Formal verification of smart contracts based on users and blockchain behaviors models

Tesnim Abdellatif[†], Kei-Leo Brousmiche[*]
[*]IRT SystemX, Paris-Saclay, France, Email: kei-leo.brousmiche@irt-systemx.fr
[†]EDF R&D, Paris-Saclay, France, Email: tesnim.abdellatif@edf.fr

***Abstract*—Blockchain technology has attracted increasing attention in recent years. One reason for this new trend is the introduction of on-chain smart contracts enabling the implementation of decentralized applications in trust-less environments. Along with its adoption, attacks exploiting smart contract vulnerabilities are inevitably growing. To counter these attacks and avoid breaches, several approaches have been explored such as documenting vulnerabilities or model checking using formal verification. However, these approaches fail to capture the blockchain and users behavior properties. In this paper, we propose a novel formal modeling approach to verify a smart contract behavior in its execution environment. We apply this formalism on a concrete smart contract example and analyze its breaches with a statistical model checking approach.**

I. Introduction

Introduced a decade ago by an anonymous contributor under the alias of Satoshi Nakamoto [10], the blockchain technology has recently gained a lot of interests from a variety of sectors such as government, finance, industry, health or research. Blockchain can be defined as a continuously growing ledger of transactions, distributed and maintained over a peer-to-peer network [15]. Based on several well-known core technologies, including cryptographic hash function, cryptographic signature or distributed consensus, it offers some key functionalities such as data persistence, anonymity, fault-tolerance, auditability, resilience, execution in a trust-less environment among others. Its applications field goes far beyond cryptocurrency, its initial purpose. For instance, blockchain can be applied to various asset management use cases (*e.g.* supply chain management, energy market) or data notarization (*e.g.* record, identity management). More recently, the introduction of smart contract has extended the functionalities of blockchains.

The term of smart contract proposed in [13] refers to a digital protocol that executes promises or terms predefined by parties who came to an agreement. Initially, smart contracts had four objectives defined as observability, verifiability (e.g by auditors or adjudicators), privity (*i.e.* limit the observability to the bare minimum members) and enforceability (*e.g.* secured transaction). Based on these propositions, some blockchains implement such functionalities through programs that are executed among the nodes. For instance, Ethereum offers a computational language named Solidity to implement smart contracts [1]. Each node participating in the blockchain has a local virtual machine and executes smart contracts, according to the transactions that have been validated and maintain a globally shared state. By inheriting the security level offered by transactions, blockchain's smart contract protocol achieves the aforementioned objectives except privity since data are duplicated in all nodes. Use cases of smart contracts are abundant and new applications are still emerging. We can mention decentralized autonomous organization (*e.g.* [12]), on-chain market places and wallets (*e.g.* [14]) or rich asset and workflow management (*e.g.* vehicle maintenance book [8]).

Due to the distributed execution environment, the complexity and the limitations of programming languages, developing smart contracts can be challenging even for experimented developers. Indeed, various attacks succeeded to make use of smart contracts' implementation breaches such as *the DAO attack* resulting in $\sim\$60M$ embezzlement [12] or the Parity wallet hacks ($\sim\$30M$ stolen in June 2017 [14] and more than $\sim\$250M$ frozen in November 2017 [11]). The vulnerabilities exploited by such attacks have been well documented by academic researches that lists, classify and explains smart contracts and EVM specific execution (*e.g.* [9], [3]). However, these approaches are not automated: a developer has to check his code according to each vulnerability that has been identified. Other approaches model the smart contract and its users using model checking techniques and game theory (*e.g.* [7], [6]). While they capture the application and users' behavior, they do not enable the detection of fraud at the blockchain level.

In this paper, we propose a new approach to model smart contract and blockchain execution protocol along with users' behaviors based on a formal model checking language. Based on these model implementation, and given their expected behavior, design vulnerabilities of the smart contracts can be analyzed using a statistical model checking tool. We illustrate our approach based on a practical and concrete example of a name registration smart contract.

This paper is organized as follows. The next section II describes an example of a simple decentralized application implemented as an Ethereum smart contract in Solidity. This contract behavior along with its users and the execution environment are then formally modeled in the section III. Based on the simulated executions of theses models, we highlight the vulnerability of the smart contract and propose alternative designs V before concluding VI.

978-1-5386-3662-6/18/$31.00 ©2018 IEEE

II. Name Register Smart Contract

We take the example of a name registry smart contract on Ethereum. The goal of this contract consists in associating a blockchain account address to a unique username. This kind of service could be used for instance to simplify currency transfer between accounts by using simple and custom usernames instead of a long and complicated address. Another way to use it would be to declare and manage web domain names.

The Solidity source code of such a contract is given below.

```
pragma solidity ^0.4.11;
contract NameReg {
    mapping (string => address) private
        nameToAddress;
    event Notify(bool _success);

    function register(string _name) public {
        if (nameToAddress[_name] != address(0)) {
            _success = false;
        } else {
            nameToAddress[_name] = tx.origin;
            _success = true;
        }
        Notify(_success);
    }
}
```

The first line specifies the version of the compiler, then the keyword *contract* declares the contract with its name similarly to a class in object oriented languages. The attribute *nameToAddress*, a hash map, will store the names/addresses links. Solidity offers to write in the "log" field of the transaction receipt (*i.e.* a data structure stored in blocks, summarizing the outcomes of a transaction) using the keyword *event*. We use this feature to notify the user whether his registration has been successful or not.

Function *register* takes a name in a string format as input and does not return any value except that the outcome of its execution will be written as a *log* as we mentioned. This function checks if the name is already assigned to an address, in which case it writes the value $false$, or stores the association between the name and the address of the caller's before writing $true$ in the *log*. The address of the calling account is retrieved using the keyword *tx.origin*: since the call of this function is done through the use of a signed transaction, the smart contract has access to the caller's address.

At a first glance, this simple code could seem robust: once a user has proposed registering a name using his account, there is no chance that his name is stolen. However, in the sections we will see that this is not true: by modeling this smart contract, users along with blockchain behaviors and simulating their executions, we demonstrate the vulnerability of this implementation.

III. Modeling and Verification Environment

We have chosen to use the BIP (Behavior Interaction Priorities) framework for its strong component-based modeling formalism and its statistical model checking engine for systems verification [4]. A formal definition for the main BIP concepts we are using are given below.

A. Atomic components

Atomic components are used to model systems elementary behaviors. They are finite-state automata, extended with variables and ports. Variables are used to store local data. Ports are action names that may be associated with variables. They are used for interaction with other components. A transition is a step, labeled by a port, from a control location to another. It has an associated Boolean condition, called a guard, and an action which is a computation defined on local variables. In BIP, data and their related computation are written in C. Formally an atomic component is defined by:

- A finite set of states $S = \{s_0, s_1, ..., s_n\}$;
- A set of ports P ;
- A set of variables V, partitioned into two sets T and U for timed and non-timed variables ;
- A set of transitions.

A transition associated to port p, from a source state s_i to a target state s_k is noted $(s_i, p, g_p, f_p, s_k)^{\tau^p}$, where g_p is the guard, that is, the condition for the transition to be triggered and f_p is the action to be executed when the transition is triggered. τ^p is the type of urgency on time guards, modeling timing constraints.

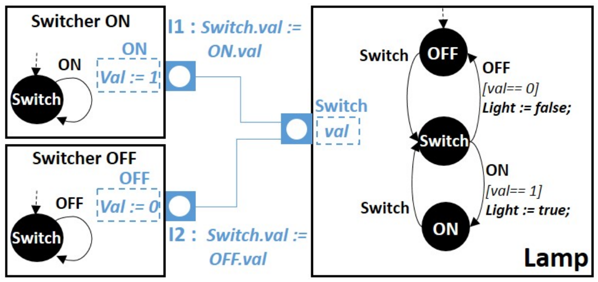

Fig. 1. Component based modeling with BIP

Figure 1 presents a system composed of three atomic components. The *lamp* component behavior consists of switching *on* and *off* the light upon the value of variable *val* given by port *Switch*. At state *OFF*, if port *Switch* is enabled, the lamp goes to state *Switch*. If guard *[val == 1]* is true, the lamp goes to state *ON* and sets variable *light* to *true*. If guard *[val == 0]* is true, it returns to state *OFF* by setting variable $light$ to false. Component *Switcher ON* have a single state *switch*. It describes the generation of a switching on command through port *On*. Port *On* is extended with variable *val*, initially set to 1. Component *Switcher OFF* behaves the same way, sending a switching off command with variable *val* set to 0.

B. Components Interactions

Composite components are defined by assembling sub-components (atomic or composite) using connectors. Connectors relate ports from different sub-components. They represent sets of interaction patterns, that are, non-empty sets of ports that have to be jointly executed. For every interaction, the connector provides the guard and the data transfer to exchange data across the ports involved in the interaction.

In the previous example, two connectors $I1$ and $I2$ enable interactions between the lamp and the switchers components. Depending on the interaction taking place, the lamp variable val takes the value of 1 or 0 from variable val in port ON or OFF.

C. SMC: the Statistical Model Checking tool

The BIP framework is currently equipped with a series of runtime verification and simulation engines [2], [5]. The SMC (Statistical Model Checking) tool is a model checking procedure to decide whether a given system B satisfies a property φ, in order to estimate the safeness of the system. Properties are logical operators described using the PB-LTL (Probabilistic Bounded Linear Time Logic) formalism. Statistical model checking refers to a series of simulation-based techniques that can be used to answer two questions:

- **Qualitative**: Is the probability for B to satisfy φ greater or equal to a certain threshold ϑ?
- **Quantitative**: What is the probability for B to satisfy φ? Note p the probability we want to estimate, SMC determines an estimation p', with a precision δ and a risk level α, so as: $\mathbb{P}(|p' - p| \leq \delta) \geq 1 - \alpha$

SMC uses SPRT (Sequential Probability Ratio Test) and SSP (Single Sampling Plan) algorithms to determine the number of simulations to reach a verdict.

IV. Smart contracts specification

We use the BIP concepts to model smart contracts implementation. We first introduce a generic modeling formalism before modeling the register smart contract example.

A. Generic smart contract specification

A smart contract can be modeled as an atomic or a compound component upon its complexity. Figure 2 is a generic representation of a smart contract specification.

The contract **interface** is defined by a set of external ports:

- Set F = $F1..Fn$ corresponds to contracts function calls. Each port is extended with data corresponding to functions inputs and the user's address.
- Set R = $R1..Rn$ corresponds to contracts return function calls. Each port is extended with a data corresponding to the return parameter.
- Set E = $E1..En$ corresponds to contracts Events.
- Set D = $D1..Dn$ corresponds to contracts data declared as public.

The Contract's **behavior** corresponds to the contract functions implementation using timed automata. Each automaton corresponds to a function call and return implementation represented by a set of transitions. Each transition execution is enabled by function calls ports or internal ports when internal computation is needed, such as the *if then else* paradigm implementation. In the next section, we give an example of the name register contract modeling.

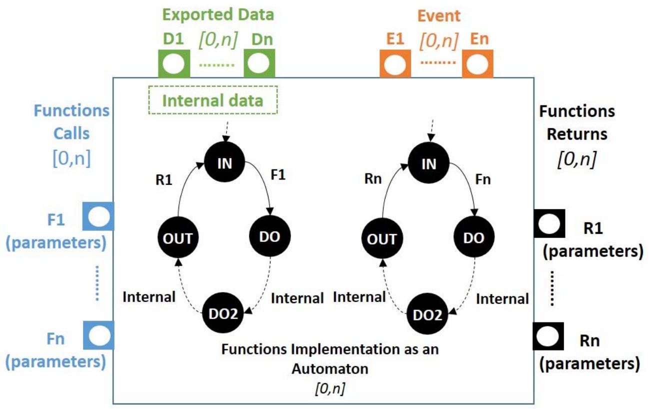

Fig. 2. Generic smart contract specification

B. The Register smart contract

Figure 3 is a modeling representation of the register contract and its external interactions given a user behavior. We model the behavior of the register smart contract given the specification in the previous section. The register contract component is defined by its:

a) Interface: : Port $Register$ and parameters (@, $alias$) of type int and string correspond to the call to the Solidity function *register*. Port $Notify$ and parameter ($success$) of type boolean model the call to the result log. Port $Return$ models the return function call to Register.

b) Behavior: is defined by an automaton: From the initial state IN, when it recieves a register call through port $Register$, it goes to state GET, checks whether the $alias$ is already registered and stores the result in a boolean $Result$. From state GET, two internal transitions are possible to reach state $SEND$. If the user address @ is not assigned to any address $[Result == 0]$ then $success$ is set to $false$. Otherwise, the address and the name are stored in the internal data $NameToaddress$ and $success$ is set to $true$. Finally, the contract executes the notification event through port $Notify$ and goes back to its initial state through port $Return$.

In order to simulate the execution of the register contract, we model a user behavior and its interactions with the smart contract through connectors. The user component models the registration mechanism with parameters ($@ : 111, alias : "User"$). It calls the register function through the $RegisterCall$ connector and takes the notification value through the $NotifyCall$ connector. The return function call is modeled by connector $ReturnCall$. We analyzed the probability for the user to register its name and it is always equal to 1. Thus, in the next section, we study advanced scenarios given external attacks behaviors.

V. Verification of the smart contract behavior

In order to build robust smart contracts and verify their safety against external attacks, it is essential to take into account the blockchain's behavior properties. In this section, we first introduce a minimal blockchain model describing

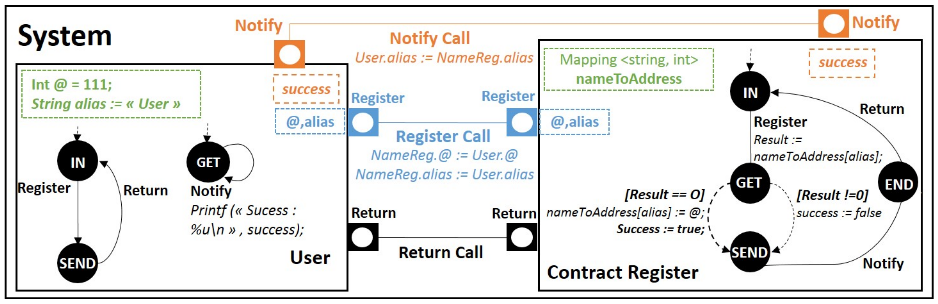

Fig. 3. User executing the register smart contract

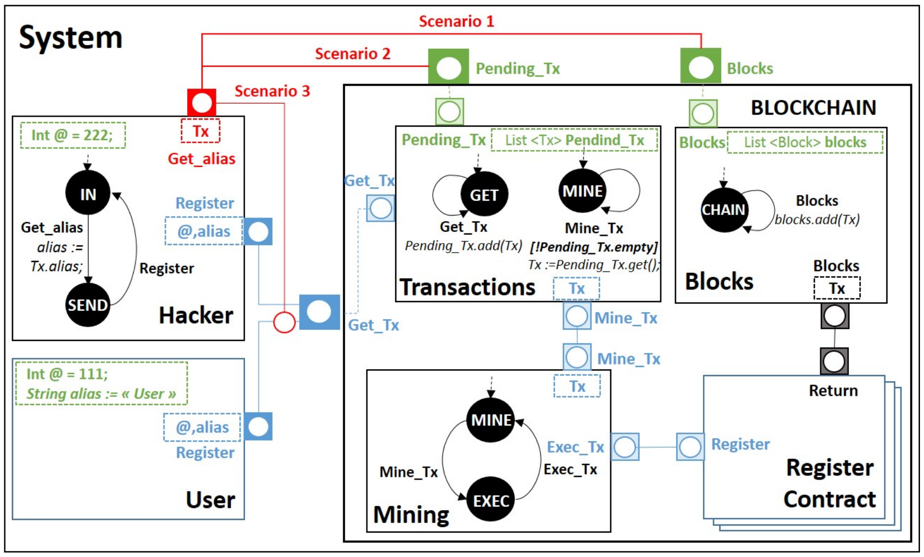

Fig. 4. Smart contract execution with blockchain and hacker behaviors

transactions execution process from the moment they are received from users to their execution results in blocks. Then we simulate and analyze the executions results of the whole system composed of the smart contract, the users and the blockchain model.

A. Modeling the Blockchain Behavior

The blockchain compound component enables external interactions through ports inherited from its internal components (see Figure 4). Port GET_Tx receives transactions from users and data ports $Pending\_Tx$ and $Blocks$ exports the pending transactions list and data blocks.

- *Component Transactions* models the reception of external transactions. It stores them in the pending transactions list until they are retrieved randomly for mining.
- *Component Mining* models the overall mining process. It mines pending transactions and sends execution commands to the corresponding smart contract. For simplification purpose, we don't take into account the number of miners and the type of consensus.
- *Component Block* encapsulates contracts execution results into blocks and exports them through the port $Blocks$.
- *Contracts components*: We consider the register contract that we have previously introduced. It executes the register function calls and sends the execution result to the Blocks component.

B. Verification Results

We analyze the safety of the register smart contract execution by introducing a hacker behavior model. The hacker purpose is to steal users identity by registering their alias with his own address. Three scenarios are possible regarding the blockchain interface.

Scenario 1: He retrieves the name after the registration of the user from the mined blocks.
Scenario 2: He retrieves the name from the pending transactions data when the user transaction is not mined yet.
Scenario 3: He gets the name from the network, that is, directly from the user call interaction with the blockchain.

We used the Statistical Model Checking (SMC) probability evaluation feature with parameters $\alpha = 0.1$ and $\delta = 0.1$. It takes as parameters the considered model and the PB-LTL property to evaluate. For each scenario, we evaluate the success for the hacker and the user to register the user alias. Figure 5 shows the SMC verification results. The PB-LTL properties are:

```
HACKER (x in [10,200]):
  P=? F[10,x](blocks.address==222 && blocks.success)
USER (x in [10,200]):
  P=? F[10,x](blocks.address==111 && blocks.success)
```

In scenario 1, the hacker can never succeed in registering the user name. The register contract rejects the registration because the alias is already registered in the *AddressToAlias* list. In scenario 2, the hacker has an average of 12% to hack the register. In scenario 3, the hacker has an average of 25% to hack the register.

We also analyzed the impact of time on the execution results. We can see that probability of success increases over time. With short time periods (*approx.* $t < 70$), registration calls don't have enough time to be executed. Time analysis can be useful for use cases where users have timing constraints (*e.g.* the need of a successful transaction within a market place turn duration).

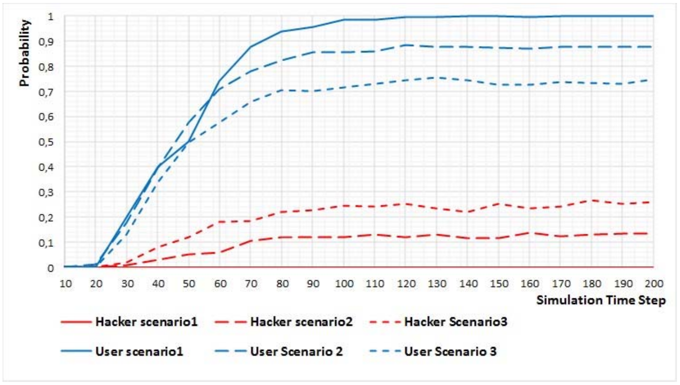

Fig. 5. Statical Model Checking Results

We analyzed the execution traces showing the success path of the simulations. In *scenario1*, the register smart contract behaves correctly and rejects the hacker tentative to register the user's name, which is already registered in the contract *AddressToAlias* map. In *scenario2*, the hacker succeeds when both the hacker and the user transactions are in the pending transaction list. Due to the random mining, the hacker transaction can be mined and executed first. In *scenario3*, the hacker intercepts the transaction while the user sends the register call transaction. Since in *scenario2*, the hacker should wait for the user's transaction to be in the pending transaction list, it explains the smaller chance for the hacker to succeed than in *scenario3*.

In order to avoid such attacks, the registration process could require two steps. First the user registers the hash of the name and then only he registers the actual corresponding name in a second transaction.

VI. Conclusions and Perspectives

We proposed a novel modeling formalism with strong semantics for smart contracts and blockchain properties. By applying this formalism on a concrete example of smart contract, we modeled its behavior and interactions with its execution environment also represented using the same approach. The simulation of the these behaviors in the BIP framework and the analysis of its results using SMC allowed to reveal scenarios where the smart contract behavior can be breached by hackers. The models presented in this paper can be extended in several ways. Our first perspective is to go further in the modeling and the analysis of blockchain properties such as the number of miners, the consensus protocol, gaz spending or timing constraints. Another perspective is the automated generation of scenarios based on the formal models to detect critical sequence of actions.

Acknowledgment

This paper has been accepted for publication in Blockchains and Smart Contracts workshop (BSC'2018). This research work has been carried out under the leadership of the Institute for Technological Research SystemX, and therefore granted with public funds within the scope of the French Program Investissements d'Avenir.

References

[1] Ethereum, February 2017. Available at: https://goo.gl/EvMgwn.
[2] T. Abdellatif, J. Combaz, and J. Sifakis. Model-based Implementation of Real-time Applications. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '10, pages 229–238, New York, NY, USA, 2010. ACM.
[3] N. Atzei, M. Bartoletti, and T. Cimoli. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *International Conference on Principles of Security and Trust*, pages 164–186. Springer, 2017.
[4] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, pages 3–12, September 2006.
[5] S. Bensalem, M. Bozga, T-H. Nguyen, and J. Sifakis. D-Finder: A Tool for Compositional Deadlock Detection and Verification. In *CAV*, volume 9, pages 614–619. Springer, 2009.
[6] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cdric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Beguelin. Formal verification of smart contracts. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security-PLAS16*, pages 91–96, 2016.
[7] Giancarlo Bigi, Andrea Bracciali, Giovanni Meacci, and Emilio Tuosto. Validation of Decentralised Smart Contracts Through Game Theory and Formal Methods. In *Programming Languages with Applications to Biology and Security*, pages 142–161. Springer, 2015.
[8] K-L Brousmiche, T. Heno, C. Poulain, E. Ben-Hamida, and A. Dalmieres. Digitizing, Securing and Sharing Vehicles Life-cycle Over a Consortium Blockchain: Lessons Learned. In *Blockchain and Smart Contracts Workshop, BSC18*, 2017.
[9] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security*, pages 79–94. Springer, 2016.
[10] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.
[11] B. Peterson. Someone deleted some code in a popular cryptocurrency wallet and as much as $280 million in ether is locked up. *Business Insider France*, 2017. Available at: https://goo.gl/uhPe2h.
[12] N. Popper. A Hacking of More Than $50 Million Dashes Hopes in the World of Virtual Currency. *The New York Times*, June 2016. Available at: https://goo.gl/7C3Zqy.
[13] N. Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.
[14] Z Wolfie. $30 Million: Ether Reported Stolen Due to Parity Wallet Breach. *CoinDesk*, July 2017. Available at: https://goo.gl/erqwNh.
[15] Z. Zheng, S. Xie, H-N. Dai, and H. Wang. Blockchain Challenges and Opportunities: A Survey. 2016.