# Model Checking of Hyperledger Fabric Smart Contracts

Elmira Ebrahimi
*Christian Doppler Laboratory for Blockchain Technologies for the Internet of Things*
*TU Hamburg*
Hamburg, Germany
elmira.ebrahimi@tuhh.de

Ehsan Khamespanah
*School of ECE*
*University of Tehran*
Tehran, Iran
ekhamespanah@ut.ac.ir

Marjan Sirjani
*School of IDT*
*Mälardalen University*
Västerås, Sweden
marjan.sirjani@mdh.se

Siamak Mohammadi
*School of ECE*
*University of Tehran*
Tehran, Iran
smohamadi@ut.ac.ir

2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA) | 979-8-3503-3991-8/23/$31.00 ©2023 IEEE | DOI: 10.1109/ETFA54631.2023.10275704

***Abstract*—Conducting interactions between shared-purpose organizations that are not entirely trustworthy of each other without centralized oversight is an idea that emerged with the advent of private blockchains such as Hyperledger Fabric and its smart contracts. It is critical to check contracts to ensure their proper functionality, as organizations may collaborate with competitors. Due to the new architecture of Hyperledger Fabric, tools in this area are limited. To formally verify the source code of contracts, we mapped Fabric contract concepts into the Rebeca modeling language. Rebeca is an actor-based language that enables the modeling of concurrent and distributed systems and is supported by a model checking tool, Afra. We have identified vulnerabilities such as deadlock and starvation by examining the desired properties. Using the model checking approach, we could debug the code and hence benefit from speeding up the transactions, creating fewer extra blocks, requiring less storage space to store the ledger, and avoiding wasting computing resources.**

***Index Terms*—Hyperledger Fabric, Smart Contracts, Model Checking.**

## I. Introduction

Blockchain [1] is a digital platform providing immutable data storage and smart contracts. Smart contracts are computer codes that automatically execute agreements between individuals once certain conditions are met. Ethereum is the most commonly used blockchain for smart contract execution [2]. From an enterprise perspective, business owners are reluctant to use it as participants are anonymous, and the access control level is not defined; therefore, private blockchains are essential.

Hyperledger Fabric [3] is a private blockchain, enabling secure interactions between shared-purpose organizations that do not fully trust one another. Hyperledger Fabric utilizes general-purpose programming languages like Go, Node.js, and Java to implement smart contracts. One downside of such languages is that they were not initially intended for direct use in developing smart contracts. As a result, some flaws in these languages are not observed in solidity [4].

Detecting vulnerabilities in smart contracts to make them safe and reliable is essential. Numerous studies have been conducted on vulnerabilities of public blockchain contracts [5]–[7], but only a few papers, like [4], [8], focused on risks related to Fabric contracts. One of the primary motivations for formal verification of Fabric's contracts is that risks in Fabric's contracts are less well-known than those in Ethereum [9], and the experience of developing a contract with Golang differs from Solidity. Static analysis tools, such as SmartCheck in Ethereum [7] and Chaincode scanner in Fabric [10], examine known risks and do not consider some run-time errors. Testing tools, such as Truffle Unit Testing [11] in Ethereum and HFContractFuzzer [12] in Fabric, examine only a subset of possible scenarios using test cases. These tools can only show the presence of bugs and cannot ensure that contracts are reliable or safe. Consequently, a thorough analysis of smart contracts, especially their formal verification, is essential for ensuring their validity. To our knowledge, only two verification tools exist for Fabric contracts [6], [13]. Still, both are in their infancy, and no product is publicly available for verifying contracts written in Golang. Therefore, providing a practical tool is one of the objectives of this study.

This research is the first to apply the model checking method to verify Golang-written Fabric smart contracts using the Rebeca language [14] and a publicly available tool, Afra. Using our method, we can check the properties of safety, liveliness, and fairness. In this study, before using formal verification method, which requires more computational resources, the code review method first checks the contract, eliminating evident vulnerabilities.

The remainder of the paper is organized as follows: Section II summarizes the technical background, including Hyperledger Fabric as the preferred blockchain for this study and formal verification. Section III discusses related work. Section IV outlines smart contract security strategies and mapping of fabric concepts to Rebeca. Section V demonstrates the feasibility of our approach by modeling four smart contracts and verifying them with the Afra tool. Section VI presents the evaluation of improvements due to debugging. Finally, Section VII presents our conclusions and further work.

## II. Background

### *A. Blockchain Technology and Smart Contract*

Blockchain orders transactions using a consensus protocol and then executes them sequentially on all peers. Sequential transaction execution limits performance and scalability. Also, transactions per second do not meet industrial needs [3]. Hyperledger Fabric is the first genuinely expandable blockchain with a modular consensus, designed to fulfill demands such as scalability, performance, and confidentiality [3]. Fabric uses the execute-order-validate architecture. Unlike previous blockchains, transactions are first executed and then placed in blocks. Smart contracts extend the blockchain platform from a simple distributed account system to a prosperous decentralized operating system [12]. A smart contract is an automated, event-driven application that executes agreements between parties on a distributed shared ledger. In Ethereum, developers use a specific domain language to develop contracts, and transactions need to pay for gas [2] to take place. Paying for a transaction is not desirable for organizations. Therefore, organizations use Fabric for enterprise usage.

### *B. Rebeca Modeling Language*

Model checking is a fully automated method for analyzing system behavior [15]. Rebeca is a language based on actors that can be used to model concurrent and distributed systems and is supported by a model checking tool, Afra [16]. System components in a Rebeca model are sets of reactive objects called rebecs, communicating with their environment through asynchronous message passing [14]. A Rebeca model consists of a set of reactive classes and an initial configuration in its main function, where a set of rebecs are created as instances of reactive classes. The reactive class's body contains declarations for the class's known rebecs, state variables, private methods, and message servers. The known rebecs are the rebecs to whom messages can be sent. Each actor's message server is responsible for responding to the message sent by the other actor. State variables contain the local state of a rebec. Each rebec has an unbounded buffer called a message queue for arriving messages. Rebeca code can be model-checked against a given set of Linear Temporal or Propositional Logic properties, defining the model's correct behavior.

## III. RELATED WORK

Smart contracts manage real assets, and cannot be patched due to their nature, so they must be developed flawlessly. Studies in this field can be divided into two categories:

### *A. Studies on Ethereum*

The first group contains static analysis tools that verify the smart contract's source code or bytecode [17]. A static analysis is an algorithm that examines a program to determine whether it satisfies a specific property. Securify [18] is an automated verification tool that examines the dependency graph of a contract symbolically. This tool had both false negatives and a large number of false positives. Zeus [6] is an automated tool with an almost zero false negative rate for determining the correctness and fairness of contracts using abstract interpretation, symbolic model checking, and Horn clauses. The eThor tool [19] is the first sound and automated static analyzer for Ethereum virtual machine (EVM) bytecode, which is based on an abstraction of the EVM bytecode semantics based on Horn clauses. Giesen et al. [5] proposed a hardening contract compiler to harden Ethereum smart contracts. The tool models a smart contract's control and data flows using a code property graph, and due to code property graph notation, it can also be applied to Fabric. The SmartCheck tool [7] is a static analysis tool that translates Solidity source code into an XML-based intermediate representation and checks it against XPath patterns to detect vulnerabilities. Although static analysis tools improve the efficiency of contract development, they cannot detect some run-time errors. Therefore, Dynamic analysis tools such as the unit testing in Truffle [11] were developed. Dynamic analysis tools analyze a program's properties while it is executing. For smart contracts, dynamic analysis is typically used to simulate attack scenarios in order to identify vulnerabilities [17]. Dynamic testing tools check parts of a contract during execution and cannot be used to prove the absence of bugs; consequently, they cannot guarantee that the contract is functioning properly. Hence, formal verification tools have gained considerable attention.

The third group converts the contract code into formal languages such as Coq, or F* [20] and then translates it into Solidity using a standard method after validation. The fourth group uses secure domain-specific languages like Flint [21] instead of Solidity, which has fewer commands. So, most of Solidity's risks do not occur in them.

### *B. Studies on Hyperledger Fabric*

Research on Fabric vulnerability diagnosis is still in the early stage due to its novelty [4], [22] and lack of practical and accessible verification tools to make contracts safer. Hyperledger Fabric uses general programming languages that were not initially created for developing contracts. As a result, incorrect use of these languages may result in inconsistencies [23]. So, developers should avoid non-deterministic chaincodes.

Chaincode Scanner [10] is the first static analysis tool in Fabric, generating the control flow and dependency graphs and then detecting errors through a database of known vulnerabilities. Yamashita et al. [4] implement a prototype static tool for detecting vulnerabilities in chaincode written in Go. Penghui Lv et al. [22] described a unique static analysis technique for performing package and function dependency analyses to provide comprehensive static characteristics of various risks. Peiru et al. [24] proposed a chaincode vulnerability detection framework and a tool that combines dynamic symbolic execution and the static abstract syntax tree analysis technology. All the above static tools check only known vulnerabilities. The HFContractFuzzer [12] was developed to create test cases through the fuzzing method and the go-fuzz tool. Unlike HFContractFuzzer, which requires manual efforts, SmartFuzzDriverGen [8] is a framework to automatically generate fuzz drivers for Fabric smart contracts using different strategies, such as sequence-based and heuristic-based.

Verifying smart contracts ensures that the contract's behavior is as expected. Two tools for verifying contracts correspond most closely to our research. Although Zeus [6] is made for Ethereum contracts, it can be adapted to Fabric. There is not yet an extension of this tool publicly available for Fabric contracts. Both the analysis and the properties must be formalized to provide reliable guarantees. While ZEUS properties are only described informally, our method formally defines the properties of smart contracts. In the VeriSmart project, Key tool [13] verifies contracts implemented in Java. One of the shortcomings of this study is that it does not consider the communication of functions with each other. The Key tool is a deductive theorem prover, so the contract verification is done semi-automatically. However, our tool is a publicly available tool that can automatically check contracts.

## IV. Mapping of Golang Smart Contracts to Rebeca Models

Verifying contracts' logic to find vulnerabilities can prevent losing confidential information. Before utilizing the model checking strategy, which requires more computational resources, the contract is first checked out using the code review, eliminating apparent flaws. Then we employ the model-checking method as the primary strategy for detecting vulnerabilities that cannot be identified using the previous method, like a deadlock. To perform model checking, as shown in Fig.1, we first mapped contract concepts to Rebeca and constructed a contract model. The desired properties are also defined. We removed the code details to avoid exploding state space while preserving the logic of the contract. The tool then analyzes whether the model satisfies the properties. If the model violates a property, the tool provides a counterexample. The modelers must determine if the bug occurs in the contract or if the model was incorrectly created. If a bug occurs, it must be reported to the developer to modify the contract code. Otherwise, the modelers must create a better model.

For modeling Fabric contracts in the Rebeca language, it is first required to map Fabric's concepts to Rebeca's. In Fabric contracts, entities that interact through contract functions can be defined in structures. We mapped this concept to reactive classes in Rebeca. The contract functions for reading data do not change the state of the database and ledger; therefore, we removed them when creating the model. We mapped other functions to message servers and private methods. In Fabric, the contract initialization is done through the init function. We do this initialization in the reactive class constructor. Rebeca has no uint variable; therefore, to check whether such variables remain positive, we examined them through assertions to prove their value is always positive.

Golang uses goroutines and channels to execute transactions concurrently. If a concurrent program is not appropriately dealt with, a race condition problem will occur, causing each node to view the ledger data in a different order. Improper channel usage can also lead to deadlock errors. Rebeca models such interactions through asynchronous message passing. We modeled channel behavior with an intermediate actor. Also, we model the events that trigger actions as messages and event handlers as message servers. We checked error handling and input sanitation through conditional statements in the model or assertions in the property file.

We use the casino game example to explain more about the mapping concept. To start the game, the casino owner tosses a coin, and the player determines the amount of money he wants to bet and his guess. If the player guesses correctly, he will receive his prize. There are three structures in the casino game contract. Listing 1 (in Golang) and Listing 2 (in Rebeca) show how the mapping of these concepts from Golang to Rebeca is done. The casino structure in the first row of Listing 1 indicates the object of the contract. The casino owner's structure (line 3) and the player's structure (line 8) of Listing 1 show entities in the contract. Reactive classes of the same name are provided for each structure, as expressed in the first, seventh, and twelfth lines of Listing 2, respectively. We have considered only essential variables for further abstraction in the contract model.

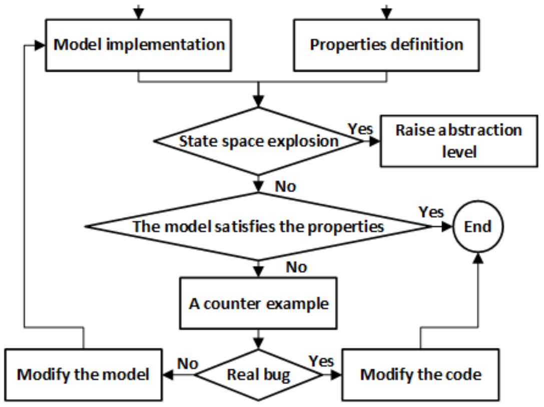

Fig. 1. Using Rebeca for Model Checking of Golang Contract Code

```
1  type casino struct {
2  }
3  type csnOwner struct {
4   ...
5   CoinResult    bool    `json:"coinResult"`
6   GameState     uint    `json:"gameState"`
7  }
8  type player struct {
9    ...
10   Balance      uint64  `json:"balance"`
11   GuessedValue bool    `json:"guessedValue"`
12   BetValue     uint64  `json:"betValue"`
13 }
```

Listing 1. Casino Game Structs in Golang.

```
1  reactiveclass Casino(5){
2    knownrebecs {
3     CasinoOwner owner;
4     Player player;
5    }
6  }
7  reactiveclass CasinoOwner(3){
8    knownrebecs{
9     Casino casino;
10   }
11 }
12 reactiveclass Player(3){
13   knownrebecs {
14    Casino casino;
15   }
16   statevars{
17    int balance;
18   }
19 }
```

Listing 2. Mapping of the Casino Game Structs to Rebeca.

For example, in the model, we only include the balance variable that deals with financial matters (line 17) in Listing 2. Listing 3 (in Golang) and Listing 4 (in Rebeca) show how the betting function is mapped to Rebeca. After receiving the player's guess, this function reduces the bet amount from the player's account and adds it to the casino account. To model this function, the `placeBet` message server in the casino reactive class and the `withdrawMoney` message server in the player reactive class are required, as shown in Listing 3. After creating the contract model, assertions and properties are defined to ensure that the model satisfies them. In a casino game, for example, the progress property is defined as `G (!(stateStart) || F(stateStop))`, checking that if the game starts, it finally ends because the previous game needs to be over to start a new one. In the following, we will describe the contracts modeled in this research. Moreover, we will discuss their vulnerabilities and show the benefits of model checking method.

```
func (t *casino)placeBet(...)peer. Response {
 ...
  if player.Balance < pBetValue {
   return shim.Error("you don't have enough money!")
  } else {
   player.BetValue = pBetValue
   player.Balance -= pBetValue
   owner.Balance += pBetValue
  }
 player.GuessedValue = pGuessedValue
 owner.GameState = stateGameBetPlaced
}
```

Listing 3. The placeBet Function in the Casino Game in Golang.

```
msgsrv placeBet(int value , boolean guess){
 if(state == GameStarted){
  betValue = value;
  state = BetPlaced;
  guessedValue = guess;
  player.withdrawMoney(value);
  balance += value;
 } else{
  player.doSomething(); }
}
msgsrv withdrawMoney(int value){
 if (balance > value ){
  balance -= value;    }
 self.doSomething();
}
```

Listing 4. Mapping of the Casino Game placeBet Function to Rebeca.

## V. OUR APPROACH: PRACTICAL RESULTS OF FABRIC CONTRACTS MODEL CHECKING

Compared to Ethereum, fewer Fabric real-world smart contracts are available in the public repositories. Consequently, we evaluated our tool and strategy using four close-to-real-world smart contracts developed by our team [25] and discovered that three, except the bank contract, include vulnerabilities, demonstrating the tool's effectiveness. Following is a brief description of the contracts and how we modeled them.

### A. Model Checking of the Bank Contract

A bank contract illustrates the characteristics of an actual bank. Listing 5 states two properties considered for the bank contract. In Rebeca, assertions must be correct in all states. The first assertion (line 8) states that the bank account's total

```
1  property {
2   define{
3    PWAUser1 = (bank.wAmount[0] < bank.withdrawLimit);
4    PWAUser2 = (bank.wAmount[1] < bank.withdrawLimit);
5    equallityofBankBalance = ( bank.balance ==
         user1.balance + user2.balance);
6   }
7   Assertion {
8    a1: bank.balance >= 0;
9   }
10  LTL{
11   safety: F (equallityofBankBalance);
12  }
13 }
```

Listing 5. Bank Contract Property File.

balance must be non-negative. Since the input validation is done correctly, a negative value cannot be sent to the contract, so this assertion is valid. The safety property (line 11) stipulates that the bank balance should equal the total of the account balances once all network transactions are completed, which is satisfied in this model. The satisfaction of all properties by the model indicates that the developer designed the contract in accordance with the intended properties.

### B. Model Checking of the Casino Contract

We used a random number to simulate the coin toss in this game. Since endorsers must receive the same random number to avoid non-determinism, the time stamp of the player's system is used for generating coin value. Listing 6 shows a set of properties considered in the game. The first assertion (line 6) states that the casino account balance must be greater than the amount given to the player if he guessed correctly. This assertion is violated after 1482 states, according to Fig.2, which is the Afra tool's counterexample output. As a result, developers must modify the contract code so that the casino balance is thoroughly checked before placing a bet, and no further bets are placed until the casino balance is added.

In a casino game, the amount withdrawn from the casino account should always be less than the casino account balance; otherwise, the casino balance may receive the maximum incorrect amount, known as a variable underflow. The second assertion in Listing 6 (line 7) examines variable underflow. The model does not satisfy this assertion; therefore, the contract code must check the casino balance using a conditional expression when executing the function and give the appropriate error message when it occurs. The balance overflow must also be prevented; otherwise, a malicious attacker can zero the casino balance by sending money to the game. Line 10 shows the progress property, which means that if a game starts, it will eventually end, and the system will not be locked because the previous game needs to be over to start a new one. This property is always satisfied by the model.

### C. Model Checking of the Local Blockchain Energy Marketplace Contract

In the local energy marketplace contract [26], consumers send the requested amount to the smart contract, using the

```
property {
 define {
  ...
 }
 Assertion {
  a1: casino.balance > (18*casino.betValue)/10;
  a2: casino.wAmount < casino.balance;
 }
 LTL {
  progress: G(!(stateStart) || F(stateStop));
 }
}
```

Listing 6. Casino Contract Property File.

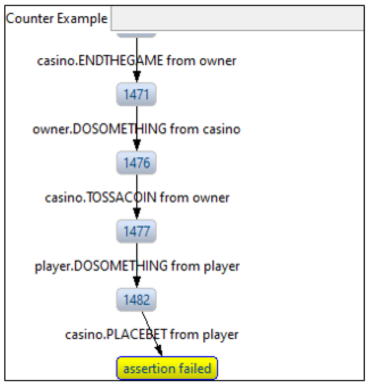
Counter Example
casino.ENDTHEGAME from owner
1471
owner.DOSOMETHING from casino
1476
casino.TOSSACOIN from owner
1477
player.DOSOMETHING from player
1482
casino.PLACEBET from player
assertion failed

Fig. 2. Afra Tool Counterexample for Casino Game Contract

allocation algorithm to fairly divide the energy among consumers. Finally, the energy token is transferred to the consumer through the physical infrastructure after transferring the energy cost to producers. The contract in this research has vulnerabilities resulting from the developer's negligence in converting the requested logic into the contract code. We developed a simpler version to model this code with two energy consumers, Bob and Tom, and a single producer, Alice. We have also set a fixed price of one coin for each energy unit. Listing 7 lists some properties for this contract. The safety property (line 14) states that the energy will be transferred to consumers before the market closes if they have made the payment. The model always meets this property. No starvation property (line 15) states that the algorithm finally allocates energy to all customers, meaning each customer must receive energy. Therefore, the modified variable, which represents the energy transfer for each customer, must be true. The implemented algorithm assigns tokens to each energy request based on Alice's production, so the model satisfies this property. The third property in (line 16) states that Alice cannot sell more energy than she produces. So, in the end, the sum of the two buyers' tokens should be less than Alice's production. The contract code model also satisfies this property.

In implementing the energy allocation algorithm, if Bob and Tom both request more than half of Alice's output, regardless of the amount requested, the algorithm must allocate energy equal to half of Alice's output to each one. In the implemented algorithm, this happens if the amount requested by Bob and Tom is equal to each other and is greater than half of Alice's output, but in a particular case where Bob and Tom request different amounts of energy, the algorithm does not work fairly and allocates more energy to the one who requested more. For example, if inputs sent to the contract are equal to 60 for Bob's requested amount and 40 for Tom's requested amount, both of which are more than half of Alice's output, which is equal to 25, instead of allocating 25 energy tokens to each, the algorithm allocates 30 tokens to Bob and 20 tokens to Tom. Hence, this violates assertion a1 (line 11).

```
1  property {
2   define {
3    consumerPaid = market.paid;
4    marketStatus = market.status;
5    modifiedAccount = market.modified;
6    equalEnergyAllocation=(market.tBuy == market.bBuy);
7    rMTHB = (market.reqTokenB > (market.tokenForSale/2));
8    rMTHT = (market.reqTokenT > (market.tokenForSale/2));
9    aliceOverSelling = ((market.tBuy + market.bBuy)
                            >  market.tokenForSale);
10  Assertion {
11   a1: ( !(rMTHB && rMTHT ) || (equalEnergyAllocation));
12  }
13  LTL {
14   safety: G(!(consumerPaid)||U(marketStatus,
         modifiedAccount));
15   noStarvation: F(modifiedAccount);
16   safety2 : F(!aliceOverSelling);
17  }
18 }
```

Listing 7. Local Energy Marketplace Property File

## D. Model Checking of the Asset Delivery Contract

There are two post offices, and each office has several postal addresses. Post office number one checks the postal address of the package received; if it is one of its addresses, the package is sent to that address; otherwise, the package is delivered to post office number two. Post office number two also checks the postal address of the package received; if it is one of its addresses, it sends the package to that address; otherwise, it delivers it to post office number one. This contract sometimes goes wrong. The bug in the logic of this contract is that if the postal address of a package is not in any of the offices, office one sends the package to office two, and office two sends it to office one, and there is a loop that causes the package never to reach the destination. Line 8 of Listing 8 indicates the property that the violation of it causes finding the bug in this contract. This feature states that, eventually, all packages must be delivered, meaning there should never be a package that is not delivered. Fig. 3 shows the tool's output. An infinite loop is seen between the two post offices in the counterexample. This loop causes the third asset never to be delivered. The primary purpose of examining the fabric contract model is to find such risks due to a weakness in the program logic that causes the contract to behave unexpectedly. Thus, identifying risks automatically using our method can gain organizations' trust, allowing them to take advantage of the platform's capabilities. In the following, we will examine the tool's efficiency and perform experiments to express the percentage of improvement after finding the contract defects and creating the correct code.

```
property {
 define{
  asset1Deliverd = asset1.deliverd;
  asset2Deliverd = asset2.deliverd;
  asset3Deliverd = asset3.deliverd;
 }
 LTL {
  noStarvation:F(asset1Deliverd) && F(asset2Deliverd)
                    && F(asset3Deliverd);
 }
}
```

Listing 8. Asset Delivery Contract Property File

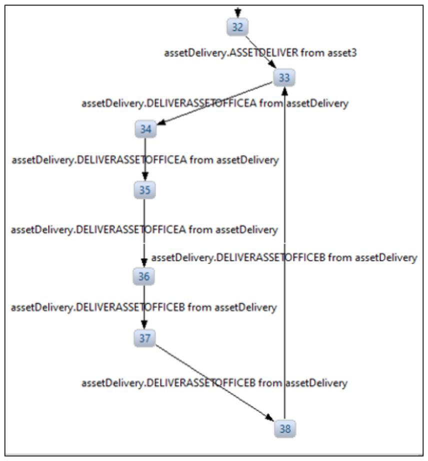
32
assetDelivery.ASSETDELIVER from asset3
33
assetDelivery.DELIVERASSETOFFICEA from assetDelivery
34
assetDelivery.DELIVERASSETOFFICEA from assetDelivery
35
assetDelivery.DELIVERASSETOFFICEA from assetDelivery
assetDelivery.DELIVERASSETOFFICEB from assetDelivery
36
assetDelivery.DELIVERASSETOFFICEB from assetDelivery
37
assetDelivery.DELIVERASSETOFFICEB from assetDelivery
38

Fig. 3. Counterexample of the Asset Delivery Contract

## VI. Evaluation of Improvements due to the Debugging

We conducted numerous experiments to demonstrate the model checking method's efficacy. Since Fabric also puts invalid transactions in blocks, a bug in the contract slows the execution of transactions, and endorsers also waste their computing resources by performing invalid transactions. In this section, the speed of transactions executed in a contract with flaws is measured relative to the correct contract after using the model checking method. Finally, we examine the increase in transaction execution speed and the rate of improvement. We examine the asset delivery contract to calculate these improvements. As mentioned, the contract has a bug that makes it impossible for the package to reach its destination if its postal address does not exist in the offices.

In Ethereum, a concept called gas [2] prevents infinite contract execution. Still, there is no such concept in the Fabric network, so contracts that are weak in implementation may run into problems. A recursive call in the asset delivery contract is an example of this. In Fabric's newer versions, contract execution has a time limit of 30 seconds by default. This means that if, after this period, the contract execution does not complete, the execution of the contract will be terminated, and Fabric will place the related transaction in the block as an invalid one. The execution timeout solves the non-termination of the transaction, but it causes each endorser to try for 30 seconds to perform a transaction stuck in the loop. A correct transaction, on the other hand, may take about 0.1 seconds; therefore, contract modeling reduces the number of such invalid transactions and increases user satisfaction. Note that the improvements obtained are due to the network configuration specific to this study, and different percentages may be obtained in other network configurations.

To experiment, we must first estimate the percentage of invalid transactions. Since there was no study in this area, we considered that in our network, 5% of transactions might be invalid. Therefore, we first send the number of 100 transactions to the contract with a bug, and for more accurate calculations, we increase this number to 200, 400, 600, 800, 1000, and 2000 transactions each time. For the percentage of invalid transactions, after every twenty valid transactions, an invalid transaction that falls into the loop is sent to the contract so that the transaction execution is closer to the real world. An invalid transaction is a transaction whose postal address does not exist in any office. Therefore, out of 100 transactions, 95 are valid, and five are invalid. We then measure the execution time of each valid and invalid transaction. To calculate the execution time of each transaction, we use the `nsec = now.UinxNano()` function, indicating the number of nanoseconds that have passed since January 1, 1970. We call this function before the execution of the transaction and after receiving the transaction's response. Then, by taking the difference between them, we obtain the execution time of each transaction. Finally, we obtain the sum of all execution times and their average in milliseconds.

For instance, the total execution time for 100 transactions is close to 4786 seconds. This high execution time is due to invalid transactions, each of which uses the endorser's computational resources for about 30 seconds. After modeling the contract, detecting the bug, and correcting it, we repeated the experiment for the same number of transactions. The total execution time for 100 valid transactions is approximately 51 seconds. We notice a significant difference between the execution times of these two experiments. This difference implies that having only 5% invalid transactions creates a heavy computational overhead for the network. As the total number of transactions increases, this overhead increases dramatically.

According to calculated averages, if the contract has a bug, each transaction request takes an average of 48 seconds, whereas, in a bug-free contract, it takes half a second. This means user satisfaction with the transaction execution is about 96 times better. Therefore, using the model checking method can significantly increase the speed of transactions. This method reduces the load on the fabric network caused by additional invalid transaction computations. In the following, we will perform these calculations and improvements. To show the improvement and efficiency of the tool, we calculate the speedup using the latency formula, which is the improvement in the execution speed of a specified number of transactions on a contract with and without bugs. According to Formula 1, for a hundred transactions, the execution speed of the transaction

TABLE I
TOTAL EXECUTION TIME IN A VALID AND INVALID STATE IN ASSETS DELIVERY CONTRACT.

| Number of transactions | Total execution time in contract without bugs (sec) | Total execution time in contract with bugs (sec) |
|---|---|---|
| 200 | 109.608 | 9611.051 |
| 400 | 210.833 | 18953.457 |
| 600 | 343.433 | 25237.918 |
| 800 | 439.552 | 51124.189 |
| 1000 | 547.439 | 60931.083 |
| 2000 | 1128.088 | 192851.356 |

has increased 94 times in total.

$$latency(ms) = \frac{Time}{workload} \qquad Speedup_{100} = \frac{4785650}{50766} = 94.2$$
$$Speedup = \frac{L_1}{L_2} = \frac{T_1}{T_2} \tag{1}$$

Performance improvements can also be measured. According to Formula 2, for a hundred transactions, after model checking and correcting the code, we have a 93% increase in performance; this means that 93 times more transactions can be done in the same amount of time before modeling. Please note that the total execution time (T) is in milliseconds.

$$\frac{T_{old} - T_{new}}{T_{new}} = \frac{4785650 - 50766}{50766} \times 100 = 93\% \tag{2}$$

It should be noted that values in this study were obtained despite an endorser node and a system with eight GB of RAM, so when an invalid transaction is performed, the system CPU becomes busy, and this invalid transaction delays the execution of the following valid transaction. There are more endorsers in real-world networks, and when one endorser executes an invalid transaction, another can respond to the valid transaction. Therefore, the execution of the valid transaction is not delayed, and side effects are reduced. If we set the consensus policy as weak or equal to the majority, all endorsers may respond to the invalid transaction. Consequently, this experiment can likewise occur in the real world.

In the same way, after calculating the total execution time up to 2000 transactions, values for other experiments are obtained. Table I shows the total transaction execution time in a valid and invalid state. According to the plot in Fig. 4, the total execution time of transactions in the contract with a bug (blue chart), compared to the bug-free contract (red chart), increased ascending as the number of transactions increased, while the execution time of each transaction in the bug-free contract is almost equal and the total execution time in it has increased slightly. Based on the results, it can be concluded that running a bug-free code is crucial. The total amount of transaction execution time creates many blocks that require more storage space. So consequently, model checking can boost transaction speed and reduce the number of such invalid transactions, thereby enhancing user satisfaction with the contract.

The red chart in Fig.5 shows how the total transaction number vs. speedup relationship changes. According to the red plot, as the total number of transactions increases, transactions execution speed in the bug-free contract will be much faster than the execution speed of transactions in the contract with bugs for the same number of transactions; this indicates that invalid transactions cause a significant delay in the execution speed of the transactions as a whole. As mentioned, the execution of an invalid transaction affects the execution time of the following valid transaction. So, to reduce this side effect and ensure our method's efficiency, we repeated the experiment when the contract was placed on a server. Thus, we simulated a server that did not return the transaction response in some cases. We designed the contract on the server so that if the postal code starts with one or two, the server adds it to the post office one or two; otherwise, the transaction is delayed for 30 seconds. In this case, the CPU is not as involved as before because it simply waits and does not perform any calculations. Fig. 5 shows a comparison of plots of the two experiments, which, as expected, using the server and reducing the side effect, the increase in total execution time is lower than before, therefore, compared to the executing contract without a server, we have less increase in the speedup.

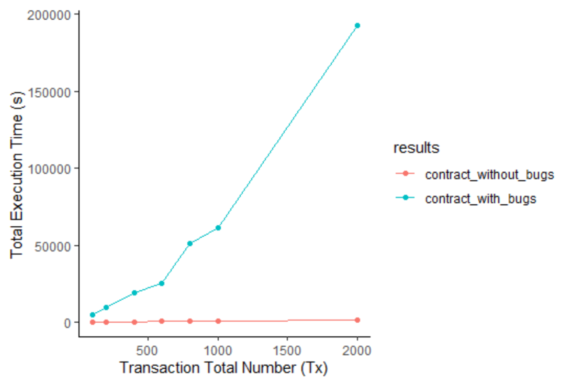

Fig. 4. Comparison of the total execution time of transactions against the number of transactions in a buggy (blue chart) and a bug-free contract (red chart).

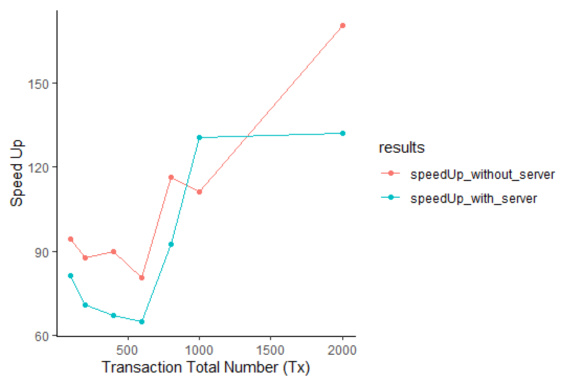

Fig. 5. The plot of speed up against total transaction in both a buggy and a bug-free asset delivery contracts by executing the contract on a server (blue chart) and without executing it on a server(red chart).

## VII. CONCLUSIONS AND FUTURE WORK

Hyperledger Fabric is one of the most popular blockchains used by enterprises. However, due to the platform's unique operating environment, it is facing a new challenge in detecting vulnerabilities. Thus, providing automated tools for verifying contracts is an essential step. Using existing strategies for reviewing the code is necessary but insufficient to ensure the validity of contracts. Therefore, this study suggested using the Rebeca modeling language and the Afra tool to verify fabric contracts. This study mapped fabric contract concepts to the Rebeca and demonstrated the method's efficiency by modeling contracts. By measuring the performance improvement, it was shown that we would have a 93% increase in performance. Significant increases in the percentage of improvement mentioned are because there are also invalid transactions in fabric blocks, unlike other blockchains. By validating the contract, the number of invalid transactions could be reduced so that endorsers respond to the transactions more quickly. It should be noted that the percentage of improvements obtained is due to the network configuration used for this research. In the real world, the number of endorsers will be higher, and the side effects of the transaction will be reduced.

The method presented in this research has performed well because, in addition to scanning the contract code through the code review method, unknown risks were also discovered through the model checking method, making it possible to deal with the automation of agreements between parties with greater confidence. In the future work of this research, we plan to improve and expand the tools for considering interactions and communications between contracts. We will compare the Afra tool with other tools after becoming publicly available. The accuracy and precision of the tool will be checked if a sufficient number of real-world and organizational contracts are reached. From the program logic, we will create a secure model in the Rebeca language and then translate it into Golang. This means we will reverse the work done in this study and generate the contract code from a secure model.

## ACKNOWLEDGMENT

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development as well as the Christian Doppler Research Association is gratefully acknowledged.

## REFERENCES

[1] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder, *Bitcoin and cryptocurrency technologies: a comprehensive introduction*. Princeton University Press, 2016.

[2] V. Buterin, "Ethereum white paper: a next generation smart contract & decentralized application platform," *First version*, vol. 53, 2014.

[3] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, and Y. Manevich, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–15.

[4] K. Yamashita, Y. Nomura, E. Zhou, B. Pi, and S. Jun, "Potential risks of hyperledger fabric smart contracts," in *2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2019, pp. 1–10.

[5] J.-R. Giesen, S. Andreina, M. Rodler, G. O. Karame, and L. Davi, "Practical mitigation of smart contract bugs," *arXiv preprint arXiv:2203.00364*, 2022.

[6] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *NDSS*, 2018, pp. 1–12.

[7] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[8] S. Pani, H. V. Nallagonda, R. K. Medicherla, and M. Rajan, "Smartfuzzdrivergen: Smart contract fuzzing automation for golang," in *Proceedings of the 16th Innovations in Software Engineering Conference*, 2023, pp. 1–11.

[9] M. Wohrer and U. Zdun, "Smart contracts: security patterns in the ethereum ecosystem and solidity," in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2018, pp. 2–8.

[10] "Chaincode scanner," accessed Apr.19, 2023. [Online]. Available: https://chaincode.chainsecurity.com/

[11] "Truffle unit testing," accessed Apr.20, 2023. [Online]. Available: https://github.com/trufflesuite/truffle

[12] M. Ding, P. Li, S. Li, and H. Zhang, *HFContractFuzzer: Fuzzing Hyperledger Fabric Smart Contracts for Vulnerability Detection*. ACM, 2021, pp. 321–328.

[13] J. Schiffl, W. Ahrendt, B. Beckert, and R. Bubel, "Formal analysis of smart contracts: applying the key system," *Deductive Software Verification: Future Perspectives: Reflections on the Occasion of 20 Years of KeY*, pp. 204–218, 2020.

[14] M. Sirjani, A. Movaghar, A. Shali, and F. S. De Boer, "Modeling and verification of reactive systems using rebeca," *Fundamenta Informaticae*, vol. 63, no. 4, pp. 385–410, 2004.

[15] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking*. MIT press, 2018.

[16] "Afra," accessed Apr.20, 2023. [Online]. Available: http://rebeca-lang.org/alltools/Afra

[17] S. Munir and W. Taha, "Pre-deployment analysis of smart contracts–a survey," *arXiv preprint arXiv:2301.06079*, 2023.

[18] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 67–82.

[19] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, "ethor: Practical and provably sound static analysis of ethereum smart contracts," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 621–640.

[20] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, and N. Swamy, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*, 2016, pp. 91–96.

[21] F. Schrans, S. Eisenbach, and S. Drossopoulou, "Writing safe smart contracts in flint," in *Conference companion of the 2nd international conference on art, science, and engineering of programming*, 2018, pp. 218–219.

[22] P. Lv, Y. Wang, Y. Wang, H. Wang, and Q. Zhou, "Potential risk detection system of hyperledger fabric smart contract based on static analysis," EasyChair, Tech. Rep. 2516-2314, 2021.

[23] S. Brotsis, N. Kolokotronis, K. Limniotis, G. Bendiab, and S. Shiaeles, "On the security and privacy of hyperledger fabric: Challenges and open issues," in *2020 IEEE World Congress on Services (SERVICES)*. IEEE, 2020, pp. 197–204.

[24] P. Li, S. Li, M. Ding, J. Yu, H. Zhang, X. Zhou, and J. Li, "A vulnerability detection framework for hyperledger fabric smart contracts based on dynamic and static analysis," in *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering 2022*, 2022, pp. 366–374.

[25] "Thesis codes," accessed May.3, 2023. [Online]. Available: https://github.com/ElmiraEbrahimi/Model-Checking-of-Hyperledger-Fabric-Smart-Contracts

[26] C. Zhang, J. Wu, Y. Zhou, M. Cheng, and C. Long, "Peer-to-peer energy trading in a microgrid," *Applied Energy*, vol. 220, pp. 1–12, 2018.