       endif
21     endpar
22   endlet
23   endif
```

Listing 4: EVM library: `r_Transaction` rule.

`r_Ret` rule is called when a function ends its execution. When the rule is invoked, the last stack frame is popped and `current_layer` is decremented (see Listing 5).

```
/* RETURN RULE */
macro rule r_Ret =
  current_layer := current_layer – 1
```

Listing 5: EVM library: `r_Ret` rule.

3.2 Modeling of a Smart Contract

Contracts in Solidity are similar to classes in object-oriented languages. Each contract can contain declarations of variables, functions, function modifiers, events, errors, struct types and enum types. When the function of a smart contract is called, either by an externally-owned account or another contract, the execution is handled by the Ethereum VM.

A Solidity smart contract is formalized by an `AsmetaL` model that includes the `EVMLibrary` library described in Section 3.1, instantiates the library domains personalizing them according to the

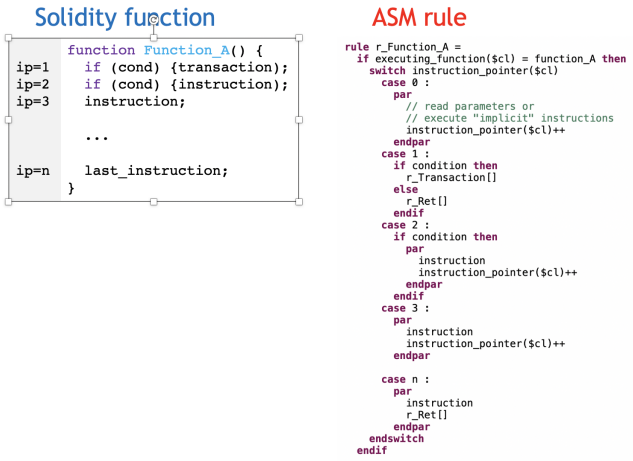

Figure 2: ASM modeling of a smart contract function.

specific smart contract, uses the library rules (i.e., `r_Transaction` and `r_Ret`) and defines contract-specific rules and functions. The translation of a single smart contract can be systematically done by following these steps:

(1) initialize the EVM library domain `User` with a unique name to identify the smart contract account, and `Function` with one unique label for each function of the smart contract.
(2) define one function for each contract *state variable*, i.e., variables specifying contract's attributes that are stored permanently on the blockchain. To this aim, the function domain will always be `User`, whereas the codomain will depend on the type of the variable.
(3) define one function for each *local variable*, i.e., function parameters or variables defined inside a function, that are stored only for the lifetime of a contract function's execution. In this case, the function domain will always be `StackLayer`, so that they will be added to the corresponding stack frame, and the codomain will depend on the type of the variable.
(4) define an ASM transition rule for each function of the smart contract, including the `fallback` function, if present. In addition, a `r_contractName` rule is defined launching the parallel execution of all the transition rules.
(5) if the contract is analyzed in isolation, the `r_Main` rule and the set of initial states in section `default init s0` must be included in the model. Otherwise, in case of interacting smart contracts, a single `r_Main` rule included in any of the contracts is used to orchestrate their behaviour (see Section 3.3 for further details).

The definition of the `AsmetaL` transition rule corresponding to a function of the contract is rather straightforward: the skeleton of the ASM function is a `switch` case on the value of the `instruction_pointer`, with as many cases as the number of instructions of the smart contract. Then, each Solidity instruction or control structures (e.g., `if`, `else`, `while`, `do`, `for`, `break`, `continue`, and `return`) are translated accordingly. Fig. 2 shows the translation of a Solidity smart contract fragment (on the left, with `ip` denoting the value of the instruction pointer) into an `AsmetaL` model (on the right). `case 0` is a "special" case where the actual arguments of the function are assigned to the local variables. `case n` includes a call to rule `r_Ret`, corresponding to the return of the function. In case the function calls another function, a call to the ASM `r_Transaction` rule is done (see `case 1`).

Example 3.1. Consider as an example the (in)famous DAO smart contract depicted in Fig. 3. It is a simplified version of the smart contract called "The DAO" ruling a decentralized, community-controlled investment fund created in 2016 (DAO stands for Decentralized Autonomous Organization). Through the contract, people were able to purchase The DAO's community token by depositing ETH (with a `Deposit` function), and that ETH became the investment funds that The DAO invested on behalf of its community of token-holding investors, with the array `balances` used to keep track of the investments of each account. Investors could then withdraw part of their investment (with a `Withdraw` function).

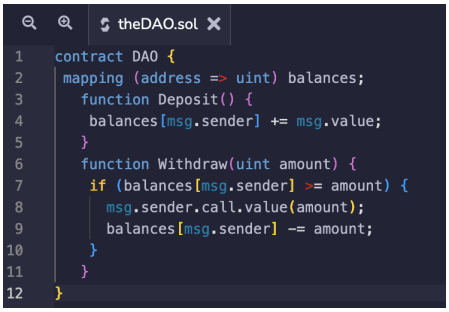

Figure 3: The DAO (vulnerable) smart contract.

Following the translation steps described above, the ASM model of the DAO contract (see Listing 6): (1) initializes the domain `User` with the unique name `dao` to identify the DAO smart contract account, and (2) populates the `Function` domain with the labels `deposit` and `withdraw` identifying the two functions of the smart contract. Then, (3) function `customer_balance` is defined to specify the smart contract `balances` array, while `value_deposit` and `value_withdraw` model the parameters of the two functions, both collecting ETHs. Then, (4) an ASM transition rule is defined for each function of the smart contract.

```
import EVMLibrary

signature:
/* ACCOUNT AND FUNCTION NAMES DEFINITION */
static dao : User
static deposit : Function
static withdraw : Function

/* CONTRACT ATTRIBUTES */
dynamic controlled customer_balance : User -> MoneyAmount

/* FUNCTIONS PARAMETERS */
dynamic controlled value_deposit : StackLayer -> MoneyAmount
dynamic controlled value_withdraw : StackLayer -> MoneyAmount
```

Listing 6: The ASM model of the DAO: domains initialization.

In Listing 7, we report the definition of the `r_Deposit` rule, modeling the `deposit` function. Since the function's body has only one instruction, the body of the corresponding ASM rule has a `switch` statement with only two possible executing paths, one for

the function's argument passing (`case 0`), the other modeling the single instruction that updates the amount of ETH invested by the user (`case 1`).

```
rule r_Deposit =
  let ($cl = current_layer) in
    let ($scl = sender($cl)) in
      if executing_function($cl) = deposit then
        switch instruction_pointer($cl)
          case 0 :
            par
              value_deposit($cl) := amount($cl)
              instruction_pointer($cl) := instruction_pointer($cl) + 1
            endpar
          case 1 :
            par
              customer_balance($scl) := customer_balance($scl) + value_deposit($cl)
              r_Ret[]
            endpar
        endswitch
      endif
    endlet
  endlet
```

Listing 7: The ASM model of the DAO: `r_Deposit` rule.

Listing 8 reports the `r_DAO` rule that launches the parallel execution of all the DAO transitions rules.

```
rule r_DAO =
  par
    r_Deposit[]
    r_Withdraw[]
    r_Fallback_DAO[]
  endpar
```

Listing 8: The ASM model of the DAO: `r_DAO` rule.

3.3 Modeling of the Attacker

In order to ensure safe interactions with other contracts and to detect possible vulnerabilities that could be exploited by malicious users, we explicitly model attackers. The attacker is just a particular case of smart contract in which there are one or more functions tailored to exploit a vulnerability. To this aim, we started to build a *library of attackers*, differing on the capabilities they have and based on well-known vulnerabilities.

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 Listing 9.

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

Listing 9: The DAO attacker: `r_Main` rule.

Example 3.2. The DAO contract of Fig. 3 is vulnerable to what is now called a *reentrancy attack*, since the `balances` state variable is updated only after the actual smart contract's ETH balance is modified (lines 8-9). This is only possible because of the order in which the smart contract is set up to handle transactions, with the vulnerable smart contract first checking the balance, then sending the funds, and then finally updating its balance. The time between sending the funds and updating the balance creates a window in which an attacking smart contract can make another call to withdraw its funds, and so the cycle continues until all the funds are drained.

DAOattack.sol

```
1  contract Attacker {
2    ...
3      function attack() {
4        dao.Withdraw(5);
5      }
6      function () payable {
7        dao.Withdraw(10);
8  }
```

Figure 4: A smart contract attacking the DAO.

Fig. 4 shows the code of a malicious smart contract exploiting the vulnerability, while Listing 10 reports the corresponding model. In order to generalize the use of the attacker model to detect the vulnerabilities of any contract, we defined in the EVM library the monitored functions `random_user` and `random_function`, which yield the name of the smart contract and the name of the function under test, respectively.

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

Listing 10: The DAO attacker: rules definition.

4 VALIDATION AND VERIFICATION

We exploit the ASMETA tools to analyze the behavior of smart contracts when running in combination with different attackers. In the following, we illustrate the DAO smart contract case study. The goal is to show that it is vulnerable to a reentrancy attack. To this aim, we combined the DAO and the attacker models by using the schema of the main rule reported in Listing 9.

We defined the following invariant (mapped into a NuSMV `AG` safety property) stating that the `balance` of the `customer` (i.e., the `attacker`, which works as the customer of the DAO contract) is ≥ 0, and a reachability property checking whether a state will be reached where the balance of the customer is negative.

```
1  invariant over customer_balance : customer_balance(attacker) >= 0
2  CTLSPEC ef(customer_balance(attacker) < 0)
```

Fig. 5 shows the results of NuSMV: contrary to what one could expect, the EF property is proved true, while the AG property is false.

```
*** Copyright (c) 2010-2014, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://minisat.se/MiniSat.html
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson

-- specification EF costumer_balance_ATTACKER < 0  is true
-- specification AG costumer_balance_ATTACKER >= 0  is false
-- as demonstrated by the following execution sequence
```

Figure 5: NuSMV result of the proof.

By using the ASMETA tools, the counter-example reported by the model checker can be transformed into a scenario, as given in Fig. 6. In the scenario, at the beginning it is specified which

```
scenario reentrancy_attack_scenario

load DAO_noAgent.asm

set random_user := attacker;
set random_function := attack;
step
check customer_balance(attacker) >= 0;

set random_user := dao;
set random_function := withdraw;
step
    // same block for more n times
    // n depends on the initial value
    // of customer_balance
set random_user := dao;
set random_function := withdraw;
step
check customer_balance(attacker) >= 0;

set random_user := dao;
set random_function := fallback;
step
check customer_balance(attacker) >= 0;
     // same block for more n-1 times
```

Figure 6: Critical scenario of the DAO model.

smart contract and which function have to be executed (the first two `set` instructions), i.e., the `attack` function of the `attacker` contract. Then, the machine makes a `step`, and the first `check` succeeds. Then, the `withdraw` function of the `dao` contract is called a number n of times greater than the initial value of the `balance` of the `customer`. In all the machine's steps, the `check` succeeds, and at each step the value of `customer_balance` is decreased by 1 (as described in example 3.1, Sect. 3.2). Then, the `fallback` function of the `dao` contract is called at least $n - 1$ times, so bringing the machine into a state where the `check` fails, and which is the state where the safety property (`AF customer_balance(attacker) >= 0`) is false.

By running the model, following the same sequence of steps, as expected, we get a state with model-invariant violation (as highlighted in red in Fig. 7).

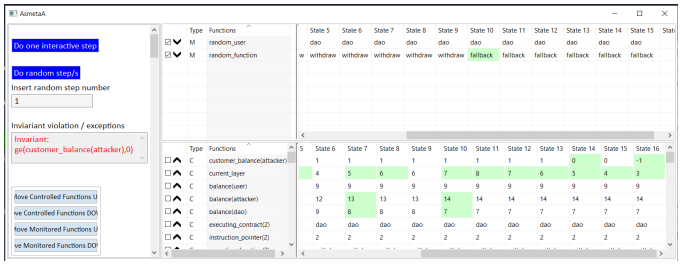

Figure 7: Execution of the DAO model with `AsmetaA`.

A similar V&V analysis has been carried out on the other models of smart contracts we developed, in combination with the available models of attackers. So far, we modeled a number of vulnerable/correct smart contracts, such as the *Simple-transfer contract*, a correct contract that transfers a property between two accounts, the *Airdrop contract*, another example of a contract distributing tokens that is vulnerable to a reentrancy attack, and the *King of the Ether Throne game*, a contract with a vulnerable management of exceptions. As for the attackers, in addition to the reentrancy attacker, at the moment we modeled an inoffensive attacker and an attacker that exploits some well-known vulnerabilities in the usage of exceptions. We were able to perform V&V on the combination of smart contracts and attackers in order to formally guarantee a smart contract to be correct both with respect to different invariants (e.g., security properties) and in different malicious scenarios.

5 RELATED WORK

Different approaches have been followed to perform, in an effective way, validation and verification of smart contracts. For Ethereum, the formal model is derived either from Solidity or bytecode. The latter case has the advantage that the bytecode is always publicly available on the blockchain, whereas only a small number of contracts have their Solidity code available on public repositories. The drawback is that vulnerabilities are intercepted upon implementation, rather than at design time, and it may not be trivial to link the error to Solidity code. The various approaches differ in the formal techniques used, ranging from symbolic execution to theorem proving and model checking. In general, they all require a strong mathematical and logical background and use complex mathematical notations; this often prevents practitioners from using them. Instead, the models we propose can be read as high-level programs, and have a simple notation and very basic control flow constructors. Moreover, since they are executable, our models are suitable for simple 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 such as safety, liveness, reachability and deadlocks, and of safe interactions with other contracts. 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 that the ones detected by the approaches that only search a specific instructions pattern in the code.

For example, the tool Oyente [13] exploits symbolic execution to detect transaction-ordering dependences, timestamp dependence, and mishandled exceptions in Ethereum smart contracts. The tool explores paths in the bytecode and uses Z3 to check for symbolic conditions indicating bugs. Osiris [18] is a more recent tool implemented on top of Oyente's symbolic execution engine. It mainly focuses on the detection of integer bugs, like overflows and truncation errors, by using symbolic execution to explore paths in the bytecode, and emits constraints checked by a solver. It also employs taint analysis to reduce false positives. MAIAN [16] is another tool that, acts directly on the bytecode: it requires some liveness and safety specifications, then makes use of a concrete validator to check if the specifications are met or not.

The work in [5] provides a general smart contract template that can be represented as a state machine; the SPIN model checker is then used to verify properties and simulate executions of a shopping smart contract use case. Many research works exploit the NuSMV model checker for smart contract verification. For instance, the work in [2] makes use of NuSMV and Linear Temporal Logic (LTL) to model a behavior encompassing different smart contracts interacting with each other; in particular, a smart contract written in GO language is translated into Finite State Machine and then to Behavior Interaction Priority (BIP) and NuSMV to check the corresponding requirements. The work in [12] presents another method to model Ethereum smart contracts with NuSMV through the design of a label that is represented by a tuple changing its values state by state. In our case, we can use the same ASM model with all the different tools available into the ASMETA toolset, not only NuSMV.

The work presented in [6] outlines a framework to analyze and verify Ethereum smart contracts by translation to F*. It presents prototype tools Solidity* and EVM* that compile contracts to F* code. F*'s type system enables generating automated queries to statically verify contract properties like safety and functional correctness. The work in [11], instead, presents a framework named ZEUS to verify the correctness and fairness of smart contracts. It uses abstract interpretation and symbolic model checking with constrained Horn clauses to check safety properties. Another approach is presented in [3], which provides a formal verification framework for Ethereum smart contracts using Isabelle/HOL. It extends an existing EVM formalization with a program logic to reason about bytecode-level properties. The logic structures bytecode into blocks of straight-line code to simplify reasoning. This approach is used to verify the properties of a Solidity-compiled escrow contract.

Recently, other approaches have been presented, which share with us the idea of using Finite State Machines (FSM) for modeling smart contracts at design time. The authors in [14] use interface automata to model smart contracts and to ensure that violations of the agreement can be detected, and contractual clauses can be enforced by all parties involved in transactions. The work in [15] presents a graphical editor for designing contracts as FSMs and an automatic Solidity code generator from models; the authors also provide a set of plugins that implement security features and design patterns, which developers can easily add to their model.

6 CONCLUSION

In this paper, we present the first results of ongoing long-term research that investigates the usage of the Abstract State Machine formal method for the specification and validation of Ethereum smart contracts. We here present the ASM model of the Ethereum virtual machine, and we define a set of ASM libraries that provide the user with built-in modeling primitive (in terms of mathematical domains and functions), which help and simplify modeling of smart contracts. We also provide models of malicious contracts as ASM libraries to be used to check if a given smart contract is vulnerable or not to a certain attack. As a running case study to show our approach, we used the DAO smart contract and its vulnerability to a reentrancy attack.

In the future, we plan to extend the library of ASM models of malicious contracts in order to increase the capability of the method to analyze smart contract vulnerabilities. Specifically, we are working on refining the model of the Ethereum virtual machine to handle exceptions and developing models of malicious contracts that exploit Solidity exceptions.

REFERENCES

[1] N. Alchemy. 2019. A short history of smart contract hacks on ethereum: A.k.a. why you need a smart contract security audit.

[2] Sarra Alqahtani, Xinchi He, Rose Gamble, and Mauricio Papa. 2020. Formal verification of functional requirements for smart contract compositions in supply chain management systems. In *Proc. of the Annual Hawaii International Conference on System Sciences*, Vol. 2020-. 5278–5287.

[3] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In *Proc. of the 7th Int. Conf. on Certified Programs and Proofs (CPP18)*, Vol. 2018-. ACM, 66–77.

[4] Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. 2011. A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience* 41, 2 (2011), 155–166. https://doi.org/10.1002/spe.1019

[5] Xiaomin Bai, Zijing Cheng, Zhangbo Duan, and Kai Hu. 2018. Formal Modeling and Verification of Smart Contracts. In *ACM Int. Conf. Proc. Series*. ACM, 322–326.

[6] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Cédric Fournet et al. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proc. of the ACM Workshop on Programming Languages and Analysis for Security (PLAS16)*. ACM, 91–96.

[7] Egon Börger and Alexander Raschke. 2018. *Modeling Companion for Software Practitioners*. Springer. https://doi.org/10.1007/978-3-662-56641-1

[8] Egon Börger and Robert Stärk. 2003. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag.

[9] E. Foundation. 2017. Ethereum.

[10] Ikram Garfatta, Kais Klai, Walid Gaaloul, and Mohamed Graiet. 2021. A Survey on Formal Verification for Solidity Smart Contracts. In *Proc. of the 2021 Australasian Computer Science Week Multiconference (ACSW '21)*. Article 3, 10 pages.

[11] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *25th Annual Network and Distributed System Security Symposium (NDSS 2018)*.

[12] Jaturong Kongmanee, Phongphun Kijsanayothin, and Rattikorn Hewett. 2019. Securing Smart Contracts in Blockchain. In *34th IEEE-ACM Int. Conf. on Automated Software Engineering Workshops (ASEW 2019)*. IEEE, 69–76.

[13] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proc. of the ACM SIGSAC Conf. on Computer and Communications Security (CCS16)*, Vol. 24-28-. ACM, NEW YORK, 254–269.

[14] Gabor Madl, Luis Bathen, German Flores, and Divyesh Jadav. 2019. Formal Verification of Smart Contracts Using Interface Automata. In *IEEE Int. Conf. on Blockchain*. 556–563.

[15] Anastasia Mavridou and Aron Laszka. 2018. Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. In *Financial Cryptography and Data Security*. 523–540.

[16] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *34th Annual Computer Security Applications Conf. (ACSAC 2018)*. 653–663.

[17] Nick Szabo. 1996. Smart contracts: building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought,(16)* 18, 2 (1996), 28.

[18] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *ACM Int. Conf. Proc. Series*. 664–676.