A Modeling and Verification Framework for Ethereum Smart Contracts[⋆]

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

Computer Science Department,
Università degli Studi di Milano, via Celoria 18, Milan, Italy
{simone.valentini,chiara.braghin,elvinia.riccobene}@unimi.it

Abstract. Blockchain has shown to be a versatile technology with applications ranging from financial services and supply chain management to healthcare, identity verification, etc. Thanks to the usage of smart contracts, blockchain can streamline and automate complex processes, eliminating the need for intermediaries and reducing administrative overhead. Smart contracts often handle valuable assets and execute critical functions, making them attractive targets for attackers. Thus, secure and reliable smart contracts are necessary.
The long-term research we present aims to face the problem of safety and security assurance of smart contracts at design time. We are investigating the usage of the Abstract State Machine (ASM) formal method for the specification, validation, and verification of Ethereum smart contracts. We provide (*i*) a set of ASM libraries that simplify smart contracts modeling, (*ii*) models of malicious contracts to be used to check the robustness of a contract against some given attacks, (*iii*) patterns of properties to be checked to guarantee the operational correctness of the contract and its adherence to certain predefined properties.

Keywords: Blockchain · Ethereum · Smart contract verification · Abstract State Machine.

1 Introduction

As blockchain-based smart contracts gain mainstream adoption, the demand for reliable and secure smart contract design and development becomes increasingly vital. Smart contracts are programs that govern high-value financial assets, but they carry a substantial risk due to their public availability, immutability, and the ability for anyone to execute them. For example, the infamous DAO hack drained $70 million worth of Ether from a vulnerable smart contract that was not properly verified [2]. Costly vulnerabilities and exploits can seriously hinder trust and acceptance in the blockchain ecosystem. Formal verification may contribute

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

to the overall maturity and spread of blockchain technology by providing a robust methodology for ensuring the correctness of smart contracts.

The field of formal verification for smart contracts has made notable progress but still faces challenges. Existing tools have several limitations: bytecode-based tools cannot reason about contracts at design time, some tools use complex notations requiring a strong mathematical base that discourages many designers or engineers, others target only a limited number of vulnerabilities [11,12].

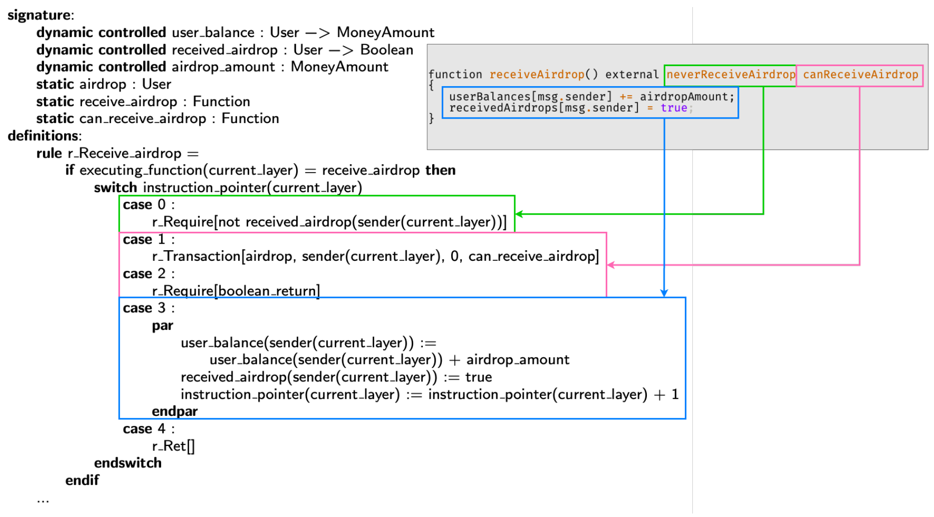
Our work aims to explore the potential of using Abstract State Machines (ASMs) [6,7] for specification and verification of Ethereum smart contracts written in Solidity. To this aim, we formalized the Ethereum Virtual Machine (EVM) and key language primitives to enable functional correctness proofs [8]. Currently, we can verify intra-contract properties and model inter-contract interactions to check the robustness of a contract against some given attacks. Our long-term vision is to build a practical verification framework using ASMs as the foundational formalism. Specific goals include:

1. Modeling the full semantics of Ethereum smart contracts;
2. Developing automatic mappings between Solidity source code and ASM models;
3. Identifying common smart contract patterns and functionalities across different blockchain platforms to define a domain-specific language in order to enhance the consistency and reliability of verification results;
4. Implementing a graphical interface to help correct-by-design smart contract development.

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

2 Smart Contract Modeling with ASM

The first and one of the most popular platforms implementing smart contracts is Ethereum [9,13]. Ethereum's structure is similar to Bitcoin, with the introduction of two main additional components: (*i*) the ability to embody some data into a transaction, and (*ii*) a stack-based virtual machine named *Ethereum Virtual Machine* (EVM) that allows code execution as a reaction to a transaction. In Ethereum, there are two types of accounts: *externally owned accounts* and *smart contract accounts*. They both have common attributes, such as an ether (ETH) balance or an identifying address, but contract accounts also provide a reference to some code stored within the blockchain, allowing them to respond to a transaction with a code execution that depends on the transaction data received. The EVM can then be considered as a distributed state machine with



```
signature:
    dynamic controlled user_balance : User −> MoneyAmount
    dynamic controlled received_airdrop : User −> Boolean
    dynamic controlled airdrop_amount : MoneyAmount
    static airdrop : User
    static receive_airdrop : Function
    static can_receive_airdrop : Function
definitions:
    rule r_Receive_airdrop =
        if executing_function(current_layer) = receive_airdrop then
            switch instruction_pointer(current_layer)
                case 0 :
                    r_Require[not received_airdrop(sender(current_layer))]
                case 1 :
                    r_Transaction[airdrop, sender(current_layer), 0, can_receive_airdrop]
                case 2 :
                    r_Require[boolean_return]
                case 3 :
                    par
                        user_balance(sender(current_layer)) :=
                            user_balance(sender(current_layer)) + airdrop_amount
                        received_airdrop(sender(current_layer)) := true
                        instruction_pointer(current_layer) := instruction_pointer(current_layer) + 1
                    endpar
                case 4 :
                    r_Ret[]
            endswitch
        endif
    ...
```

```
function receiveAirdrop() external neverReceiveAirdrop canReceiveAirdrop
{
   userBalances[msg.sender] += airdropAmount;
   receivedAirdrops[msg.sender] = true;
}
```

Fig. 1. An example of smart contract translation to ASMETA model.

a global state encompassing account balances, storage, and code that is updated when transactions are executed, rather than just a distributed ledger.

Smart contracts are written in Solidity, a high-level programming language, and then compiled into low-level bytecode. In [8], we showed how to use Abstract State Machines to model the EVM and smart contracts through the ASMETA framework [3] to exploit its powerful toolset. We presented some ASMETA libraries that, like the Ethereum Virtual Machine, provide rules and functions to model the Ethereum accounts with their specific attributes, and some primitives to model smart contract execution and their possible interactions. In particular, the library defines a stack structure that models the Ethereum execution stack and two rules to move along the stack, called `r_Transaction` and `r_Ret`. The rule `r_Transaction` models a transaction execution by increasing the stack size and storing within the stack layer all the relevant information on the transaction and the new execution environment, e.g., the executing contract and the instruction pointer; the rule also increases the value of caller's instruction pointer. The second one stops the execution by decreasing the stack structure and resuming the last execution.

In Fig. 1, we highlight the rationale for the translation of a Solidity smart contract into `AsmetaL`, the textual ASMETA's notation, presented in detail in [8]. The Solidity code on the right is the main function `receiveAirdrop()` of the Airdrop contract, which distributes liquidity to the accounts asking for it. The function first checks that two conditions are satisfied: (*i*) that the claiming account has not already received liquidity (i.e., the `neverReceivedAirdrop` modifier checks that `receivedAirdrops` value is false); and (*ii*) that the claimer is an eligible account (i.e., the `canReceiveAirdrop` modifier returns true, meaning that the claimer is a regular user, or a smart contract that can receive the

Airdrop). If both conditions are satisfied, then the contract adds the airdrop amount to the sender's balance and sets the sender's `receivedAirdrops` value to true.

Solidity functions are translated to an ASMETA rule that mainly consists of a `case` rule, which is essential to refer to the Solidity instructions within the function, and allows the instruction pointer to jump to a specific instruction when needed. The left part of Fig. 1 shows the transition rule corresponding to the function `receiveAirdrop()`. The modifiers checking the two initial conditions are translated into two `r_Require` rules that stop the function execution in case the condition is false. In particular, since the second condition, in order to check the eligibility of the claiming smart contract, makes a function call to the `canReceiveAirdrop` function implemented by the contract, it calls the `r_Transaction` rule that stops the execution of the current function and proceeds with the execution of the called function. More specifically, a new frame to the stack is added, with the `instruction_pointer` set to 0, and a new value for the `executing_function` and `executing_contract` fields. The `r_Ret[]` rule models the end of function execution by removing the frame on top of the stack structure and turning back to the previous layer. Please, notice that the Airdrop smart contract is vulnerable to a reentrancy attack since the `receivedAirdrops` value is updated only after the `canReceiveAirdrop` modifier is executed: a malicious user can make the `canReceiveAirdrop` function to call the `receiveAirdrop()` function multiple times, allowing the attacker to earn the airdrop amount more than once.

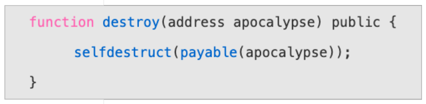
3 Smart Contract Verification with ASM

Smart contracts are often susceptible to various vulnerabilities that can lead to security breaches, financial losses, and other undesirable consequences. Design-time error detection would allow developers to catch and rectify issues before the smart contract is deployed on the blockchain, enhancing the overall security, reliability, and trustworthiness of smart contracts.

For the verification of smart contracts at design time, we used the ASMETA tool-set, which allows different forms of model analysis. In particular, 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. In particular, we can verify both intra-contract properties and vulnerabilities exploitable by malicious users through inter-contract interactions.

An example of intra-contract properties is a logical error inside a contract definition. For instance, consider the Airdrop contract described in the previous section, and suppose that the contract contains a logical error in a function `destroy` that invokes the Solidity library function `selfdestruct` without checking the sender, i.e., allowing any account to terminate the contract, remove the bytecode from the Ethereum blockchain, and send the contract funds to a specified address. The `destroy` function can be modeled by the rule in Fig. 2.

```
rule r_Destroy =
    if executing_function(current_layer) = destroy then
        switch instruction_pointer(current_layer)
            case 0 :
                par
                    selfdestroy(executing_contract) := true
                    instruction_pointer = instruction_pointer + 1
                endpar
            case 1 :
                r_Ret[]
        endswitch
    endif
```



```
function destroy(address apocalypse) public {
    selfdestruct(payable(apocalypse));
}
```

Fig. 2. The AsmetaL rule (on the left) of the Solidity function destroy (on the right).

The vulnerability can be detected through the AsmetaSMV tool by using the following CTL specification, asserting that, if the executing contract is airdrop, the executing function is autodestroy and the instruction pointer is 0, then the transaction sender (i.e., sender(current_layer)) that invoked the method has to be the contract owner:

```
CTLSPEC ag((executing_contract=airdrop and executing_function=autodestroy and instruction_pointer=0)
        implies (owner(airdrop)=sender(current_layer)))
```

Other more complex vulnerabilities involve the interaction among contracts. In Section 2, we mentioned how the Airdrop contract is vulnerable to the reentrancy attack. This attack involves a malicious contract that interrupts the normal flow of the canReceiveAirdrop modifier and executes some loop calls to the receiveAirdrop() function. The receiveAirdrop()'s body instructions updating the user's balance (e.g., user_balance attribute) and recording that the user has received the airdrop are triggered after the exit of each reentrancy call, when the neverReceiveAirdrop modifier is still true. Once the attacker model has been imported into the model of the main contract and the two models are executed in parallel (so to build a multi-agents ASM with an agent running the Airdrop contract and an attacker agent running the malicious one), it is possible to state some CTL formulas on the two interacting contracts. E.g., we can check if the user_balance attribute exceeds the airdrop_amount value through the following property, detecting the reentrancy attack.

```
CTLSPEC ag((forall $u in User with user_balance($u) <= airdrop_amount))
```

Given the heterogeneous nature of the possible types of vulnerabilities (e.g., unchecked external codes, gas limit and out-of-gas issues, timestamp dependencies reentrancy attacks, unintended exposure of sensitive data, or errors in the flow logic), starting from the taxonomy of vulnerabilities causes in [4], we have started to build a catalog (available at[1]) of:

– *patterns of properties* to guarantee the operational correctness of the contract and its adherence to certain predefined intra-contract properties;
– *models of malicious contracts* to check the robustness of a contract against those vulnerabilities that allow malicious contracts to manipulate the behavior of the vulnerable contract (i.e., due to unsafe inter-contract interaction).

[1] https://github.com/smart-contract-verification/ethereum-via-asm

4 Related Work

Several approaches and tools have been developed to formally verify smart contracts, particularly those written in languages like Solidity. However, challenges and research areas still require further development and improvement. For example, the languages and tools used for formal verification may not fully capture the specific features and behaviors of smart contracts. In addition, while formal verification tools can prove simple properties and catch common vulnerabilities, proving complex properties or verifying more sophisticated aspects of a smart contract's behavior remains challenging. Scalability is also a relevant issue, formal verification can be computationally expensive and time-consuming, especially for complex smart contracts or large codebases. In addition, smart contracts often interact with external systems, such as oracles and other smart contracts. Formal verification tools often struggle to model and verify the behavior of such external dependencies. At the moment, our approach also has some of these limitations, however with our approach, we are able to effectively handle external interactions.

Among the most relevant approaches, Certora [1] is a formal verification tool specifically designed for Ethereum smart contracts. It supports the verification of correctness properties and functional specifications. Other relevant works are remarkable for their usage of symbolic execution, like Oyente [10], which performs a symbolic execution analysis on Ethereum smart contracts. Instead, other works are focused on automatic translation from smart contract code, in particular, F* language [5] is a functional language that allows performing some verification; however, this method has a limited application since it does not support the translation of the full Solidity language.

5 Conclusion

In this paper, we presented the first results of ongoing long-term research that investigates the usage of the Abstract State Machine formal method for the specification and verification of Ethereum smart contracts. We developed libraries within the ASMETA framework that allow specifying Solidity contracts in ASMs, simulating their behavior, and formally verifying key properties. As demonstrated through examples, this enables detecting at design-time various vulnerabilities, such as access control issues, or reentrancy attacks. The long-term vision is to build an integrated environment that supports correct-by-design smart contract development, by providing automatic mappings between Solidity and ASMs, a GUI for ASM modeling, and seamless access to the verification back-end. There are also opportunities for extending this approach to other blockchain platforms beyond Ethereum. Overall, formal methods like ASMs can potentially increase the reliability and security of smart contracts, which is essential for their widespread adoption across domains. Further research is needed in this direction.

References

1. Certora Technology White Paper. `https://docs.certora.com/en/latest/docs/white\protect\discretionary{\char\hyphenchar\font}{}{}paper/index.html`, Accessed: 2024-02-20
2. Alchemy, N.: A short history of Smart Contract hacks on Ethereum: A.k.a. why you need a smart contract security audit (2019)
3. Arcaini, P., Gargantini, A., Riccobene, E., Scandurra, P.: A model-driven process for engineering a toolset for a formal method. Software: Practice and Experience **41**(2), 155–166 (2011). https://doi.org/10.1002/spe.1019
4. Atzei, N., Bartoletti, M., Cimoli, T.: A Survey of Attacks on Ethereum Smart Contracts SoK. In: Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204. p. 164–186. Springer-Verlag, Berlin, Heidelberg (2017)
5. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., et al.: Formal verification of smart contracts: Short paper. In: Proceedings of the 2016 ACM workshop on Programming Languages and Analysis for Security. pp. 91–96 (2016)
6. Börger, E., Raschke, A.: Modeling Companion for Software Practitioners. Springer (2018). https://doi.org/10.1007/978-3-662-56641-1
7. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer Verlag (2003)
8. Braghin Chiara, Riccobene Elvinia, V.S.: State-based modeling and verification of smart contracts. Accepted to 39th ACM/SIGAPP Symposium on Applied Computing (2024)
9. Foundation, E.: Ethereum (2017)
10. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC conference on Computer and Communications Security. pp. 254–269 (2016)
11. Madl, G., Bathen, L., Flores, G., Jadav, D.: Formal Verification of Smart Contracts Using Interface Automata. In: IEEE Int. Conf. on Blockchain. pp. 556–563 (2019)
12. Mavridou, A., Laszka, A.: Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. In: Financial Cryptography and Data Security. pp. 523–540 (2018)
13. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)