electronics

MDPI

Article

A Formal Verification Framework for Security Issues of Blockchain Smart Contracts

Tianyu Sun and Wensheng Yu *

Beijing Key Laboratory of Space-ground Interconnection and Convergence, School of Electronic Engineering, Beijing University of Posts and Telecommunications, Beijing 100876, China; stycyj@bupt.edu.cn

* Correspondence: wsyu@bupt.edu.cn; Tel.: +86-135-2121-7036

Received: 13 January 2020; Accepted: 3 February 2020; Published: 3 February 2020

Abstract: Blockchain technology has attracted more and more attention from academia and industry recently. Ethereum, which uses blockchain technology, is a distributed computing platform and operating system. Smart contracts are small programs deployed to the Ethereum blockchain for execution. Errors in smart contracts will lead to huge losses. Formal verification can provide a reliable guarantee for the security of blockchain smart contracts. In this paper, the formal method is applied to inspect the security issues of smart contracts. We summarize five kinds of security issues in smart contracts and present formal verification methods for these issues, thus establishing a formal verification framework that can effectively verify the security vulnerabilities of smart contracts. Furthermore, we present a complete formal verification of the Binance Coin (BNB) contract. It shows how to formally verify the above security issues based on the formal verification framework in a specific smart contract. All the proofs are checked formally using the Coq proof assistant in which contract model and specification are formalized. The formal work of this paper has a variety of essential applications, such as the verification of blockchain smart contracts, program verification, and the formal establishment of mathematical and computer theoretical foundations.

Keywords: blockchain; BNB; Coq proof assistant; Ethereum; formal verification; smart contracts

1. Introduction

Blockchain technology, which supports financial transactions in the Bitcoin system [1], is essentially a decentralized database. It provides a globally-consistent append-only ledger that does not rely on a central trusted authority. As the underlying technology of Bitcoin, a blockchain is a string of data blocks generated by cryptographic association. Each data block contains a batch of information about network transactions of Bitcoin. Blockchain technology has significant achievements in finance, politics, industry, and society, with potential use in legal, medical, and supply chain industries. Ethereum [2] is an open-source public blockchain platform that implements an internal Turing-complete scripting language called Solidity [3]. It provides a decentralized Ethereum Virtual Machine (EVM) to process point-to-point contracts through its unique encrypted currency, Ethereum (ETH). The Ethereum blockchain offers smart contracts [4], which are high-level programming abstractions that are compiled down to EVM bytecode. Smart contracts can deal with different business logic to make full use of the ability of the Ethereum blockchain. It makes the expansion of the blockchain stronger, which makes Ethereum the largest blockchain development platform at present. With a large number of applications, smart contracts have also exposed many security issues. In recent years, attacks against smart contracts have occurred frequently. Some attacks triggered by contract vulnerabilities have caused huge losses [4–7]. Therefore, the security of smart contracts is particularly important.

In recent years, with the rapid development of computer science, especially the emergence of interactive theorem proving tools like Coq [8,9], Isabelle/HOL [10], and so on, the formal verification

technology has made significant progress [11–15]. Formal verification technology has achieved remarkable achievements in both the formalization of mathematics and the certification of properties of programming languages. In 2005, the international computer experts Gonthier and Werner offered the successful machine proving of the famous "four-color theorem" using Coq [12]. After six years of hard work, Gonthier completed the machine verification of the "odd order theorem" in 2012 [13], which made Coq more and more popular in academia. Wiedijk pointed out that relevant research groups around the world have completed or plan to complete the formal proof of 100 famous mathematical theorems including Gödel incompleteness theorem, prime number theorem, Fermat last theorem, and so on [15]. In the aspect of program verification, Xavier Leroy developed CompCert C in 2008, which is a high-assurance compiler for almost all of the C language, programmed and proved correct in Coq [16]. In 2016, Zhong Shao and Ronghui Gu had successfully developed a practical concurrent OS kernel and verified its functional correctness in Coq [17]. Andrew W. Appel, Benjamin C. Pierce, Zhong Shao, and others launched the Deep Specification project, which focuses on the specification and verification of the full functional correctness of software and hardware in 2016 [18]. These results show that formal verification technology plays an essential role in the field of program verification.

The formal verification method is based on the mathematical model and reasoning. It is more rigorous and reliable. Therefore, the correctness verification of smart contracts based on the formal method can effectively validate whether a smart contract is credible and accurate. In this paper, we apply the formal verification method to the detection of security issues in smart contracts. The specific contributions of this paper are as follows. First, through a detailed study of smart contracts, we summarize five kinds of security issues in the blockchain smart contracts. These issues are integer overflow, the function specification issue, the invariant issue, the authority control issue, and the behavior of the specific function. Furthermore, we present a specific formal verification method for each of these five types of security issues. Second, we establish a formal verification framework to examine the correctness of smart contracts. This formal verification framework can be used in all smart contracts to verify the significant security issues that exist. It establishes a foundation for further development. Third, based on the proof assistant Coq, we present a complete formal verification of the Binance Coin (BNB) contract in combination with the above method and framework. This is its first formal verification in Coq to our knowledge. We built the mathematical model and function specification for the BNB contract. Finally, we completed the verification of the function specification and properties at the mathematical level, which sufficiently proves the security and reliability of the smart contract. It shows how to complete the formal verification of a specific blockchain smart contract.

The organization of the paper is as follows: In Section 2, we discuss some related work. In Section 3, a brief overview of blockchain technology, smart contracts, and the Coq theorem prover is given. In Section 4, formal verification methods for five kinds of security issues of blockchain smart contracts are presented. Moreover, a formal verification framework for smart contracts is built. In Section 5, a complete formal verification of the BNB contract is presented. The entire verification is divided into four parts. It should be pointed out that all proofs are verified in the Coq proof assistant. Finally, in Section 6, we draw conclusions and discuss some potential further work.

2. Related Work

There are many studies that provide security assurance for smart contracts in different ways. In the paper [19], Alharby and Moorsel analyzed the environmental security issues of smart contracts, especially the blockchain security. Torres, Schütte, and State [20] developed OSIRIS, a tool designed to scan integer vulnerabilities in Ethereum smart contracts. Bragagnolo et al. [21] presented a SmartInspect architecture to inspect smart contracts. It addresses the lack of inspectability of a deployed contract by analyzing the contract state using decompilation techniques. In addition to security assurance, the correctness verification of smart contracts is particularly important. Bartoletti and Pompianu [22] studied the two most popular platforms, Bitcoin and Ethereum, and present a programming model for smart contracts. In the paper [23], Coblenz designed the programming

language Obsidian to make smart contracts safer. It allows programmers to write the correct programs easier.

Because of the mathematical characteristics of formal verification, formal methods have received more and more attention to the security verification of smart contracts. It can more effectively and completely verify the correctness of smart contracts. In the paper [24], Magazzeni, McBurney, and Nash predict that the formal verification of smart contracts has excellent potential for development in the future. In recent years, many works try to solve the security problem of smart contracts using formal verification. For instance, Bhargavan, Delignat-Lavaud, Swamy, and others outline a framework to analyze and formally verify both the runtime safety and the functional correctness of Solidity contracts in F* [25]. Yoichi Hirai [26] completed the formal verification of Deed Contract in Ethereum name service using Isabelle/HOL. Then he defined the EVM, which enables interactive theorem provers to reason about Ethereum smart contracts in Lem [27]. Amani's work [28] presents an approach to the verification of smart contracts at the level of EVM bytecode. Grishchenko, Maffei, and Schneidewind [29] presented a complete small-step semantics of EVM bytecode and formalized a large fragment thereof in the F*. The SECBIT group completed the formal verification of the ERC20 contract framework in [30]. Yang and Lei [31] developed a definitional interpreter FEther, which supports hybrid symbolic executions of Ethereum smart-contract formal verifications in Coq.

Our present work refers to the formal method of the above-cited work, especially the proof of the ERC20 contract in SECBIT's work. We summarize the advantages and problems encountered in their formalization. Then we study all kinds of security issues of smart contracts and summarize five kinds of common security vulnerabilities. For each security vulnerability, we propose a specific formal verification scheme. Meanwhile, we develop a new formal verification framework for the security of blockchain smart contracts. Then, we apply this framework to the verification of the BNB contract.

3. Background

3.1. Blockchain Technology

Blockchain is a new application mode that combines some computer technologies, such as distributed data storage, peer-to-peer transmission, consensus mechanism, smart contracts, and encryption algorithm. In 2008, a person or group of people using the name Satoshi Nakamoto presented the concept of "blockchain" in the bitcoin white paper [1]. Blockchain is an essential concept of Bitcoin, which is essentially a decentralized distributed ledger. It can record transaction information efficiently and permanently. Blockchain technology, which mainly solves the problems of high cost, low efficiency, and insecure data in third parties, has the characteristics of decentralization, transparency, and immutability [7]. Currently, blockchain networks can be divided into public blockchains, private blockchains, hybrid blockchains, and consortium blockchains. Blockchain technology has been applied in the fields of finance, the Internet of Things, public services, supply chain, and so on. In recent years, it has received increasing attention.

As the underlying technology of Bitcoin, a blockchain is a string of data blocks generated using cryptographic methods. As shown in Figure 1, in the blockchain, each block contains a cryptographic hash of the previous block, a timestamp, a nonce, and a Merkle root which consists of transaction data. The hash of the previous block links two blocks to form a blockchain. The timestamp ensures the security of the blockchain because no one can change the timestamp once it is recorded. A nonce is an arbitrary number that can be used just once in the cryptographic communication. It is a counter for the workload proof algorithm. Every valid transaction data of the blockchain can get a hash. The Merkle root hash records enormous transaction data using the Merkle tree (or hash tree) structure. This storage structure ensures the security of the data stored in the blockchain.

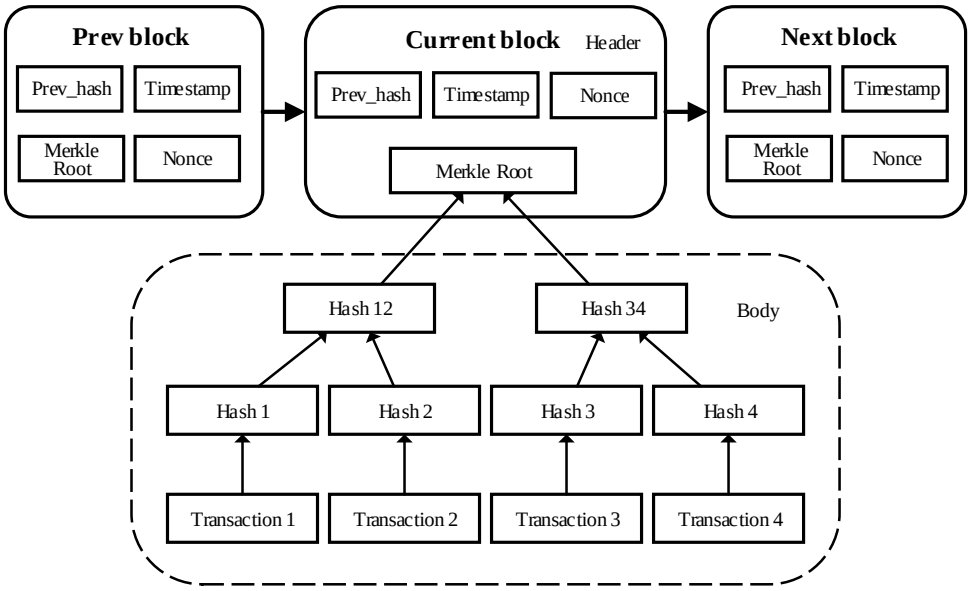

Figure 1. Block structure of blockchain system.

3.2. Smart Contracts

A smart contract is a computer protocol that automatically executes the terms of a contract. It was first proposed by the computer scientist, legal scholar, and cryptographer Nick Szabo in 1996 [32]. However, due to the lack of relevant theoretical knowledge and technology, smart contracts have not been practiced in that era. In recent years, with the increasing popularity of blockchain technology, smart contracts have attracted much attention. The blockchain technology has provided a trusted environment for smart contracts that allow the performance of credible transactions without third parties. Blockchain-based smart contracts have the characteristics of decentralization, being programmable, and immutability. It is widely used in finance, insurance, the Internet of things, and other fields. There are many specific application examples of smart contracts, such as digital payment, cloud computing, shared economy, and so on.

Unfortunately, with the rapid development of blockchain smart contracts, the number of attacks on smart contracts is gradually increasing. The DAO, the world's largest crowdfunding project deployed on the Ethereum, was attacked on June 2016 due to significant flaws in its smart contracts [6]. More than three million ETH was separated from the DAO resources pool, which leads to $60 million economic losses. In September 2017, due to a security vulnerability in the Ethereum multi-signature wallet Parity, more than 150,000 ETH (about $30 million) was stolen. Moreover, in April 2018, a significant vulnerability in integer overflow occurred in BEC's smart contract. A hacker attacked the BEC smart contract and transferred an immense amount of BEC tokens to two addresses successfully, which caused massive amounts of BEC in the market to be sold and made the value of BEC almost zero on that day. In 2018, Atzei's analysis, which used the MAIAN analysis tool to perform a security analysis of nearly one million Ethereum smart contracts, found that 34,200 smart contracts are vulnerable [5]. Faced with a large number of vulnerable contracts, the issue of the security verification of smart contracts deserves imperative and comprehensive studies.

3.3. Coq Proof Assistant

Proof assistants (or interactive theorem provers) are software tools that assist with the development of formal proofs by human–machine collaboration. The user can formalize mathematical definitions and theorems and write formal proofs in the proof assistant. The correctness of these

formalizations and proofs is checked by the computer, typically down to the level of necessary logical inference. There are many different proof assistants in the world, widely used ones include Coq, Isabelle, HOL Light, Agda, and ACL2, among many others. They differ in their underlying logic and in the infrastructure they provide. In this paper, we focus on the Coq proof assistant.

The Coq system has been developed since 1983. In recent years, it has attracted a large community of users in both academia and industry. Coq proof assistant is designed to develop mathematical proofs, write formal specifications, and verify the correctness of programs. The specification language of Coq, also called *Gallina*, can represent programs as well as properties of these programs and proofs of these properties. The kernel of the Coq system is a simple proof-checker, which includes a type-checking algorithm, and all logical judgments in Coq are typing judgments. On top of this kernel, the Coq development environment has a robust mathematical model foundation, flexible proof-automation tactics, and favorable expansibility. Besides, an extensive standard library is provided, which contains essential contents axiomatizations about sets, lists, sorting, and arithmetic. The foundation of the Coq system is an expressive formal language called the calculus of inductive constructions (CIC), which is a λ-calculus with a productive type system. Using the Curry–Howard isomorphism, programs, properties, and proofs can be formalized in CIC. To describe formal developments using the Coq proof assistant, as shown in Figure 2, we briefly introduce how to define natural numbers and construct a function.

```
1  Inductive nat : Type :=
2    | O : nat
3    | S : nat -> nat.
4
5  Fixpoint nat_blt (n m: nat) {struct n} : bool :=
6    match n, m with
7    | O, O => false
8    | O, S _ => true
9    | S _, O => false
10   | S n', S m' => nat_blt n' m'
11   end.
```

Figure 2. Examples of formal development in Coq.

Using the command `Inductive` which can define a simple inductive type and its constructors, we inductively define nature numbers as data type `nat` (lines 1-3, Figure 2). The type `nat` has two constructors, `O` and `S`, respectively. The definition `nat` states that `O` is a nature number (`nat`) and if `n` is a nature number then `S n` is also a nature number. For example, the term `S(S O)`, also written as number 2, is `nat`. According to the formal definition of nature numbers, we can construct a function in Figure 2 (lines 5-11). We construct the recursive function `nat_blt` to compare two nature numbers `(n:nat)` and `(m:nat)`. If n is smaller than m, then the function outputs `true`. Conversely, the function outputs `false`. We implement it with the command `Fixpoint` which allows defining functions by pattern matching over inductive objects using a fixed point construction. The point of the `{struct n}` annotation is to tell the system which argument decreases along the recursive calls. In the function `nat_blt`, we first match the variable `n` and then match the variable `m`. The whole matching process is divided into four cases corresponding to different output results.

4. Formal Verification Framework

The formal method is a powerful technique for verifying the correctness of smart contracts. It has more advantages over the existing test and security audit. Smart contracts are a very complex system with relatively complex logic. On the basis of the unambiguous mathematical language, the formal method can precisely define the correctness and security of smart contracts. Through strict mathematical proof in Coq proof assistant, we can fully cover every logical behavior of the smart contract and give formal descriptions of function specification and high-level properties.

4.1. Security Issues of Smart Contracts

In this subsection, we deeply study common security problems in smart contracts and summarize five types of main security issues. Meanwhile, we propose a specific formal verification method for each security issue.

4.1.1. Integer Overflow

Integer overflow is the most common security issue in smart contracts. This security vulnerability is usually inadvertently introduced by programmers. It may cause some features of the contract to fail. In the most severe case, it may lead to hacker attacks and economic losses. For example, BEC, SMT, and EDU have been attacked by hackers due to integer overflow security vulnerabilities, which results in the value of the tokens becoming zero.

We formally verify the integer overflow issue in smart contracts by building a safe math library in Coq. Because the parameter types in smart contracts are unsigned integer types, we can represent unsigned integers directly through natural numbers. In order to ensure the reliability of the entire construction process, we did not use the conversion functions of `nat` and `bool` provided in Coq. Instead, we build functions `nat_blt`, `nat_ble`, and `nat_neq` from scratch. Furthermore, safe arithmetic and safe mapping are defined step by step.

First we define a function `nat_ble` in Figure 3 (lines 1-5) according to the function `nat_blt` in Figure 2. Since the function `nat_blt` constructs the relation "$<$" on boolean, it is obvious that the function `nat_ble` defines the relation "$\leq$" on boolean. As shown in Figure 3 (lines 7-13), on the basis of the definition of `nat` in Coq, we construct a conversion function that converts the equality relationship of natural numbers to boolean values. We use the recursive definition method in the same way that function `nat_ble` is defined. The function `nat_beq` can judge whether two natural numbers are equal. If two natural numbers are equal, the output is true. Otherwise, the output is false.

```
1  Definition nat_ble (n m: nat) : bool :=
2    match nat_blt m n with
3      | true => false
4      | false => true
5    end.
6
7  Fixpoint nat_beq (n m: nat) {struct n} : bool :=
8    match n, m with
9    | O, O => true
10   | O, S _ => false
11   | S _, O => false
12   | S n', S m' => nat_beq n' m'
13   end.
```

Figure 3. Definitions of conversion function.

Figure 4 declares some basic data types for defining safe computational functions of smart contracts. All integer types are modeled as `nat` in Coq formalization. The type `uint256` is a 256-bit unsigned integer type. We normally define integer variables in contract as `uint256` type. The `uint8` type is used to define decimals. Both the `time` type and the `address` type are defined as `nat`. We define `av` in the line 6 and `aav` in the line 7 to represent the Solidity "mapping" type in Coq. The type `av` represents the mapping (`address` $\Rightarrow$ `uint256`) and `aav` represents the mapping (`address` $\Rightarrow$ (`address` $\Rightarrow$ `uint256`)). The mapping about address and value occurs more frequently in most smart contracts. We can assign them to other types in the specific contract. `maxuint256` is the maximum of the type uint256 and `Error` is the return value when an exception occurs. The command `Parameter` declares a global variable. In order to avoid interference with normal parameters of type `uint256`, we set the value of `Error` to be greater than `maxuint256` in line 11.

```
1  Definition uint256 := nat.
2  Definition uint8 := nat.
3  Definition time := nat.
4  Definition address := nat.
5
6  Definition av := address -> uint256.
7  Definition aav := (address * address) -> uint256.
8
9  Parameter maxuint256 : uint256.
10 Parameter Error : nat.
11 Axiom Eraxiom : Error > maxuint256.
```

Figure 4. Basic data types of smart contracts.

Next, as shown in Figure 5, we define safe arithmetic, which consists of some basic operators. These operators, which can prevent overflow, include addition, subtraction, multiplication, and division. The function `safe_plus` implements the safe addition of two uint256-type variables. By the judging condition in line 2, Figure 5, it ensures that the sum of two numbers must be greater than or equal to each addend. The function `safe_minus` implements the safe subtraction. Since the return value needs to be a positive number, the number `b` must be less than or equal to the number `a` (line 6, Figure 5). We define the safe multiplication in lines 8-10. According to the division and the function `nat_beq`, it can determine whether there is an overflow. The safe division is defined as the function `safe_div`. We ensure that the dividend is not zero and prevent overflow by the condition (line 13, Figure 5), where `mode` is a modulo function.

```
1  Definition safe_plus (a b: uint256) :=
2    if (nat_ble a (a + b)) && (nat_ble b (a + b))
3    then (a + b) else Error.
4
5  Definition safe_minus (a b: uint256) :=
6    if (nat_ble b a) then (a - b) else Error.
7
8  Definition safe_mult (a b: uint256) :=
9    if (nat_beq a 0) || (nat_beq (a * b / a) b)
10   then (a * b) else Error.
11
12 Definition safe_div (a b: uint256) :=
13   If (nat_blt 0 b) && (nat_beq a (b * (a / b)+ a mod b))
14   then (a / b) else Error.
```

Figure 5. Definitions of operators in the safe math library.

The functions defined above are all for variables of `uint256` type. In smart contracts, we usually do arithmetic on mapping types. Therefore, we construct the safe arithmetic of mapping in Figure 6. In Figure 4 (lines 6-8), we define two mapping types `av` and `aav` that are common in smart contracts. Therefore, we build separate safe arithmetic functions for these two types. First, a function `dadd_beq` (lines 1-4, Figure 6) for judging two variables of type `(address * address)` is constructed according to the function `nat_beq`. It is to determine whether two variables are equal. Then we define two functions `map_rep` and `map_repd` in Figure 6 (lines 6-9). Based on these two functions, we can replace a value in the mapping or add a value to the mapping. Taking function `map_rep` as an example, we look for addresses equal to the address `(a:address)` in the domain of mapping `(m:av)` (line 7, Figure 6). The values of those addresses are assigned to the value `(b:uint256)`. The λ-abstraction meets our requirements of the above development and it is implemented through function `fun` in Coq. As shown in lines 11-12, we define functions that map values to zero. Finally we define the safe arithmetic on the mapping in lines 14-25. `av_inc` and `aav_inc` are addition on mapping. `av_dec` and `aav_dec` are subtraction on mapping. The multiplication on mapping is defined as `av_mul` and `aav_mul`. The division on mapping is defined as `av_div` and `aav_div`.

```
Definition dadd_beq (a1 a2: address * address): bool :=
  match a1, a2 with
  | (x1, y1), (x2, y2) => andb (nat_beq x1 x2) (nat_beq y1 y2)
  end.

Definition map_rep (m: av) (a: address) (b: uint256) : av :=
  fun a' => if (nat_beq a a') then b else m a'.
Definition map_repd (m: aav) (a1 a2: address) (b: uint256) : aav :=
  fun a' => if (dadd_beq (a1, a2) a') then b else m a'.

Definition map_emp : av := fun _ => 0.
Definition map_empd : aav := fun _ => 0.

Definition av_inc (m: av) (a: address) (v: uint256) := map_rep m a (safe_plus (m a) v).
Definition av_dec (m: av) (a: address) (v: uint256) := map_rep m a (safe_minus (m a) v).
Definition av_mul (m: av) (a: address) (v: uint256) := map_rep m a (safe_mult (m a) v).
Definition av_div (m: av) (a: address) (v: uint256) := map_rep m a (safe_div (m a) v).
Definition aav_inc (m: aav) (a1 a2: address) (v: uint256) :=
  map_repd m a1 a2 (safe_plus (m (a1, a2)) v).
Definition aav_dec (m: aav) (a1 a2: address) (v: uint256) :=
  map_repd m a1 a2 (safe_minus (m (a1, a2)) v).
Definition aav_mul (m: aav) (a1 a2: address) (v: uint256) :=
  map_repd m a1 a2 (safe_mult (m (a1, a2)) v).
Definition aav_div (m: aav) (a1 a2: address) (v: uint256) :=
  map_repd m a1 a2 (safe_div (m (a1, a2)) v).
```

Figure 6. Definitions of safe operators on mapping.

As shown in Figure 7, notations can be introduced to ease the reading and writing of specifications. This also allows us to stay close to the way mathematicians would write. Moreover, we can also define precedence levels and associativity rules of notations in Coq. So far, we have completed the construction of the safe math library in Coq. The safe arithmetic and safe mapping will be used when defining the function specification of smart contracts. By verifying the function specification, we can determine whether there is an integer overflow issue in the smart contract.

```
Notation "m [ a ->> v ]" := (map_rep m a v) (at level 50, left associativity).
Notation "m [ a ->> + v ]" := (av_inc m a v) (at level 50, left associativity).
Notation "m [ a ->> - v ]" := (av_dec m a v) (at level 50, left associativity).
Notation "m [ a ->> * v ]" := (av_mul m a v) (at level 50, left associativity).
Notation "m [ a ->> \ v ]" := (av_div m a v) (at level 50, left associativity).
Notation "m [ a1, a2 ->> v ]" := (map_repd m a1 a2 v) (at level 50, left associativity).
Notation "m [ a1, a2 ->> + v ]" :=(aav_inc m a1 a2 v)(at level 50, left associativity).
Notation "m [ a1, a2 ->> - v ]" :=(aav_dec m a1 a2 v)(at level 50, left associativity).
Notation "m [ a1, a2 ->> * v ]" :=(aav_mul m a1 a2 v)(at level 50, left associativity).
Notation "m [ a1, a2 ->> \ v ]" :=(aav_div m a1 a2 v)(at level 50, left associativity).
```

Figure 7. Notations of safe operators on mapping.

4.1.2. Function Specification Issue

The second type of security issue is the function specification issue of smart contracts. Nowadays, there is no uniform function specification for the implementation of many smart contracts. Smart contracts are interactive multi-person collaboration programs. If the function in the contract is not standardized, misunderstanding of the contract behavior will arise, which will lead to a large number of security problems. We present a formal verification method for the function specification issue in smart contracts. As shown in the following figure, we define an abstract structure of function specification in Coq by analyzing the construction of functions in smart contracts.

First, we declare the specific state of the contract in Figure 8 (lines 1-2). The command `Record` defines a record type `State` in Coq. Its constructor has the default name `Build_State`. The identifiers

`st1`, `st2`, and `st3` are parameters of the record. These parameters include token name, total supply, owner address, token balance, etc., which may be different types. We use these parameters to describe the state of the contract. Then we define a abstract event model `Event` in Figure 8 (lines 4-9). In line 5, we define a creator event `Creator`, which is usually called in an initialization function. The parameter `Sa` is the address of function sender and `cr1`, `cr2`, and `cr3` represent the initial parameters. We define events for specific functions as `Fun1`, `Fun2`, and `Fun3`. The number of input parameters for these events depends on the specific function. The event `Throw` indicates that an exception is thrown and the parameter `Ca` is the contract address. Finally, as shown in Figure 8 (lines 12-15), we define the abstract structure of function specification `Specification`. The specification of a function is generally composed of four parts. In the first part, the `require` represents requirements via the command 'require' calls in Solidity. These requirements are formalized as the type `Prop`. The `transfer` implements the state transition on the basis of the behavior of functions in the second part. The safe arithmetic we defined in Figure 5 will have important applications in it. The `event` records events generated via event calls in the third part. We implement it through the definition `list` provided in the Coq library. In the fourth part, the identifier `return` define the return value of the function. The type `returns` will change depending on the type of return value. All parameters in these abstract models can be changed according to the specific smart contract.

```
1  Record State : Type :=
2    { st1 : uint256; st2 : address; st3 : av }.
3
4  Inductive Event : Type :=
5    | Creator (Sa: address) cr1 cr2 cr3: Event
6    | Fun1 (Sa: address) A: Event
7    | Fun2 (Sa: address) A B: Event
8    | Fun3 (Sa: address) A B C: Event
9    | Throw (Ca: address): Event.
10
11 Record Specification: Type :=
12   { require : State -> Prop;
13     event : State -> list Event -> Prop;
14     transfer : State -> State -> Prop;
15     return : returns }.
```

Figure 8. Abstract structure of function specification.

In the verification of a specific contract, we define the contract specification based on the above abstract model. Then we describe the Solidity code. Finally, on the basis of the mathematics theorem proof, we verify whether the Solidity code of the contract satisfies the contract specification.

4.1.3. Invariant Issue

There are some invariant quantities in the process of contract creation, function calls, and environment changes. The third kind of issue is about these invariant quantities. When we build a smart contract, we always hope that some parameters are not changed with the external environment and function calls. For example, the total amount of tokens needs to remain the same in most contracts.

As shown in Figure 9, we present the abstract model of contract status, message calls, and current environment based on the Coq proof assistant. The type `Contract` (lines 1-2) records the status of the current contract, including the contract address `adr` and the state `st` of each variable in the contract. These variables may include the token name, total supply, owner address, token balance, and so on. Then we define the abstract message call model `Message` in lines 4-8. The parameter `data` is the complete call data, including call information for each function. We define the input parameter `mdata` based on functions in a specific contract. The parameter `(gas:uint256)` represents the current remaining gas in the contract. The "gas" is the name of a special unit used by Ethereum. It measures how much work action or a series of actions needs to perform. We can inquire about the address of the

message sender through (sender:address). The (value:uint256) represents the amount of money attached to the message. Finally, in lines 10-15, we define the environment type Environment of the contract. The contract environment consists of some information about the current block, including block hash, miner's address, difficulty, gas limit, and time stamp. In the specific contract, we can increase or decrease the parameters according to the actual situation.

```
1  Record Contract (State: Type) : Type :=
2    { adr : address; st : State }.
3
4  Record Message (mdata: Type) : Type :=
5    { data : mdata;
6      gas : uint256;
7      sender : address;
8      value : uint256 }.
9
10 Record Environment : Type :=
11   { blockhash : uint256;
12     coinbase : address;
13     difficulty : uint256;
14     gaslimit : uint256;
15     timestamp : time }.
```

Figure 9. The abstract model of smart contract.

After defining the abstract model of contract status, the specific contract state can directly defined by the record type state in Figure 8. Moreover, as shown in Figure 10, we define a structural model mdata of the data in message calls. We can adjust it according to the specific contract. The message data of creator function is declared as md_creator in line 2 of Figure 10. The parameters cr1, cr2 and cr3 represent the initial parameters. The constructors md_fun1, md_fun2, and md_fun3 are message data for specific functions (lines 3–4, Figure 10). The number of input parameters for these events depends on the specific function. According to the specific function of the smart contract, then we construct an evaluation step that includes any of the possible invocations in the contract. When the environment is changed, we construct an evaluation step for the environment as well. Finally, the invariants are determined according to the specific requirements of the contract. We prove in the form of theorems that function calls and environmental changes do not change the invariants.

```
1 Inductive mdata : Type :=
2   | md_creator cr1 cr2 cr3 : mdata
3   | md_fun1 A : mdata
4   | md_fun2 A B : mdata
5   | md_fun3 A B C : mdata.
```

Figure 10. Structural model of message data.

4.1.4. Authority Control Issue

The fourth is the issue of the authority control of smart contracts. Generally, an administrator (or owner) is set up in the smart contract. Administrators generally have super privileges, such as burning tokens, freezing accounts, closing transfers, and so on. The security risks of such contracts are relatively significant. Once the administrator's private key is stolen, it can easily cause huge losses.

As shown in Figure 11, we construct a formal verification method for the authority control issue. We verify the issue according to the mathematical theorem Owner_Prop. The input parameter msg is the message call of the function. The spec is the specification of the function. The parameters C and C' are specific contract model. The E is the event list of the contract. In lines 2-3, we get the requirements, event, and state transition of the function spec. Furthermore, we specify that the message sender is the owner of the contract C in line 3. Then we can prove the specific property about the authority of

the owner. The first `Prop` in line 4 is the property about the variable account. The second `Prop` is the specific property about authority control. The specific `Prop` of different contracts are different.

```
1 Theorem Owner_Prop: forall msg spec C C' E,
2   (require spec) (st C) /\ (event spec) (st C) E /\
3   (transfer spec) (st C) (st C') -> sender msg = owner (st C) ->
4   (forall account, account <> (sender msg) /\ Prop -> Prop).
```

Figure 11. Verification of the authority control issue.

4.1.5. Behavior of the Specific Function

The fifth kind of problem is about the behavior of the specific function in the smart contract. There may be various functions in the smart contract, each of which has a different purpose. The behavior of different functions can have different effects. In general, the function mainly operates on the following aspects: total supply, the balance of an account, contract status, and contract authority. Next, we consider the specific properties of functions. We will not discuss the permission change caused by functions because of the existence of the fourth security issue of smart contracts.

As shown in Figure 12, we define an abstract model `Fun_Prop` for verification of the function behavior. The input parameters `C` and `C'` are different contract model. The `E` is the event list of the contract. The parameter `spec` represents a concrete function specification model. Line 2 defines the requirements of the function. The event and state transition of the function are obtained from lines 2-3. Then we can define the specific property of the function according to the above information. According to different functions, we can present different formal theorem descriptions.

```
1 Theorem Fun_Prop: forall C C' E spec,
2   (require spec) (st C) /\ (event spec) (st C) E /\
3   (transfer spec) (st C) (st C') -> Prop.
```

Figure 12. Verification of the function behavior.

4.2. Formal Verification Framework for Blockchain Smart Contracts

According to the formal verification methods for five types of security issues in the previous subsection, we present a formal verification framework for blockchain smart contracts in Figure 13. We construct the basic theory, as shown in the green box of Figure 13. The data types of smart contracts are the theory of the initial construction. We define the functions `nat_beq` and `nat_ble` which are conversion functions between natural numbers and Boolean according to the type `(uint256:nat)` in data types. Then a safe math library is constructed based on the functions. It can verify the integer overflow issue in smart contracts. In addition, we can use the `Ltac` strategy provided in Coq to enhance the automation of the proof. As shown in the light blue box, we next define the abstract model of the contract. The abstract model mainly includes the abstract contract state model, abstract event model, abstract message call model, and abstract environment model. These abstract models are built on the basis of the underlying theory. The orange box represents the Solidity code for a specific contract. According to the abstract models, we can define the state model, event model, and message data model of the contract in the yellow box. As shown in the dark blue box, the functional specification of the contract is defined according to the abstract structure `Spec`, which takes the above three models as input parameters. Then we verify whether the function in the contract conforms to the functional specification. If the contract function satisfies the specification, then we verify that it satisfies the properties of the function. The properties of functions are mainly divided into two parts: authority control and specific function behavior. Finally, we verify the invariant issue in the process of environmental change and message calls. The formal verification framework we developed can be used to verify the security of most blockchain smart contracts.

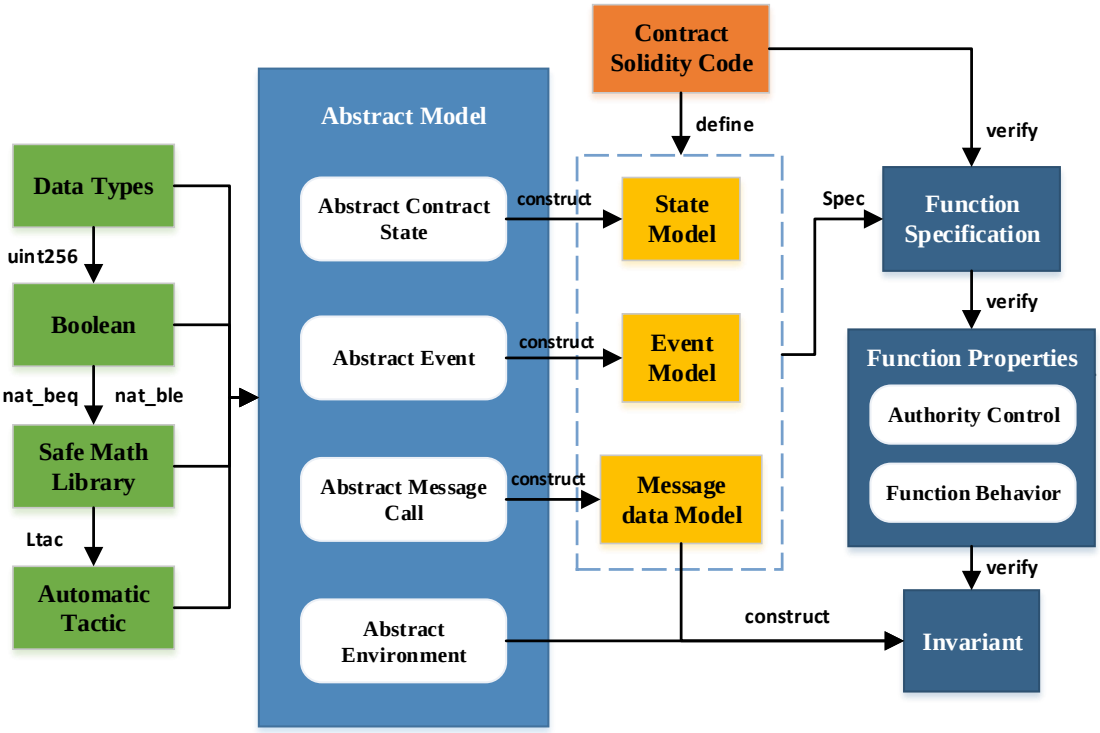

Figure 13. Formal verification framework for blockchain smart contracts.

5. Formal Verification of BNB contract

BNB is a platform token issued by Binance Exchange and runs on the Ethereum blockchain with the ERC20 standard. Binance is one of the three largest cryptocurrency exchanges in the world, with three million registered users. Binance Exchange offers to trade in more than 45 virtual coins, including Bitcoin (BTC), Ethereum (ETH), Litecoin (LTC), and BNB. BNB has a strict limit of a maximum of 200 million tokens and sold 100 million tokens (50%) to the outside world. The smart contract of BNB is one of the most used ERC20 contracts on the Ethereum platform. This section shows how to verify a specific smart contract based on the formal verification framework constructed above. We take the BNB contract as an example and present a detailed process of the formal verification in Coq. It thoroughly verifies the five kinds of security issues mentioned above.

5.1. Model of BNB contract

According to the abstract model of the formal verification framework proposed in the previous section, we define the specific model of the BNB contract. First, as shown in Figure 14, we formalize the specific state model of the BNB contract. The state variables of the BNB contract are the name, symbol, decimals, total supply, owner, the balance of address, freeze token of address, and allowance. Since name and symbol of the contract are a string type, we introduce the String library in coq and define `name` and `symbol` as type `string` (lines 2-3, Figure 14). The identifier `decimals` represents the number of decimal places in the token. The `totalSupply` is the total supply of the contract and the `owner` defines the address of the contract administrator. Both `balanceOf` and `freezeOf` are mappings of type `av`. The mapping `balanceOf` is the balance of an account address and the mapping `freezeOf` represents the frozen token of an account. The identifier `allowance` is a mapping of type `aav` and defines a quota for one address to another. The specific contract is defined in line 11 based on the stat model and the abstract model `Contract`.

```
1  Record State : Type :=
2    { name: string;
3      symbol: string;
4      decimals: uint8;
5      totalSupply: uint256;
6      owner: address;
7      balanceOf: av;
8      freezeOf: av;
9      allowance: aav }.
10
11 Definition contract := Contract State.
```

Figure 14. The state model of the Binance Coin (BNB) contract.

Then we define the specific event model of the BNB contract in Figure 15. There are four events in the Solidity code of the BNB contract. In line 2, the `Transfer` defines the transfer event, which generates a public event on the blockchain that will notify clients. The event `Burn` notifies clients about the amount burnt in line 3. The event `Freeze` and `Unfreeze` notify clients about the amount frozen and unfrozen in lines 4-5. In addition, we added an extra event `Throw` for the exception in line 6.

```
1 Inductive Event : Type :=
2   | Transfer (Sa: address) (from: address) (to: address) (val: uint256): Event
3   | Burn (Sa: address) (from: address) (val: uint256): Event
4   | Freeze (Sa: address) (from: address) (val: uint256): Event
5   | Unfreeze (Sa: address) (from: address) (val: uint256): Event
6   | Throw (Ca: address): Event.
```

Figure 15. The event model of the BNB contract.

Finally, we define the specific message model of the BNB contract. As shown in Figure 16 (lines 1-10), we construct the message data `mdata` of the BNB contract. The constructors of `mdata` are consistent with functions of the contract. The `md_BNB` is the message data of the creator function. The input parameters of it are initial supply, token name, decimal units, and token symbol. The identifier `md_transfer` represents the transfer function which has two input parameters. The message data of the TransferFrom function is defined as `md_transferFrom` in the line 5. The input parameters of it are the `from` address, the `to` address and the value of transfer `val`. The input parameters of function `md_approve` are the spender address and the value of token. As shown in lines 7-10, the functions `md_burn`, `md_freeze`, `md_unfreeze`, and `md_withdrawEther` all have only one input parameter of type `uint256`. According to the definition of the message data, we define the specific message model in line 12. The identifier `Message` is the abstract message call model defined in Figure 9.

```
1  Inductive mdata : Type :=
2    | md_BNB (initialSupply: uint256) (tokenName: string)
3      (decimalUnits: uint8) (tokenSymbol: string): mdata
4    | md_transfer (to: address) (val: uint256): mdata
5    | md_transferFrom (from: address) (to: address) (val: uint256): mdata
6    | md_approve (spender: address) (val: uint256): mdata
7    | md_burn (val: uint256): mdata
8    | md_freeze (val: uint256): mdata
9    | md_unfreeze (val: uint256): mdata
10   | md_withdrawEther (amount: uint256): mdata.
11
12 Definition message := Message mdata.
```

Figure 16. The specific message data and model of the BNB contract.

5.2. Verification of function Specification

For space reasons, below we take the freeze function as an example to introduce the formalization of the functional specification of the BNB contract. As shown in Figure 17, according to the abstract structure of function specification, we define the function specification in Coq after analyzing the freeze function. The input parameter for the freeze function is `val` and it has type `uint256`. In addition, we consider the impact of the contract address, environment, and message call on the function. We also use them as input parameters of functions in line 2. In lines 3-15, we construct an item of type `Spec` by constructor `Build_Specification`. In line 4, we present the requirements that the value must be greater than zero and the balance of message sender must be greater than or equal to the value. Since more attention is paid to the specific properties of requirements, we directly use the relationship '>' and '>=' of natural numbers in the Coq library. Then we define the state transition in lines 5-14. The balance of message sender is decreased by value, while the frozen token is increased by value. Because the arithmetic in the BNB contract is safe, we implement it according to the safe math library defined in the previous section. Finally, we define the return value `(true:bool)` of freeze function in line 15.

```
1  Definition funcspec_freeze (val: uint256) :=
2    fun (this: address) (env: Environment) (msg: message) =>
3      (Build_Specification bool
4        (fun S : State => val > 0 /\ balanceOf S (sender msg) >= val
5        (fun S S' : State =>
6          name S' = name S /\
7          symbol S' = symbol S /\
8          decimals S' = decimals S /\
9          totalSupply S' = totalSupply S /\
10         balanceOf S' = (balanceOf S) [ (sender msg) ->> - val ] /\
11         freezeOf S' = (freezeOf S) [ (sender msg) ->> + val ] /\
12         allowance S' = allowance S /\
13         owner S' = owner S)
14       (fun S E => E = (Freeze (sender msg) (sender msg) val) :: nil)
15       (true)).
```

Figure 17. Definition of the specification of the freeze function.

```
1  Inductive PrimitiveStmt :=
2    | DSL_require
3      (cond: State -> env -> message -> bool)
4    | DSL_balanceOf_dec
5      (addr: State -> env -> message -> address)
6      (expr: State -> env -> message -> uint256)
7    | DSL_balanceOf_rep
8      (addr: State -> env -> message -> address)
9      (expr: State -> env -> message -> uint256)
10   | DSL_freezeOf_inc
11     (addr: State -> env -> message -> address)
12     (expr: State -> env -> message -> uint256)
13   | DSL_emit
14     (evt: State -> env -> message -> Event).
15
16 Inductive Stmt :=
17   | DSL_prim (stmt: PrimitiveStmt)
18   | DSL_seq (stmt: Stmt) (stmt': Stmt).
```

Figure 18. Definition of domain specific language (DSL) statements in the freeze function.

After completing the specification definition of the freeze function, we need to verify the Solidity code of the function. We convert the Solidity code in the BNB contract to Coq code through a domain specific language (DSL) which was developed by SECBIT [30]. DSL is able to complete correctness proofs of Ethereum token contracts. First, we define a Solidity implementation of the freeze function

in DSL. Figure 18 describes the definition of DSL statements in the freeze function. In lines 1-14, the function `PrimitiveStmt` describes the behavior in the freeze function. In order to better display the code, we use `env` instead of `Environment`. The function `Stmt` construct a sequence consisting of constructors in `PrimitiveStmt`. Its structure is similar to the list structure.

Then Figure 19 (lines 1-2) define a record type `Result` which is the result of each behavior in freeze function. The identifier `res_st` is the state at the end of the behavior. The `res_evts` records the list of event. The identifier `res_stop` determines whether the function stops. There are two definitions of the type `Result` in lines 4-5.

```
1 Record Result : Type :=
2   { res_st : State; res_evts : eventlist; res_stop : bool }.
3
4 Definition Next (sta: State) (evts: eventlist) : Result := Build_Result sta evts false.
5 Definition Stop (sta: State) (evts: eventlist) : Result := Build_Result sta evts true.
```

Figure 19. Result of statement execution.

In Appendix A, we introduce the semantics of DSL statements. As shown in Figure 20, we present a formal definition of the Solidity code of the freeze function based on the semantics of DSL statements. We first declare some hypothetical parameters in lines 1-4. The variable `_value` is the input parameter of the freeze function. To match the types defined in the DSL, we define the corresponding variables `Value` and `Zero` for `_value` and the nature number `0`. At the same time we specify that for any state `st`, environment `env`, and message call `msg`, the `(Value st env msg)` is always equal to `_value` and the `(Zero st env msg)` is always equal to `0`. In lines 6-11, we formalize the Solidity code of the freeze function in the BNB contract. We have achieved the consistency between the formalization and Solidity code by the notations, which increases the readability of the formal work. The return value of the freeze function is defined as `freeze_ret` in line 12.

```
 1 Variable _value : uint256.
 2 Variable Value Zero : State -> env -> message -> uint256.
 3 Hypothesis Value_immutable: forall sta env msg, Value sta env msg = _value.
 4 Hypothesis Zero_immutable: forall sta env msg, Zero sta env msg = 0.
 5
 6 Definition freeze_dsl : Stmt :=
 7   (@ Require (Value > Zero);
 8    @ Require (BalanceOf [msg.sender] >= Value);
 9    @ BalanceOf [msg.sender] -= Value;
10    @ FreezeOf [msg.sender] += Value;
11    ( @ emit Ev_Freeze (msg.sender, Value) )).
12 Definition freeze_ret : bool := true.
```

Figure 20. Formal definition of Solidity code of the freeze function.

```
 1 Theorem freeze_sat_spec: forall sta env msg this,
 2   require bool (funcspec_freeze _value this env msg) sta ->
 3   forall result, result = dsl_exec freeze_dsl sta env msg this nil ->
 4   transfer bool (funcspec_freeze _value this env msg) sta (ret_st result) /\
 5   event bool (funcspec_freeze _value this env msg) (ret_st result) (ret_evts result) /\
 6   return bool (funcspec_freeze _value this env msg) = freeze_ret.
 7
 8 Theorem freeze_throw: forall sta env msg this,
 9   ~ require bool (funcspec_freeze _value this env msg) sta ->
10   dsl_exec freeze_dsl sta env msg this nil = Stop sta (Throw this :: nil).
```

Figure 21. Specification verification of the freeze function.

Finally, as shown in Figure 21, we verify the specification of freeze function according to the proof of two mathematical theorems. The first theorem `freeze_sat_spec` (liens 1-6) is to prove that the

property, event, and the return value of the function satisfy the specification when the requirements of function specification are satisfied.

This theorem is proved according to the definitions of `freeze_dsl` and `freeze_ret`. The second theorem `freeze_throw` (lines 8-10) is to prove that the function must revert to the initial state when the requirements of function specification are not satisfied. It can be directly proved on the basis of the condition and the definition of `freeze_dsl`. According to the formal proof of these two theorems, it is verified that the Solidity code of the freeze function satisfies the function specification.

5.3. Verification of Invariant

In order to verify the invariant issue in the BNB contract, we construct the evaluation step in Appendix A. Then we define the evaluation step for the contract environment in Figure 22. We consider the general case where the block hash and time stamp change.

```
1 | Definition Env_Step (env1 env2: Environment) : Prop :=
2 |   blockhash env2 <> blockhash env1 /\ timestamp env2 >= timestamp env1.
```

Figure 22. Evaluation step for the environment.

According to our analysis, there is no fixed invariant in the BNB contract because of the presence of the `burn` function. Nevertheless, we can verify that the sum of the account balances in the BNB contract is always equal to the total supply. As shown in Figure 23, we formalize the above proposition. We define a function `SumMap` that can sum the mapping in lines 1-4. The function has two input parameters, one is the mapping, and the other is the sum of the mappings. The function can be divided into three cases. If the mapping is `map_emp`, the sum of the mapping is `0`. The constructor `Sum_inc` can add a value `v'` to the map. The `Sum_dec` can remove an address `m` from the map. Therefore, the function `Inv` is defined in lines 6-7, that is, the sum of the account balances in the BNB contract is the total supply of the account. Then we prove that this property will not change under any circumstances.

```
1 | Inductive SumMap : av -> uint256 -> Prop :=
2 | | Sum_emp : SumMap map_emp 0
3 | | Sum_inc : forall m v a v', SumMap m v -> m a = 0 -> SumMap (m [a ->> v']) (v + v')
4 | | Sum_dec : forall m v a, SumMap m v -> SumMap (m [a ->> 0]) (v - (m a)).
5 |
6 | Definition Inv (env: Environment) (S: State): Prop :=
7 |   SumMap (balanceOf S) (totalSupply S).
```

Figure 23. The invariant of the BNB contract.

As shown in Figure 24, we formally describe the above property as a theorem in Coq. The input parameter `a` is the address of contract. The `env1` is the current environment and `env2` is the changed environment. The current state is `S1` and the changed state is `S2`. The `msg` is the message call and `E` is the event list. We construct an evaluation step through the function `Step`. The `Build_Contract` can construct a specific state model of the BNB contract. The function `Env_Step` implements a change in the contract environment. To prove this theorem, we need to discuss each function in the BNB contract. We complete the proof of the theorem `Pro_Inv` on the basis of the automatic arithmetic library in Coq.

```
1 | Theorem Pro_Inv: forall a env1 msg S1 env2 S2 E,
2 |   Step env1 (Build_Contract a S1) msg (Build_Contract a S2) E ->
3 |   Env_Step env1 env2 -> Inv env1 S1 -> Inv env2 S2.
```

Figure 24. Verification of the invariant in the BNB contract.

5.4. Contract Properties

We prove some properties of the BNB contract in this subsection. First, we consider the authority control issue of the BNB contract. As shown in Figure 25, we prove that the owner of the BNB contract cannot transfer tokens in an arbitrary account. The theorem `Owner_not_trans_token` is constructed based on the theorem model `Owner_Prop` in Figure 11. The parameter `to` is the address of the recipient and `value` is the token value of the transmission. The input parameter `spec` defines the specification of transfer function. The address of `account` is different from the addresses of owner and `to`. The theorem is to prove that the balance of `account` does not change when the transfer function is executed. We prove the theorem according to the specification of the transfer function and the arithmetic of mappings. In addition, we also verify the impact of other functions on the authority issue in Coq. It is not shown here for space reasons.

```
1 Theorem Owner_not_trans_token: forall to val this en msg spec C C' E,
2   spec = funcspec_transfer to val this en msg ->
3   (require (list bool) spec) (st C) /\ (event (list bool) spec) (st C) E /\
4   (transfer (list bool) spec) (st C) (st C') ->
5   sender msg = owner (st C) ->
6   (forall account, account <> (sender msg) /\ account <> to ->
7   balanceOf (st C) account = balanceOf (st C') account).
```

Figure 25. Verification of the authority control issue of the transfer function.

Then we verify the behavior issue of the specific function in the BNB contract. As shown in Figure 26, we prove that the total supply of the BNB contract is fixed with the transfer function. We present a formalization of the theorem `Freeze_totalSupply` according to the theorem `Fun_Prop` in Figure 12. The concrete properties of the theorem are described in detail in line 5. This theorem is proved according to the specification of the freeze function. Because the total supply of contract in the specification of freeze function is constant. The formal proof of the properties of other functions in the BNB contract can be found in the source code.

```
1 Theorem Freeze_totalSupply: forall en C C' E  msg val spec,
2   spec = funcspec_freeze val (adr C) en msg ->
3   (require bool spec) (st C) /\ (event bool spec) (st C) E /\
4   (transfer bool spec) (st C) (st C') ->
5   (totalSupply (st C)) =  (totalSupply (st C')).
```

Figure 26. Verification of the behavior of the freeze function.

6. Conclusions and Future work

As one of the most critical technologies in blockchain systems, smart contracts are widely used in different research topics. However, smart contracts expose many security problems as well. This paper applies formal techniques to the security verification of smart contracts. We summarize five types of security issues in smart contracts and propose a corresponding formal verification method for each issue. Thus, a formal verification framework for the security of smart contracts is established. On the basis of this framework, we complete the formal verification of the security issues of the BNB contract. In this paper, all the proofs of the verification have been implemented using the Coq proof assistant. The complete source files containing the formalization and proofs are accessible at:

https://github.com/BKLSIC/BNB_Contract/

Overall, the proofs of the formal verification consist of about 3,500 lines of code. It has been tested and should compile under Coq 8.9.1. Table 1 provides a detailed account of the formalization in terms

of script files. To help navigate through the script files, we indicate the related sections in the paper. The count in terms of lines of code distinguishes between specifications and proofs.

Table 1. Overview of the formal verification of the BNB contract.

File	Reference	Specification	Proof
Data_Types.v	Section 4.1	10	0
Boolean.v	Section 4.1	300	150
Safe_Math.v	Section 4.1	200	150
Abstract_Model.v	Section 4.1	50	0
Automatic.v	Section 4.2	150	60
Model.v	Section 5.1	40	0
Specification.v	Section 5.2	750	720
Invariant.v	Section 5.3	150	400
Properties.v	Section 5.4	40	300

In the future, we will complete the verification of more smart contracts based on the formal verification framework of this paper. With more security issues and different contract standards, we will improve the formal verification framework of smart contracts. Our goal is to adapt the framework to the security verification of all smart contracts. In addition, we plan to develop some automatic tactics which can increase the automation of formal proof. It will be a meaningful exploration and attempt in the field of automated verification of smart contracts. In addition, we will also try to combine access control technology with formal verification technology and apply it in the verification of actual security issues of blockchain smart contracts.

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

Funding: This work was supported in part by National Natural Science Foundation (NNSF) of China under grant 61936008, 61571064.

Acknowledgments: We are grateful to the anonymous reviewers, whose comments much helped to improve the presentation of our research in this article.

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

Abbreviations

The following abbreviations are used in this manuscript:

EVM Ethereum Virtual Machine
BNB Binance Coin
BTC Bitcoin
ETH Ethereum
LTC Litecoin
DSL Domain Specific Language

Appendix A. Semantics of DSL Statements and Evaluation Step of the BNB Contract

In this appendix, we introduce the formalization of semantics of DSL statements and the evaluation step of the BNB contract. Figure A1 defines semantics of DSL statements. The function

`dsl_exec_prim` define the specific result of each behavior in the freeze function. This function performs pattern matching on the input parameter `stmt`. If it is `DSL_require`, we consider whether the function condition is satisfied. As shown in lines 6-13, the `DSL_balanceOf_dec` reduce the balance of address `addr` by `expr` and the `DSL_balanceOf_rep` assign the balance of address `addr` to `expr`. If the `stmt` is `DSL_freezeOf_inc`, we increase the freeze token of address `addr` by `expr`. The `DSL_emit` adds event `evt` to the event list of the contract. In Figure A1 (lines 20–28), the function `dsl_exec` implements the judgment of all behaviors in the freeze function and summarizes all the results. Furthermore, Figure A2 defines some notations that makes the DSL syntax close to Solidity code. So far, the DSL for freeze function of BNB contract has been built. Below we formally describe the Solidity code of the freeze function according to DSL.

```
1  Fixpoint dsl_exec_prim (stmt: PrimitiveStmt) (sta: State) (env: Env)
2    (msg: message) (this: address) (evts: eventlist) : Result :=
3    match stmt with
4    | DSL_require cond => if cond st env msg then Next sta evts
5       else Stop sta (Throw this :: nil)
6    | DSL_balanceOf_dec addr expr => Next (Build_State (name sta)
7       (symbol sta) (decimals sta) (totalSupply sta) (owner sta)
8       ((balanceOf sta) [(addr sta env msg) ->> - (expr sta env msg)])
9       (freezeOf sta) (allowance sta)) evts
10   | DSL_balanceOf_rep addr expr => Next (Build_State (name sta)
11      (symbol sta) (decimals sta) (totalSupply sta) (owner sta)
12      ((balanceOf sta) [(addr sta env msg) ->> (expr sta env msg)])
13      (freezeOf sta) (allowance sta)) evts
14   | DSL_freezeOf_inc addr expr => Next (Build_State (name sta) (symbol sta)
15      (decimals sta) (totalSupply sta) (owner sta) (balanceOf sta)
16      ((freezeOf sta) [(addr sta env msg) ->> + (expr sta env msg)])
17      (allowance sta)) evts
18   | DSL_emit evt => Next sta (evts ++ (evt sta env msg :: nil)) end.
19
20 Fixpoint dsl_exec (stmt: Stmt) (sta: State) (env: Env) (msg: message)
21   (this: address) (evts: eventlist) {struct stmt}: Result :=
22   match stmt with
23   | DSL_prim stmt' => dsl_exec_prim stmt' sta env msg this evts
24   | DSL_seq stmt stmt' =>
25     match dsl_exec stmt sta env msg this evts with
26     | Build_Result sta'' evts'' stop  =>
27       if stop then Build_Result sta'' evts'' stop
28       else dsl_exec stmt' sta'' env msg this evts'' end end.
```

Figure A1. Semantics of DSL statements.

```
1  Definition dsl_ge_le := fun x y (sta: State) (env: Env) (msg: message) =>
2    (negb (Nat.leb (x sta env msg) (y sta env msg))).
3  Definition dsl_ge := fun x y (sta: State) (env: Env) (msg: message) =>
4    (negb (ltb (x sta env msg) (y sta env msg))).
5  Infix ">" := dsl_ge_le. Infix ">=" := dsl_ge.
6
7  Notation "'BalanceOf'" := (fun (sta: State) (env: Env) (msg: message) => balanceOf sta).
8  Notation "'BalanceOf' '[' add ']'" := fun (sta: State) (env: Env) (msg: message) =>
9    (BalanceOf sta env msg) (add sta env msg)).
10 Notation "'msg.sender'" := (fun (sta: State) (env: Env) (msg: message) => msg_sender msg).
11 Notation "'Require' '(' cond ')'" := (DSL_require cond) (at level 200).
12 Notation "'BalanceOf' '[' add ']' '-=' exp" := (DSL_balanceOf_dec add exp) (at level 0).
13 Notation "'FreezeOf' '[' add ']' '+=' exp" := (DSL_freezeOf_inc add exp) (at level 0).
14 Notation "@ stmt_prim" := (DSL_prim stmt_prim) (at level 200).
15 Notation "stmt0 ';' stmt1" := (DSL_seq stmt0 stmt1) (at level 220).
```

Figure A2. Notations that makes the DSL syntax close to Solidity.

In order to verify the invariant issue in the BNB contract, we construct the evaluation step in Figure A3. Based on message calls and function specifications, the `Step` discusses the situation of function calls and the changes in the contract. Each function in the BNB contract corresponds to an execution step. For example, we define the step of the creation function BNB in lines 2-6.

```
1  Inductive Step : Environment -> contract -> message -> contract -> eventlist -> Prop :=
2  | step_BNB: forall en C msg C' E ga send val ins nam dec sym spec,
3     adr C = adr C'-> msg = Build_Message (md_BNB ins nam dec sym) ga send val->
4     spec = funcspec_BNB ins nam dec sym (adr C) en msg ->
5     (require (list bool) spec) (st C) /\ (event (list bool) spec) (st C) E /\
6     (transfer (list bool) spec) (st C) (st C') -> Step en C msg C' E
7  | step_transfer: forall en C msg C' E ga send val to v spec,
8    adr C = adr C' -> msg = Build_Message (md_transfer to v) ga send val ->
9    spec = funcspec_transfer to v (adr C) en msg ->
10   (require (list bool) spec) (st C) /\ (event (list bool) spec) (st C) E /\
11   (transfer (list bool) spec) (st C) (st C') -> Step en C msg C' E
12 | step_approve : forall en C msg C' E ga send val spend v spec,
13   adr C = adr C' -> msg = Build_Message (md_approve spend v) ga send val ->
14   spec = funcspec_approve spend v (adr C) en msg ->
15   (require bool spec) (st C) /\ (event bool spec) (st C) E /\
16   (transfer bool spec) (st C) (st C') -> Step en C msg C' E
17 | step_transferFrom : forall en C msg C' E ga send val from to v spec,
18   adr C = adr C'-> msg = Build_Message (md_transferFrom from to v) ga send val ->
19   spec = funcspec_transferFrom from to v (adr C) en msg ->
20   (require bool spec) (st C) /\ (event bool spec) (st C) E /\
21   (transfer bool spec) (st C) (st C') -> Step en C msg C' E
22 | step_burn: forall en C msg C' E ga send val v spec,
23   adr C = adr C' -> msg = Build_Message (md_burn v) ga send val ->
24   spec = funcspec_burn v (adr C) en msg ->
25   (require bool spec) (st C) /\ (event bool spec) (st C) E /\
26   (transfer bool spec) (st C) (st C') -> Step en C msg C' E
27 | step_freeze: forall en C msg C' E ga send val v spec,
28   adr C = adr C' -> msg = Build_Message (md_freeze v) ga send val ->
29   spec = funcspec_freeze v (adr C) en msg ->
30   (require bool spec) (st C) /\ (event bool spec) (st C) E /\
31   (transfer bool spec) (st C) (st C') -> Step en C msg C' E
32 | step_unfreeze: forall en C msg C' E ga send val v spec,
33   adr C = adr C' -> msg = Build_Message (md_unfreeze v) ga send val ->
34   spec = funcspec_unfreeze v (adr C) env msg ->
35   (require bool spec) (st C) /\ (event bool spec) (st C) E /\
36   (transfer bool spec) (st C) (st C') -> Step en C msg C' E
37 | step_withdrawEther: forall en C msg C' E ga send val a spec,
38   adr C = adr C' -> msg = Build_Message (md_withdrawEther a) ga send val ->
39   spec = funcspec_withdrawEther a (adr C) en msg ->
40   (require (list bool) spec) (st C) /\ (event (list bool) spec) (st C) E /\
41   (transfer (list bool) spec) (st C) (st C') -> Step en C msg C' E.
```

Figure A3. Evaluation step of the BNB contract.

Appendix B. Source Code of the BNB Contract

As shown in Algorithm 1, we give the partial source code of the BNB contract. This includes several main functions involved in the article, such as BNB function, transfer function, burn function, and freeze function. The complete contract source code can be found in the formal working file.

Algorithm 1 Partial source code of BNB contract.

```
pragma Solidity ^0.4.8;

contract BNB is SafeMath {
    string public name;
    string public symbol;
    uint8 public decimals;
    uint256 public totalSupply;
    address public owner;
    mapping (address => uint256) public balanceOf;
    mapping (address => uint256) public freezeOf;
    event Transfer (address indexed from, address indexed to, uint256 value);
    event Burn (address indexed from, uint256 value);
    event Freeze (address indexed from, uint256 value);

    function BNB (uint256 initialSupply, string tokenName, uint8 decimalUnits, string tokenSymbol)
    {
        balanceOf[msg.sender] = initialSupply;
        totalSupply = initialSupply;
        name = tokenName;
        symbol = tokenSymbol;
        decimals = decimalUnits;
        owner = msg.sender;
    }

    function transfer (address _to, uint256 _value) {
        if (_to == 0x0) throw;
        if (_value <= 0) throw;
        if (balanceOf [msg.sender] < _value) throw;
        balanceOf [msg.sender] = SafeMath.safeSub (balanceOf [msg.sender], _value);
        balanceOf [_to] = SafeMath.safeAdd (balanceOf [_to], _value);
        Transfer (msg.sender, _to, _value);
    }

    function burn (uint256 _value) returns (bool success) {
        if (balanceOf [msg.sender] < _value) throw;
        if (_value <= 0) throw;
        balanceOf [msg.sender] = SafeMath.safeSub (balanceOf [msg.sender], _value);
        totalSupply = SafeMath.safeSub (totalSupply,_value);
        Burn (msg.sender, _value);
        return true;
    }

    function freeze (uint256 _value) returns (bool success) {
        if (balanceOf [msg.sender] < _value) throw;
        if (_value <= 0) throw;
        balanceOf [msg.sender] = SafeMath.safeSub (balanceOf [msg.sender], _value);
        freezeOf [msg.sender] = SafeMath.safeAdd (freezeOf [msg.sender], _value);
        Freeze (msg.sender, _value);
        return true;
    }
}
```

References

1. Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. Available online: https://bitcoin.org/bitcoin.pdf (accessed on 18 May 2008).
2. Buterin, V. Ethereum: A next-generation smart contract and decentralized application platform. Available online: https://github.com/ethereum/wiki/wiki/White-Paper (accessed on 18 May 2019).
3. Ethereum Solidity Documentation. Available online: https://Solidity.readthedocs.io/en/develop/ (accessed on 12 December 2019).
4. Luu, L.; Chu, D.H.; Olickel, H.; et al. Making smart contracts smarter. In Proceedings of the ACM SIGSAC Conf. Comput. Commun. Securit, New York, USA, 24-28 October 2016; pp. 254–269.
5. Atzei, N.; Bartoletti, M.; Cimoli, T. A Survey of Attacks on Ethereum Smart Contracts (SoK). In Proceedings of the Int. Conf. Princ. Secur. Trust, Uppsala, Sweden, 22-29 April 2017; pp. 164–186.
6. The DAO Attacked: Code Issue Leads to $60 Million Ether Theft. Available online: https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft/ (accessed on 17 June 2017).
7. Liu, J.; Liu, Z.T. A Survey on Security Verification of Blockchain Smart Contracts. *IEEE Access* **2019**, *7*, 77894–77904.
8. The Coq Proof Assistant Reference Manual. Available online: https://coq.inria.fr/distrib/current/refman/ (accessed on 20 May 2019).
9. Bertot, Y.; Castéran, P. *Interactive theorem proving and program development, Coq'Art: The calculus of inductive constructions*; Springer: Heidelberg, Germany, 2004.
10. Nipkow, T.; Paulson, L.C.; Wenzel, M. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*; Springer-Verlag: Berlin, Germany, 2002.
11. Avigad, J. The Mechanization of Mathematics. *Notices Am. Math. Soc.* **2018**, *65*, 681–690.
12. Gonthier, G. Formal proof – the Four Color Theorem. *Notices Am. Math. Soc.* **2008**, *55*, 1382–1393.
13. Gonthier, G.; Asperti, A.; Avigad, J.; et al. Machine-checked proof of the Odd Order Theorem. In Proceedings of the Interactive Theorem Proving, Rennes, France, 23-26 July 2013; pp. 163–179.
14. Hales, T.C. Formal proof. *Notices Am. Math. Soc.* **2008**, *55*, 1370–1380.
15. Wiedijk, F. Formal proof – getting started. *Notices Am. Math. Soc.* **2008**, *55*, 1408–1414.
16. The CompCert C verified Compiler: Documentation and User's Manual. Available online: http://compcert.inria.fr/man/manual.pdf (accessed on 17 September 2019).
17. Gu, R.H.; Shao, Z.; Chen, H.; et al. CertiKOS: An Extensible Architecture for Building Cerified Concurrent OS Kernels. In Proceedings of the USENIX Symposium on Operating Systems Design and Implementation, Savannah, GA, USA, 2-4 November 2016; pp. 653–669.
18. Appel, A.W.; Beringer, L.; Chlipala, A.; et al. Position paper: the science of deep specifcation. *Phil. Trans. R. Soc. A* **2017**, *375*, 1–24.
19. Alharby, M.; Moorsel, A.V. Blockchain-based smart contracts: A systematic mapping study. In Proceedings of the Fourth International Conference on Computer Science and Information Technology (CSIT 2017), Yerevan, Armenia, 25-29 September, 2017; pp. 125–140.
20. Torres, C.F.; Schütte, J.; State, R. Osiris: Hunting for integer bugs in ethereum smart contracts. In Proceedings of the Annual Computer Security Applications Conference, San Juan, USA, 3–7 December, 2018; pp. 19–34.
21. Bragagnolo, S.; Rocha, H.; Denker, M.; et al. SmartInspect: Solidity smart contract inspector. In Proceedings of the International Workshop Blockchain Oriented Software Engineering, Campobasso, Italy, 20 March, 2018; pp. 9–18.
22. Bartoletti, M.; Pompianu, L. An empirical analysis of smart contracts: Platforms, applications, and design patterns. In Proceedings of the International Conference on Financial Cryptography and Data Security, Sliema, Malta, 3-7 April, 2017; pp. 494–509.
23. Coblenz, M. Obsidian: A safer blockchain programming language. In Proceedings of the IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), Buenos Aires, Argentina, 20-28 May, 2017; pp. 1–11.
24. Magazzeni, D.; Mcburney, P.; Nash, W. Validation and verification of smart contracts: A research agenda. *Computer* **2017**, *50*, 50–57.

25. Bhargavan, K.; DelignatLavaud, A.; Swamy, N.; et al. Short Paper: Formal verification of smart contracts. In Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, Vienna, Austria, 24 October 2016; pp. 91–96.
26. Hirai, Y. Formal Verification of Deed Contract in Ethereum Name Service. Available online: https://yoichihirai.com/deed.pdf (accessed on 1 November 2016).
27. Hirai, Y. Defining the ethereum virtual machine for interactive theorem provers. In Proceedings of the Int. Conf. Financial Cryptogr. Data Secur., Sliema, Malta, 7 April 2017; pp. 520–535.
28. Amani, S.; Begel, M.; Bortin, M.; et al. Towards verifying ethereum smart contract bytecode in Isabelle/Hol. In Proceedings of the ACM Sigplan International Conference, Los Angeles, USA, 8-9 January 2018; pp. 66–77.
29. Grishchenko, I.; Maffei, M.; Schneidewind, C. A Semantic Framework for the Security Analysis of Ethereum smart contracts. In Proceedings of the Principles of Security and Trust, Thessaloniki, Greece, 14-20 April 2018; pp. 243–269.
30. SECBIT. Correctness proofs of Ethereum token contracts. Available online: https://github.com/sec-bit/tokenlibs-with-proofs (accessed on 20 July 2018).
31. Yang, Z.; Lei, H. FEther: An Extensible Definitional Interpreter for Smart-Contract Verifications in Coq. *IEEE Access* **2019**, *7*, 37770–37791.
32. Szabo, N. Smart Contracts: Building Blocks for Digital Markets. Available online: https://kameir.com/smart-contracts/ (accessed on 1996).