electronics

Article

Formal Analysis of Reentrancy Vulnerabilities in Smart Contract Based on CPN

Yaqiong He [1], Hanjie Dong [1], Huaiguang Wu [1,*] and Qianheng Duan [2]

[1] College of Computer and Communication Engineering, Zhengzhou University of Light Industry, Zhengzhou 450001, China
[2] Henan Key Laboratory of Network Cryptography Technology, Zhengzhou 450001, China
* Correspondence: hgawu@126.com

Abstract: A smart contract is a special form of computer program that runs on a blockchain and provides a new way to implement financial and business transactions in a conflict-free and transparent environment. In blockchain systems such as Ethereum, smart contracts can handle and autonomously transfer assets of considerable value to other parties. Hence, it is particularly important to ensure that smart contracts function as intended since bugs or vulnerabilities may lead, and indeed have led, to substantial economic losses and erosion of trust for blockchain. While a number of approaches and tools have been developed to find vulnerabilities, formal methods present the highest level of confidence in the security of smart contracts. In this paper, we propose a formal solution to model a smart contract based on colored Petri nets (CPNs). Herein, we focus on the most common type of security bugs in smart contract, i.e., reentrancy bugs, which led to a serious financial loss of around USD 34 million for the Cream Finance project in 2021. We present a hierarchical CPN modelling method to analyze potential security vulnerabilities at the contract's source code level. Then, modeling analysis methods such as correlation matrix, state space report and state space graph generated via CPN Tools simulation are exploited for formal analysis of smart contracts. The example shows the full state space and wrong path in accordance with our expected results. Finally, the conclusion was verified on the Ethereum network based on the Remix platform.

Keywords: smart contract; CPN; reentrancy vulnerability; Petri net

Citation: He, Y.; Dong, H.; Wu, H.; Duan, Q. Formal Analysis of Reentrancy Vulnerabilities in Smart Contract Based on CPN. *Electronics* **2023**, *12*, 2152. https://doi.org/10.3390/electronics12102152

Academic Editor: Hung-Yu Chien

Received: 19 February 2023
Revised: 3 May 2023
Accepted: 4 May 2023
Published: 9 May 2023

1. Introduction

Since the introduction of Bitcoin in 2009 [1], the first decentralized cryptocurrency has gained considerable attention and has been adopted to a considerable extent. As of 2022, there are approximately 20,457 cryptocurrencies that can be tracked via CoinMarketCap. A cryptocurrency is administrated via distributed network nodes without involving a trusted third party or central agency. The nodes share and secure a public and append-only ledger of transactions, i.e., blockchain. Blockchain is a distributed and decentralized digital ledger technology that allows multiple parties to record, verify and share transactions in a secure and transparent manner without the need for a central authority [2]. In recent years, the popular Ethereum [3] blockchain started the era of Blockchain 2.0 by introducing the Turing-complete smart contract, with a wide range of applications including auctions [4], elections [5], crowdfunding, and cross-industry finance far beyond cryptocurrency [6].

Tens of thousands of transactions have been processed on Ethereum every day since its launch in July 2015, and its capitalization has reached more 100 billion as of July 2022. As such, we only consider smart contracts on Ethereum blockchain in this work.

The term "smart contract" was first proposed in the 1990s by computer scientist and cryptographer Nick Szabo [7], who defined a smart contract as "a set of commitments, specified in the digital form, including protocols of the contract participants [8]". However, due to the lack of trusted environments, smart contracts did not receive widespread

attention and research at that time. The emergence of blockchain technology has redefined and made smart contracts possible. In the context of blockchain, a smart contract is a self-executing computer program that runs on blockchain. It will automatically execute any task when predefined conditions are met.

The security of smart contracts has become a major concern for the healthy development of Ethereum. A smart contract on the blockchain often carries financial value worth millions of dollars [9]. Holding so much wealth, however, makes smart contracts lucrative and profitable targets for malicious attackers. A bug, flaw or error in a smart contract may produce an unexpected result or behave in unintended ways, resulting in devastating consequences. For instance, by exploiting the reentrancy vulnerability in the DAO [10] smart contract's source code, an attacker managed to drain more than USD 60 million worth of Ether in 2016. In addition, the SpankChain contract [10] also suffered a reentrancy attack in 2018, which caused USD 40,000 to be stolen for lack of proper authority management. Similarly, in January 2020, the Uniswap [11] and Lendf.Me [12] projects were subject to the same vulnerability again. It can be seen that a security vulnerability may cause irreparable losses to the project or investors. Reentrancy vulnerabilities are some of the most serious vulnerabilities in smart contracts, and are discovered and exploited from time to time, undermining the trust for smart contract-based applications [13].

Traditional informal analysis methods mainly focus on the analysis of logic control behavior, which is no longer enough to deal with the complex and changeable on-chain data state environment. Formal verification methods are based on mathematical models and reasoning, making them more rigorous and reliable. In recent years, formal methods have received more and more attention in the security verification of smart contracts. In this paper, we conduct a formal analysis method based on CPNs [14] to analyze and verify the reentrancy vulnerabilities of smart contracts from different perspectives. We chose this formalism because of its ability to combine the strong description ability of Petri nets with the expressive power of programming languages. In order to describe the TheDAO contract more accurately, we added concepts such as data attributes, key data elements and key transitions, and described the smart contract from the perspective of data flow and control flow, where the data state reflects the change of the transaction state, and the control flow reflects some key operations during the transaction process. Then, we utilized modeling tools in CPNs to hierarchically model the simplified TheDAO contract, including the whole contract, non-attack operation and attack operation, respectively, which avoid the loss of context-based information and the influence of under-approximation. Furthermore, we adopted the simulation tools in CPN to show the specific execution process of the contract. The results not only revealed the logic loopholes at the Solidity language level, but also identified the malicious attack behavior against custom specifications of contacts in real life. Moreover, we used the state analysis method to analyze the model and obtained an executable attack transition sequence. Finally, the correctness of the conclusion was confirmed on the Remix platform.

To summarize, the main contributions of this study are as follows:

- We propose a formal verification for detecting reentrancy vulnerability, introducing the concepts of data attributes, key data elements and key transitions.
- We leverage CPN Tools to hierarchically model the simplified TheDAO contract and describe smart contracts from the perspectives of data flow and control flow.
- We not only discover the logic loopholes at the Solidity language level, but also find the malicious attack behavior against custom specifications of contacts.
- We verify our results on the Remix platform.

The remainder of this paper is structured as follows. Section 2 provides essential background knowledge on smart contracts. Section 3 describes the property specification and model building of TheDAO. In Section 4, we conduct experimental analysis from various aspects and verify our results based on the Remix platform. Section 5 discusses the related work about reentrancy vulnerability detection. Finally, we conclude this paper and explore future work in Section 6.

2. Background

In this section, we introduce background knowledge about smart contracts, colored Petri nets and TheDAO contracts.

2.1. Ethereum and Smart Contract

There are many blockchain platforms that support the operation of smart contracts, such as EOS, BCOS, Fabric and CITA. Among them, Ethereum is the largest, oldest and most influential one [15]. It uses Solidity [16] programming language to develop smart contracts and provides a large number of application program interfaces (APIs). The deployment of smart contracts is realized by sending a transaction to the blockchain; the entire process is shown in Figure 1. The receiver of the transaction is empty and contains bytecode and other information. After the miner successfully packages the transaction, an address is returned. This address is the contract address, and the invocation or access of the contract needs to go through this address, which is the unique identifier of the contract. Smart contracts are compiled and interpreted and executed using Ethereum virtual machines (EVMs) installed on each Ethereum node. Ethereum accounts fall into two categories: external accounts and internal accounts. External accounts invoke contracts by sending transactions, which are broadcast to all nodes of the Ethereum network, packaged and verified by miners, and all transaction records are stored in the blockchain. On the other hand, internal accounts call other contracts through message calls, which transfer internal parameters and will not synchronize with the blockchain. Therefore, the contract invoked via message invocation between contracts will not be recorded in the chain.

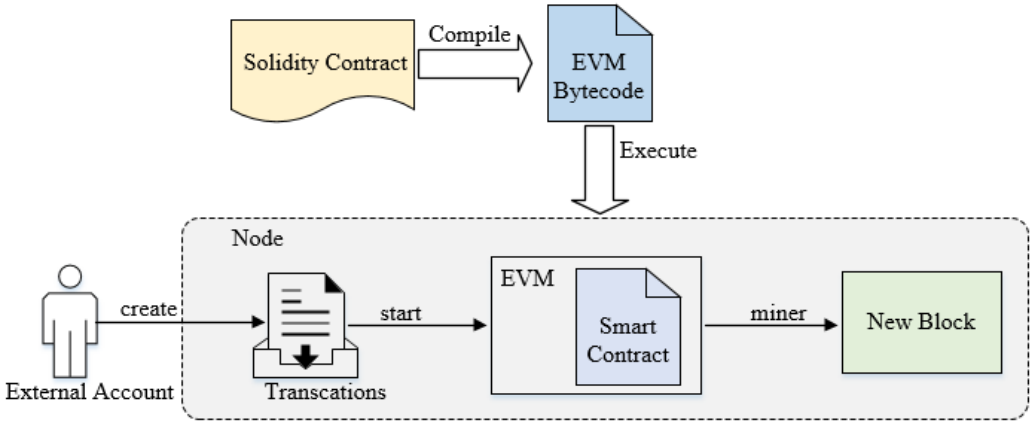


Figure 1. Smart contract deployment flowchart.

2.2. Features of Smart Contracts

A couple of reasons could account for why smart contracts are particularly prone to errors. Compared with ordinary programs, smart contracts have many different features. Ethereum smart contracts run on the blockchain, and their complete lifecycle includes development, compilation, deployment, invocation and destruction, many of which are triggered by Ethereum transactions with the flow of assets between contract participants. Hence, smart contract programming requires an "economic thinking" perspective, which is undoubtedly a challenging task for traditional programmers. [17] In addition, smart contract programs also have special mechanisms such as gas restrictions, delegated calls and code that cannot be modified, updated or fixed due to the immutability of blockchains. These differences bring new security risks and attack surfaces to smart contracts; practitioners are required to take into consideration all possible situations and environments that may be encountered in the future. As a result, a rigorous security analysis of smart contracts is of great significance before deployment.

2.3. Colored Petri Net

Petri net is a formal model for modeling discrete event concurrent systems, and was proposed by the German scholar Carl Adam Petri in the 1960s [18]. It can not only describe the structure of the system, but also simulate its operation. Petri nets include two types of nodes, place and transition, and add a token distribution that represents state information to the set of places. Places and transitions are connected by directed arcs, and the tokens in the place represent the resources or data that can be used. Despite the advantages of concurrency and synchronization, Petri nets have the problem of space explosion. Colored Petri nets (CPNs) were founded in 1981 by Professor Jensen [19] to overcome the bottleneck problem of Petri nets. On the basis of Petri nets, the problem scale of basic Petri nets is simplified by adding a color set and variable definition, which greatly complements the abstraction and description ability of Petri nets. The built-in mechanism of CPNs includes hierarchical structure, fusion set, entity folding, etc., reducing the possibility of state space explosion. CPN Tools is a modeling tool for editing, simulating and analyzing colored Petri nets. Using CPN Tools, it is possible to study the behavior of the modeled system by means of simulation, and generate state space reports [20]. The generated state space directed graph can not only precisely locate the wrong path, but also display the status of each node in the path, which is convenient for analyzing the cause and improving the contract code. The formal definition of CPN [21] is given as follows.

Definition 1 (Colored Petri Net**).** *A Colored Petri Net is a nine-tuple* $CPN = (P, T, A, \Sigma, V, C, G, E, I)$*, where:*

(1) P *is a finite set of places.*
(2) T *is a finite set of transitions.*
(3) A *is a set of directed arcs and* $A \subseteq (P \times T) \cup (T \times P)$.
(4) Σ *is a finite set of color set types.*
(5) V *is a finite set of type variables, and* $Type[v] \in \sum$ *for* $\forall v \in V$.
(6) C *is the color function, which is the mapping from place set to color set such that* $C : P \to \sum$.
(7) G *is the guard function, which is the mapping such that* $G : T \to EXPR_v$ *and* $Type[G(t)] = Bool$.
(8) E *is an arc expression function, which is the mapping such that* $E : A \to EXPR_V$ *and* $Type[E(a)] = C(p)_{MS}$*, where* p *is the place connected to the arc and* MS *represents the polymorphic set.*
(9) I *is the initial function of* p*, which is the mapping such that* $I : P \to EXPR_\varnothing$*, and* $Type[I(p)] = C(p)_{MS}$.

In the process of firing the transition, the number of tokens in each place will change. The vector formed by the number of tokens corresponding to each place is called the identifier M, $M : P \to \{0, 1, 2 \ldots\}^k$, the initial identifier is M_0. The corresponding component of P is denoted as $M(p)$; there may be several transitions that have the right to fire under the initial identifier M_0. The rules of transition occurrence are as follows:

(1) For the transition $t \in T$, if $\forall p \in P : p \in \bullet t \to M(p) \geq 1$, then we use $M \xrightarrow{t}$ to denote that t is enabled at M, denoted by $M[t\rangle$.
(2) If $M[t\rangle$, a new marking M' can be produced through notation $M \xrightarrow{t}$, denoted by $M[t\rangle M'$. For $\forall p \in P$:

$$M[p] = \begin{cases} M[p] - W[p,t], if\ p \in \bullet t - t\bullet \\ M[p] + W[p,t], if\ p \in t\bullet - \bullet t \\ M[p] - W[p,t] + W[t,p], if\ p \in t\bullet \cap \bullet t \\ M[p], otherwise \end{cases} \tag{1}$$

Definition 2. *Let $N = (P, T; F)$ be a Petri net, where $P = \{p_1, p_2, \ldots, p_m\}$, $T = \{t_1, t_2, \ldots, t_m\}$, then the incidence matrix of N is $A = [a_{ij}]_{n \times m}$ with n rows and m columns, where:*

$$a_{ij} = a_{ij}^{+} - a_{ij}^{-}, i \in \{1, 2, \ldots, n\}, j \in \{1, 2, \ldots, m\} \tag{2}$$

$$a_{ij}^{+} = \begin{cases} 1, if(t_i, p_j) \in F \\ 0, otherwise \end{cases}, i \in \{1, 2, \cdots, n\},\ j \in \{1, 2, \cdots, m\} \tag{3}$$

$$a_{ij}^{-} = \begin{cases} 1, if(p_j, t_i) \in F \\ 0, otherwise \end{cases}, i \in \{1, 2, \cdots, n\},\ j \in \{1, 2, \cdots, m\} \tag{4}$$

For the convenience of the discussion, we introduce two matrices with n rows and m columns, i.e., $A^{+} = [a_{ij}^{+}]_{n \times m}$, $A^{-} = \left[a_{ij}^{+}\right]_{n \times m}$ and call them the output and input matrices of N.

Definition 3. *Let $N = (P, T; F)$ be a Petri net in which M_0 is the initial identifier and A is the incidence matrix of N. If $M \in R(M_0)$, then there exists an n-dimensional vector of non-negative integers such that:*

$$M = M_0 + A^{\mathrm{T}} X \tag{5}$$

This is called the state equation of Petri net. The n-dimensional vector X is called the occurrence number vector.

2.4. The Reentrancy Attack

Although smart contracts are widely adopted in various applications, they are not always secure. The majority of concerns with smart contract security began in 2016 with the notorious DAO attack. The Decentralized Autonomous Organization (DAO) is an open source project of the German startup Slock.it [22], which raises money through Ethereum and locks it into a smart contract. Each investor participating in the crowdfunding obtained the corresponding DAO token as shares according to the amount of investment, and jointly decided the invested project through voting, making the entire community fully autonomous. The DAO completed the crowdfunding on 28 May 2016, raising a total of 11.5 million ETH worth USD 149 million at the time, which was by far the largest crowdfunding project. The part code of the token exchange contract is shown in Figure 2. The contract owner can deposit and withdraw normally by calling the deposit () function in line 2 and the withdraw () function in line 5.

```
1  contract EtherStore{
2      function deposit() public payable{
3          balances[msg.sender]+=msg.value;
4      }
5      function.withdraw(uint256 amount) public payable{
6          require(balances[msg.sender]>=amount);
7          require(this.balance>=amount);
8          msg.sender.call.value(amount)();
9          balances[msg.sender]-=amount;
10     }
11 }
```

Figure 2. Simplified TheDAO contract code.

However, the DAO experiment failed soon after its launch when hackers exploited a vulnerability in the recursive calls in TheDAO smart contract (i.e., a reentry vulnerability), resulting in the hijacking of USD 60 million worth of Ether [23]. The reentrancy bug is easy to exploit on deployed blockchain contracts and has already become one of the most severe vulnerabilities in practice. The principal reason for this attack is the fallback mechanism. By default, there is a function without function names and parameters in Solidity smart contracts, called fallback function, which is automatically triggered when the identifier of the function call does not match any of the existing functions in a smart contract or if no data are supplied at all. Figure 1 simply implements deposit and withdrawal operations. This

code seems to be logically flawless, but the problem lies in the *call. value ()* function. In fact, the vulnerability of the Ethereum smart contract has a lot to do with its own grammatical characteristics. Unlike the *send ()* and *transfer ()* functions, with which only 2300 gas is used to process the transfer operation, the *call. value ()* function will consume all the remaining gas of the contract for external calls. If the target address of the transfer is a contract, the fallback function in the contract will be called automatically. When the attacker deploys a contract that maliciously and recursively calls the transfer operation before updating the balance, all the balances in the public wallet contract will be withdrawn. The whole principle is shown in Figure 3.

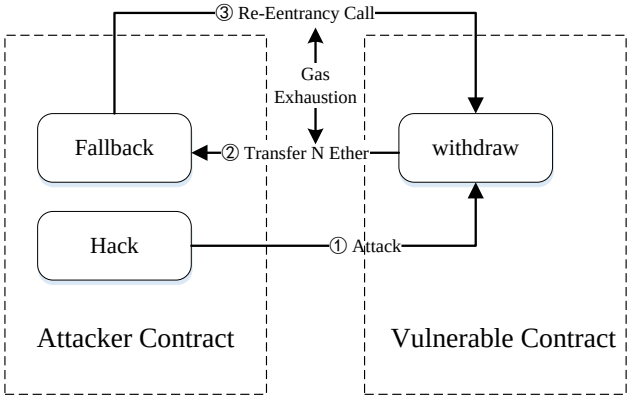


Figure 3. Reentrancy vulnerability principle.

In particular, the attacker contract first calls the deposit () to send 1 Ether to the vulnerable contract, as shown in Figure 4. Then, the attacker contract executes a withdraw operation, thus triggering the withdraw () in the vulnerable contract. When the attacker receives 1 Ether, the fallback function is also triggered. Next, the fallback function repeatedly calls the withdraw () of vulnerable contract. However, at this time, the balance is not updated in time. Therefore, before clearing the balance, the attacker can exchange a large amount of Ether with a small amount of DAO coins by constantly calling the *fallback* function. This not only brought huge losses to the crowdfunding participants, but also caused the first hard fork in the history of Ethereum [24], which attracted widespread concern in the industry.

```
contract Attack{
    function AttackEtherStore() public payable{
        EherStore.withdraw(1 ether);
    }
    function() payable{
        if (EherStore.balance > 1 ether){
            EherStore.withdraw(1 ether);
        }
    }
}
```

Figure 4. The DAO Attack.

3. Related Work

3.1. Reentracy Vulnerability Detection

ReGuard [25] is a fuzzing-based analyzer designed to specially detect a reentrancy bug in Ethereum smart contracts. Specifically, ReGuard consists of three key modules: Contract Transformer, Fuzzing Engine and Core Detector. It first parses the code of smart contracts into an intermediate representation, and then performs source-to-source transformation from IR to C++ using the contract transformer module. The transformed C++ contracts retain the original behavior and interfaces for the well-designed fuzzing engine. As the fuzzing engine runs, ReGuard executes a smart contract to identify reentrancy via runtime trace analysis, in which the trace is analyzed through Core Detector. A major challenge faced by fuzzing lies is that it tests programs in a random way [26].

Sereum [27] is a novel design and implementation of smart contract security technology which leverages the state updates of smart contracts to detect malicious behavior without knowing the relevant semantics about the Solidity language. On the basis of existing Ethereum clients, it extends to perform run-time monitoring by introducing a taint engine and attack detector. The attack detector exploits the taint engine to prevent suspicious states, making it the first solution to apply taint-tracking to smart contracts. By running Sereum on massive Ethereum transactions, results demonstrate high accuracy against attacks with little runtime overhead.

ReDetect [28] is a symbolic execution-based detection tool that targets reentrancy vulnerabilities of smart contracts at the EVM bytecode level. Therefore, for Solidity-based contracts, it is necessary to compile the contract into EVM bytecode by means of compiler solc in the preprocessing phase. The EVM bytecode file is then converted to EVM assembly code for constructing control flow graphs (CFGs) with CFG Builder. After that, the proposed tool explores each possible path where a reentrancy vulnerability may exist and devises five path filters to reduce false positives.

Clairvoyance [29] is a cross-function and cross-contract static analysis for finding reentrancy vulnerabilities in smart contracts. It takes as input the source code (Solidity) of contracts, and then builds a cross-contract call graph and CFG to identify vulnerable call chains that start with a suspicious contract address or object and end with a function call. By comparing Clairvoyance with three mainstream tools on 17,770 real contracts, it was found to present the best detection accuracy and was proven to be an accurate and efficient detection method.

Reentrancy Analyzer (RA) [30] is a static analysis tool that combines symbolic execution and equivalence checking with the Z3 satisfiability modulo theories (SMT) solver to detect reentrancy attack within smart contracts. Analysts can implement the analysis of inter-contract flows at the EVM bytecode level without much prior knowledge of attacks in contracts. RA mainly verifies whether the program behavior with reentrancy attacks is equal to behavior in normal executions. The biggest advantage of this tool is that it completely eliminates false positives and negatives.

Samreen et al. [31] present a semi-automated framework that targets reentrancy vulnerabilities in Ethereum smart contracts, which combines static and dynamic analyses to detect bugs at runtime. In addition, deep learning technology has also been utilized for automatically discovering potential bugs. For example, Qian et al. propose a sequential model [32], namely bidirectional long-short term memory with attention, for the precise detection task of reentrancy bugs, which promotes the future research interests in this field.

3.2. Formal Verification of Smart Contract

Apart from the above vulnerability detection methods, there are many formal verification methods, among which theorem proving and model checking are the most widely used in the era of smart contracts. Bai et al. [33] proposed a formal verification framework for an intelligent shopping scenario. They used Promel to model the intelligent shopping line and SPIN was for testing the model, verifying the effect of formal methods on smart contracts. Qu et al. adopted the communication Sequence Process (CSP) theory to formally model concurrent programs [34]. They provided a framework for using CSP and FDR to check smart contract vulnerabilities, and successfully detected concurrent vulnerabilities in Ethereum public smart contracts. Amani et al. proposed a method for verifying Ethereum smart contracts at the EVM bytecode level [35] and extended the existing EVM formalization in Isabelle/HOL through sound program logic. Sun et al. used formal methods to detect security problems in smart contracts [36]. They summarized five types of security problems in smart contracts and proposed corresponding formal verification methods for each problem.

Kalra et al. proposed a framework called ZEUS [37] to verify the correctness and fairness of smart contracts using abstract interpretation and symbolic model checking, as well as powerful constrained Horn clauses. Park et al. presented a formal verification

tool UPPAAL [38] to model and verify the Dutch auction trading system based on smart contracts, and set time constraints for each template and state in the modeling process. Dharanikota S et al. presented CELESTIAL [39], an open-source framework that allows programmers to translates the contracts and the specifications to F*, to formally verify real-world Ethereum smart contracts from different application domains. Petri nets, a mature formal modeling method, are gradually applied to the behavior description and expression of smart contracts. Aiming at the satisfaction analysis of behavior attribute of smart contracts, Liu et al. [40] modeled, analyzed and verified the normal execution of smart contract and the process of attacking the contract with CPN Tools. Duo W et al. [41] introduced a formal analysis method of Ethereum smart contracts based on CPN, which improved the existing bytecode program logic, and defined more complete Hoare pre- and post-conditions, thereby making it suitable for CPN modelling. Garfatta et al. [42] proposed a CPN-based formal verification method for Solidity smart contracts. The main idea behind this approach is to transform the Solidity smart contracts to CPN and verified contract-specific properties. Thereafter, ref. [43] continued to extend their work with respect to two aspects. One is taking the concept of function calls in the transformation into consideration. The other is to define the correctness properties of smart contracts through Linear Temporal Logic formalization. Dwivedi et al. [44] believe that the performance of conventional contracts is often slow and expensive, and hard to enforce. Machine-readable smart contracts on blockchain promise to improve collaboration efficiency and effectiveness, but lack legal relevance. To this end, they develop the smart-legal-contract ontology to define the ontological concepts of rights and obligations for smart contracts. CPN Tools are utilized to design, develop and analyze processing states of smart contracts to trace the performance of contractual rights and obligations. In another study [45], Dwivedi et al. identify the critical smart contract language properties to make smart contracts legally binding by conducting a systematic literature review, further promoting the development of legally enforceable smart contracts and smart contract languages.

4. Property Specification and Model Building

The essence of reentrancy is that the involvement of an attack leads to inconsistency in the data state of the entire transaction process, violating the relevant properties of the transaction such as transaction integrity and fund consistency. Therefore, we exploit set D to describe the data elements used in the transaction process. It is also a collection of token types. For some data elements that play a key role in the attack process such as user's contract balance, public wallet balance, etc., a key token type is established in the model, which is recorded as S. Predicates are assigned to transitions that characterize validation conditions in a transaction.

In this section, we establish the property specification according to a simplified DAO contract, and exploit CPN Tools to conduct top-down hierarchical modeling, including the overall model and attack-free model, as well as the attacker model based on the transaction attack behavior.

4.1. Attribute Specifications

By analyzing the contents of the contract, it is stipulated that the contract needs to meet the following conventions.

(1) Investors must ensure sufficient Ether when exchanging Ether for DAO tokens, and can withdraw at any time without affecting the operation of the contract;
(2) Investors can only perform one of the two operations of deposit (ether to DAO coin) or withdraw (DAO coin to Ether) at a time;
(3) When investors apply for deposit and withdrawal, the two balances of investors and the public wallet of crowdfunding should be updated in real time;
(4) When the DAO coin balance in the investor's contract account is insufficient, the investor cannot apply to use the DAO coin to exchange Ether;

(5) Before and after the investor conducts any transaction, the investor's DAO coin balance and the sum of the Ether balance after conversion should be consistent.

If the DAO satisfies the above stipulation conditions during the execution process, it proves that the contract is safe.

4.2. Top-Lever Model

In order to describe the execution process of the overall model and explain the related results, we list the relevant definitions in Table 1. The color sets and variables used in the model are defined as follows:

```
colset Account = INT;
colset Money = INT;
colset Bank = INT;
colset Cfd = INT;
colset Gas = INT;
colset Take = INT;
colset Number = INT;
colset Hacker = INT;
var userbal:INT;
var b,c:INT;
var m,g:INT;
var count:INT.
```

Table 1. CPN Model Definitions.

Type	Annotation
P[place]	Input data flow places
T[transition]	Input control flow transitions
CS[colset]	Color set name
IM[place]	Initial place value
M[place]	Local place marking value
ARC[node1,node2]	Bound transmission value on the arc, node1, node2 as input and output node

Based on CPN Tools, the top-level model of the DAO is shown in Figure 5, which includes three places and two alternative transitions. It should be noted that the conversion between Ether and DAO coins is not only considered in the process of model construction, and the numbers are blurred to represent Ether and DAO coins.

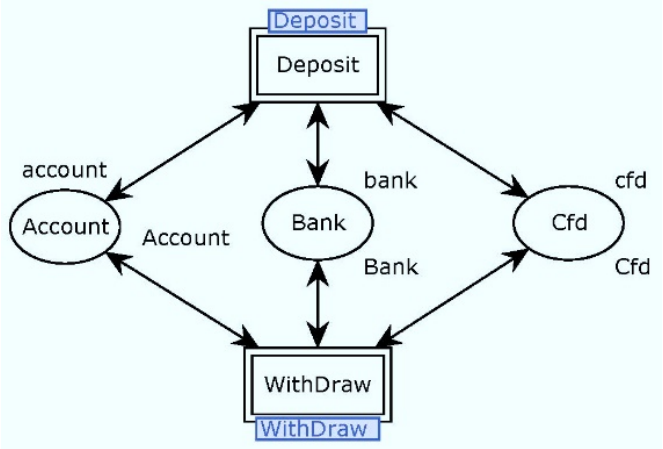


Figure 5. The top-level model of the contract.

In Figure 3, Account represents the investor account involved in crowdfunding, Bank represents the investor contract account and *Cfd* represents the total contract account of the crowdfunding project.

T[Deposit] and T[WithDraw] both are substitution transitions and represent the deposit operation and withdrawal operation, respectively. IM[Account] = *{account = 6}*, IM[Bank] = *{bank = 0}*, IM[Cfd] = *{Cfd = 25}*.

4.3. Attack-Free Model

The model without attack is shown in Figures 6 and 7. They are the lower implementations of the top-level model that replace the transitions deposit and withdraw. In the Deposit layer, the model contains 8 places and 3 transitions, while in the WithDraw layer, it contains 14 banks and 3 transitions. The relevant information of data stored by nodes in Figure 4 of the underlying model is explained as follows. The names of transitions and places in Figure 5 are similar to those in Table 2. To save space, we will not explain them one by one here.

In CPN Tools, we assign initial values to the model, such that P[Account]:1‘6, P[Bank]:1‘0, P[Cfd]:1‘25, i.e., IM[Account] = {account = 6}, M[Bank] = {bank = 0}, [Cfd] = {Cfd = 25}, IM[Gas_Need] = {gas = 1}. Since the data in the place are single, they can be abbreviated as IM[Account] = 6, IM[Bank] = 0, IM[Cfd] = 25. From the initial value, we observe the execution process of the DAO model without attack. The state sequence in the running process of the model is represented by Sn (n = 1, 2, 3, . . .). Firstly, the running process of the Deposit layer of the simulation model is as follows.

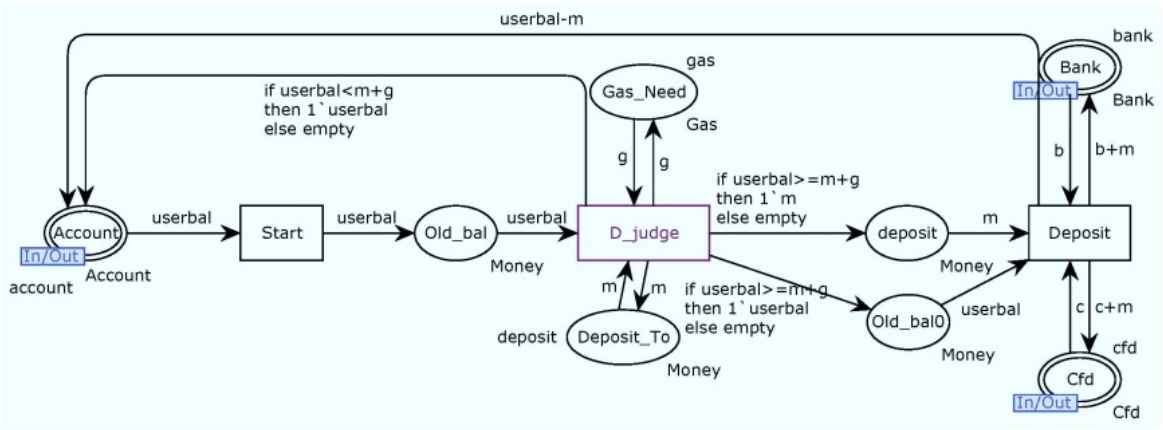


Figure 6. Deposit Layer.

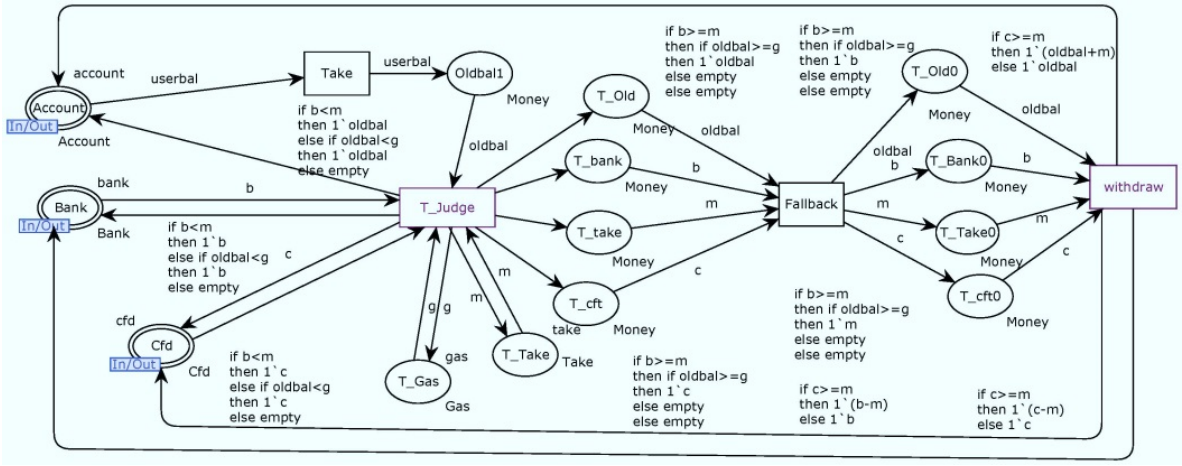


Figure 7. WithDraw Layer.

Table 2. Part Nodes Definitions.

Type	Annotation
P[Account]	Investor Account
P[Gas_Need]	Consumed gas
P[Deposit_To]	Plan to deposit balance
P[deposit]	Deposit balance
P[Old_bal0]	Investor account original balance
P[Bank]	Investor account balance
P[Cfd]	Crowdfunding balance

① S_1 [Start > S_2.

ARC < Account, start >= *{userbal = 6}*, T[Start] is enabled. The investor sends a transaction request, and the state of the model changes from S_1 to S_2 after T[Start] occurs, and M[Old_bal] = 6, M[Account] = 0.

② S_2 [D_Judge > S_3.

ARC < Old_bal, D_judge >= {*userbal = 10*}, ARC < Gas_Need, D_judge >= {g = 1}, ARC < Deposit_To, D_Judge >= {*deposit = 4*}, T[D_judge] is fired. Now the state of the model is transformed from S_2 to S_3. Then, M[deposit] = 5, M[Old_bal0] = 6, M[Gas_Need] = 1, M[Deposit_To] = 5, M[Old_bal] = 0.

③ S_3 [Deposit > S_4.

ARC < deposit, Deposit >= {*m = 5*}, ARC < Old_bal0, Deposit >= {*userball = 6*}, ARC < Bank, Deposit >= {*b = 0*}, ARC < Cfd, Deposit >= {*c = 25*}. T[Deposit] is enabled. At this time, M[Account] = 1, M[Bank] is bonded to 5 and M[Cfd] = 30. After that, no transition can be triggered and the model finishes running.

Next, the WithDraw layer is executed by the simulation tool, and the initial state of the model is IM[Account] = 6, IM[Bank] = 0, IM[Cfd] = 25, IM[T_Gas] = 1. The execution process is as follows.

① S_1 [Start > S_5.

ARC < Account, Start >= {*account = 6*}, T[Take] is fired. The investor sends a transaction request, and the state of the model changes from S_1 to S_5, and M[Old_bal1] = 10.

② S_5 [T_Judge > S_6.

ARC < Old_bal1, T_judge >= {oldbal = 1}, ARC < T_Gas, T_judge >= {g = 1}, ARC < T_Take, T_Judge >= {m = 4}, ARC < Bank, T_Judge >= {b = 5}, ARC < Cfd,T_Judge >= {c = 25}. When the transition condition is satisfied, T[D_judge] is triggered to fire, and the state of the model is converted from S_5 to S_6. At this time, M[T_Gas] = 1, M[T_Take] = 5. M[T_Old] = 1, M[T_bank] = 5, M[T_take], M[T_cft] = 30.

③ S_6 [Fallback > S_7.

ARC < T_Old, Fallback >= {oldbal = 1}, ARC < T_bank, Fallback >= {b = 5}, ARC < T_Take, Fallback >= {m = 5}, ARC < T_cft, Fallback>= {c = 30}. T[Fallback] is enabled, then the state changes from S_7 to S_8. In addition, M[T_Old0] = 1, M[T_bank0] = 5, M[T_Take0] = 5, M[T_cft0] = 30.

④ S_7 [withdraw > S_8.

ARC < T_Old0, withdraw >= {oldbal = 1}, ARC < T_bank0, withdraw >= {b = 5}, ARC < T_Take0, withdraw >= {m = 5}, ARC < T_cft0, withdraw >= {c = 25}. At this point, T[withdraw] is triggered and the model state is converted from S_8 to S_9. M[Account] is bound to 6, M[Bank] is bound to 0 and M[Cfd] is bound to 25. At this time, the currency withdrawal operation is normal, and the model operation runs to completion.

4.4. Attacker Model

Since the attack mainly targets the WithDraw layer, we simulate the attacker's behavior in Section 4.3 to model the attack. This is done to analyze whether the model can have reentrancy vulnerabilities. Figure 6 shows the attack layer. Based on the WithDraw layer model above, the behavior of the attacker is added to the WithDraw layer of the original model (Figure 8). Due to the concurrency of the Petri net model, we only consider the execution of the model after adding the Attack transition by adding the suppression arc and the place P[Hacker]. The result is the WithDraw layer of the attack model (Figure 9). The attack layer model is then merged with the new WithDraw layer to form the DAO attack model. When the initial state value is assigned, the malicious trading behavior of the attacker can be simulated. In the simulation tool, the initial state of the contract is set as follows.

- P[Account]:1'6
- P[Bank]:1'0
- P[Cfd]:1'25
- P[Gas_Need]1'1
- P[Deposit_To]:1'5
- P[T_Gas]:1'1
- P[T_Take]:1'5

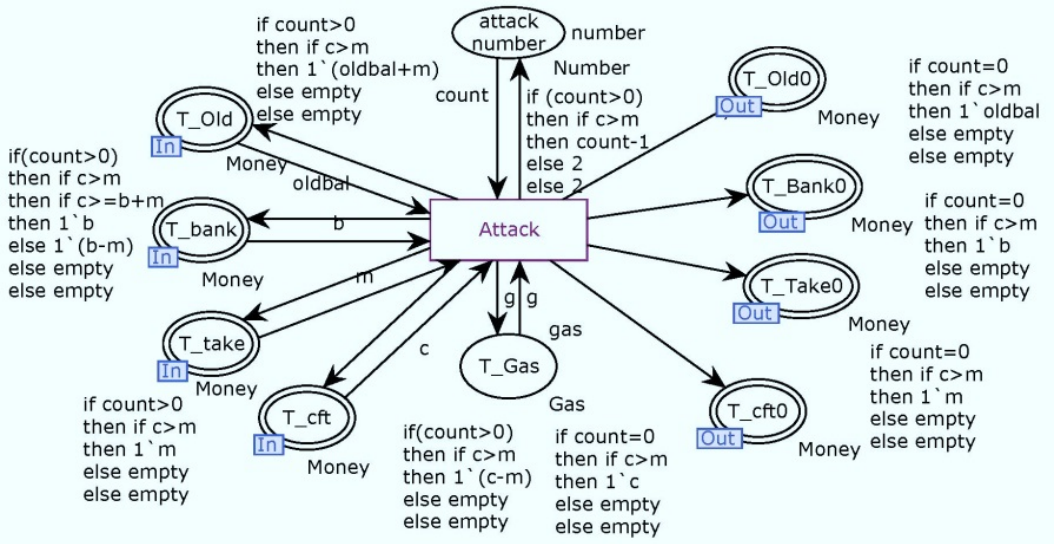


Figure 8. Attack Layer Model.

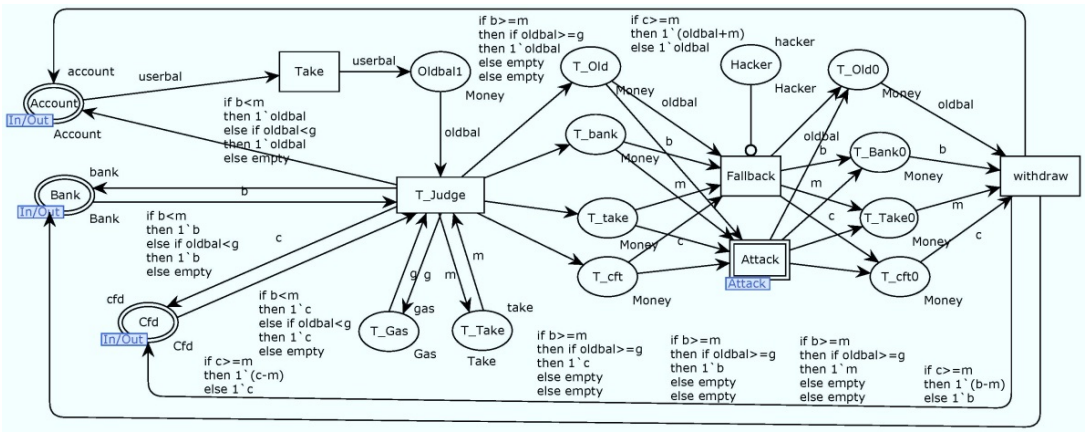


Figure 9. WithDraw Layer with Attack.

The attack process of the new model is simulated as follows.

① S_6 [Attack > S_7′

ARC < T_Old, Attack >= {*oldbal* = 1}, ARC < T_bank, Attack >= {*b* = 5}, ARC < T_Take, Attack >= {*m* = 5}, ARC < T_cft, Attack >= {*c* = 30}. At this point, T[Attack] is fired, and the state of the model is changed from S_6 to S_7′. The Attack transition is set to be executed twice to simplify the simulation execution process. M[Account] is bound to 11 and M[T_Cfd] is bound to 20.

② S_7′ [withdraw > S_8′

When the number of attacks is reduced to 0, the contract returns to the WithDraw layer for normal operation. At this time, T[withdraw] is executed, and M[Account] is finally updated to 16, while M[Bank] and M[Cft] are both updated to 0 and 15, respectively.

5. Experimental Analysis of Reentrancy Model

5.1. Non-Secure State Reachability Analysis

The principle of matrix-based reachability analysis is to establish a correlation matrix. Given the vector M_0 representing the initial state and the unsafe state of CPN represented by a non-negative integer vector M, they are then substituted into the state equation to be solved. If there is no solution, it means that the state is safe. In contrast, if there is a solution, it means that there is a security loophole in the contract, and the sequence of attack transitions can be determined.

The association matrix of the CPN model is shown in Tables 3–5, which are the submatrix M_1 Deposit layer, submatrix M_2 of the WithDraw layer and the submatrix M_3 of the Attack layer, respectively.

Table 3. Submatrix M_1 of Incidence Matrix.

	Old_bal	**Gas_Need**	**Deposit_To**	**Deposit**	**Old_bal0**	**Account**	**Bank**	**Cfd**
Start	Money							
D_Judge		-Gas	-Money	Money	Money			
Deposit						Account	Bank	Cfd

Table 4. Submatrix M_2 of Incidence Matrix.

	Oldbal1	**T_Gas**	**T_Take**	**T_Old**	…	**T_Old0**	**T_Bank0**	**T_Take0**	**T_cft0**
Take	Money								
T_Judge	-Money	-Gas	-Take	Money					
Fallback				-Money		Money	Money	Money	Money
withdraw						-Money	-Money	-Money	-Money

Table 5. Submatrix M_3 of Incidence Matrix.

	T_Old	**T_bank**	**T_take**	**T_cft**	…	**T_cft0**	**Account**	**Bank**	**Cfd**
Fallback									
Attack	-Money	-Money	-Money	Money		Money			
withdraw							Account	Bank	Cfd

The initial state of the contract is M_0 = [0, Gas, Money, 0,0, Money, Money, Money, 0, Gas, Take, 0, 0 . . . , 0]. We assume that an unsafe state is M = [0, Gas, Money, 0, 0, 0 . . . , Hacker, Attack_number, Money, Money, Money], indicating the data abnormality caused by a successful attack of a hacker. M and M_0 are substituted into the state equation $M = M_0 + A^{\mathrm{T}}X$. To solve the state equation, we can use the solution identification formula of the linear function to determine that the equation has a solution. From this, we obtain

an executable transition sequence: σ = T[Start] T[D_Judge] T[Deposit] T[Take] T[T_Judge] T[Attack] T[withdraw]. It can be concluded that the execution of the contract can reach this unsafe state; that is, there is at least one vulnerability in the smart contract that the attacker can exploit to steal illegal assets without hindrance.

5.2. CPN Tools State Space Analysis

In the fourth part of our study, we adopted CPN Tools to model the whole DAO contract, non-attack operation and attack operation. By observing the results before and after executing the model, we extracted the changes identified via Account, Bank and Cfd from the generated state space and summarized them, as shown in Tables 6–8.

Table 6. Account, Bank, Cfd Identification Change at Normal Deposit Operation Without Attack.

Deposit Layer	Account	Bank	Cfd
Before Execution	6	0	25
After Execution	1	5	30

Table 7. Account, Bank, Cfd Identification Change at Normal WithDraw Operation Without Attack.

WithDraw Layer	Account	Bank	Cfd
Before Execution	1	5	30
After Execution	6	0	25

Table 8. Account, Bank, Cfd Identification Change With Attack.

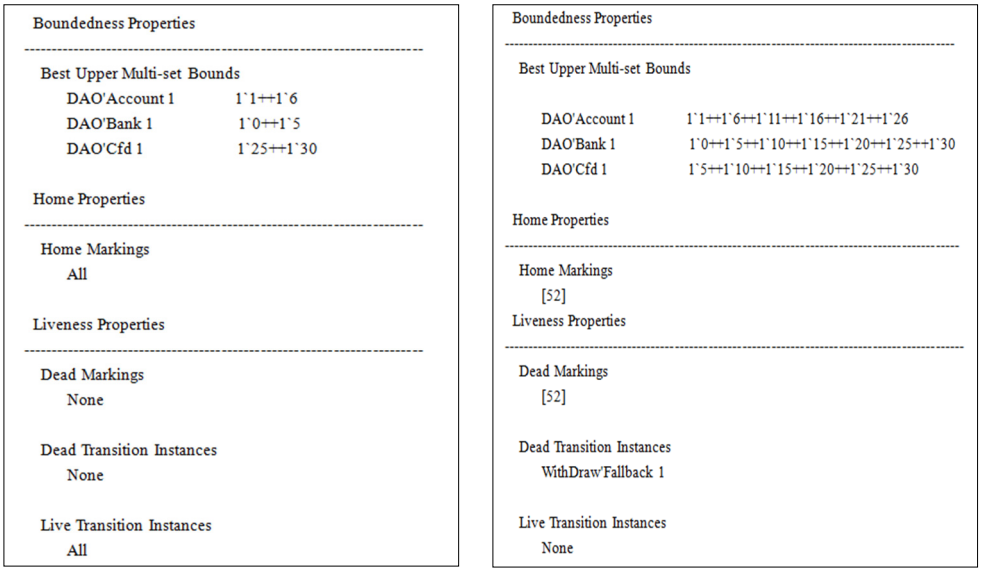
Attack Model	Account	Bank	Cfd
Before Execution	6	0	25
During Execution	1	5	30
After Execution	16	0	15

It can be seen from Tables 6 and 7 that in the no-attack mode, the contract is executed normally, and the property Specifications (1)–(4) are satisfied. However, when the contract is attacked, the user can perform deposit operations without any exception, but when the user performs withdrawal operations (Table 8), it can be found that the Account balance is 16 and the Cfd Account balance is 15. This indicates that during the execution process, the malicious attacker not only withdraws the 5 Ether from the Account he deposited in the contract, but also steals an additional 10 Ether from the crowdfunded contract wallet. As a result, the investor's balance, the investor's contract account balance and the public wallet of the crowdfunding did not change in real time according to the attribute specification. Moreover, after any transaction by the investor, the sum of the investor's token balance and is not consistent with that of the Ether balance, violating attribute Specifications (3) and (5). Hence, the contract is in an insecure state.

5.3. CPN Tools Status Report Analysis

Using CPN Tools to generate partial status state reports of the non-attack model and the attack model, which are shown in (a) and (b) of Figure 10.

In the attack-free mode, all markings are home states. That is to say, starting from the current marking along any path, one can return to this marking, which denotes that the smart contract works well. At the same time, based on the SML language, ASK-CTL is used to verify whether the initial marking satisfies the property of home state. The execution language and results are shown in Figure 11, where val it=true:bool indicates that the judgment is true, i.e., the initial identifier is a home state. Figure 10a shows that the model without attack satisfies the property of activity without deadlock and dead transition, which means that all the identifiers in the model can generate the next identifier, and there are no unactivated transitions. The contract Specifications (1)–(5) are satisfied.



(**a**) (**b**)

Figure 10. Status Space Report. (**a**) Attack-free model; (**b**) Attack Model.

```
val IsInitialMarking = fn : Node -> bool
val myASKCTLformula =
  NOT (EXIST_UNTIL (TT,NOT (EXIST_UNTIL (TT,NF ("initial marking",fn))))) : A
val it = true : bool

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

Figure 11. Home Marking ASK-CTL without Attack.

When there is an attack behavior, as can be seen from Figure 10b the identifier [52] is both a home state identifier and a dead identifier. This is because the presence of malicious attackers makes Fallback become a dead transition. ASK-CTL was used to analyze the initial marking and the marking [52], respectively. According to the judgment result in Figure 12, it can be seen that there is no dead transition in the attack model. The initial identification is no longer a home state, and the marking [52] is dead. This leads to an abnormal data balance and the contract being unable to operate continuously, thereby violating the property Specification (1).

```
val IsInitialMarking = fn : Node -> bool
val myASKCTLformula =
  NOT (EXIST_UNTIL (TT,NOT (EXIST_UNTIL (TT,NF ("initial marking",fn))))) : A
val it = false : bool

fun IsInitialMarking n = (n=InitNode) ;
val myASKCTLformula =INV(POS(NF("initial marking",IsInitialMarking))) ;
eval_node myASKCTLformula InitNode;
val myASKCTLformula = NOT (MODAL TT) : A
val it = true : bool

val myASKCTLformula = NOT(MODAL(TT)) ;
eval_node myASKCTLformula 52;
```

Figure 12. ASK-CTL of Home marking and dead transition.

5.4. CPN Tools State Diagram Analysis

Based on the CPN Tools simulation tool, the state space is generated. Through observation, it can be seen that there are abnormal data transactions in the state space. Taking the user Account as an example, a series of abnormal transaction values such as 11, 16, 21, 26 appear during the execution of the smart contract. Starting from the first marking M_{10} of the value 11, we can trace a counter-example path, as shown in Figure 13. According to the expanded state of marking M_9 and M_{10}, the balance in T'Old (i.e., Account account) changed during the process from M_9 to M_{10}, while the Bank account balance is always 5. From this, we can determine that an exception has occurred, thus violating property Specifications (3) and (5). Furthermore, the logical defect in the TheDAO contract can also been located from the location of the exception.

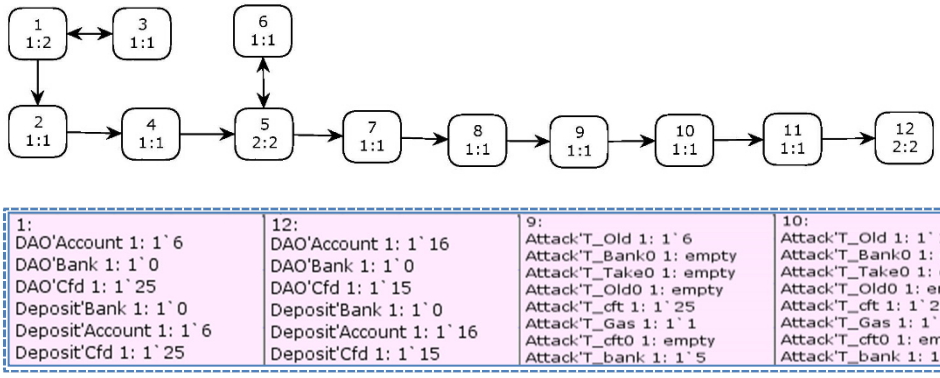


Figure 13. Local abnormal graph path and state details.

5.5. Conclusions Verification Based on the Remix Platform

In fact, if there are enough attacks, the attacker can transfer the entire balance of the crowdfunding contract account. In order to confirm the correctness of the conclusion, we conducted an experiment through the Remix platform [46]. First we used an account (0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c) to play the victim. Then we clicked the "Deploy" button to deploy the contract in Remix IDE and deposited some Ether. Here, we put 25 Ether into the contract wallet. At this time, we could find that the balance of the contract was 25 Ether by clicking the wallet, indicating that the deposit operation was successful, as shown in Figure 14.

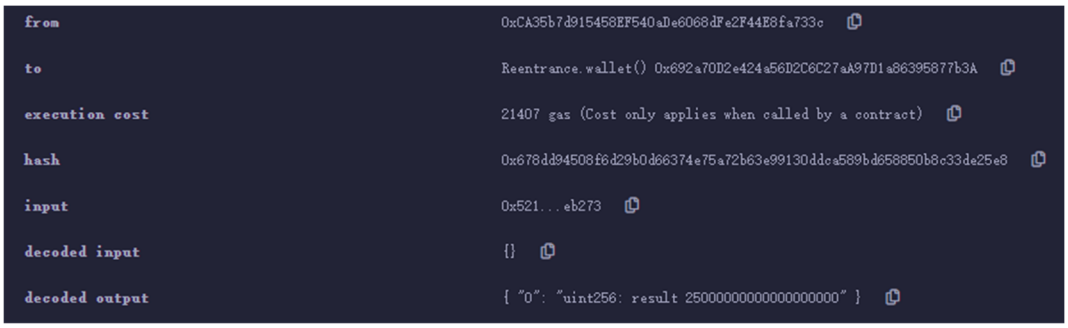


Figure 14. Execution Process.

Next, we use another account (0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db) to act as an attacker, deploy the contract and store 5 Ether in the victim's wallet. At this time, as we can see from Figure 15, the account of the victim's contract becomes 30 Ether.

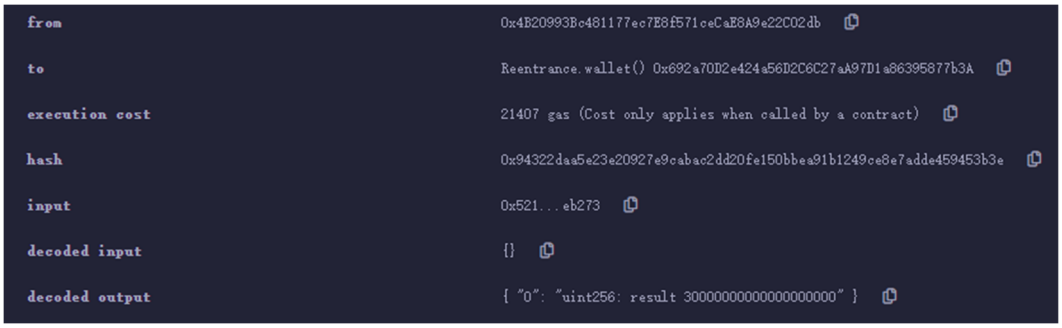
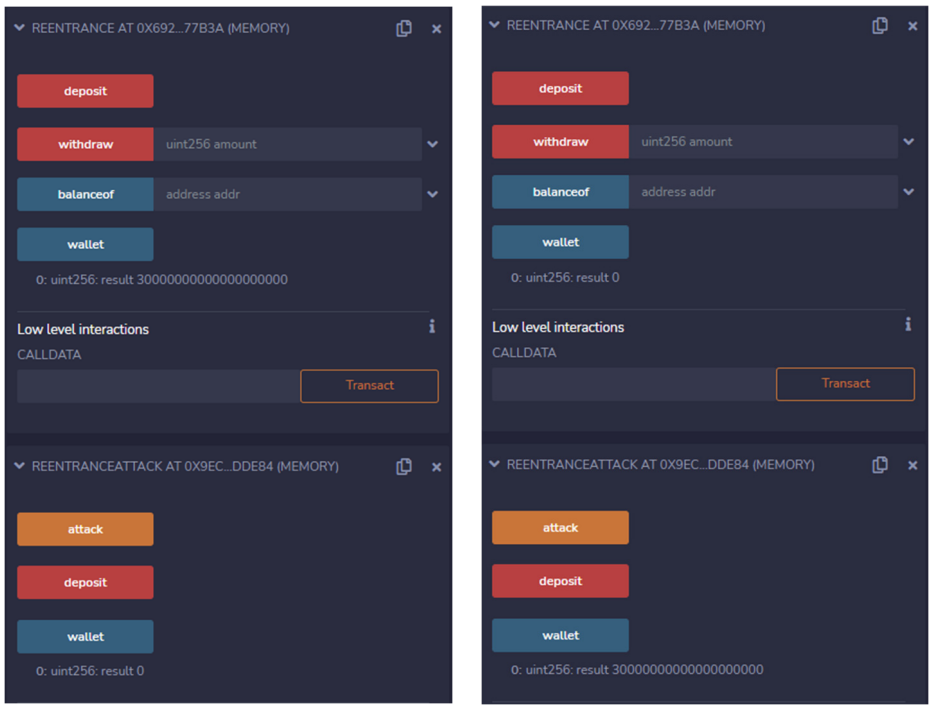


Figure 15. Execution Process.

The attacker calls the attack function to simulate an attack and then calls the wallet function of the attacked contract to check the balance of the contract; we can find that the balance is zero, which indicates 30 Ether in the attacked contract have been transferred to the attacker's contract. The result is shown in Figure 16, which proves that the contract does have vulnerabilities that can be attacked. In addition, it also further shows that CPN modeling can correctly verify the potential reentrancy vulnerability of smart contracts. However, we primarily focused on the reentrancy bug in this paper. In fact, there are many other potential vulnerabilities in smart contracts that can pose a significant security risk.



(**a**) (**b**)

Figure 16. Remix Simulation Results. (**a**) Contract Account; (**b**) Attacker Account.

5.6. Comparison with State-of-the-Art Methods

In this section, we selected 50 smart contracts from the SBcurated [47] dataset, which consists of real-world contracts with vulnerabilities and specially constructed contracts without vulnerabilities. We analyze all these contracts to determine whether they suffer reentrancy vulnerabilities. Finally, the results are shown in Table 9. We selected four metrics to compare the performance of analysis tools. Specifically, they refer to:

- TP (True Positive): the number of actual vulnerabilities that are correctly identified by the tool.

- FP (False Positive): the number of non-existent vulnerabilities that are incorrectly identified by the tool.
- FN (False Negative): the number of actual vulnerabilities that are missed by the tool.
- TN (True Negative): the number of non-existent vulnerabilities that are correctly identified by the tool.

Table 9. Comparison with other state-of-the-art method.

Tool	TP	FP	FN	TN	Accuracy	TPR	FPR
Oyente	20	5	10	15	70%	66.7%	25%
Mythril	12	6	18	14	52%	40%	33.3%
Slither	10	10	12	18	56%	31.2%	35.7%
CPN	35	0	0	15	100%	100%	100%

In general, the higher the true positive rate (TPR) and the lower false positive rate (FPR), the better the tool is. TPR = TP/ (TP + FN), and FPR = FP/ (FP + TN). As Table 9 shows, the CPN-based Tool has higher TPR than Oyente, Mythril and Slither up to 100%, while obtaining no FPs. Mythril and Slither obtain similar FPR. By comparison, we can see that our CPN Tool can detect all the reentrancy vulnerabilities without FPs, while the other three methods have both false positives and false negatives. However, the drawback of the CPN-based approach lies in the requirement of human involvement in modeling and analysis, as well as the immaturity of automated modeling techniques.

6. Conclusions and Future Work

With the popularization of blockchain platform-level applications, the amount of money involved in smart contracts is growing exponentially. However, smart contracts face many security threats due to their program characteristics and special execution environment. This paper proposes a formal verification method for smart contracts based on CPN. Firstly, the concept of data flow and control flow is proposed to better reflect the change of data state in the process of transaction execution. The simplified DAO contract is modeled hierarchically and the malicious behavior of attackers is reproduced via the simulation tool CPN Tools. Finally, the potential transaction vulnerabilities in the contract are successfully verified using analysis methods such as state space.

As a class of digital contracts, smart contracts are driven by data participation and have certain legal effects. A well-constructed smart contract behavior model must be able to meet the different needs of participants in terms of operational functions and data status. Data behavior that violates the data state requirements can reflect the defects of the smart contract. The more accurate the data attributes contained in the contract are, the more accurately the contract reliability can be judged based on the data state. In combination with our current work, we intend to improve previously proposed WFDC-nets [48,49] and define a novel Colored Petri net with Data Constraints (CFDC-net). The WFDC-net model for workflow analysis is extended to a general model that can be applied to a wide range of business processes and the different data attributes contained in the smart contract are formally defined. Then, we explore the refinement model of multi-party and multiple transaction behaviors to provide a model basis for behavior analysis, ensuring that the actual execution behavior of smart contracts is consistent with its data specification.

In summary, formal verification of smart contracts based on Petri nets is of great significance. In the future, we will continue to carry out research on Petri nets aiming at security and optimization of smart contracts.

Author Contributions: Conceptualization, H.W. and Y.H.; methodology, H.D.; software, H.D.; validation, H.D. and Y.H.; formal analysis, H.D.; writing—original draft preparation, Y.H.; writing—review and editing, H.W.; visualization, H.D; supervision, Q.D. and H.W.; project administration, H.W.; funding acquisition, H.W. All authors have read and agreed to the published version of the manuscript.

Funding: The work is supported by Major Public Welfare Projects Foundation of Henan Province (201300210200) and Open Foundation of Henna Key Laboratory Cryptography (LNCT2020-A08).

Data Availability Statement: https://github.com/coderlrj/Reetrancy-CPN-Model, 3 May 2023.

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

References

1. Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Bus. Rev.* **2018**, *1*, 21260.
2. Crosby, M.; Pattanayak, P.; Verma, S.; Kalyanaraman, V. Blockchain technology: Beyond bitcoin. *Appl. Innov.* **2016**, *2*, 71.
3. Buterin, V. A next-generation smart contract and decentralized application platform. *White Pap.* **2014**, *3*, 2–11.
4. Blass, E.O.; Kerschbaum, F. Strain: A secure auction for blockchains. In *European Symposium on Research in Computer Security*; Springer: Cham, Switzerland, 2018; pp. 87–110.
5. Zou, W.; Lo, D.; Kochhar, P.S.; Dinh Le, X.-B.; Xia, X.; Feng, Y.; Chen, Z.; Xu, B. Smart contract development: Challenges and opportunities. *IEEE Trans. Softw. Eng.* **2019**, *47*, 2084–2106. [CrossRef]
6. Wang, S.; Ouyang, L.; Yuan, Y.; Ni, X.; Han, X.; Wang, F.-Y. Blockchain-enabled smart contracts: Architecture, applications, and future trends. *IEEE Trans. Syst. Man Cybern. Syst.* **2019**, *49*, 2266–2277. [CrossRef]
7. Szabo, N. Formalizing and securing relationships on public networks. *First Monday* **1997**, *2*, 9. [CrossRef]
8. Szabo, N. Smart contracts: Building blocks for digital markets. *EXTROPY J. Transhumanist Thought* **1996**, *18*, 28.
9. Finding the greedy, prodigal, and suicidal contracts at scale. In Proceedings of the 34th Annual Computer Security Applications Conference, San Juan, PR, USA, 3–7 December 2018; Volume 1, pp. 653–663.
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.* **2019**, *21*, 19–32. [CrossRef]
11. Usman, T.A.; Selçuk, A.A.; Özarslan, S. An Analysis of Ethereum Smart Contract Vulnerabilities. In Proceedings of the 2021 International Conference on Information Security and Cryptology (ISCTURKEY), Ankara, Turkey, 2–3 December 2021; pp. 99–104.
12. Wang, Y.; Gou, G.; Liu, C.; Cui, M.; Li, Z.; Xiong, G. Survey of security supervision on blockchain from the perspective of technology. *J. Inf. Secur. Appl.* **2021**, *60*, 102859–102871. [CrossRef]
13. Ferreira Torres, C.; Iannillo, A.K.; Gervais, A. The eye of horus: Spotting and analyzing attacks on ethereum smart contracts. In *International Conference on Financial Cryptography and Data Security*; Springer: Berlin/Heidelberg, Germany, 2021; pp. 33–52.
14. Liu, Z.; Qian, P.; Wang, X.; Zhuang, Y.; Qiu, L.; Wang, X. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Trans. Knowl. Data Eng.* **2021**, *35*, 1296–1310. [CrossRef]
15. Gehlot, V.; Nigro, C. An introduction to systems modeling and simulation with colored petri nets. In Proceedings of the 2010 Winter Simulation Conference, Baltimore, MD, USA, 5–8 December 2010; pp. 104–118.
16. Tolmach, P.; Li, Y.; Lin, S.W.; Liu, Y.; Li, Z. A survey of smart contract formal specification and verification. *ACM Comput. Surv.* **2021**, *54*, 1–38. [CrossRef]
17. Pierro, G.A.; Tonelli, R. Paso: A web-based parser for solidity language analysis. In Proceedings of the 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE), London, ON, Canada, 18 February 2020; pp. 16–21.
18. Schneidewind, C.; Grishchenko, I.; Scherer, M.; Maffei, M. eThor: Practical and provably sound static analysis of ethereum smart contracts. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual, 9–13 November 2020; pp. 621–640.
19. Peterson, J.L. Petri nets. *ACM Comput. Surv.* **1977**, *9*, 223–252. [CrossRef]
20. Jensen, K.; Kristensen, L.M. Colored Petri nets: A graphical language for formal modeling and validation of concurrent systems. *Commun. ACM* **2015**, *58*, 61–70. [CrossRef]
21. Jensen, K.; Christensen, S.; Kristensen, L.M. CPN tools state space manual. In *Department of Computer Science*; University of Aarhus: Aarhus, Denmark, 2006.
22. Jensen, K. *Colored Petri Nets: Basic Concepts, Analysis Methods and Practical Use*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 1996.
23. DuPont, Q. Experiments in algorithmic governance: A history and ethnography of "The DAO," a failed decentralized autonomous organization. In *Bitcoin and Beyond: Cryptocurrencies, Blockchains and Global Governance*; Routledge: London, UK, 2017; pp. 157–177. [CrossRef]
24. Jentzsch, C. Decentralized Autonomous Organization to Automate Governance. *White Pap.* **2016**. Available online: https://lawofthelevel.lexblogplatformthree.com/wp-content/uploads/sites/187/2017/07/WhitePaper-1.pdf (accessed on 3 February 2023).
25. Hard-Fork. 2020. Available online: https://www.investopedia.com/terms/h/hard-fork.asp (accessed on 28 January 2023).
26. Liu, C.; Liu, H.; Cao, Z.; Chen, Z.; Chen, B.; Roscoe, B. Reguard: Finding reentrancy bugs in smart contracts. In Proceedings of the IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), Gothenburg, Sweden, 27 May 2018; pp. 65–68.
27. Feng, X.; Wang, Q.; Zhu, X.; Wen, S. Bug searching in smart contract. *arXiv* **2019**, arXiv:1905.00799.
28. Rodler, M.; Li, W.; Karame, G.O.; Davi, L. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv* **2018**, arXiv:1812.05934.

29. Yu, R.; Shu, J.; Yan, D.; Jia, X. ReDetect: Reentrancy Vulnerability Detection in Smart Contracts with High Accuracy. In Proceedings of the 17th International Conference on Mobility, Sensing and Networking (MSN), Exeter, UK, 13–15 December 2021; pp. 412–419.
30. Ye, J.; Ma, M.; Lin, Y.; Xue, Y.; Sui, Y.; Peng, T. Clairvoyance: Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Virtual Event, Australia, 21–25 December 2020; pp. 274–275.
31. Chinen, Y.; Yanai, N.; Cruz, J.P.; Okamura, S. RA: Hunting for re-entrancy attacks in ethereum smart contracts via static analysis. In Proceedings of the IEEE International Conference on Blockchain (Blockchain), Rhodes, Greece, 2–6 November 2020; pp. 327–336.
32. Samreen, N.F.; Alalfi, M.H. Reentrancy vulnerability identification in ethereum smart contracts. In Proceedings of the IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE), London, ON, Canada, 18 February 2020; pp. 22–29.
33. Qian, P.; Liu, Z.; He, Q.; Zimmermann, R.; Wang, X. Towards automated reentrancy detection for smart contracts based on sequential models. *IEEE Access* **2020**, *8*, 19685–19695. [CrossRef]
34. Bai, X.; Cheng, Z.; Duan, Z.; Hu, K. Formal modeling and verification of smart contracts. In Proceedings of the 2018 7th International Conference on Software and Computer Applications, Kuantan, Malaysia, 8–10 February 2018; pp. 322–326.
35. Qu, M.; Huang, X.; Chen, X.; Wang, Y.; Ma, X.; Liu, D. *Formal Verification of Smart Contracts from the Perspective of Concurrency. International Conference on Smart Blockchain*; Springer: Cham, Switzerland, 2018; pp. 32–43.
36. 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, Los Angeles, CA, USA, 8–9 January 2018; pp. 66–77.
37. Sun, T.; Yu, W. A formal verification framework for security issues of blockchain smart contracts. *Electronics* **2020**, *9*, 255. [CrossRef]
38. Kalra, S.; Goel, S.; Dhawan, M.; Sharma, S. Zeus: Analyzing safety of smart contracts. *Ndss* **2018**, *1*, 1–12. [CrossRef]
39. Park, W.S.; Lee, H.; Choi, J.Y. Formal Modeling of Smart Contract-based Trading System. In Proceedings of the 2021 23rd International Conference on Advanced Communication Technology (ICACT), PyeongChang, Republic of Korea, 7–10 February 2021; pp. 48–52.
40. Dharanikota, S.; Mukherjee, S.; Bhardwaj, C.; Rastogi, A.; Lal, A. Celestial: A smart contracts verification framework. *Proc. 2021 Form. Methods Comput. Aided Des.* **2021**, *2*, 133–142.
41. Liu, Z.; Liu, J. Formal verification of blockchain smart contract based on colored petri net models. In Proceedings of the 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), Milwaukee, WI, USA, 15–19 July 2019; pp. 555–560.
42. Duo, W.; Xin, H.; Xiaofeng, M. Formal analysis of smart contract based on colored petri nets. *IEEE Intell. Syst.* **2020**, *35*, 19–30. [CrossRef]
43. Garfatta, I.; Klai, K.; Graïet, M.; Gaaloul, W. A Solidity-to-CPN Approach Towards Formal Verification of Smart Contracts. In Proceedings of the 2021 IEEE 30th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Bayonne, France, 27–29 October 2021; pp. 69–74.
44. Garfatta, I.; Klai, K.; Graïet, M.; Gaaloul, W. Model checking of vulnerabilities in smart contracts: A solidity-to-CPN approach. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, Virtual, 25–29 April 2022; pp. 316–325.
45. Dwivedi, V.; Norta, A. A legal-relationship establishment in smart contracts: Ontological semantics for programming-language development. In Proceedings of the Advances in Computing and Data Sciences: 5th International Conference (ICACDS), Nashik, India, 23–24 April 2021; pp. 660–676.
46. Dwivedi, V.; Pattanaik, V.; Deval, V.; Dixit, A.; Norta, A.; Draheim, D. Legally enforceable smart-contract languages: A systematic literature review. *ACM Comput. Surv.* **2021**, *54*, 1–34. [CrossRef]
47. Jain, S.M. *Introduction to Remix IDE. A Brief Introduction to Web3: Decentralized Web Fundamentals for App Development*; Apress: Berkeley, CA, USA, 2022; pp. 89–126.
48. Gupta, V.; Kumar, A.; Pathak, A. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. *J. King Saud Univ. -Comput. Inf. Sci.* **2020**, *1*, 273–282.
49. He, Y.; Liu, G.; Xiang, D.; Sun, J.; Yan, C.; Jiang, C. Verifying the Correctness of Workflow Systems Based on Workflow Net With Data Constraints. *IEEE Access* **2018**, *6*, 11412–11423. [CrossRef]