Article

A Dynamic Behavior Verification Method for Composite Smart Contracts Based on Model Checking

Jun Jin [1], Wenhao Zhan [1], Haisheng Li [2], Yi Ding [1] and Jie Li [1,3,4,*]

[1] School of Information, Beijing Wuzi University, Beijing 101149, China
[2] Beijing Key Laboratory of Big Data Technology for Food Safety, Beijing Technology and Business University, Beijing 100048, China
[3] School of Computer Science and Engineering, Beihang University, Beijing 100191, China
[4] Yunnan Key Laboratory of Blockchain Application Technology, Kunming 650233, China
* Correspondence: lijiebwu@163.com

Abstract: A composite smart contract can execute smart contracts that may belong to other owners or companies through external calls, bringing more security challenges to blockchain applications. Traditional static verification methods are inadequate for analyzing the dynamic execution of these contracts, resulting in misjudgment and omission issues. Therefore, this paper proposes a model checking approach based on dynamic behavior that verifies the security and business logic of composite smart contracts. Utilizing automata, the method models contracts, users, attackers, and extracts properties, focusing on six types of common security vulnerabilities. A thorough case study and experimental evaluation demonstrate the method's efficiency in identifying vulnerabilities and ensuring alignment with business requirements. The UPPAAL tool is employed for comprehensive verification, proving its effectiveness in enhancing smart contract security.

Keywords: smart contracts; model checking; solidity; UPPAAL; formal methods; security vulnerabilities

MSC: 03B70

Citation: Jin, J.; Zhan, W.; Li, H.; Ding, Y.; Li, J. A Dynamic Behavior Verification Method for Composite Smart Contracts Based on Model Checking. *Mathematics* **2024**, *12*, 2431. https://doi.org/10.3390/math12152431

Academic Editors: Vincenzo Vespri and Antanas Cenys

Received: 16 June 2024
Revised: 19 July 2024
Accepted: 1 August 2024
Published: 5 August 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Blockchain is a distributed ledger shared between peer-to-peer network nodes that follows a certain consensus protocol [1], allowing transactions to be processed without the need for a trusted third party. Therefore, business activities can be completed in an efficient manner. In addition, the immutability of blockchain also ensures distributed trust, and any transaction stored in the blockchain cannot be tampered with. All historical transactions are auditable and traceable [2]. The initial popularity of blockchain technology is attributed to Bitcoin [3], which uses decentralized, untrusted systems to record cryptocurrency transactions on distributed ledgers. Subsequently, Ethereum [4] ushered in the era of blockchain 2.0, expanding the functionality and applications of blockchain by introducing smart contracts. In blockchain application scenarios, smart contracts refer to computer programs that run in a blockchain environment and can be automatically executed. Most smart contracts are written in Turing complete languages such as Solidity [5], Go, Java, etc.

With the rapid development of blockchain smart contracts, their application in the financial field is becoming increasingly widespread, and attacks against smart contracts are also increasing. Some attacks triggered by contract vulnerabilities have caused huge losses [6–9]. For example, the famous Decentralized Autonomous Organization (DAO) attack is due to a significant flaw in smart contracts, which separated over 3 million Ethers from the DAO resource pool, resulting in an economic loss of USD 60 million [10]. Obviously, the security and correctness of smart contracts are very crucial. Formal verification methods use mathematical models and reasoning to make them more rigorous and reliable [11]. Symbolic execution [12], Theorem proving [13], model checking tools [14], and

some other approaches [15,16] are proposed, but they mainly consider the verification of one smart contract. Complex business logic needs the collaborative invocation among several smart contracts. Thus, in the context of composite smart contracts, the other calling and called smart contracts are usually unknown, meaning the composite smart contracts brings more complex security issues. The authors of [17] only verify the correctness of whether the business logic meets the users' requirements. Some works [18] are focused on the verification of security issues. However, the approach is based on the static analysis. For composite smart contracts where other external contracts are called during contract execution, the static analysis fails to reflect the execution process of the vulnerability and cannot be fully applicable.

Therefore, this paper adopts a model checking method to simulate the dynamic execution process of composite contracts, for the verification of security-vulnerable properties and business logic properties of composite smart contracts. UPPAAL model checker [19] offers an intuitive, graphically integrated environment for modeling, verifying, and simulating real-time systems. It use the more rigorous and efficient computation tree logic (CTL) to specify the properties of composite smart contracts. Thus, our method uses UPPAAL for modeling and verification. The contribution of this paper is as follows:

(1) We take six types of security vulnerabilities in blockchain smart contracts as examples, including reentrancy, access control, privileged functions exposure, cross-contract invocation, denial of service, and miner privilege. A modeling approach based on the execution semantics of Solidity code is proposed. This method not only models composite contracts but also constructs models for users and potential attackers.

(2) A method for the dynamic behavior analysis and verification of composite contracts is proposed. We simulate and analyze the dynamic calling process of the contracts that triggers the aforementioned security vulnerabilities, identify the location of the vulnerability code, and derive specific security vulnerability and business logic verification properties.

(3) We use UPPAAL to model a financial composite smart contract case, verify the security and correctness of smart contracts with security and business logic properties, and evaluate the verification time. The two results demonstrate the effectiveness of our method.

The second section of this paper introduces the relevant work in the field of formal verification of smart contracts. Section 3 proposes the automaton modeling process for Solidity smart contracts and the formal specification of six types of security vulnerability properties. Section 4 takes smart contract financial services as a case study for modeling, defines security vulnerability properties and business logic properties, verifies the model, and conducts result analysis. Finally, a summary and outlook are presented in Section 5.

2. Related Work

Since the 2016 DAO incident that led to the Ethereum hard fork, formal verification research of smart contracts has gained increasing popularity. Formal verification technology has practical significance in ensuring the correctness and security of smart contracts. The verification of smart contracts initially focused solely on correctness. In [15,16], the authors used the model checker Spin to verify correctness, to understand whether smart contracts comply with the specifications for given behaviors and some properties like state accessibility, no deadlock, and no livelock. In [20], the authors proposed a method for correctness verification of composite smart contracts, but it does not address security vulnerabilities. In [17], the authors proposed a graphical notation to specify the interaction between contracts, as well as a framework for verifying, generating, and deploying multiple smart contracts. However, this paper focused on the contract interaction and deployment, so it did not specify security properties from the perspective of normal typical vulnerabilities. Almakhour et al. [18] used FSM to model composite smart contracts to verify both correctness and security properties, but did not consider the external user interactions or invocations by other smart contracts. Nam et al. [21] utilized the ATL to

analyze smart contracts on the blockchain, representing the interactions between users and smart contracts as a two-player game, and employed model checker for multi-agent systems (MCMAS) to verify relevant properties. Chen et al. [22] focused on arithmetic bugs detection with higher precision and recall, such as integer over/under flows and division-by-zeros. These works did not consider correctness verification.

There is also research on automated verification methods and tools for smart contracts, capable of validating contracts from various perspectives. For instance, Oyente [12], as a pioneer in smart contract verification, employed a variety of static analysis techniques for contract validation. Tools like DefectChecker [23] utilized symbolic execution to accomplish automated verification of contracts, while SPCon [24] primarily focused on identifying vulnerabilities in access control, and Wang et al. [25] predominantly detected flaws present in token contracts.

Smart contracts also have a temporal nature, meaning there are some time constraint issues that are influenced by time. For instance, the miner's privilege vulnerability occurs because miners can alter timestamps, thereby gaining indirect control over the execution of smart contracts. The aforementioned studies and tools are incapable of handling security verification with temporal constraints. Zhao et al. [26] proposed the use of timed automata for modeling contracts and then employing the UPPAAL tool to verify temporal properties.

Nevertheless, most of the verification approaches conduct static analysis of contracts, which cannot reflect the actual execution process of composite contracts, and few works have focused on verification with time constraints. Therefore, this paper proposes a dynamic method based on contract execution logic to verify the security and correctness of composite smart contracts. This method, based on the automaton model and model checking techniques, models and verifies composite smart contracts, respectively, and is capable of discovering six types of significant security vulnerabilities within composite smart contracts, as well as correctness issues that do not conform to the users' business logic requirements. Finally, CTL formula is used to represent all properties and all properties are verified using the UPPAAL checker. This method is validated using a set of different Solidity smart contracts.

3. Composite Contract Model and Verification Method

Model checking is one of the most commonly used methods in the field of formal verification, typically divided into three stages: system modeling, specification, and verification. During the system modeling phase, a finite state machine (FSM) pattern, which is recommended for developing smart contracts, is suitable for modeling the behavior of smart contracts [27]. FSMs are a specific type of automaton. The concept of an automaton is broader and not limited to a finite set of states. In the specification phase, modal logics such as linear temporal logic (LTL), CTL, and alternating-time temporal logic (ATL) are often employed to describe various requirements of the system, including its applications and safety. The verification phase is used to ascertain whether the system model conforms to the properties that the system should possess. If it does not conform, the system is considered to have corresponding issues that require modification, re-modeling, and re-verification.

The method proposed in this paper for the dynamic behavior of composite smart contracts is based on automata and model checking. The entire process is divided into three stages: ① automata modeling, ② property formalization, and ③ property verification. It includes four main components, namely, Solidity source code, automaton model, verification properties, and model checking tool UPPAAL. The framework of our research method is shown in Figure 1.

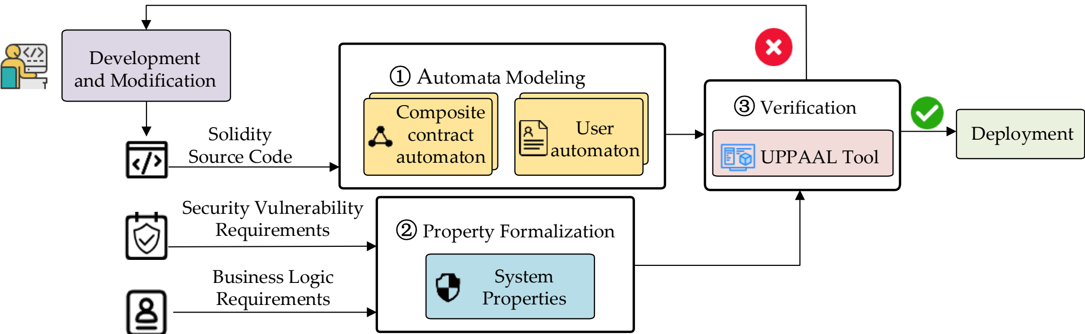


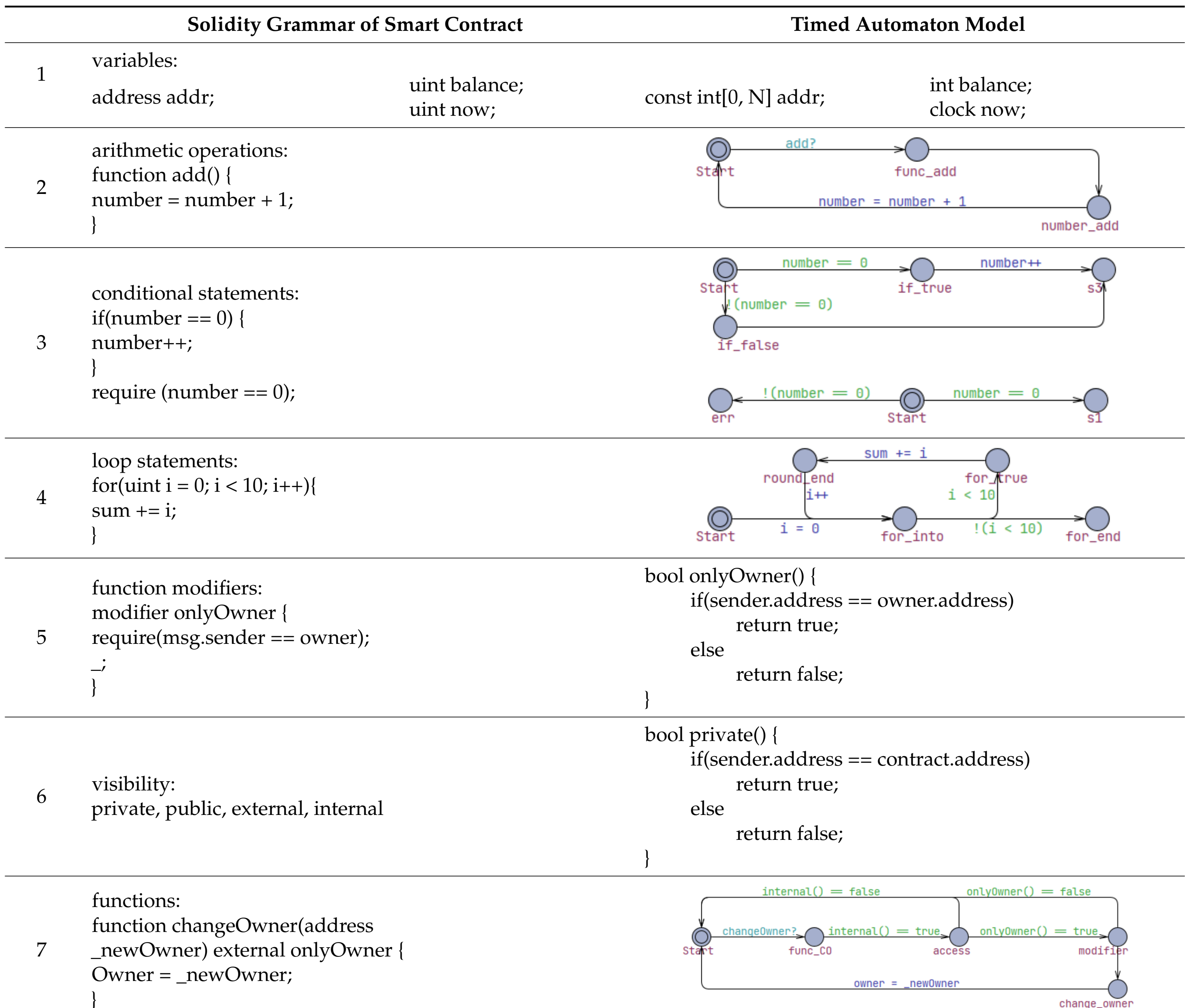
Figure 1. Outline of our verification framework.

In order to describe the dynamic behavior of the contract execution process, this paper models each Solidity contract as a time automaton, and in addition to modeling a corresponding number of timed automata, composite smart contracts also need to model other contracts that interact with them. For example, contracts and users can call other contracts, and the dynamic behavior of their business logic or execution process is closely related to the behavior of the called contract. Therefore, these individuals also need to be modeled as automata. After modeling, the purpose of verifying the composite contract can be achieved by verifying whether the automaton satisfies certain conditions, referred to as properties in the following text. The correct operation of composite contracts depends on accurate and errorfree business logic, which must meet business needs without security issues. This paper establishes a time automaton model for the contract and the initiator based on the dynamic behavior of contract execution, based on some common security vulnerabilities of smart contracts. Then, a set of business logic properties and security vulnerability properties are defined according to the application requirements and key vulnerabilities of the contract, and CTL is used to represent all the business and security properties that need to be satisfied. Finally, by verifying whether the model meets the above properties, we can verify whether the composite contract meets business requirements and whether there are any security vulnerabilities. Because the system properties to be described and validated include time, the UPPAAL model checking tool is used to implement modeling and define and validate all properties.

3.1. Smart Contract Automata Model

An automaton is a mathematical model that represents a set of states, along with the transitions and actions among them. If represented graphically, an automaton's states are depicted as nodes, with the transitions between states represented as edges. State transitions are triggered when certain conditions are met or specific operations are performed; hence, each edge is characterized by three elements: a condition (guard), a variable update operation (update), and/or a synchronization operation (sync). The synchronization operation is used for communication between automata sent from one automaton (sending message is denoted by "!") to another automaton (receiving message is denoted by "?"). When the conditions on the edge (as shown by the if statements, function modifier, visibility, and loop statements in Table 1) are met and the required signal has arrived, or a variable update operation has occurred, the transition from one state to another will occur.

Table 1. Smart contract automaton modeling rules.



	Solidity Grammar of Smart Contract		Timed Automaton Model	
1	variables: address addr;	uint balance; uint now;	const int[0, N] addr;	int balance; clock now;

2	arithmetic operations:
	function add() {
	number = number + 1;
	}
			add?
			Start
			func_add
			number = number + 1
			number_add	

3	conditional statements:
	if(number == 0) {
	number++;
	}
	require (number == 0);
			number == 0
			number++
			Start
			if_true
			s3
			!(number == 0)
			if_false
			err
			!(number == 0)
			Start
			number == 0
			s1	

4	loop statements:
	for(uint i = 0; i < 10; i++){
	sum += i;
	}
			sum += i
			round_end
			for_true
			i++
			i < 10
			Start
			i = 0
			for_into
			!(i < 10)
			for_end	

5	function modifiers:
	modifier onlyOwner {
	require(msg.sender == owner);
	_;
	}
			bool onlyOwner() {
			if(sender.address == owner.address)
			return true;
			else
			return false;
			}	

6	visibility:
	private, public, external, internal
			bool private() {
			if(sender.address == contract.address)
			return true;
			else
			return false;
			}	

7	functions:
	function changeOwner(address _newOwner) external onlyOwner {
	Owner = _newOwner;
	}
			internal() == false
			onlyOwner() == false
			changeOwner?
			internal() == true
			onlyOwner() == true
			Start
			func_CO
			access
			modifier
			owner = _newOwner
			change_owner	

To model a smart contract, refer to the modeling rules shown in Table 1; row 1 to row 5 are the basic contract statements. Row 1: About variables. Integer variables are used in the model to record common types. To represent the timestamp of the current block, one can use the uniquely self-incrementing variable of clock type that is specific to UPPAAL. Row 2: In arithmetic operation, each assignment operation is described in the model as a transition activity between states. Row 3: Model *if()* and *require()* in the conditional statements. If the condition is met, the state will transfer to state *if_true*, and the statement *num*++ will be executed normally. Otherwise, the state will transfer to *if_false*, and the statement block in the curly braces will be skipped to execute the next statement. If the required condition is met, it will transfer to state *s1* and continue to execute subsequent codes normally. Otherwise, it will transfer to abnormal state *err*. Row 4: In the loop statement *for()* structure, if the loop condition is met, it will move to the state *for_true*. Until the condition is not met, the state will transfer to the state *for_end* to complete the loop execution. Row 5: The function modifier is modeled as a function in C language form according to the specific judgment logic. Before executing the business logic code, judge and verify whether the

content of the modifier meets the requirements. For example, only the owner can continue to execute. Row 6: Similar to row 5, the four types of visibility domains (*private*, *public*, *external*, *internal*), as predefined modifiers, are also modeled as specific functions, in which the corresponding judgment can be made on whether the execution can continue. Row 7: A single contract function is modeled as an automaton, starting from the state *Start* by default. When the called contract function receives a message with the same name as itself, such as *changeOwner?* (Usually, this message is sent from another contract function, such as *changeOwner!*), the state will transfer to another state with its function name, such as *changeOwner_C0*, which means that the call occurred. Next, judge the function's visibility domain and function modifiers, such as the external and onlyOwner of the function. If these judgments are true, the function body will execute, the corresponding state transfers to *modifier*. The execution owner is reassigned, the state transfers to *change_owner*. After executing *Owner = _newOwner*, the state returns to *Start*.

Some typical library functions of Solidity are predefined as templates during modeling. For example, *call.value()* means that when *call.value()* is initiated, the receiver's *fallback()* function is automatically called for withdraw operations.

3.2. Composite Smart Contract Formal Properties and Verification

Formal verification of properties is responsible for providing contract properties to validate the security and correctness of a single or composite smart contract. In order to cover the complete range of security issues in composite smart contracts, two types of properties are defined: security properties and business logic properties. The former models 6 types of smart contract security vulnerabilities as examples (reentrancy attack, access control, privileged function exposure, cross-contract invocation, denial of service, miner privilege), then extracts implementation logic and dynamic interactions between different contracts from each security vulnerability model, defines CTL properties, and finally detects the existence of vulnerabilities by verifying whether the model satisfies the CTL properties. As for business logic properties, users define them based on the business logic context of composite smart contracts, similarly describing them as CTL properties and verifying them. The following section will introduce automaton modeling and verification of each vulnerability with CTL properties using the six vulnerabilities as examples.

3.2.1. Reentrancy

Smart contracts can call the code of other external contracts and send tokens to external addresses, but when performing these operations, the system automatically triggers the function *fallback* of the recipient. Therefore, this external call may be exploited by malicious attackers to trigger recursive calls between contracts by the "re-entry" of the function *fallback*, resulting in unexpected actions, such as unauthorized transfers.

In the code block 1, the bank contract's *balances[msg.sender]* records the balance of each account. Normally, callers can recharge their own account using the *recharge()* function and withdraw their entire balance using the *withdraw()* function. The attacker contract launches an attack on the bank contract through the *attack()* function. The *bank.recharge()* function first recharges a certain amount specified by *msg.value*, and then calls the *bank.withdraw()* function, with the *call.value()* function transferring funds to the attacker contract. When this function is called, the system automatically invokes the fallback function of the attacker contract (i.e., the unnamed function of the receiver), leading to a recursive call. If *attackCount* < 5, which it can be, the *bank.withdraw()* function is continuously called, transferring funds to the receiver, and triggering the fallback function. This process repeats until condition *attackCount < 5* is no longer satisfied, then the attacker contract receives 5 times the recharge amount, i.e., *5* msg.value*.

Code block 1. Solidity reentrancy vulnerability sample code

```
1   contract Bank {
2       ...
3       function recharge() payable public {
4           balances[msg.sender] += msg.value;
5       }
6       function withdraw() public {
7           require(msg.sender.call.value(balances[msg.sender])());
8           balances[msg.sender] = 0;
9       }
10  }
11
12  contract Attacker {
13      ...
14      function attack() payable {
15          attackCount = 0;
16          Bank bank = Bank(bankAddr);
17          bank.recharge.value(msg.value)();
18          bank.withdraw();
19      }
20      function () payable {
21          if(msg.sender == bank.Addr && attackCount < 5) {
22              attackCount += 1;
23              Bank bank = Bank(bankAddr);
24              bank.withdraw();
25          }
26      }
27  }
```

To verify the execution process of the above composite contract, the automata of attacker user contract, bank contract, and attacker contract established by UPPAAL are shown in Figure 2a–c. The execution of the reentrancy attack automaton is initiated by the attacker user sending the message *attack!*, starting from the initial state *Start* in Figure 2a. Figure 3 presents the simulation execution sequence. When the bank attacker contract receives *attack?* from the state *Start*, it initializes the *attackCount = 0* and the *bankAddr = 0*. Subsequently, it assigns a value to the recharge amount *msg_value = 10* and its own contract address to *msg_sender*, then sends a *recharge!* call to the bank contract's *recharge()* function.

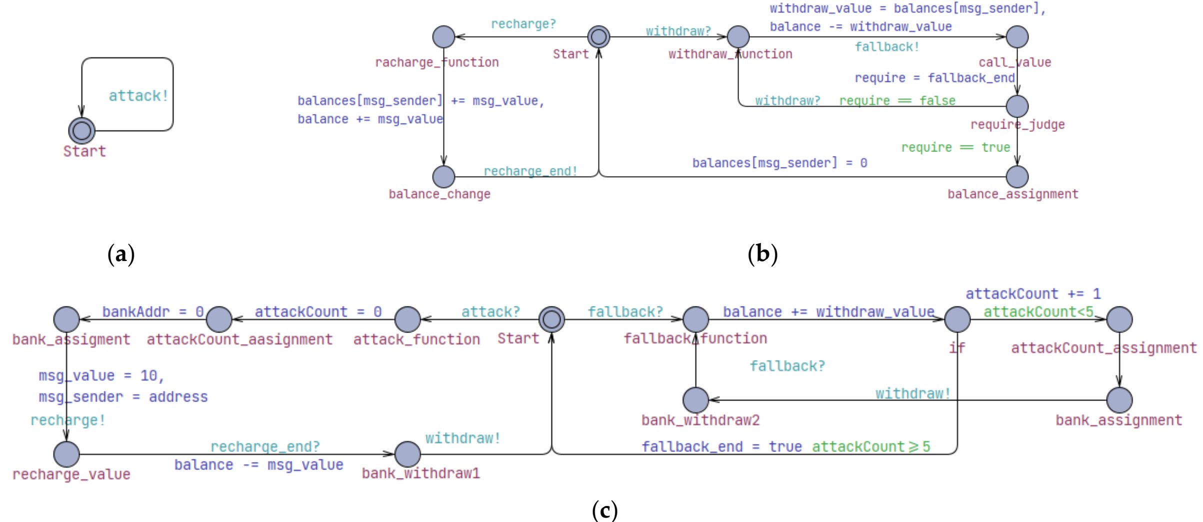


Figure 2. Reentrancy vulnerability model. (**a**) attacker user model; (**b**) bank contract model; (**c**) bank attacker contract model.

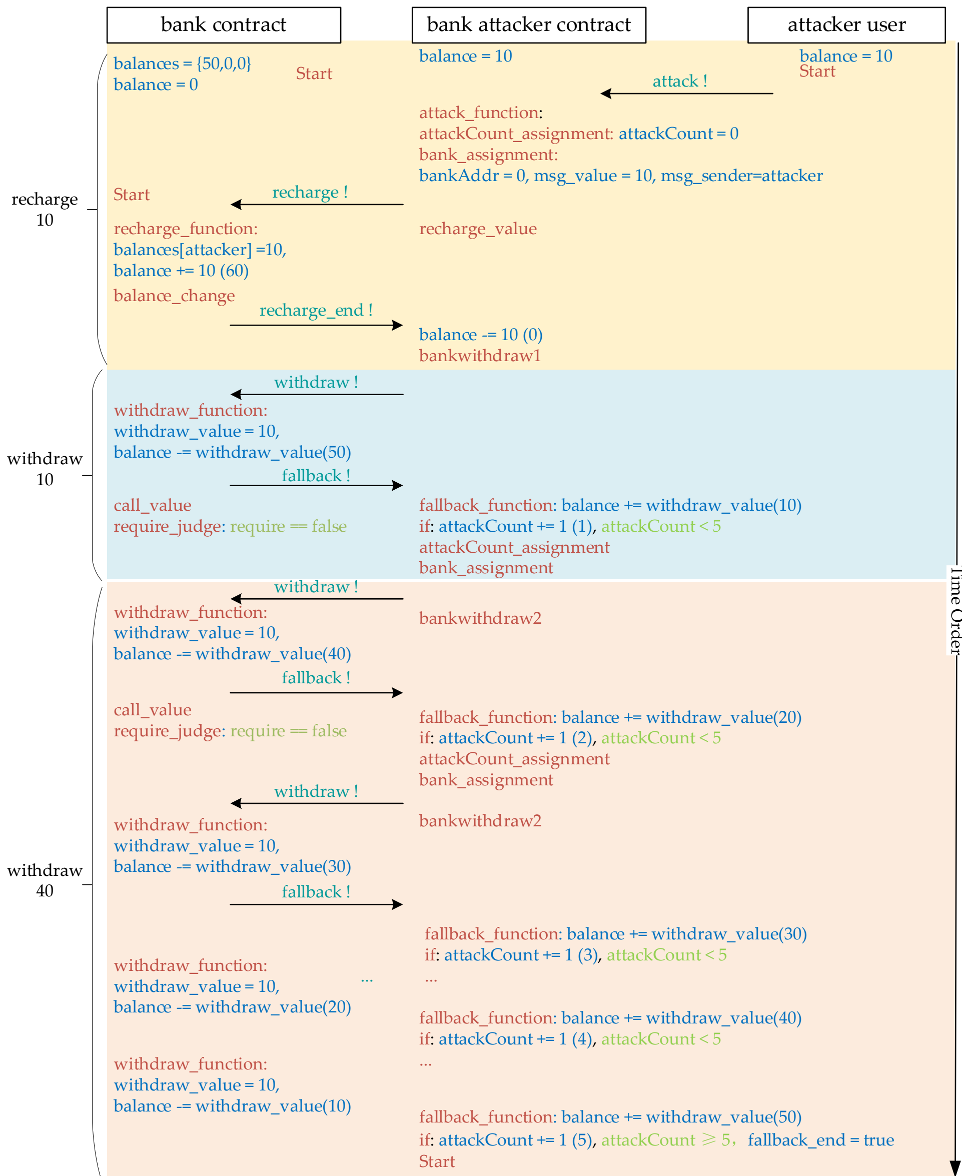


Figure 3. Reentrancy vulnerability simulation execution sequence.

After receiving the *recharge?* in the state *Start*, the bank contract recharges *msg_value* to both its own account and the attacker's account, and then sends the *recharge_end!* message to the attacker contract. After receiving the *recharge_end?* message, the attacker contract subtracts *msg_value* from its own balance, and then sends a *withdraw!* message to the bank contract.

After receiving the *draw?* message, the bank contract assigns message sender's account balance to variable *withdraw_value*, and subtracts *withdraw_value* from its own balance, because function *call.value()* invokes the anonymous function of attacker contract by default (row 7), so here we use the message *fallback!* sending to the attacker contract in this automaton. Then it uses *require* to determine whether the attacker contract has completed function *fallback* executions, i.e., variable *fallback_end* is true. If *fallback_end* is false, the bank contract continues to wait for the message *withdraw?* again. If *fallback_end* is true, it sets message sender's *balances [msg_sender]* to 0 and returns to its initial state *Start*.

After receiving the message *fallback?*, the attacker contract adds the *withdraw_value* to its own balance. It then checks the value of *attackCount*, and if *attackCount* < 5, it increments *attackCount* by 1. Subsequently, it calls the function *withdraw()* of the bank contract again by sending the message *withdraw?*. To differentiate the second and subsequent abnormal calls

to the function *withdraw()* from the first one, the names of the states are *bank_withdraw2* and *bank_withdraw1*, respectively. If the condition *attackCount >= 5* is true, the contract sets *fallback_end = true* to indicate that the function *fallback()* has completed execution, and then returns to the initial state *Start*.

CTL properties:

$$\text{E<> attacker.bank_withdraw2 \&\& bank.withdraw_function} \quad (1)$$

The CTL property (1) is derived based on the execution process of the automata. The property is used to verify whether there is a scenario where the bank contract has received a call message for the function *withdraw*, i.e., it is in the state *withdraw_function* and the attacker contract calls the function *withdraw()* again, i.e., it is in the state *bank_withdraw2*. If such a scenario exists, it indicates that the function *fallback()* in the attacker contract can call the bank contract's function *withdraw()* again, enabling repeated transfer operations, which signifies the presence of a reentrancy vulnerability.

3.2.2. Access Control

Access control vulnerabilities typically arise from imprecise or erroneous definitions of certain conditions during the writing of smart contracts, such as function modifiers. Attackers can exploit these vulnerabilities to maliciously execute functions that they should not have permission to access, causing losses to the entire contract system.

Code block 2. Solidity access control vulnerability sample code

```
1   contract Wallet {
2       bool tokenTransfer;
3       address walletAddress;
4       mapping(address => uint256) _balances;
5
6     modifier isTokenTransfer {
7           if(!tokenTransfer){
8               revert();
9           }
10          _;
11   }
12
13     modifier onlyFromWallet {
14          require(msg.sender != walletAddress);
15          _;
16     }
17
18     function transfer(address to, uint value) public isTokenTransferreturns(bool success) {
19          require(_balances[msg.sender] >= value);
20          _balances[msg.sender] = _balances[msg.sender] − value;
21          _balances[to] = _balances[to] + value;
22          Transfer(msg.sender, to, value);
23          return true;
24     }
25
26     function enableTokenTransfer()external onlyFromWallet {
27          tokenTransfer = true;
28     }
29
30     function disableTokenTransfer() external onlyFromWallet {
31          tokenTransfer = false;
32     }
33  }
```

Code block 2 demonstrates the sample code of access control, the intention of the modifier *onlyFromWallet* at line 13 of the sample code of access control is to use "==" to determine whether the call originated from the wallet itself, but mistakenly use "!=" at line 14. As a result, after being decorated with *onlyFromWallet*, the functions *enableTokenTransfer()* at line 26 and *disableTokenTransfer()* at line 30 can only be executed by accounts other than *walletAddress*. Consequently, an attacker can modify the variable *tokenTransfer* to true by *enableTokenTransfer()* and then uncontrollably invoke the function *transfer()* at line 18 to carry out transfer operations.

The automaton model shown in the Figure 4 represents the owner user, attacker user, and wallet contract. Figure 5 illustrates the execution sequence of access control. Both the owner and attacker automata perform the same actions, where, upon initiation, they set the *msg_sender* to their own address and send the messages *enableTokenTransfer!* or *disableTokenTransfer!*. The wallet contract automaton, upon receiving these messages, checks if the function *OnlyFromWallet()* returns true, and then proceeds to respectively set *tokenTransfer* to false and true before returning to the state *Start*.

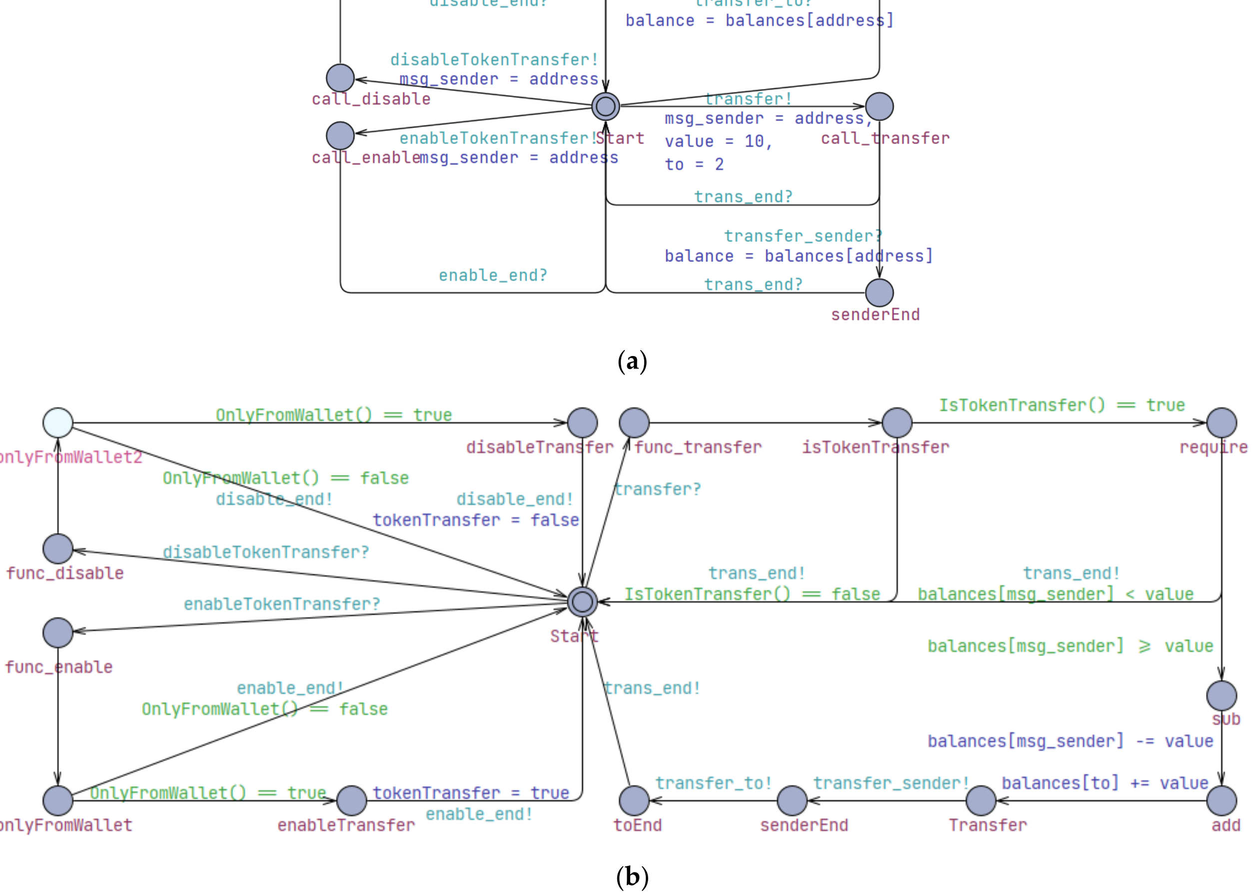


Figure 4. Access control vulnerability model. (**a**) owner and attacker user contract model; (**b**) wallet contract model.

Due to the mistake at line 14 of code, the attacker user is able to freely control the opening and closing of the wallet because the modifier *onlyFromWallet()* returns true. In contrast, the owner user is restricted to return to the initial state from *OnlyFromWallet* and *onlyFromWallet2*, as the modifier *OnlyFromWallet()* returns false for them. Similarly, the owner user is unable to manually open the wallet switch through the function *enableToken-Transfer()*.

If the attacker user closes the wallet by setting *tokenTransfer* to false, then when the owner user attempts to send a transfer, the wallet contract will execute the action specified

by the modifier *isTokenTransfer()*, which returns false. Consequently, the owner user automaton will always return to the state *Start* without successfully invoking the function *transfer()* in the wallet contract to complete the transfer.

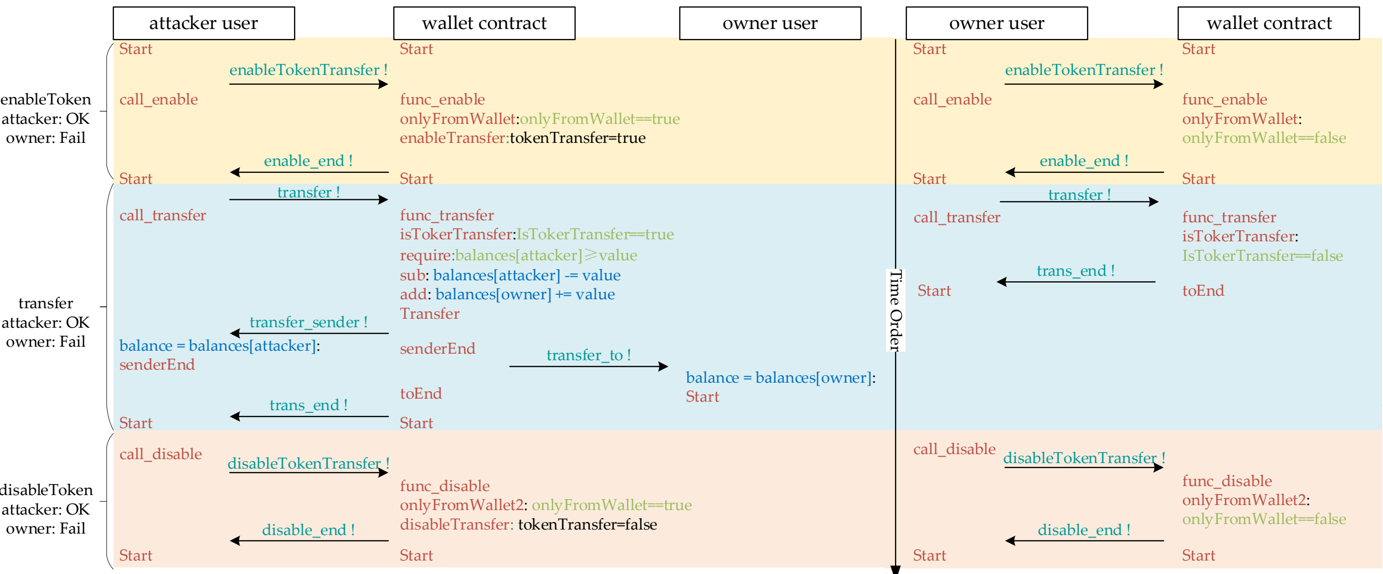


Figure 5. Access control vulnerability simulation execution sequence.

CTL properties:

$$
\begin{aligned}
&\text{(a1) E<> wallet.disableTransfer \&\& attacker.call_disable;}\\
&\text{(a2) E<> wallet.enableTransfer \&\& attacker.call_enable;}\\
&\text{(b1) E<> wallet.disableTransfer \&\& owner.call_disable;}\\
&\text{(b2) E<> wallet.enableTransfer \&\& owner.call_enable.}
\end{aligned} \tag{2}
$$

We derived CTL properties in Formula (2), where (a1) and (a2), respectively, indicate that the attacker user reaches the state *call_disable* or *call_enable*, and the wallet contract also reaches the state *disableTransfer* or *enableTransfer* accordingly. It implies that the attacker user can control the switch of the wallet contract. In addition, (b1) and (b2) signify that when the owner user reaches the state *call_disable* or *call_enable*, the wallet contract can perform the *disableTransfer* or *enableTransfer* operations. The owner user cannot alter the switch of the wallet contract, so when (a1) or (a2) satisfies, and (b1) or (b2) is not satisfied, it can be confirmed that there exists an access control vulnerability in the contract.

3.2.3. Privilege Function Exposure

Functions with no permission modifiers in Solidity can be invoked by anyone by default. The *selfdestruct()* is a built-in function in Solidity that can destroy the current contract and send the balance of the contract to a specified address. Therefore, if the function *selfdestruct()* in a contract does not have any permission modifiers, anyone can call this function to gain the privilege of destroying the contract. The code block 3 illustrates the scenario described above. At line 3 of the wallet contract, the function *destroyContract()* lacks any permission modifiers, allowing an attacker to freely call the function *selfdestruct(_to)* to destroy the current contract. Automatically, the contract, which is about to be terminated, will transfer all its balance to the designated address *_to*.

Code block 3. Solidity privilege function exposure vulnerability sample code

```
1 contract Wallet {
2     . . .
3     function destroyContract(address _to) {
4         selfdestruct(_to);
5     }
6 }
```

The models in Figure 6 illustrate the owner and attacker users interacting with the wallet contract in the context of exploiting a privileged function exposure. Both types of users have the capability to invoke the function *func_destroy* by sending a message *destroy!* to the wallet contract. Figure 7 represents the simulation execution sequence. Since the destructor function *destroyContract(address _to)* is not protected by any modifiers, upon receiving the request message *destroy?*, the wallet contract model can directly execute a transfer operation to the user via the function *selfdestruct()*. The user receives the request *trans_to?*, which increments their account balance, and then sends a message *destroy_end?* to the wallet contract to return to the state *Start*. Upon receiving the *destroy_end!* request, the wallet contract resets its account balance to 0 and transitions to the state *Destroyed*, rendering the contract inactive.

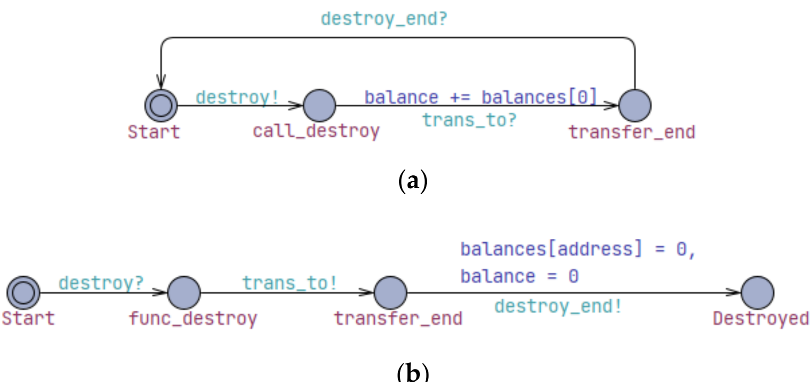


Figure 6. Privilege function exposure model. (**a**) Common models for both attacker and owner users; (**b**) wallet contract model.

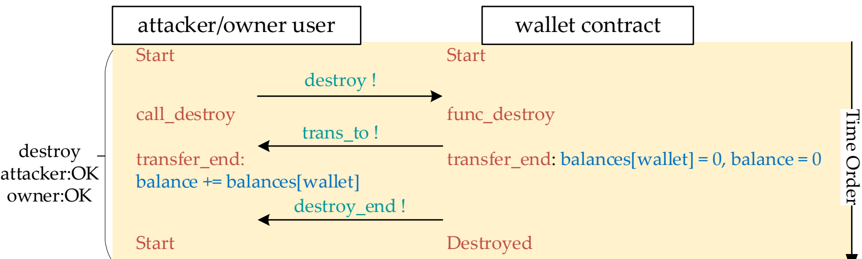


Figure 7. Privilege function exposure vulnerability simulation execution sequence.

CTL properties:

$$\begin{aligned}&\text{(a) E<> wallet.transfer_end \&\& attacker.transfer_end;}\\&\text{(b) E<> wallet.transfer_end \&\& owner.transfer_end.}\end{aligned} \quad (3)$$

From the execution process of the automata, we can obtain CTL properties in Formula (3). It is indicated that when the wallet contract reaches the state *transfer_end*, the attacker or owner users are also able to reach the state *transfer_end*. If this situation exists, it means that attacker and owner users, meaning any user, can perform the destruct operation of

the wallet contract. When both (a) and (b) are true, it means the presence of a privileged function exposure vulnerability.

3.2.4. Cross-Contract Invocation

Solidity provides three functions: *call()*, *delegatecall()*, and *callcode()* to facilitate interaction and invocation between contracts. Among these, the function *call()* poses a security risk if not handled properly. Attackers can impersonate the current contract and invoke internal functions of itself or other contracts, leading to cross-contract vulnerability.

In the code block 4, at line 7, the function *authorityTransfer()* enables transfer operations, requiring the caller to be the current contract. The function *callFunc()* in the CallBug contract, at line 3, allows anyone and any contracts to invoke the internal function *call()*. If its parameter *data* is constructed as *authorityTransfer()*, then the require statement *this* == *msg.sender* cannot work, which allows anyone or any contract to execute the subsequent operations after the require statement.

Code block 4. Solidity cross-contract invocation vulnerability sample code

```
1   contract CallBug {
2       ...
3       function callFunc(bytes data) public {
4           this.call(data);
5       }
6
7       function authorityTransfer(uint256 _amount) {
8           require(this == msg.sender);
9           //secret operations...
10      }
11  }
```

Figure 8 shows the automata of the attacker user and the CallBug contract and Figure 9 represents its simulation execution sequence. Normally the attacker sends the message *authorityTransfer!* to the CallBug contract, after receiving it the CallBug calls the function *authorityTransfer()* to perform the *require()* statement. Since *msg_sender != CallBug.address*, the CallBug contract sends the message *authorityTransfer_end!* to the attacker and returns to the state *Start*. When the attacker receives the message *authorityTransfer_end?*, it also returns to the state *Start*. Therefore, normally the require statement acts as a guard to prevent unauthorized cross-contract calls.

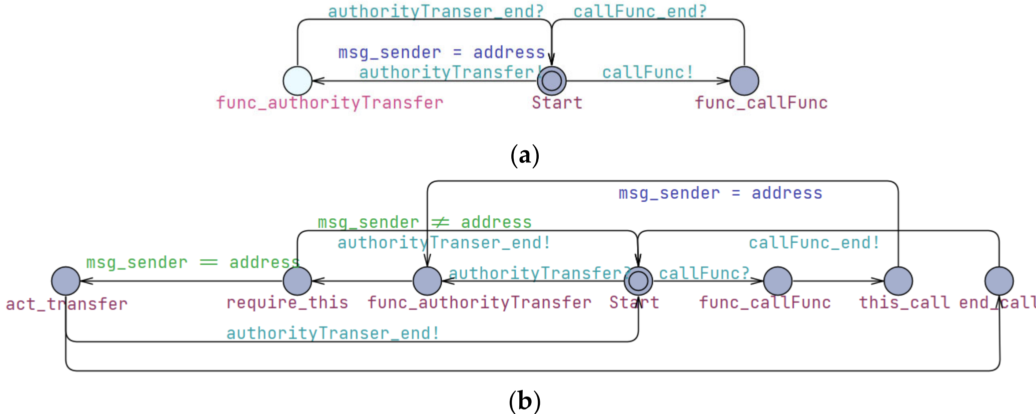


Figure 8. Cross-contract invocation vulnerability model. (**a**) attacker user model; (**b**) CallBug contract model.

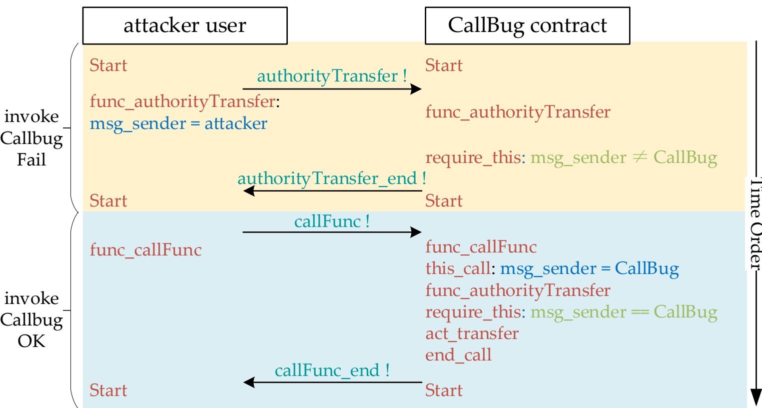


Figure 9. Cross-contract invocation vulnerability simulation execution sequence.

However, if the attacker sends the message *callFunc!* to the CallBug contract, and after receiving it, contract CallBug then calls the function *callFunc()* and reaches the state *func_callFunc*. Since the function *authorityTransfer()* is called through the internal function *callFunc()*, the value of the variable *msg_sender* is *CallBug.address*, thus the require statement *this* == *msg.sender* in function *authorityTransfer()* is true, allowing the subsequent transfer operations to be executed. Previously, the attacker is unable to invoke the function *authorityTransfer()*, but now by first calling function *callFunc()*, it is able to call the function *authorityTransfer()* to execute transfer operations.

CTL properties:

(a) E<> attacker.func_authorityTransfer && CallBug.act_transfer;
(b) E<> attacker.func_callFunc && CallBug.act_transfer. (4)

We derive Formula (4), which indicates whether the CallBug contract reaches the state *act_transfer* when an attacker user accesses *func_authorityTransfer* or *func_callFunc*. If conditions (a) and (b) are not satisfied, it means that attacker users cannot directly or indirectly invoke the function *authorityTransfer()* for transfer operations, indicating the absence of a cross-contract calling vulnerability in the contract CallBug. If condition (a) or (b) is met, it implies that attacker users can cause the CallBug contract to reach the state *act_transfer* by invoking the function *authorityTransfer()* or *callFunc()*, enabling direct or indirect transfers. In such a case, it indicates the presence of a cross-contract calling vulnerability in this contract.

3.2.5. Denial of Service

In smart contracts, attackers can exploit contract resources to prevent other users from executing normal operations within a certain period by monopolizing available resources. This can lock funds in the attacked contract. In a transfer application, users can create a contract that does not accept tokens. If another contract needs to send tokens to this contract address, and then it is able to transit to a new state, the contract will never reach the new state because the transfer operation cannot be completed.

For example, in a transfer application, attacker users can create a contract, such as contract B, that does not accept tokens. If another contract, such as A, needs to send tokens to the address of contract B before entering a new state, as contract B does not accept tokens, the transfer operation of contract A cannot be completed, and contract A will never reach a new state.

The auction contract shown in code block 5 is used for bidding, and the function *bid()* at line 5 is responsible for updating the latest bid situation. Firstly, it checks if the current bid amount *msg.value* is greater than the highest bid in history *highestBid*, then it proceeds to refund the previous highest bidder. If both conditions are met and the refund is completed, the action contract updates the highest bidder and the highest bid amount.

One attacker may create a POC contract to illegally win the bids. The POC contract uses the function *attack()* to win the bid with the highest bid. When other users participate in the bidding and offer the highest bid, the require statement at line 7 triggers the refund operation in POC contract at line 19, i.e., the unnamed fallback function decorated with payable. Due to the use of the function *revert()* in the function, the refund operation cannot complete, causing the require condition at line 7 to always return false. As a result, the subsequent codes at line 8 and 9 cannot be executed, allowing the POC contract to win the bid at a lower price.

Code block 5. Solidity denial-of-service vulnerability sample code

```
1   contract Auction {
2       address public currentLeader;
3       uint256 public highestBid;
4
5       function bid() public payable {
6           require(msg.value > highestBid);
7           require(currentLeader.send(highestBid));
8           currentLeader = msg.sender;
9           highestBid = msg.value;
10      }
11  }
12
13  contract POC {
14      ...
15      function attack() public payable {
16          Auction auction = Auction(auctionAddr);
17          auction.bid.value(msg.value)();
18      }
19      function () external payable {
20          revert();
21      }
22  }
```

The automata shown in Figure 10 depicts the interactions among the attacker user, bidder user, auction contract, and POC contract. Figure 11 illustrates its simulation execution sequence. Normally, a bidder user in Figure 10c directly sends a message *bid!* to the auction contract, specifying the bid amount variable *msg_value* = 10 and their address as *msg_sender*. The attacker user in Figure 10a, in order to prevent other bidders from successfully placing higher bids, sends a message *attack!* to the POC contract to call the function *attack()*. Upon receiving the message *attack?*, the POC contract in Figure 10d internally sends a message *bid!* to the auction contract, setting *msg_value=20* and *msg_sender* to its own address.

When the auction contract in Figure 10b receives the message *bid?*, it checks whether the condition *msg_value* > *highestBid* is true or not. As *20* > *10*, the condition is true and the auction automaton continues execution to state *send_currentLeader*. If the condition is false, it returns to the state *Start*. The contract then checks if the current leader address is a contract address; if not, it refunds the bid amount to the bidder user's address. When the user bids again with 40, triggering the function *refund*, the auction contract sends a message *fallback!* to invoke the function *fallback* in the receiving contract.

Upon receiving the message *fallback?*, the POC contract sends a *revert!* to execute the refund operation. The auction contract, upon receiving *revert?*, restores the account balance and returns to the state *Start*. After the successful refund operation, it updates *currentLeader* and *highestBid*, and returns to the state *Start*.

CTL properties:

$$\text{E<> poc.revert_end \&\& auction.require_false} \quad (5)$$

By verifying the presence of denial-of-service vulnerabilities in the contract through the property (5), when the auction contract reaches the state *require_false* by executing the 7th line, the POC contract reaches the state *revert_end* after the function *revert()* is executed. If the above situation exists, it means that the POC contract can successfully prevent higher bids in the auction by rejecting payments.

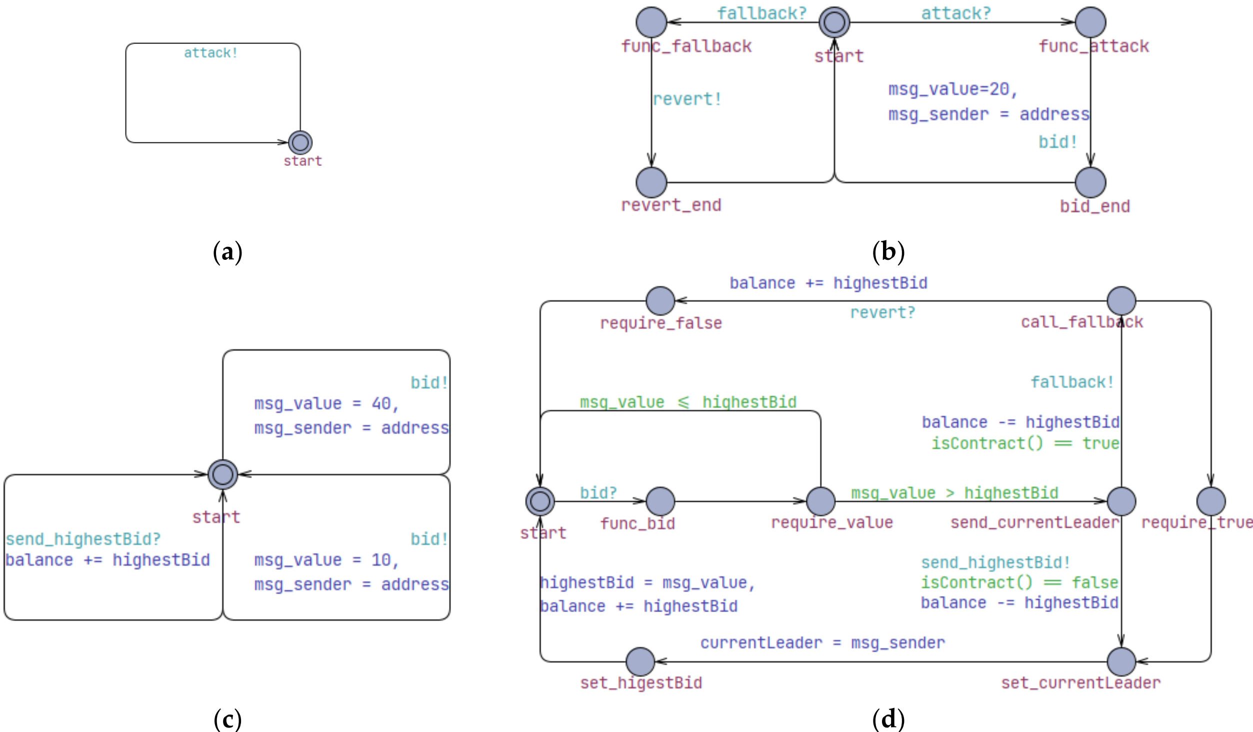


Figure 10. Denial-of-service vulnerability model. (**a**) attacker user model; (**b**) POC contract model; (**c**) bidder user model; (**d**) auction contract model.

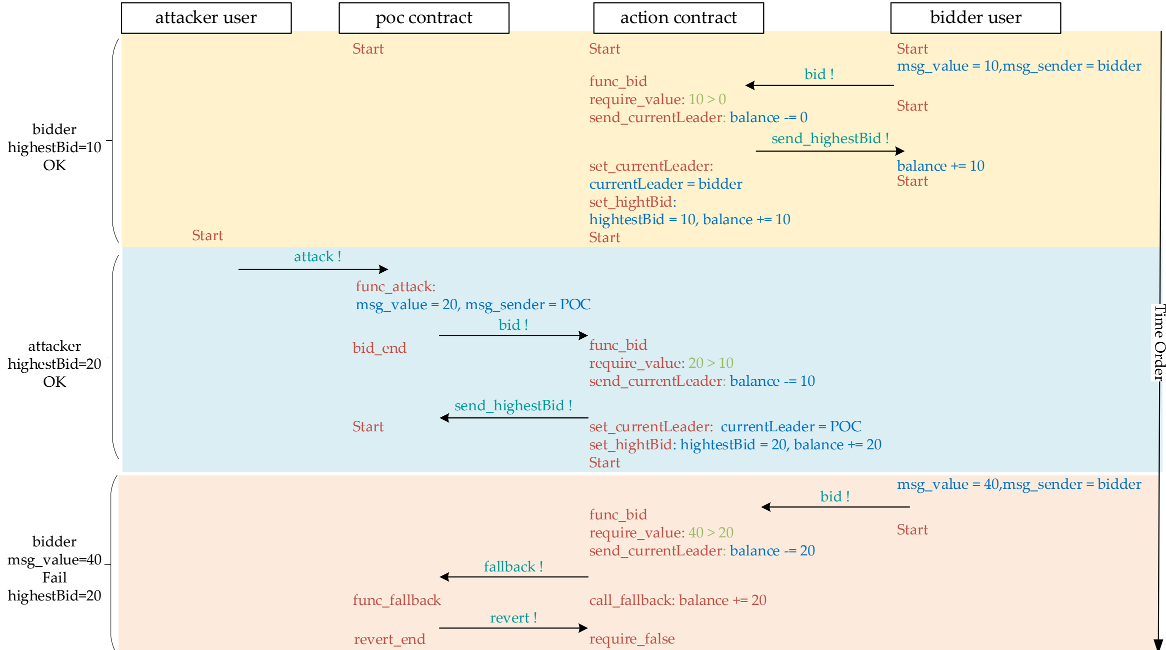


Figure 11. Denial-of-service vulnerability simulation execution sequence.

3.2.6. Miner Privilege

The miner privilege vulnerability mainly refers to contract vulnerabilities that rely on timestamps. Block timestamps are widely used in various conditional statements based on time-changing states, such as generating random numbers and locking funds for a period of time. If miners have the ability to slightly adjust the timestamp (the adjusted value is still legal), and the smart contract mistakenly uses the block timestamp, this can have serious consequences. Taking the lottery roulette contract code shown in the code block 6 as an example, the function rollback at line 3 is used for a single bet. Firstly, the require statement at line 4 is used to limit the player's betting amount to meet the condition *msg.value == 10 Ether*. Then, the require statement in the 5th line is used to limit each block to only contain one bet transaction of 10 Ether. Finally, the 7th line determines that if the current timestamp is a multiple of 15, the player can win the full balance of the contract. Clearly, if miners help players adjust timestamps, players will easily win.

Code block 6. Solidity miner privilege vulnerability sample code

```
1   contract Roulette {
2       ...
3       function () public payable {
4           require(msg.value == 10 Ether);
5           require(now != pastBlockTime);
6           pastBlockTime = now;
7           if (now % 15 == 0) {
8               msg.sender.transfer(this.balance);
9           }
10      }
11  }
```

The automata for players, miners, and roulette contracts are shown in Figure 12. Figure 13 illustrates its simulation execution sequence. The variable *balance* for each of these three models is initialized to 100. The player and miner automata are mostly the same, but miner users can control the timestamp *now* based on the current clock variable *t*, so the value of *now* is assigned with a multiple of 15 (satisfying the condition for generating new blocks within 900s of the previous block). After variables configuration, the miner automaton sets the betting amount *msg_value* to 10 ether (accordingly both its account balance *balance* and roulette contract balance balance[address] decreased by 10) and sends the message *fallback!* to the roulette contract to call its function *fallback.* After receiving the *fallback?*, the roulette automaton begins to execute the function *fallback.* The first step is to determine whether the betting amount *msg_value* == 10 ether. If they are equal, the roulette automaton reaches the state *request_time*. If the current timestamp *now* equals to *passBlockTime*, the roulette automaton sends the message *fallback_end!*, returns the received betting amount 10 Ether to the miner, and goes back to the state *Start*. If the current timestamp *now* is not equal to *pasteBlockTime*, the roulette automaton updates *pasteBlockTime=now* and checks whether the current timestamp *now* is a multiple of 15 or not. As miners can control the timestamp *now*, if the condition *now%15* == 0 is met, all balances of the roulette contract will be transferred to the betting party, most likely to be miners, the roulette automaton reaches the state *player_win* and sends the message *fallback_end!*. After receiving the message *fallback_end?*, the miner automaton updates its balance. Clearly, because a miner automaton can set the current timestamp *now* to a multiple of 15, it can always win the bet.

CTL properties:

$$\text{A[] roulette.player_win imply miner.bid} \tag{6}$$

By analyzing the execution process of the above model, we can obtain property (6), which is used to verify whether there is a mining privilege risk in the contract. It indicates that if the roulette contract reaches the state *player_win*, the user is always the miner. If

this property is satisfied, it indicates that there is a hidden privilege risk for miners in the roulette contract.

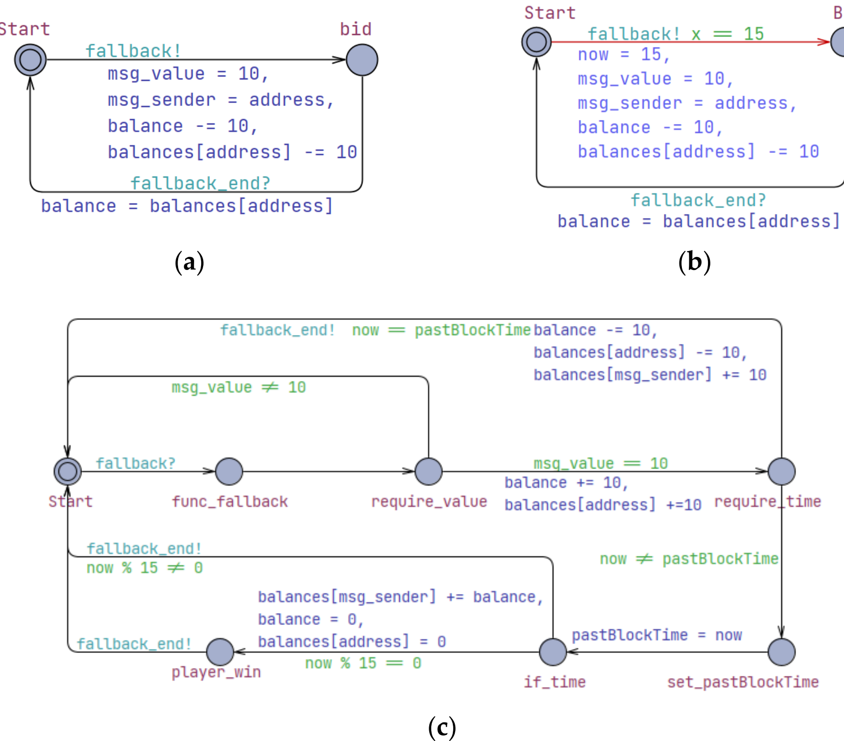


Figure 12. Miner privilege vulnerability model. (**a**) player model; (**b**) miner model; (**c**) roulette contract model.

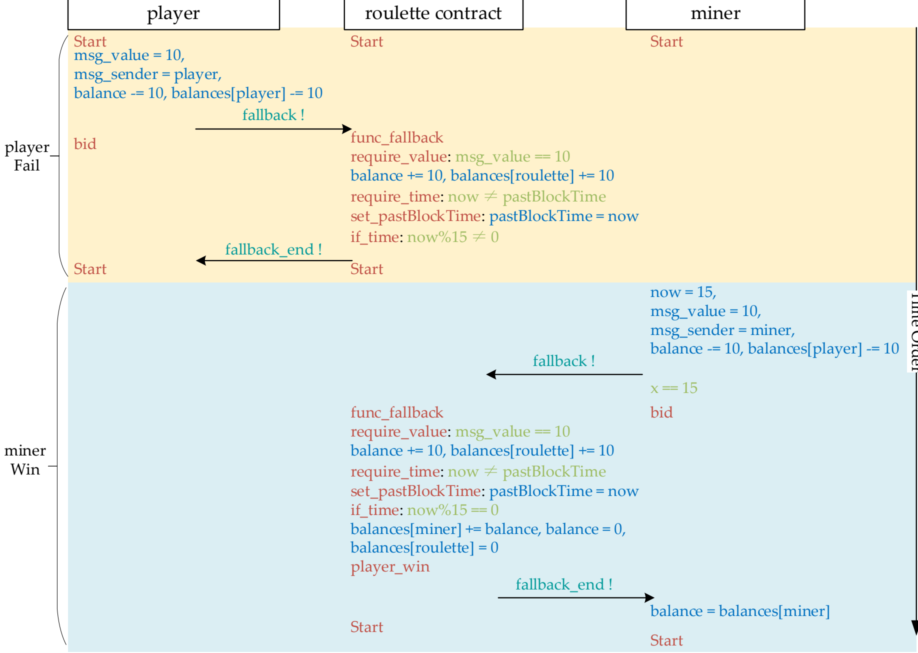


Figure 13. Miner privilege vulnerability simulation execution sequence.

4. Case Study

In order to illustrate the model and formal verification schemes described in Section 3, this section takes the composite contracts of financial services as a case study. We use UPPAAL to verify if the model has six types of security vulnerabilities and whether it meets

the business logic requirements. The experimental environment is configured with an Intel i7-1165G7 2.80 GHz CPU with 16 GB of memory, running Windows 11 and UPPAAL version 5.0.

Based on the Solidity code of the financial services composite contract, we model every financial service and user as a automaton shown in Figure 14. The model is divided into three layers, with a total of nine contracts. The lowest layer is the service contract layer for providing financial services to the outside world, including four contracts, which are bank, wallet, auction, and roulette. The bank contract at layer 3 of Figure 14 primarily includes functions for recharging, withdrawing, and the destructor operation when this contract becomes invalid. The wallet contract includes two types of transfer functions: the function *transfer* for peer-to-peer transactions and the function *transferToContract* for transactions to the other contracts. The function *setOwner* is utilized to designate the current owner of the wallet. The functions *enableTokenTransfer* and *disableTokenTransfer* are employed to control the wallet's operational status, enabling or disabling the transfer capabilities as required. In the auction contract, the function *bid* is used to obtain the bid amount, refund funds to the previous highest bidder, and update the current successful bidder. In the roulette contract, the function *fallback* is employed to transfer the entire balance of the contract to the winner when the user's lottery time meets the contract's timestamp requirements.

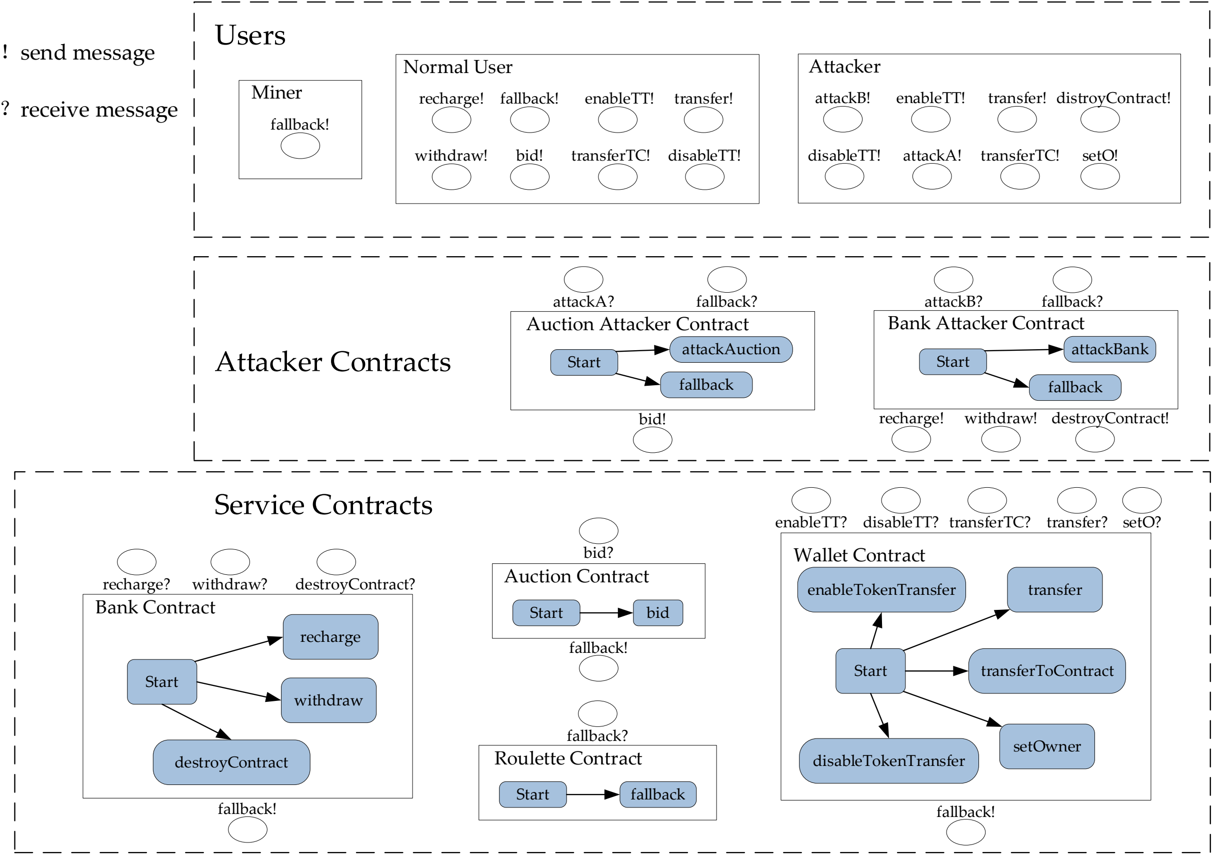


Figure 14. Case studies of composite smart contracts for financial services.

In order to verify the dynamic behavior of the composite contract, as shown in the top layer of Figure 14, in addition to the normal user, miner, and attacker are also designed. The attack contracts in the intermediate layer include bank attacker contract and auction attacker contract. Normal users interact with the four bottom-layer service contracts for regular service communication. Miner users control the current timestamp to meet the lottery time requirements of the roulette contract, making themselves the winners.

Attacker users can initiate attacks on service contracts either directly by sending messages to the wallet contract such as *enableTT!*, *disableTT!*, *setO!*, *transfer!*, and *transferTC!*, and to the bank contract with messages like *destroyContract!*, or by activating attack contracts in the intermediate layer by sending messages like *attackA!* and *attackB!*. The bank attacker contract can launch an attack on the bank contract using the function *attackBank*, repeatedly calling its function *withdraw* (by sending the message *withdraw!*) to gain illicit profits, and can also call the function *destroyContract* (by sending the message *destroyContract!*) to destroy the bank contract. The auction attacker contract can initiate an attack on the auction contract through the function *attackAuction*, which can keep it in a busy state for a certain period to prevent other users from successfully bidding.

4.1. Security Vulnerability Properties Verification

Table 2 illustrates the CTL properties for security vulnerability verification. Bank contracts may have security vulnerabilities such as reentrancy and exposure of privileged functions. If the verification result of reentrancy vulnerability is true, it indicates that the attacker can use the bank attacker contract to repeatedly call the function *withdraw()* and achieve the goal of multiple withdrawals. The problem code is located at line 7 of reentrancy sample code. The operation *call.value()* in the function *withdraw()* invokes the *fallback()* of the payment contract, but the *fallback()* can call the function *withdraw()* again, which causes duplicate transfers. If the verification result of privileged function exposure is true, it indicates that both the attacker and the contract owner, i.e., all users, can perform a deconstruction operation on the contract and obtain the full balance within the contract. The problem code is shown at line 3 of privilege function exposure sample code, and the important function *destroyContract()* lacks a modifier to restrict the caller when defined.

Code block 7. Solidity sample code of cross-contract invocation vulnerability

```
1    function transferToContract (address from, address to, unit256 amount, string
     custom_fallback) public {
2        to.call.value(0) (bytes4 (keccak256 (custom_fallback)), from, amount);
3    }
```

Table 2. Security vulnerability verification results of the financial service composite contract case.

Contract Name	Security Vulnerability	CTL Properties	Verification Result
Bank	Reentrancy	E<> attacker.bank_withdraw2 && bank.withdraw_function	True
	Privilege function exposure	E<> bank.transfer_end && attacker.transfer_end	True
		E<> bank.transfer_end && owner.transfer_end	True
Wallet	Access control	E<> wallet.disableTransfer && attacker.call_disable	True
		E<> wallet.enableTransfer && attacker.call_enable	True
		E<> wallet.disableTransfer && owner.call_disable	False
		E<> wallet.enableTransfer && owner.call_enable	False
	Cross-contract invocation	E<> attacker.func_setOwner && wallet.setOwner	False
		E<> attacker.func_transferTC && wallet.setOwner	True
Auction	Denial of service	E<> attacker.revert_end && auction.require_false	True
Roulette	Miner privilege	A[] roulette.player_win imply miner.bid	True

The wallet contract may have access control and cross-contract invocation vulnerabilities. For the four properties pertaining to access control vulnerabilities, if properties of line 4 and 5 in Table 2 both evaluate to true, and properties of line 6 and 7 both evaluate to false, it can be verified that the contract contains vulnerabilities of this type. This indicates

that the attacker can perform switch operations on the wallet, while the contract owner cannot. The code is located at line 14 of code block 2, where the condition in the modifier *onlyFromWallet* is set as *msg.sender != walletAddress*, allowing an attacker's address to meet this condition and, thus, enabling them to set the wallet switch.

Regarding the cross-contract invocation vulnerability in Table 2, when the property at line 8 in Table 2 evaluates to false and the property at line 9 in Table 2 evaluates to true, it verifies the existence of a cross-contract invocation security vulnerability in the wallet contract. This means that the attacker cannot directly set the contract owner, but can indirectly achieve the change in the contract owner through the function *transferToContract()*. The code is located at line 2 of the code block 7 where the operation *call.value* in the function *transferToContract()* accepts the called function name and address variables as parameters; the attacker can exploit this parameter to circumvent the restriction that the caller must be the contract itself, thereby achieving the invocation of any internal function of the contract, such as setOwner(), allowing the attacker to utilize these parameters to invoke any internal function *setOwner()* of the contract.

In the auction contract, the verification of the denial-of-service vulnerability shows that the attacker can exploit the auction attacker contract to prevent the auction contract from refunding the current bid, causing the update of the current highest bidder and the highest bid operation at lines 4–5 to fail. Ultimately, when the auction deadline is reached, the attacker can successfully bid at a lower price. The code is shown at line 7 of code block 5, where the refund of the current leader's highest bid can be utilized by the receiving contract to implement denial of payment, preventing other bidders from bidding.

In the roulette contract, the verification of miner privilege vulnerability shows that miner users can alter the current timestamp. As long as there is a winning player, then that player must be a miner user. According to the verification, the code is located at line 7 of code block 6, because the condition for sending contract balance to the winning player depends on the current timestamp, which can be controlled by miner users.

4.2. Business Logic Properties Verification

To verify the business logic properties, corresponding CTL properties can be proposed from the perspective of contract function design. As shown in Table 3, the property in row 1 is used to verify the correctness of the balance of all user accounts in the bank contract. For example, if there is a situation where the bank contract's own balance does not equal the sum of each bank account balance, then the CTL property is satisfied, and the verification result is passed, indicating an error in the bank contract. The CTL property can also be used to find attackers. The property indicated in row 2 means that in any case, if the bank contract's own balance is 0, but the attacker contract's balance is always greater than the balance of all other accounts. If the CTL property is satisfied, it implies that the attacker contract is the attacker.

Table 3. Business logic verification results of the financial service composite contract case.

Contract Name	CTL Properties	Verification Result
Bank	E<> bank.balance != bank.balances[bank.address] + bank.balances[bank_attacker.address] + bank.balances[attacker_user.address]	True
	A[] bank.balance == 0 imply bank_attacker.balance > attacker_user.balance && bank_attacker.balance > bank.balance	True
Auction	E<> msg_value == 40 && highestBid == 20 && auction.require_false && poc.revert_end	True
Roulette	A[] roulette.balance == 0 imply balances[miner.address] > 100 && balances[player1.address] <= 100 && balances[player2.address] <= 100	True

The CTL property for auction contract at row 3 can be utilized to identify contract attackers when a specific bidding action occurs. This property indicates that both conditions, *msg_value == 40* and *highestBid == 20*, are satisfied simultaneously and at this point, the auction contract executes the operation *require_false*, while the auction attacker contract executes the operation *revert_end*. If such a scenario exists, it implies that the POC contract with a bid amount of 20 is preventing the user with a bid amount of 40 from participating in the auction by refusing to accept payment. Should this property verification pass, it would suggest that the auction attacker contract is the attacker. Line 4 is the roulette contract, which contains a CTL property that signifies when the contract's balance is 100, and it implies that the miner user's balance exceeds 100 while each player user's balance is less than 100. By utilizing this property, it is possible to pinpoint the account address of the malicious miner user.

To evaluate the verification efficiency of smart contracts, an experimental validation was conducted on the banking case study presented in this chapter. The results, as depicted in Figure 15, indicate that the verification time for individual properties does not exceed 10 milliseconds, and the overall verification efficiency is below 1 s, demonstrating that this method is highly efficient and possesses practical applicability.

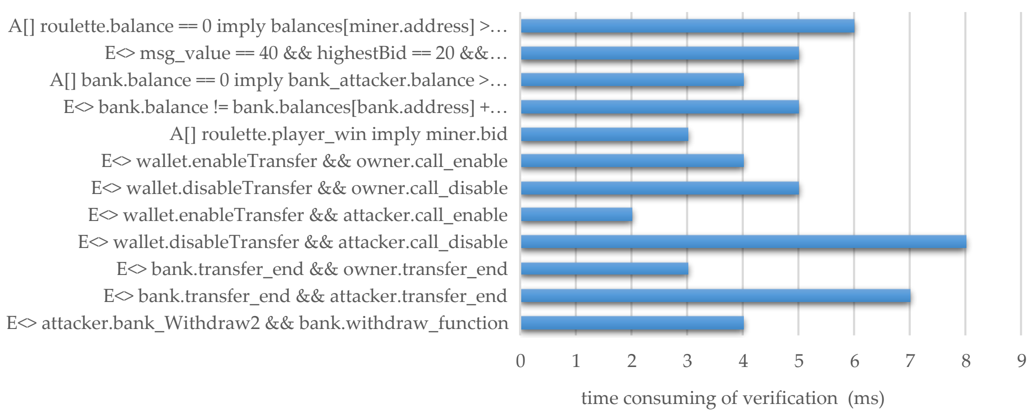


Figure 15. Time consumed for properties verification.

5. Conclusions

This paper proposed a dynamic behavior analysis and verification method for composite smart contracts, addressing six typical categories of security and business logic vulnerabilities that are not typically covered by traditional static analysis methods. Modeling composite services using automata, with the incorporation of temporal factors, CTL is employed to articulate business logic and security properties. Finally, a case study and time evaluation experiment involving a composite smart contract in financial services are conducted by UPPAAL checker to validate the method. The case and experimental results demonstrate that the proposed method can be utilized for analyzing and verifying the security and business logic vulnerabilities of composite smart contracts.

In future work, the current verification framework is predicated on known attack patterns and is incapable of verifying contract robustness against unknown types of attacks. Future plans include analyzing and verifying contracts that may have potential vulnerabilities. Secondly, methods that integrate knowledge graph extraction will be utilized to extract key information. Furthermore, the automatic translation of Solidity into UPPAAL model files (.xml) will be implemented. These approaches will collectively enhance the efficiency of model construction. All these works will be both meaningful and challenging.

Author Contributions: Conceptualization, J.J.; methodology, J.L.; software, W.Z.; formal analysis, J.L.; writing—original draft, W.Z. and J.J.; writing—review and editing, J.J., H.L. and Y.D.; supervision, Y.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Beijing Natural Science Foundation (grant number M22040), the Open Research Fund of Beijing Key Laboratory of Big Data Technology for Food Safety from Beijing Technology and Business University (grant number BTBD-2021KF06), the Open Research Fund of Yunnan Key Laboratory of Blockchain Application Technology (grant number 202105AG070005, project number YNB202105), the Beijing Wuzi University Youth Research Fund (grant number 2022XJQN24), and the Science and Technique General Program of Beijing Municipal Commission of Education (grant number KM201910037003).

Data Availability Statement: Data are contained within the paper.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Yaga, D.; Mell, P.; Roby, N.; Scarfone, K. Blockchain Technology Overview. *arXiv* **2019**, arXiv:1906.11078.
2. Zheng, Z.; Xie, S.; Dai, H.-N.; Chen, W.; Chen, X.; Weng, J.; Imran, M. An Overview on Smart Contracts: Challenges, Advances and Platforms. *Future Gener. Comput. Syst.* **2020**, *105*, 475–491. [CrossRef]
3. Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008. Available online: https://assets.pubpub.org/d8wct41f/31611263538139.pdf (accessed on 18 July 2024).
4. Wood, G. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum Proj. Yellow Pap.* **2014**, *151*, 1–32.
5. Wohrer, M.; Zdun, U. Smart Contracts: Security Patterns in the Ethereum Ecosystem and Solidity. In Proceedings of the 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), Campobasso, Italy, 20 March 2018.
6. Huang, Y.; Bian, Y.; Li, R.; Zhao, J.L.; Shi, P. Smart Contract Security: A Software Lifecycle Perspective. *IEEE Access* **2019**, *7*, 150184–150202. [CrossRef]
7. Wang, Z.; Jin, H.; Dai, W.; Choo, K.-K.R.; Zou, D. Ethereum Smart Contract Security Research: Survey and Future Research Opportunities. *Front. Comput. Sci.* **2021**, *15*, 1–18. [CrossRef]
8. Zou, W.; Lo, D.; Kochhar, P.S.; Le, X.-B.D.; Xia, X.; Feng, Y.; Chen, Z.; Xu, B. Smart Contract Development: Challenges and Opportunities. *IEEE Trans. Softw. Eng.* **2019**, *47*, 2084–2106. [CrossRef]
9. Kushwaha, S.S.; Joshi, S.; Singh, D.; Kaur, M.; Lee, H.-N. Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract. *IEEE Access* **2022**, *10*, 6605–6621. [CrossRef]
10. Mehar, M.I.; Shier, C.L.; Giambattista, A.; Gong, E.; Fletcher, G.; Sanayhie, R.; Kim, H.M.; Laskowski, M. Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack. *J. Cases Inf. Technol. (JCIT)* **2019**, *21*, 19–32. [CrossRef]
11. Singh, A.; Parizi, R.M.; Zhang, Q.; Choo, K.-K.R.; Dehghantanha, A. Blockchain Smart Contracts Formalization: Approaches and Challenges to Address Vulnerabilities. *Comput. Secur.* **2020**, *88*, 101654. [CrossRef]
12. Luu, L.; Chu, D.-H.; Olickel, H.; Saxena, P.; Hobor, A. Making Smart Contracts Smarter. In Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS), Vienna, Austria, 24–28 October 2016.
13. Amani, S.; Bégel, M.; Bortin, M.; Staples, M. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP), Los Angeles, CA, USA, 8–9 January 2018.
14. Wang, Y.; Lahiri, S.K.; Chen, S.; Pan, R.; Dillig, I.; Born, C.; Naseer, I. Formal Specification and Verification of Smart Contracts for Azure Blockchain. *arXiv* **2018**, arXiv:1812.08829.
15. Bai, X.; Cheng, Z.; Duan, Z.; Hu, K. Formal Modeling and Verification of Smart Contracts. In Proceedings of the 7th International Conference on Software and Computer Applications, Kuantan, Malaysia, 8–10 February 2018.
16. Yang, Z.; Dai, M.; Guo, J. Formal Modeling and Verification of Smart Contracts with Spin. *Electronics* **2022**, *11*, 3091. [CrossRef]
17. Nelaturu, K.; Mavridoul, A.; Veneris, A.; Laszka, A. Verified Development and Deployment of Multiple Interacting Smart Contracts with VeriSolid. In Proceedings of the 2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), Toronto, ON, Canada, 2–6 May 2020.
18. Almakhour, M.; Sliman, L.; Samhat, A.E.; Mellouk, A. A Formal Verification Approach for Composite Smart Contracts Security Using FSM. *J. King Saud Univ.-Comput. Inf. Sci.* **2023**, *35*, 70–86. [CrossRef]
19. Behrmann, G.; David, A.; Larsen, K.G. A Tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems*; Springer: Berlin/Heidelberg, Germany, 2004; pp. 200–236.
20. Alqahtani, S.; He, X.; Gamble, R.; Mauricio, P. Formal Verification of Functional Requirements for Smart Contract Compositions in Supply Chain Management Systems. In Proceedings of the 53rd Hawaii International Conference on System Sciences, Maui, HI, USA, 7–10 January 2020.
21. Nam, W.; Kil, H. Formal Verification of Blockchain Smart Contracts via ATL Model Checking. *IEEE Access* **2022**, *10*, 8151–8162. [CrossRef]
22. So, S.; Lee, M.; Park, J.; Lee, H.; Oh, H. Verismart: A Highly Precise Safety Verifier for Ethereum Smart Contracts. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020.
23. Chen, J.; Xia, X.; Lo, D.; Grundy, J.; Luo, X.; Chen, T. DefectChecker: Automated Smart Contract Defect Detection by Analyzing EVM Bytecode. *IEEE Trans. Softw. Eng.* **2022**, *48*, 2189–2207. [CrossRef]

24. Liu, Y.; Li, Y.; Lin, S.W.; Artho, C. Finding permission bugs in smart contracts with role mining. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Virtual, Republic of Korea, 18–22 July 2022.
25. Wang, W.; Huang, W.; Meng, Z.; Xiong, Y.; Miao, F.; Fang, X.; Tu, C.; Ji, R. Automated inference on financial security of Ethereum smart contracts. In Proceedings of the 32nd USENIX Security Symposium (USENIX Security), Anaheim, CA, USA, 9–11 August 2023.
26. Zhao, Y.Q.; Zhu, X.Y.; Li, G.Y.; Bao, Y.L. Time Constraint Patterns of Smart Contracts and Their Formal Verification. *J. Softw.* **2022**, *33*, 2875–2895.
27. Ethereum. Solidity Documentation. 2022. Available online: https://docs.soliditylang.org/en/v0.8.11 (accessed on 18 July 2024).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.