2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)

Formal Verification of Blockchain Smart Contract based on Colored Petri Net Models

Zhentian Liu
College of Computer Science
Inner Mongolia University
China, Huhhot
1602942580@qq.com

Jing Liu
College of Computer Science
Inner Mongolia University
China, Huhhot
liujing@imu.edu.cn

***Abstract*—A smart contract is a computer protocol intended to digitally facilitate and enforce the negotiation of a contract in undependable environment. However, the number of attacks using the vulnerabilities of the smart contracts is also growing in recent years. Many solutions have been proposed in order to deal with them, such as documenting vulnerabilities or setting the security strategies. Among them, the most influential progress is made by the formal verification method. In this paper, we propose a formal verification method based on Colored Petri Nets (CPN) to verify smart contracts in blockchain system. First, we develop the smart contract models with possible attacker models based on hierarchical CPN modeling, then the smart contract models are executed by step-by-step simulation to validate their functional correctness, and finally we utilize the branch timing logic ASK-CTL based model checking technology in the CPN tools to detect latent vulnerabilities in smart contracts. We demonstrate that our CPN modeling based verification method can not only detect the logical vulnerabilities of the smart contract, but also consider the impacts of users behavior to find out potential non-logical vulnerabilities in the contracts, such as the vulnerabilities caused by the limitations of the Solidity language.**

***Index Terms*—blockchain, smart contract, formal verification, CPN**

I. INTRODUCTION

In 2008, Nakamoto published a white paper on bitcoin [1], marking the birth of the blockchain technology. The blockchain can be seen as a decentralized distributed “account book” [2]. It guarantees the secure transactions in an untrusted environment. Smart contracts are digital agreements promised by contract participants [3]. They execute automatically and none participant can disobey. The blockchain provides a credible environment for smart contracts, greatly facilitating the development of smart contracts. The most common deployment platform for smart contracts is Ethereum [4].

However, due to the complexity of the distributed environment and the limitations of the programming language, there are many vulnerabilities in most smart contracts [5]. In June 2016, The DAO was attacked by hackers, causing more than 3 million ETH stolen. After that, the security vulnerability appeared in the Parity, which had resulted in more than 150,000 ETH embezzlement. Atzei et al.used the MAIAN analysis tool to perform a security analysis of nearly 1 million smart contracts [6]. Their analysis found that 34,200 smart contracts were vulnerable, which might cause huge losses. Facing with such painful losses ,how to ensure the security and reliability of smart contracts has attracted close attention. And many solutions have been proposed to protect the security of the contracts [7]–[9]. Among them, the formal methods analyze the contract based on mathematics and find the unknown vulnerabilities in the contract, which has got significant achievements [10]. So we focus on how to use formal methods to verify the smart contracts.

At present, some formal verification methods for smart contracts have been proposed. Paper [11], [12] verified the contract by means of theorem proving. Hu et al. used the Promela language to model a shopping contract and then used the SPIN model checking tool to verify the contract [13]. None of the above methods take into account the impact of user’s behaviors. Tesnim et al. combined the BIP framework to model component [14]. Their verification focuses on the user’s behaviors and the impact of the environment, which is a more comprehensive method. But their method is abstract. Therefore, to solve these problems, we propose a formal verification method of blockchain smart contracts based on the Colored Petri Nets (CPN). Firstly, we use CPN tools to model smart contracts and their attackers. With the Simulation tool, we can see every step of the contract execution. Then we combine the improved computing timing logic ASK-CTL with State Space Tools for model checking. So that we can find vulnerabilities in smart contracts when considering user’s behaviors. For a more detailed explanation, we use a simple crowdfunding smart contract (CFD) as an example. The advantage of our approach is that not only can we find logical vulnerabilities in static contracts, but we can also discover where there are vulnerabilities under the user’s malicious attack.

This paper is organized as follows. Section II introduces a concrete example to support subsequent modeling and analysis. Section III uses the CPN tools to formally model contracts and attackers. Section IV analyzes and validates the model with ASK-CTL and State Space Tools. Section V is a conclusion.

II. A CONCRETE EXAMPLE

A. *Crowdfunding smart contract*

To illustrate in more detail how to model and validate smart contracts with CPN, we introduce a simple crowdfunding

978-1-7281-2607-4/19/$31.00 ©2019 IEEE
DOI 10.1109/COMPSAC.2019.10265

smart contract (CFD). The core code is as follows.

```
contract Bank{
function () payable{}
function addToBalance() payable{
userBalances[msg.sender] =
    userbalances[msg.sender] + msg.
    value;
}
function withdrawBalance(){
uint amountToWithdraw = userBalances[
    msg.sender];
if(msg.semder.call.value(
    amountToWithdraw)() == false) {
throw;
}
userBalances[msg.sender] = 0;
}}
```

This contract is based on The DAO smart contract, which mainly implements the function of depositing money into a smart contract account (*addToBalance()*) and withdrawing money into the user's balance (*withdrawBalance()*). In contract, *function()* has no parameters, no return value, and is anonymous. Such a type of function is defined as the fallback function.

When the contract is executed, the user's withdraw operation involves the flow of the Ethereum. Due to the limitations of the Solidity language, when the contract sends the Ethereum but does not call any function, the fallback function must be executed to complete the transaction. If the contract doesn't have fallback function, the transaction is canceled. Therefore the *function()* is necessary to complete the withdraw operation.

B. Attacker smart contract

To consider the impact of users behaviors, we modify the fallback function and write the attacker contract. In this contract, We set the number of attacks to 2. The core content is as follows.

```
contract Attack{
function Attack(address addr) payable
{addressOfBank = addr;
 attackCount = 2;
}
function payable{
while(attackCount>0){
attackCount -- ;
Bank bank = Bank(addressOfBank);
bank.withdrawBalance();
}
}}
```

In section III, we use the CPN tool to model the above CFD contract and the attacker contract. Through the model checking, we can judge whether there are vulnerabilities in the contract. By this example, we say that CPN modeling is an effective method to model and verify smart contracts.

III. Modeling

A. Attribute modeling of CFD

For the above contract, we analyze the attributes that should be satisfied, and abstract the attribute constraints of the model as follows.

1) The operation selected by each user is random.
2) Each user can only select one operation at a time, deposite money or withdraw money.
3) After the user completes the operation, the crowdfunding balance, user's balance and contract account balance should be updated in real time.
4) The user's contract account balance can not be withdrawn when the balance is insufficient.
5) If the user's balance is insufficient, the money cannot be deposited into the contract account balance.
6) The sum of the user's contract account balance and the user's balance is a fixed constant (that is the user's total assets are fixed).

In the subsequent model analysis process, it is determined whether the model is correct by verifying whether the model conforms the above attributes. The relevant concepts of CPN can be referred to the literature [16].

B. Behavior modeling of CFD

We propose a two-layer CFD contract model. The top layer is the CFD layer model, which establishes the overall structure of the contract. The bottom layer is the AddTo layer and the WithDraw layer, which implement the functions of depositing and withdrawing respectively. Besides, we design the Attacker layer as an attack model, so that we can consider the impact of the environment.

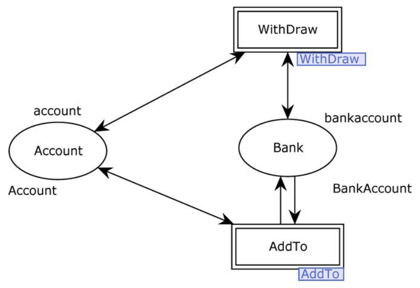


Fig. 1. CFD Layer.

a) The CFD layer: The CFD layer model is shown in Fig. 1. The place Account represents the user, the place Bank represents the contract account, and the Withdraw and AddTo are two substitution transitions, representing the user's withdrawing and depositing operations. In order to facilitate the verification of the model, we assume that there is a contract user with id=1 and the total asset of the user is 5. The crowdfunding contract currently raises 20. The initial marking is as follows.

```
val account = 1`{id=1,balance=5};
val bankaccount = 1`{bankid=1,
    bankbalance=0,sumbalance=20};
```

b) The AddTolayer: The AddTo layer is the bottom implementation of the AddTo substitution transitions, which is modeled for the user's depositing operations shown in Fig. 2. There are 3 transitions and 8 places in the model, where

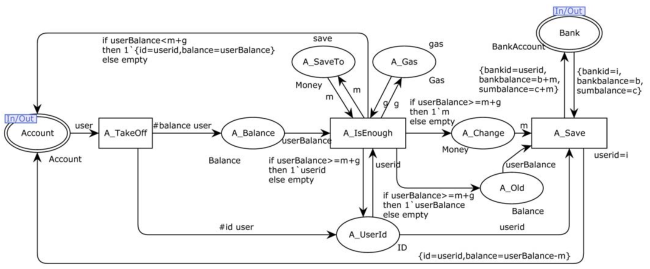


Fig. 2. AddTo Layer.

Account and Bank are the interfaces connected to the top layer. The place A_Balance is the user's balance, A_UserId is the user's id, A_Gas is the gas consumed by the contract, A_SaveTo is the money you want to deposit, A_Old is the original user's balance and A_Change is the assets that will be deposited into the contract account balance when the depositing operation is performed. We assume that the gas consumed by each operation is 1 and the amount of money you want to deposit is 4.

Based on this model, the initial marking is M0. Then we use Simulation tool to execute a depositing operation.

step 1 *M0[A_TakeOff>M1*. A_TakeOff transition fired and contract status is converted from M0 to M1. At this point, E(Account, A_TakeOff) <user > =1'{id=1, balance=5}, G(A_TakeOff)= true. That is, the input arc of the A_TakeOff transition binds a user variable with an id of 1 and the guard expression is true. After the A_TakeOff transition fired, the user's information is extracted.

step 2 *M1[A_IsEnough>M2*. A_IsEnough transition fired and contract status is converted from M1 to M2. At this point, E(A_Balance,A_IsEnough) <userBalance> =5, E(A_UserId,A_IsEnough) <userid> =1, E(A_SaveTo,A_IsEnough) <m> =4, E(A_Gas,A_IsEnough) <g> =1, G(A_IsEnough)= true. It is judged whether the user's balance is greater than or equal to the assets to be deposited into the contract account balance. If the condition is met, the next operation is performed, and if not, exiting the depositing operation. Since balance is greater than m (ie. 5>4), we can proceed to the next step.

step 3 *M2[A_Save>M3*. A_Save transition fired and contract status is converted from from M2 to M3. At this point, E(A_UserId,A_Save) <userid> =1, E(A_Old,A_Save) <userBalance> =10, E(A_Change,A_Save) <m> =4, G(A_Save)<userid=i> = true. That is, the user deposits the user's balance into the contract account balance. At this time, {id=1, balance=1}, {bankid=1, bankbalance=4, sumbalance=24}, 4 assets are deposited in the contract account balance.

In the depositing operation, the 1st and 2nd attributes are guaranteed in step 1, the 5th attribute is guaranteed in step 2, the 3rd attribute is guaranteed in step 3, the 6th attribute is satisfied in all steps and the 6th attribute is not involved. So the AddTo layer model is correct. The attributes are described in section III-A.

c) The WithDraw layer: The WithDraw layer is the bottom implementation of the WithDraw substitution transition, which is modeled for the user's withdrawing operations shown in Fig. 3. There are 4 transitions and 15 places in the model, where Account and Bank are the interfaces connected to the top layer. The place W_OBalance is the user's balance, W_UserId is the user's id, W_Gas is the gas consumed by the contract, W_Take is the money you want to withdraw, W_Balance is the original user's balance, W_Money is the assets that will be withdrawn from the contract account balance when the withdrawing operation is executed, S_Money is the total assets of the CFD contract and B_Money is the user's contract account balance. Besides, the five places connected to the output arc of the fallback transition are respectively corresponding to the five places connected to the input arc of the fallback transition. As the same, to verify the impact of the user's behavior, we assume that the gas consumed by each operation is 1 and the amount of money to withdraw is 4.

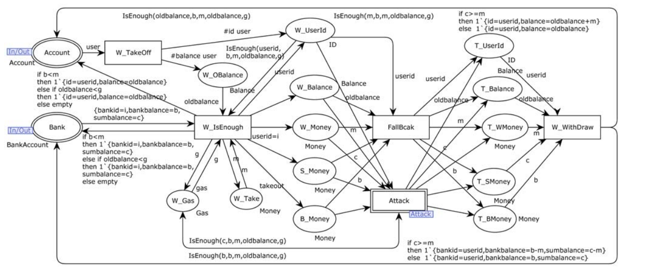


Fig. 3. WithDraw Layer.

Based on this model, the initial marking is M0. Then we use Simulation tool to execute a withdrawing operation.

step 1 *M0[A_TakeOff>M4*. A_TakeOff transition fired and contract status is converted from M0 to M4. At this point, E(Account,W_TakeOff) <user > =1'{id=1,balance=5}G(W_TakeOff)= true. The user's information is extracted.

step 2 *M4[W_IsEnough>M5*. W_IsEnough transition fired and contract status is converted from M4 to M5. At this point, E(W_OBalance,W_IsEnough) <oldbalance> =1, E(W_UserId,W_IsEnough) <userid> =1, E(W_Take,W_IsEnough) <m> =4, E(W_Gas,W_IsEnough) <g> =1, E(BankAccount,W_IsEnough) <bankAccount>=4, G(W_IsEnough)<userid=i> = true. After the W_IsEnoughh transition fired, it is judged whether the user's account balance is greater than or equal to the assets to be extracted. If the condition is met, the next operation is executed, and if not, exiting the withdrawing operation. Since bankAccount is equal to m (ie. 4=4), we can execute the next step.

step 3 *M5[FallBack>M6*. FallBack transition fired and contract status is converted from M5 to M6. At this point, E(W_UserId,FallBack) <userid> =1, E(W_Balance,FallBack) <oldbalance> =1, E(W_Money,FallBack) <m> =4, E(S_Money,FallBack) <c> =24, E(B_Money,FallBack) <b> =4, G(FallBack)= true. Here, FallBack transition only to complete the transfer transaction of the contract.

step 4 *M6[W_WithDraw>M7*. W_WithDraw transition fired and contract status is converted from M6 to M7. At this point, E(T_UserId,W_WithDraw) <userid> =1, E(T_Balance,W_WithDraw) <oldbalance> =1, E(T_WMoney,W_WithDraw) <m> =4, E(T_SMoney,W_WithDraw) <c> =14, E(T_BMoney,W_WithDraw) <b> =4, G(W_WithDraw)= true. After the W_WithDraw transition fired, the balance is updated to {id=1, balance=5}, {bankid=1, bankbalance=0, sumbalance=20}. That is, 4 assets are withdrawn from the smart account balance.

It can be seen from step 1 that M0 can reach both M4 and M1(shown in IV-C.b), and can only reach one of M1 or M4, which meet the 1st and 2nd attribute constraints. The 4th attribute is guaranteed in step 2, the 3rd attribute is guaranteed in step 4, the 6th attribute is satisfied in all steps and the 5th attribute is not involved. The attributes are described in section III-A So the WithDraw layer model is correct.

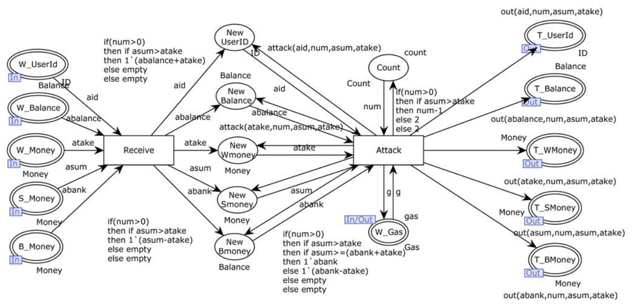


Fig. 4. Attacker Layer.

d) The attacker layer: In order to detect the impact of the user's malicious behavior on the smart contract, we introduce the attacker model to attack the contract shown in Fig. 4. This model has 2 transitions and 17 places. The double-line ellipse is the interface of the places in WithDraw layer. The place NewUserID is the user's id, NewBalance is the user's balance, NewWmoney is the assets that will be withdrawn from the contract account balance when the withdrawing operation is executed, NewSMoney is the total assets of the CFD contract, NewBMoney is the user's contract account balance and Count is the number of attacks by the attacker. For the convenience of observation we set it to 2 times.

When the user executes step 3 in the WithDraw layer, G(FallBack)=true and G(Receive)=true. The FallBack transition and the Receive transition are confusing relationship (both can be fired). If the FallBack transition fired, the execute process is as follows.

step 1 *M5[Receive>M6'*. Receive transition fired and contract status is converted from M5 to M6'. At this point, E(W_UserId,Receive) <userid> = 1, E(W_Balance,Receive) <oldbalance> = 1, E(W_Money,Receive) <m> = 4, E(S_Money,Receive) <c> = 24, E(B_Money,Receive) <b> = 4, G(Receive)=true. That is, the input arc of the Receive transition binds the variables, so that we enter the attack model.

step 2 *M6'[Attack>M7'*. Attack transition fired and contract status is converted from M6 to M7'. At this point, E(NewUserID,Attack) <aid> =1, E(NewBalance,Attack) <abalance> =1, E(NewWmoney,Attack) <atake> =4, E(NewSMoney,Attack) <asum> =24, E(NewBMoney,Attack) <abank> =4, E(Count,Attack) <num> =2, E(W_Gas,Attack) <g> =1, G(Attack)=true. Since the W_WithDraw transition of the WithDraw layer cannot be fired, the balance of the contract account always equals to the assets that you want to withdraw. Therefore, the Attack transition iteratively fired 2 times. That is, *M7'[Attack>M8'*, which realizes twice withdrawing operation. At this time, the balance is updated to {id=1, balance=9}, {bankid=1, bankbalance=4, sumbalance=16}.

step 3 When the number of attacks is 0 or the total assets of the CFD contract is less than 4, the attacker stops the attack. At this point, E(W_UserId, Receive) <userid> =empty, E(W_Balance, Receive) <oldbalance> =empty, E(W_Money, Receive) <m> =empty, E(S_Money,Receive) <c> =empty, E(B_Money,Receive) <b> =empty. Then, the contract returns to the WithDraw layer to execute normal operations. After the WithDraw transition fired, the balance is updated to {id=1, balance=13}, {bankid=1, bankbalance=0, sumbalance=12}.

In the simulation process, the 3rd and 4th attributes are guaranteed in step 2, the 1st, 2nd and 5th attributes are not involved. The attributes are described in section III-A From the last data, we find that the total assets of the user becomes 13 (balance+bankbalance=13), however the initial assets of the user are 5. Obviously the 6th attribute is not satisfied. Therefore by simulating the execution of the contract, we find that the attack behavior makes it possible for the user to illegally obtain the assets. That is, the CFD contract has vulnerabilities.

IV. Model checking

In order to find potential vulnerabilities in the contract, this section performs model checking for both non-attack and attack scenarios. Firstly, using the State Space Tools to analyze the model properties. Then, using the computing timing logic ASK-CTL and CPN ML language to analyze what counterexamples may exist, so that the model does not comply the 6th attribute. Finally, using the state space tool to draw a counterexample. Thereby finding the location of the vulnerability that may be attacked. The semantics of ASK-CTL can be found in paper [17].

A. *Model checking without attack*

We use state space tools to analyze the liveness, security and boundedness of CFD contract models, and get a complete state space report without considering the attack behavior. From the liveness report we can see that there are no dead markings and dead transitions in the model. All transitions have the possibility of being fired. So the contract is deadlock-free. From the boundedness report we can see that each place has one token at most, which satisfies the security attribute and boundedness attribute. So the model is correct. That is the logic of the CFD contract is correct.

Next, we use the improved computing timing logic ASK-CTL and CPN ML language to find a counterexample that does not satisfy the 6th attribute. The state formula is as follows.

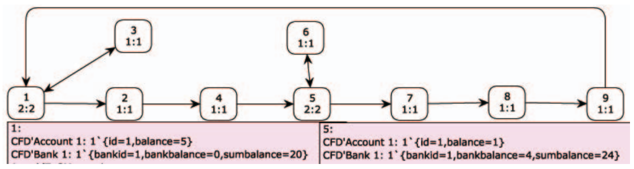
```
fun IsInitialMarking n = (n=InitNode)
    ;
val myASKCTLformula1=INV(POS(NF("
    initial marking",IsInitialMarking)
    ));
eval_node myASKCTLformula1 InitNode;
```

This formula judges whether the initial markings is Home Markings and returns a true result. Since the user can iteratively depositing, withdrawing and the assets exchanged each time is 4, it can always returns to the initial markings, which is in line with the actual situation. Then we execute the transition formula as follows.

```
fun WithDraw1 a =(Bind.WithDraw'
    W_TakeOff (1,{user={id=1,balance
    =1}})= ArcToBE a);
fun WithDraw2 a =(Bind.WithDraw'
    W_TakeOff (1,{user={id=1,balance
    =5}})= ArcToBE a);
val B1=MODAL(POS(AF("StateBE",
    WithDraw1)));
val B2 =MODAL(POS(AF("StateBE",
    WithDraw2)));
val myASKCTLformula3 =INV(AND(B1,B2))
    ;
eval_node myASKCTLformula3 InitNode;
```

This formula detects the status of the balance and returns a true result. It can be seen that when there is no attacker, the model is livessness, safety, boundedness and conforms the 6th attribute. So there is no counterexample.

Using the state space tool to generate the state space shown in fig. 5. The initial user's balance is 5, the user's contract account balance is 0, and the totle assets of the CFD contract is 20. After a depositing operation, it reaches state 5. The user's balance is 1, the user's contract account balance is 4, the totle assets of the CFD contract is 24. So the depositing operation is correct. Finally, after a withdrawing operation, the state returns to state 1. So there are no logical vulnerabilities in the CFD smart contract under the non-attack scenario.



1 2:2
2 1:1
3 1:1
4 1:1
5 2:2
6 1:1
7 1:1
8 1:1
9 1:1
1:
CFD'Account 1: 1`{id=1,balance=5}
CFD'Bank 1: 1`{bankid=1,bankbalance=0,sumbalance=20}
5:
CFD'Account 1: 1`{id=1,balance=1}
CFD'Bank 1: 1`{bankid=1,bankbalance=4,sumbalance=24}

Fig. 5. State space of CFD without attack.

B. *Model checking under attack*

When an attacker exists, using the state space tool to analyze the liveness, security, and boundedness of the model. It is found that the model also satisfies these attributes, that is, the model with the attacker is still correct.

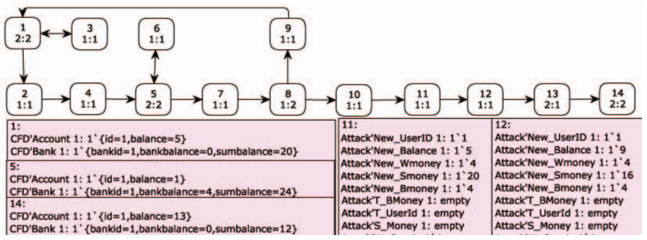
Next, we use the computing timing logic ASK-CTL and CPN ML language to find a counterexample that does not satisfy the 6th attribute. Similarly, using the above state formula to judge whether the initial markings is Home Markings, the return result is false. That is, the contract model does not always return to the state of id=1, balance=5. Then using the above transition formula to judge the status of the balance, and the return result is still false. Therefore, we find that the normal execution of the contract is affected when an attacker attacks. In order to find the specific location of the vulnerability, continue to execute the following formula.

```
fun IsOverWithDraw n a =(Bind.
    WithDraw'W_TakeOff (1,{user={id=1,
    balance=n}})= ArcToBE a);
val myASKCTLformula4 =MODAL(POS(AF("
    Executed",IsOverWithDraw n)));
eval_node myASKCTLformula4 InitNode;
```

From the results, we find that the balance has 5 abnormal values of 9, 13, 17, 21, 25. Selecting a abnormal value randomly (e.g. 13) and executing the following formula to find the abnormal status.

```
fun error n=let val X=(st_Mark.CFD'
    Account 1 n) in X="CFD'Account 1:
    1`{id=1,balance=13}"end
val errornode=PredAllNodes(fn n =>
    error n);
```

By returning the result, we find that the abnormal status with a balance value of 13 has M14, M55, M106. Since M14 produces a balance=13 error for the first time, we use the state space tool to draw a counterexample that produces the M14 shown in Fig. 6.



```
1:
CFD'Account 1: 1`{id=1,balance=5}
CFD'Bank 1: 1`{bankid=1,bankbalance=0,sumbalance=20}
5:
CFD'Account 1: 1`{id=1,balance=1}
CFD'Bank 1: 1`{bankid=1,bankbalance=4,sumbalance=24}
14:
CFD'Account 1: 1`{id=1,balance=13}
CFD'Bank 1: 1`{bankid=1,bankbalance=0,sumbalance=12}
```

```
11:
Attack'New_UserID 1: 1`1
Attack'New_Balance 1: 1`5
Attack'New_Wmoney 1: 1`4
Attack'New_Smoney 1: 1`20
Attack'New_Bmoney 1: 1`4
Attack'T_BMoney 1: empty
Attack'T_UserId 1: empty
Attack'S_Money 1: empty
```

```
12:
Attack'New_UserID 1: 1`1
Attack'New_Balance 1: 1`9
Attack'New_Wmoney 1: 1`4
Attack'New_Smoney 1: 1`16
Attack'New_Bmoney 1: 1`4
Attack'T_BMoney 1: empty
Attack'T_UserId 1: empty
Attack'S_Money 1: empty
```

Fig. 6. A counterexample under attack.

As can be seen, the initial user's balance is 5, the user's contract account balance is 0, and the totle assets of the CFD contract is 20. After a depositing operation, it reaches state 5. The user's balance is 1, the user's contract account balance is 4, the totle assets of the CFD contract is 24. So the depositing operation is still correct.

Then continue to execute the withdrawing operation. When the status 14 is reached, the user's balance is 13, the user's contract account balance becomes 0, and the total assets of the CFD contract is 12. Therefore, the user uses the contract vulnerability to execute the illegal withdrawing operation, and obtains 8 assets that are not their own. While the CFD contract lost 8 assets.

To find how it occurred, we observe the state between state 5 and state 14. Through the state 11, we find that the user's balance is 5, so 4 assets are withdrawn from the contract account balance. But the user's contract account balance is still 4, and the total assets of the CFD contract is 20. Finally, to state 12, the user's balance is 9, 4 assets are illegal withdrawn from the contract account balance again. But the user's contract account balance is still 4. According to the detected counterexample, corresponding to the code position, we find that because the smart contract invokes the fallback function when trying to transfer the Ethereum, and the attacker rewrites the internal code of the function, which iteratively called the *WithDraw()* function twice, thus resulting three times withdrawing operations. So, it can be seen that our method can find the location of the vulnerability in the contract.

Through the above model checking method, it can be found that even if there is no logical vulnerability in the smart contract, the absolute security of the contract can not be guaranteed. We should consider more about the impact of malicious behaviors on contracts, find potential vulnerabilities in smart contracts, and improve the security of contracts.

V. Conclusion

Formal verification of blockchain smart contracts is an important way to improve the reliability and security of the contract. We analyze the necessity of introducing formal methods into smart contract verification, and propose a new CPN-based smart contract verification method. To illustrate the feasibility of this method, we model a simple crowdfunding smart contract and its attacker, using ASK-CTL and state space tools for model checking, so that we can find the location of the vulnerability in the contract. This method not only analyzes the static logical structure of the contract, but also simulates the dynamic interaction of the user's malicious behavior, which can more comprehensively verify whether the contract has a vulnerability. In addition, CPN has streaming characteristics. By simulation tool, we can see the status of each step of the contract execution, making it easier to discover potential vulnerabilities in smart contracts and reducing unnecessary losses.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of Chain (No. 61662051).

References

[1] Nakamoto S. Bitcoin: A peer-to-peer electronic cash system[J]. Consulted, 2008.
[2] Dejan Vujičić,Dijana Jagodić,Siniša Randić, et al. Blockchain Technology, Bitcoin, and Ethereum: A Brief Overview.17th International Symposium INFOTEH-JAHORINA, 21-23 March,2018
[3] Nikolic I, Kolluri A, Sergey I, et al. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale[J]. 2018.
[4] Dejan Vujicic,Dijana Jagodic,Sinisa Randic, et al. Blockchain Technology, Bitcoin, and Ethereum: A Brief Overview.17th International Symposium INFOTEH-JAHORINA, 21-23 March,2018.
[5] Li X, Jiang P, Chen T, et al. A Survey on the security of blockchain systems[J]. Future Generation Computer Systems, 2017.
[6] Atzei N, Bartoletti M, Cimoli T. A Survey of Attacks on Ethereum Smart Contracts (SoK)[C]// International Conference on Principles of Security and Trust. Springer, Berlin, Heidelberg, 2017:164-186.
[7] Watanabe H, Fujimura S, Nakadaira A, et al. Blockchain contract: Securing a blockchain applied to smart contracts[C]// IEEE International Conference on Consumer Electronics. IEEE, 2016:467-468.
[8] Matsuo S. How formal analysis and verification add security to blockchain-based systems[C]// Formal Methods in Computer Aided Design. IEEE, 2017.
[9] Magazzeni D, Mcburney P, Nash W. Validation and Verification of Smart Contracts: A Research Agenda[J]. Computer, 2017, 50(9):50-57.
[10] Destefanis G, Marchesi M, Ortu M, et al. Smart contracts vulnerabilities: a call for blockchain software engineering?[C]// International Workshop on Blockchain Oriented Software Engineering. IEEE, 2018.
[11] Bhargavan K, Swamy N, Zanella-Bguelin S, et al. Formal Verification of Smart Contracts: Short Paper[C]// ACM Workshop. ACM, 2016:91-96.
[12] Amani S, Begel M, Bortin M, et al. Towards verifying ethereum smart contract bytecode in Isabelle/HOL[C]// The, ACM Sigplan International Conference. ACM, 2018:66-77.
[13] Xiaomin BAI, Zijing CHENG, Zhangbo DUAN, Kai HU. Formal Modeling and Verification of Smart Contracts[J]. 2018:322-326.
[14] Tesnim Abdellatif, Kei-Leo Brousmiche. Formal verification of smart contracts based on users and blockchain behaviors models. IFIP NTMS International Workshop on Blockchains and Smart Contracts (BSC), 2018.
[15] Jensen K, Kristensen L M, Coloured Petri nets: Modeling and validation of concurrent system[J]. Berlin: Springer, 2009:95-188.
[16] Jensen K, Kristensen L M. Coloured Petri Nets - Modelling and Validation of Concurrent Systems.[M]. Springer, 2009.
[17] Cheng A, Mortensen K H. Model Checking Coloured Petri Nets Exploiting Strongly Connected Components[J]. Proceedings of the International Workshop on Discrete Event Systems Wodes96 Institution of Electrical Engineers Computing & Control Division, 1997, 26(519).