# Smart Contract Security
A Software Lifecycle Perspective

HUANG, Yongfeng; BIAN, Yiyang; LI, Renpu; ZHAO, J. Leon; SHI, Peizhong

**Published in:**
IEEE Access

**Published: 01/01/2019**

**Document Version:**
Final Published version, also known as Publisher’s PDF, Publisher’s Final version or Version of Record

**License:**
CC BY

**Publication record in CityU Scholars:**
Go to record

**Published version (DOI):**
10.1109/ACCESS.2019.2946988

**Publication details:**
HUANG, Y., BIAN, Y., LI, R., ZHAO, J. L., & SHI, P. (2019). Smart Contract Security: A Software Lifecycle Perspective. *IEEE Access*, *7*, 150184-150202. Article 8864988. Advance online publication. https://doi.org/10.1109/ACCESS.2019.2946988

Received September 23, 2019, accepted October 7, 2019, date of publication October 11, 2019, date of current version October 28, 2019.

*Digital Object Identifier 10.1109/ACCESS.2019.2946988*

# Smart Contract Security: A Software Lifecycle Perspective

**YONGFENG HUANG[1,3], YIYANG BIAN[2,3], RENPU LI[1], J. LEON ZHAO[3], AND PEIZHONG SHI[1,3]**
[1]School of Computer Engineering, Jiangsu University of Technology, Changzhou 213001, China
[2]School of Information Management, Nanjing University, Nanjing 210000, China
[3]Department of Information Systems, City University of Hong Kong, Hong Kong

Corresponding author: Yiyang Bian (bianyiyang@nju.edu.cn)

This work was supported in part by the Shenzhen Science and Technology Innovation Committee under Grant JCY20170818100156260, in part by the National Natural Science Foundation of China under Grant 61472166, Grant 61672270, and Grant 61602216, in part by the Humanities and Social Science Research Fund of the Ministry of Education in China under Grant 18YJC840045, in part by the Natural Science Foundation of Jiangsu Province under Grant BK20161199, and in part by the Natural Science Foundation of the Jiangsu University of Technology under Grant KYY16033.

**ABSTRACT** Smart contract security is an emerging research area that deals with security issues arising from the execution of smart contracts in a blockchain system. Generally, a smart contract is a piece of executable code that automatically runs on the blockchain to enforce an agreement preset between parties involved in the transaction. As an innovative technology, smart contracts have been applied in various business areas, such as digital asset exchange, supply chains, crowdfunding, and intellectual property. Unfortunately, many security issues in smart contracts have been reported in the media, often leading to substantial financial losses. These security issues pose new challenges to security research because the execution environment of smart contracts is based on blockchain computing and its decentralized nature of execution. Thus far, many partial solutions have been proposed to address specific aspects of these security issues, and the trend is to develop new methods and tools to automatically detect common security vulnerabilities. However, smart contract security is systematic engineering that should be explored from a global perspective, and a comprehensive study of issues in smart contract security is urgently needed. To this end, we conduct a literature review of smart contract security from a software lifecycle perspective. We first analyze the key features of blockchain that can cause security issues in smart contracts and then summarize the common security vulnerabilities of smart contracts. To address these vulnerabilities, we examine recent advances in smart contract security spanning four development phases: 1) security design; 2) security implementation; 3) testing before deployment; and 4) monitoring and analysis. Finally, we outline emerging challenges and opportunities in smart contract security for blockchain engineers and researchers.

**INDEX TERMS** Blockchain, Ethereum, information security, smart contract, software engineering, software lifecycle.

## I. INTRODUCTION

As a decentralized and tamper-proof ledger, blockchain has been portrayed as an ultimate security technology in many respects, such as artificial intelligence (AI) [1]–[4], big data [5], [6], Internet of Things (IoT) [7]–[9] and digital property (e.g., Deepfake [10] and Proof of delivery [11]). However, blockchain technology still faces numerous security issues [12]–[14]. Especially with the increase of decentralized applications (Dapps) running on blockchains, smart contract security is becoming more and more significant (see Figure 1) and has attracted much attention from researchers [16]–[19]. Although the consensus protocol ensures the faithful execution of smart contracts, smart contracts still have many security issues. These security issues are especially severe in public blockchains because the environment where smart contracts execute is mostly decentralized, and a vulnerable smart contract is hard to patch. For this reason, an attacker in the DAO [20] was able to exploit a bug in a smart contract to repeatedly siphon off money, which caused the investors to lose approximately $50 million in cryptocurrency value. Therefore, effective security solutions for smart contracts are urgently needed.

The associate editor coordinating the review of this manuscript and approving it for publication was Francisco J. Garcia-Penalvo.

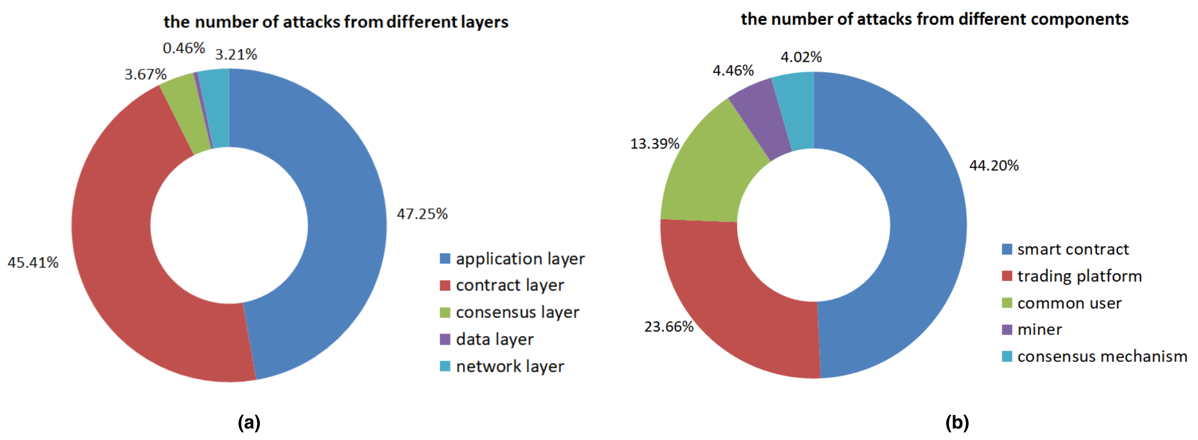


**FIGURE 1.** The number of attacks from smart contracts accounts for a significant proportion of the number of attacks from both (a) different layers and (b) different components (data statistics as of March 15, 2019, from BCSEC [15]).

This paper proposes a specific software lifecycle approach to rationalize how to tackle the issue of smart contract security. Compared to previous works [16], [19], [21] concerning this topic, our paper excels in several aspects.

1) Previous works have focused only on vulnerability detection. Our research has a wider focus and includes not only vulnerability detection but also security modeling, security monitoring, bug bounties, etc. Specifically, our paper offers a novel perspective for understanding smart contract security in a visualized manner, which enables developers to track, control, and avoid blockchain project risks systematically.

2) By considering smart contract security as a new type of software weakness mitigation in a blockchain setting, our paper systematically analyzes the causes of security issues in smart contracts for the first time.

3) Most previous works surveyed smart contract security only in Ethereum. However, our research outcome has broad impact and is not limited to Ethereum. Without precedent, we have compared security vulnerabilities of smart contracts in Ethereum [22] and Fabric [23], two widely used blockchain platforms, from three aspects: programming languages, blockchain platforms, and misunderstanding of common practices. The same comparison can be extended to other blockchain platforms.

The rest of this paper is organized as follows. Section II analyzes the causes of security issues in smart contracts. A discussion of security vulnerabilities in different platforms and anomalous activities in smart contracts is presented in Section III. Section IV examines the existing solutions of smart contract security in terms of security themes from the perspective of the software lifecycle. Then, the emerging challenges and opportunities are proposed in Section V. Finally, Section VI concludes the paper.

## II. ANALYSIS OF SECURITY ISSUES IN SMART CONTRACTS

A smart contract can be assumed as a mapping of a legal agreement in reality. Once a smart contract is confirmed by the consensus protocol and submitted to the blockchain, it will be run in terms of the prior negotiation without the interference of any third party. Owing to success in Ethereum [22], smart contracts have been widely supported by most of the current blockchains, such as Fabric [23], Corda [24], and EOS [25]. However, several key features of current blockchains may cause security issues in smart contracts.

1) *Decentralization and tamper-proofing are double-edged swords.* Blockchain is decentralized and tamper-proof. Moreover, smart contracts can be developed and deployed by pseudonymous malicious people (only public addresses or public keys are known to others in most public blockchains). Therefore, a vulnerable smart contract is hard to patch and can easily become out of control once deployed.

2) *Open-source code and public ledgers.* Generally, a smart contract is open-source. Contract transactions and data may be visible to an adversary. The exposure leads to the fact that smart contract vulnerability is easy to exploit.

3) *Immaturity of blockchain platforms and smart contract languages.* Dapps development in the blockchain is different from traditional application development. Developers of Dapps need a thorough understanding of the operations on the blockchain. Otherwise, the intention of smart contract developers is often inconsistent with smart contract implementation. Moreover, the blockchain technologies are evolving so fast that design flaws may exist in blockchain platforms or smart contract languages. Developers of Dapps are always confronted with changing platform features. Thus, common software weaknesses [26] (we have illustrated some weaknesses related to smart contract security in Table 1) may be amplified on blockchain platforms.

4) *Misunderstanding of common practices.* Smart contract developers often do not thoroughly understand the principles of some practices of the blockchain. For example, a misunderstanding of cryptography may lead to security being for granted, which does not facility true security.

**TABLE 1.** Common software weakness.

| Weakness | Explanation |
|---|---|
| *Improper Behavioral Workflow* | When several behaviors must be performed, the software does not ensure that the behaviors are performed in the required sequence. |
| *Improper Access Control* | The code incorrectly gives access to a resource to an unauthorized actor. |
| *Incorrect Calculation* | The program performs a calculation that generates incorrect or unintended results that may be later used in security-critical decisions or resource management. |
| *Improper Initialization* | The software does not initialize or incorrectly initializes a resource, which might leave the resource in an unexpected state. |
| *Race Condition* | A code sequence can run concurrently with other code sequences, where a shared resource can be modified by the other code sequence that is operating concurrently. |
| *Inclusion of Functionality from Untrusted Control* | The code includes executable functionality from an untrusted source that is out of control. |
| *Use of Insufficiently Random Values* | The program may use insufficiently random numbers or values in a security context that depends on predictable numbers. |
| *Improper Handling of Exceptional Conditions* | The program does not correctly handle exceptional conditions that rarely occur, which may be exploited by malicious actors. |
| *Improper Cryptographic Understanding* | The developer incorrectly understands the principles of cryptography, which may be exploited by malicious actors. |

5) *Pseudonymous transactions*. On most of the public blockchains, transactions are pseudonymous. This feature also stimulates criminal activities such as money laundering [27] or Ponzi schemes [28] in smart contracts.

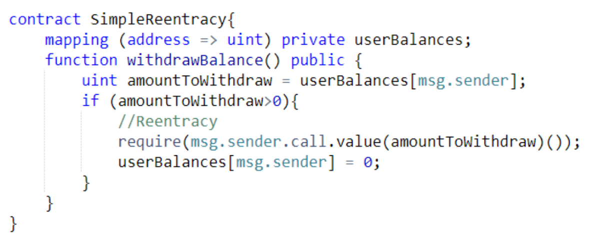
Smart contracts are often used to transfer financial assets. Security issues in smart contracts may lead to a large number of financial losses. An example is the DAO hack in Ethereum [20], which caused a hard fork of the blockchain to nullify the malicious transactions. Therefore, the risk of security issues in smart contracts is generally more severe than that in traditional applications.

## III. CLASSIFICATION OF SECURITY ISSUES IN SMART CONTRACTS

Different vulnerabilities may exist in different blockchain systems. In this section, we mainly discuss security vulnerabilities in smart contracts on Ethereum [22] and Fabric [23]. As the two most widely applied blockchains, Ethereum and Fabric adopt two different but representative technologies in smart contract implementation. Ethereum utilizes the domain-specific language (DSL) Solidity to write smart contracts, whereas Fabric uses a general-purpose programming language (e.g., Go or Java) to develop smart contracts (also called chaincodes in Fabric). Different platforms and languages cause these two types of smart contracts to exhibit distinguishing security vulnerabilities.

### A. SECURITY VULNERABILITIES IN ETHEREUM SMART CONTRACTS

Based on the previous works of [16], [19], [21], [29], [30], we summarize primary security vulnerabilities in Ethereum smart contracts in Table 2. We think that these vulnerabilities arise from three aspects: the Solidity language, the blockchain platform, and a misunderstanding of common practices. We relate these vulnerabilities to common software weaknesses. In Table 2, we offer a preliminary understanding of security vulnerabilities in Ethereum smart contracts. In the following, we will restate these vulnerabilities, explain their rationale, and propose simple schemes to fix them.



```
contract SimpleReentracy{
    mapping (address => uint) private userBalances;
    function withdrawBalance() public {
        uint amountToWithdraw = userBalances[msg.sender];
        if (amountToWithdraw>0){
            //Reentracy
            require(msg.sender.call.value(amountToWithdraw)());
            userBalances[msg.sender] = 0;
        }
    }
}
```

**FIGURE 2.** An example of reentrancy.

1) *Reentrancy*. A reentrancy attack may occur when a contract calls an external contract that takes over the control flow and calls back into the calling contract before the first invocation is finished. This attack may have an unexpected consequence. As shown in Figure 2, when an external contract calls the function *withdrawBalance*, the function *withdrawBalance* will call the fallback of *msg.sender* when calling *require*. In turn, the fallback function can call into *withdrawBalance* again. Thus, the contract *SimpleReentracy* will send the balance to *msg.sender* repeatedly because the assignment statement "*userBalances* [*msg.sender*] = 0" has not been executed thus far.

*Fix Scheme*. Do not call an external function until the developer has done all of the internal work he needs to do.

2) *Unprotected selfdestruct*. This vulnerability arises from a logic flaw in Solidity. A smart contract incorrectly permits access to an unauthorized actor. For example, due to missing or insufficient access controls, malicious parties can call the function *selfdestruct* (see Figure 3) to destruct the contract. Thus, the balance in the destructed contract will be transferred to an unauthorized account.

*Fix Scheme*. Consider removing the *selfdestruct* functionality unless it is needed.

3) *Integer underflow* (overflow). An underflow (overflow) occurs when an arithmetic operation reaches the minimum

**TABLE 2. Security vulnerabilities in Ethereum smart contracts.**

| Cause | No | Vulnerability name | Weakness (see Table.1) | Example | Severity |
|---|---|---|---|---|---|
| Solidity language | 1 | *Reentrancy* | Improper behavioral workflow | the DAO [20] | severe |
| | 2 | *Unprotected selfdestruct* | Improper Access Control | "I accidentally killed it" bug [31] | severe |
| | 3 | *Integer underflow (overflow)* | Incorrect Calculation | BeautyChain Token Bug [32] | severe |
| | 4 | *Locked money* | Improper Initialization | Transferring to 0x0 address [33] | severe |
| | 5 | *Delegatecall to untrusted contracts* | Inclusion of Functionality from Untrusted Control | see Figure 5 | severe |
| Blockchain platform | 6 | *Transaction order dependence* | Race condition | see Figure 6 | medium |
| | 7 | *Weak randomness from chain attributes* | Use of Insufficiently Random Source | see Figure 7 | medium |
| | 8 | *Timestamp dependence* | Inclusion of Functionality from Untrusted Control | theRun [34] | medium |
| Misunderstanding of common practices | 9 | *Mishandled exceptions* | Improper Handling of Exceptional Conditions | see Figure 8 | severe |
| | 10 | *Replay attack* | Improper Cryptographic Understanding | see Figure 9 | severe |

```
contract SimpleSuicide {
    function sudicideAnyone() {
        //unprotected selfdestruct
        selfdestruct(msg.sender);
    }
}
```

**FIGURE 3. An example of unprotected self-destruct.**

```
contract IntegerUnderflowMinimal {
    uint public count = 0;
    function run(uint256 input) public {
        count -= input; //underflow minimal
    }
}
```

**FIGURE 4. An example of integer underflow.**

(maximum) of a type. As shown in Figure 4, an integer underflow occurs when a subtract operation attempts to create a value that is outside of the unit type range (0∼255), which will cause a misjudgment of the count value. This vulnerability exists in many platforms and smart contract languages.

*Fix Scheme*. Use SafeMath [35] libraries for all arithmetic operations in the smart contract.

4) *Locked money*. This vulnerability stems from the immaturity of blockchain platforms. An operator sometimes forgets to enter the address he expects to transfer to. Because the default value of an empty field for an address is 0x0 in some wallets such as Parity, money is often locked in this address.

*Fix Scheme*. Check the address before transferring.

5) *Delegatecall to untrusted contracts*. This vulnerability mainly arises from the Solidity language. A special function in contract *A*, namely *delegatecall*, may call a function of another untrusted contract *B* (e.g., *callee* in Figure 5). When calling into contract *B*, the context such as *msg.sender* is still identical to the previous context of contract *A*. Calling into untrusted contract *B* is very dangerous, because the code in *B* can change any storage values of *A* and thus can completely control the balance of *A*.

*Fix Scheme*. Use the function *delegatecall* with caution and ensure that you never call into an untrusted contract.

```
contract DelegateCallProxy {
  address owner;
  function DelegateCallProxy public {
    owner = msg.sender;
  }
  function forward(address callee, bytes _data) public {
    require(callee.delegatecall(_data));
  }
}
```

**FIGURE 5. An example of delegatecall to untrusted contracts.**

```
contract EthClaimReward {
    address public owner;
    bool public claimed;
    uint public reward;

    function EthClaimReward() public {
        owner = msg.sender;
    }

    function setReward() public payable {
        require (!claimed);
        require(msg.sender == owner);
        owner.transfer(reward);
        reward = msg.value;
    }

    function claimReward() public {
        require (!claimed);
        msg.sender.transfer(reward);
        reward = 0;
        require (!claimed);
    }
}
```

**FIGURE 6. An example of transaction order dependence.**

6) *Transaction order dependence*. This vulnerability arises from a feature of the blockchain. As shown in Figure 6, a contract EthClaimReward will give a reward to the first person who solves a math problem (the contract owner will initialize the value of the reward by calling the function *setReward*). Assuming that *Alice* solves the problem and submits the answer by calling *claimReward* with a standard gas price, *Eve* can see the answer that *Alice* just submitted because the ledger is public. Now *Eve* can resubmit the answer with a much higher gas price, and thus *Eve*'s transaction probably

```
contract UnsafeDependenceOnBlock {
    uint8 answer;
    function UnsafeDependenceOnBlock() public payable {
        require(msg.value == 1 ether);
        answer =
            uint8(keccak256(block.blockhash(block.number - 1), now));
    }
    function isComplete() public view returns (bool) {
        return address(this).balance == 0;
    }
    function guess(uint8 n) public payable {
        require(msg.value == 1 ether);
        if (n == answer) {
            msg.sender.transfer(2 ether);
        }
    }
}
```

**FIGURE 7.** **An example of weak randomness.**

gets processed and committed before *Alice*'s. In that case, *Eve* will receive the reward even though *Alice* was the first person to solve the problem.

*Fix Scheme*. Use a commitment scheme that is carried out in two phases. (I) Instead of submitting a transaction with the answer, any person who solves the problem first submits a transaction with a hash of [*salt*, *address*, *answer*], where *salt* may be a random value and [] is a combination operator of multiple values. The contract stores this hash value and the sender's address. (II) To claim the reward, any person who submits the hash in the previous phase will continue to submit a transaction with his *salt* and *answer*. The contract then calculates the hash of [*salt*, *msg.sender*, *answer*] and checks this hash against the stored hash. The person who first submits the matched hash receives the reward.

7) *Weak randomness from chain attributes*. Because many contracts in Ethereum are open-source, it is challenging to create a sufficiently strong source of randomness in Ethereum. Attackers can easily predict the random number generated by an algorithm with custom seeds using the corresponding block information. An example is gambling Dapps, such as *UnsafeDependenceOnBlock* in Figure 7, where a pseudo-random number generator is used to pick the winner, and thus the answer can be easily guessed because the value of *block.number* is predictable when people guesses.

*Fix Scheme*. Use a commitment scheme that is carried out in two phases: (I) the commit phase, during which a random number for the answer is chosen and specified; in this phase, the answer is quite unpredictable; and (II) the reveal phase during which the answer is revealed and verified; in this phase, the answer becomes a determined value.

8) *Timestamp dependence*. Smart contracts often use a block timestamp to trigger conditions to execute some critical operations. For example, a smart contract may depend on a block timestamp to send out money (see theRun [34]). Malicious miners can adjust the timestamp to a specific value that influences the timestamp-dependent condition and favors them.

*Fix Scheme*. Do not use a block timestamp as a random seed to trigger conditions. Meanwhile, use the previous commitment scheme.

9) *Mishandled exceptions*. This vulnerability mainly arises from Solidity. If an external function in a contract contains

```
contract DoS_Auction {
    address highestBidder;
    uint highestBid;
    function bid() payable {
        require(msg.value >= highestBid);
        if (highestBidder != address(0)) {
            //fails may leas to Dos
            highestBidder.transfer(highestBid);
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
    }
}
```

(a)

```
contract DoS_Auction_fix {
    address highestBidder;
    uint highestBid;
    mapping(address => uint) refunds;
    function bid() payable external {
        require(msg.value >= highestBid);
        if (highestBidder != address(0)) {
            // record the refund that this user can claim
            refunds[highestBidder] += highestBid;
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
    }
    function withdrawRefund() external {
        uint refund = refunds[msg.sender];
        refunds[msg.sender] = 0;
        msg.sender.transfer(refund);
    }
}
```

(b)

**FIGURE 8.** **An example of mishandled exceptions.**

many operations that may use up gas, calling such a costly function may trigger an exception. A mishandled exception may cause an attack, such as DOS (denial of service), on the on-going contract. As shown in Figure 8 (a), if a malicious bidder in an auction becomes a leader, he can remain the leader forever because he can prevent anyone else from successfully calling the function *bid* via a costly fallback function.

*Fix Scheme*. Set up a pull payment system to isolate each external call into an independent transaction from the function *bid* so that the recipient of the call can initiate the independent transaction. As shown in Figure 8 (b), the function *withdrawRefund* enables users to withdraw funds by themselves rather than pushing funds to them in the function *bid*, which reduces the chance of calling an external function in *bid*.

10) *Replay attack*. Digital signatures can be used for identity authentication. However, by intercepting and replaying the user's previous signature, a malicious user can impersonate a specific user [36]. Figure 9 shows a smart contract that aims to solve the problem of users being unable to transfer their tokens unless they have enough ether to pay for the transaction fees. By invoking *transferProxy*, a user *_from* can transfer tokens with a value of *_value* to another user *_to* with a valid elliptic curve signature represented by (*_v*, *_r*, *_s*). A replay attack is performed as follows. (I) *Alice*, i.e., *A*, initiates a transaction by calling *transferProxy* (*A, B, 100, 3, sig*) in which 100 Token will be sent to *Bob*, i.e., *B*, and 3 Token will be paid to Proxy as a service fee. To identify herself, *Alice* then signs the transaction with her signature

**TABLE 3.** Security vulnerabilities in fabric chaincodes.

| Cause | No | Vulnerability name | Rationale |
|---|---|---|---|
| **Go language** (nondeterminism arising from Go) | 1 | *Global variables* **(Global state variables [37])** | Global variables are only global to a single node. If such a node breaks down, global state variables might no longer be consistent over all peers. Put a global state to the ledger will lead to inconsistency. |
| | 2 | *KVS structure iteration* **(Iterate over map [37])** | Because (key, value) is returned in a random order in Go, use of the range to iterate over the entries of a map is not deterministic. |
| | 3 | *Reified object addresses* | Addresses of memory depend on the environment. |
| | 4 | *Concurrency of program* **(Goroutines [37])** | Concurrency easily leads to nondeterministic behavior in Go. |
| | 5 | *Random number generation* | Each peer may obtain different results when generating a random number. |
| | 6 | *System timestamp* | It is difficult to ensure timestamp functions are executed at the same time for each peer. |
| | 7 | *Field declarations* **(Field declarations [37])** | Like global variables, the value of a variable declared in a structure field may not sustain the same value among peers. |
| | 8 | *Access resources outside of the blockchain* **(Blacklisted imports [37])** | The results of accessing the resources outside of the blockchain (e.g., a web service, system command execution, external file access, and external library call [38]) may be inconsistent among peers. |
| **Blockchain platform** (undesired behavior arising from platform features) | 9 | *Range query risk* **(Phantom reads [37])** | Some range query methods, such as GetQueryResult, are not re-executed during the validation phase, which means that phantom reads (dirty data) are not detected. |
| | 10 | *Read your write* **(Read after write [37])** | For a write statement to take effect, a transaction first has to be committed. Therefore, when reading a value that has already been written during the same transaction, its old value will be retrieved from the ledger. This is not the expected behavior. |
| **Misunderstanding of common practices** | 11 | *Unhandled errors* **(Unhandled errors [37])** | Ignored errors might lead to faulty execution. Therefore, developers should not ignore return values related to errors. |
| | 12 | *Unchecked input arguments* **(Unchecked Arguments [37])** | The input arguments should always be checked to avoid accessing a nonexistent element. |

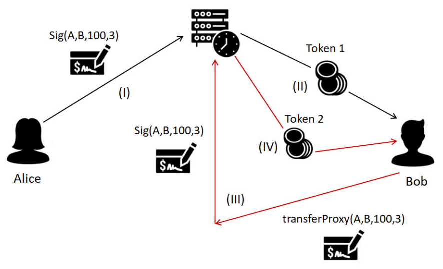
of *sig*(*A, B, 100, 3*). (II) The transaction is carried out by Proxy. *Bob* gets 100 Token from *Alice* the first time. (III) *Bob* initiates a new transaction by calling *transferProxy* (*A, B, 100, 3, sig*), where *sig* is a replay of *sig*(*A, B, 100, 3*). (IV) The new transaction is carried out by Proxy. *Bob* gets 100 Token for a second time, this time without authorization.

*Fix Scheme*. Add an incremental nonce, the name of the blockchain and the address of the calling user into the signature. Use the information submitted by the calling user and the shared nonce to construct the hash value (i.e., byte32 *h* in *transferProxy*) to verify the signature [36].

## B. SECURITY VULNERABILITIES IN FABRIC CHAINCODES

There are very few papers [37], [38] focused on security vulnerabilities in Fabric chaincodes. Based on [37] and [38], we summarize primary security vulnerabilities in Fabric chaincodes in Table 3, where we consider Go as the programming language because Go is most widely used in Fabric chaincode development.

Similar to the vulnerabilities in Ethereum smart contracts, the vulnerabilities in Fabric chaincodes also arise from three aspects: the Go language, the blockchain platform, and a misunderstanding of common practices. Different from common software weaknesses, most security vulnerabilities in Fabric chaincodes arise from the nondeterministic behavior of Go, which may lead to consensus failure. Examples of the vulnerabilities in Table 3 can be found in [38] (e.g., Listing 2 in [38]) or [39] (readers are able to run a demo on the homepage of [39] to get a report that includes examples of nine security vulnerabilities in [37]). Moreover, because



(a)

```
contract TransferProxy_Contract {
    //_v,_r and _s represend signature
    function transferProxy(address _from, address _to,
        uint256 _value, uint256 _fee,
        uint8 _v, bytes32 _r, bytes32 _s) public returns (bool){
        if(balances[_from] < _fee + _value || _fee > _fee + _value)
            revert();
        bytes32 h = keccak256(_from,_to,_value,_fee);
        if(_from != ecrecover(h,_v,_r,_s)) revert();
        if(balances[_to] + _value < balances[_to] ||
            balances[msg.sender] + _fee < balances[msg.sender])
            revert();
        balances[_to] += _value;
        balances[msg.sender] += _fee;
        balances[_from] -= _value + _fee;
        return true;
    }
}
```

(b)

**FIGURE 9.** An example of a replay attack [36].

Fabric has no native cryptocurrency, it is difficult to determine the severity of these vulnerabilities.

## C. DISCUSSION OF VULNERABILITIES ON DIFFERENT PLATFORMS

Different platforms and programming languages cause the vulnerabilities in smart contracts on different platforms to exhibit distinguishing characteristics regarding three aspects.

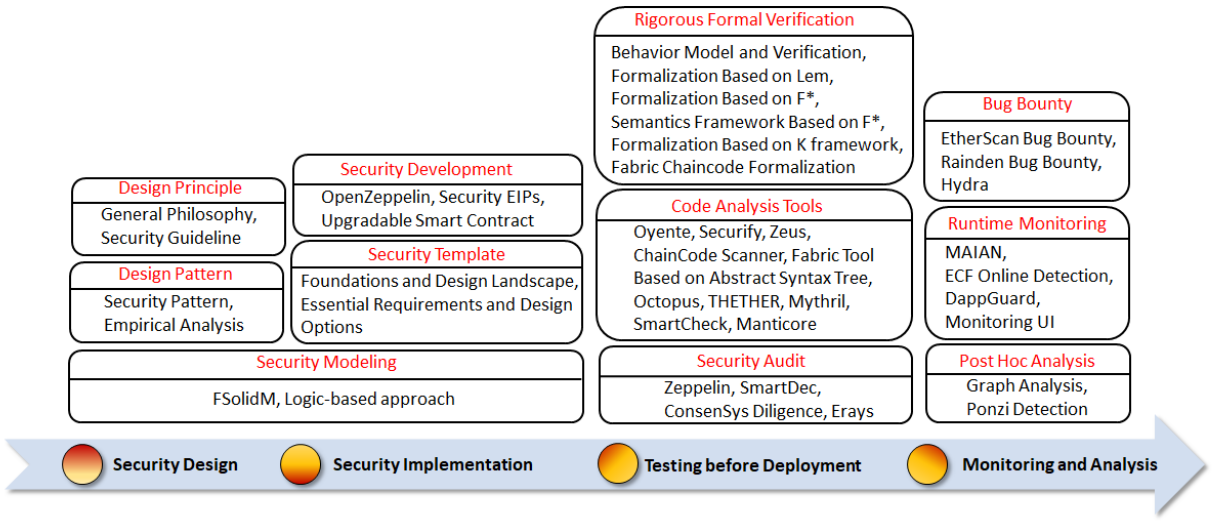

Rigorous Formal Verification
Behavior Model and Verification, Formalization Based on Lem, Formalization Based on F*, Semantics Framework Based on F*, Formalization Based on K framework, Fabric Chaincode Formalization
Bug Bounty
EtherScan Bug Bounty, Rainden Bug Bounty, Hydra
Design Principle
General Philosophy, Security Guideline
Security Development
OpenZeppelin, Security EIPs, Upgradable Smart Contract
Code Analysis Tools
Oyente, Securify, Zeus, ChainCode Scanner, Fabric Tool Based on Abstract Syntax Tree, Octopus, THETHER, Mythril, SmartCheck, Manticore
Runtime Monitoring
MAIAN, ECF Online Detection, DappGuard, Monitoring UI
Design Pattern
Security Pattern, Empirical Analysis
Security Template
Foundations and Design Landscape, Essential Requirements and Design Options
Security Modeling
FSolidM, Logic-based approach
Security Audit
Zeppelin, SmartDec, ConsenSys Diligence, Erays
Post Hoc Analysis
Graph Analysis, Ponzi Detection
Security Design
Security Implementation
Testing before Deployment
Monitoring and Analysis

**FIGURE 10.** The overview of security themes from a smart contract lifecycle perspective.

#### 1) SOME VULNERABILITIES ARISE FROM SPECIFIC LANGUAGES

These vulnerabilities, such as no.1∼ no.5 in Table 2 and no.1∼ no.5 in Table 3, may only appear in the corresponding platform. For instance, the vulnerability of *unprotected self-destruct* arises from a flaw of Solidity. Accordingly, this vulnerability may only exist in Ethereum (no. 3 in Table 2 may be an exception because this vulnerability may exist in many programing languages). Moreover, because nondeterminism may exist in general-purpose programming languages (e.g., Go), some vulnerabilities, such as *global variables* and *iterate over map*, may appear in Fabric chaincodes. However, due to specific restrictions on Solidity (e.g., there are no explicit instructions in Solidity to generate random numbers or access files outside of the EVM), these nondeterministic vulnerabilities do not exist in Ethereum smart contracts.

#### 2) SOME VULNERABILITIES ARISE FROM FEATURES OF THE SPECIFIC BLOCKCHAIN PLATFORM

These security vulnerabilities, such as no. 6∼ no. 8 in Table 2 and no. 9∼ no. 10 in Table 3, only appear on the corresponding platform. For example, the vulnerability of *transaction order dependence* only occurs when an accounting measure for runtime with gas [22] exists in the blockchain. This vulnerability does not exist in Fabric chaincodes. Alternately, the vulnerability of *read your write* does not exist in Ethereum smart contracts.

#### 3) SOME VULNERABILITIES ARISE FROM A MISUNDERSTANDING OF COMMON PRACTICES

These vulnerabilities may appear on most blockchain platforms, including Ethereum and Fabric.

To date, there are at least dozens of blockchain platforms that support smart contracts. Saini summarized the characteristics of 40 different smart contract platforms [40]. These platforms may use different programming languages and have distinguishing platform features. It is difficult to compare all of these platforms. However, the same analysis regarding their comparison to Fabric chaincodes or Ethereum smart contracts concerning vulnerabilities can be conducted based on the above three aspects.

### D. ANOMALOUS ACTIVITIES

In most of the existing blockchain platforms, pseudonymous transactions provide a nest for criminal smart contracts [27]. Because a smart contract is hard to patch for bugs, how to monitor anomalous activities in a smart contract after its deployment and apply appropriate countermeasures should be considered in advance. Some approaches for effectively detecting and preventing the proliferation of malicious behaviors in smart contracts are encouraged [28], [41].

## IV. SECURITY SOLUTIONS FOR SMART CONTRACTS

Smart contract security is systematic engineering that should be explored from a global perspective. Accordingly, we classify current smart contract security solutions in terms of the evolution of the contract lifecycle**.** Similar to the traditional software lifecycle, we divide the contract lifecycle into four phases: *security design*, security *implementation*, *testing before deployment*, and *monitoring and analysis*. In any of these phases, contract security is paramount. Figure 10 illustrates the state of the art of smart contract security based on these phases.

The horizontal axis in Figure 10 shows the evolution of the smart contract lifecycle. We clustered related research works into different themes spanning one or more phases. Each theme is represented by a rounded rectangle with a red title. For each theme, different types of related works, which are separated by commas, are shown in the body of the corresponding rectangle. We illustrate these themes in terms of different lifecycle phases in the following sections.

**TABLE 4. Smart contract security solutions in the phase of *security design*.**

| Security Themes | Related Work | Platforms | Keywords | Open source Tool |
|---|---|---|---|---|
| *Design Principle* | General Philosophy [42] | Ethereum | Prepare for failure, Rollout carefully, Keep contracts simple, Stay up to date, Be aware of blockchain properties | No |
| | Security Guideline [43] | EOS | | |
| *Design Pattern* | Security Pattern [44] | Ethereum | Check-effects-interaction, Emergency stop, Mutex, Speed bump, Rate Limit, Balance Limit | No |
| | Empirical Analysis [45] | Ethereum | Authorization, Time constraint, Oracle, Math, Termination | No |
| *Security Modeling* | FSolidM [46] | Ethereum | Finite state machine, Code generator, Plugins for security enhancement | Yes |
| | Logic-based approach [47] | Ethereum | Procedure languages, Logic-based languages | No |

## A. SECURITY DESIGN

As shown in Figure 10, three themes (i.e., *design principle*, *design pattern*, and *security modeling*) span the phase of *security design*. These themes focus on how contracts are designed to avoid security issues. Notably, models constructed in the theme of *security modeling* may be transformed into contract implementation directly. We have summarized all related works in Table 4, where platforms, keywords, and whether open-source tools are available or not are listed. Meanwhile, the corresponding websites are shown in the APPENDIX if open-source tools are available.

### 1) DESIGN PRINCIPLE

To design secure smart contracts, reference [42] proposed a general philosophy for Ethereum and reference [43] proposed security guidelines for EOS. Both of them proposed five essential design principles (see the keywords in Table 4), which present the methodology for designing secure contracts on the blockchain. For example, the principle of *Prepare for failure* indicates that the contract code must be able to respond to bugs gracefully due to the lack of patching schemes. For instance, if an attack such as DAO [20] occurs, the contract must be able to pause to avoid further financial losses.

### 2) DESIGN PATTERN

In software engineering, a design pattern describes an abstraction of a reoccurring problem and presents a standard solution. Six security patterns (see the keywords in Table 4) are outlined in [44] to address vulnerabilities in Ethereum. For example, the *mutex* pattern can provide a solution to reentrancy vulnerability, and the *emergency stop* pattern can stop the execution of a contract if malicious behaviors are detected. By manually inspecting the source code of 811 Solidity contracts, Bartoletti *et al.* [45] identified some common design patterns (see the keywords in Table 4), some of which, such as *authorization* and *time constraint*, can also address some security issues in the blockchain.

### 3) SECURITY MODELING

Due to semantic inconsistency between the code implementation and the contract requirements, the process of coding a contract using a Turing-complete language such as Solidity

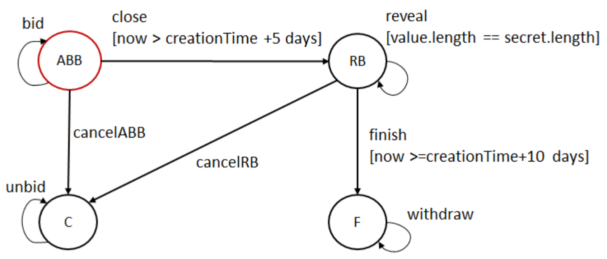

bid
close
[now > creationTime +5 days]
reveal
[value.length == secret.length]
ABB
RB
cancelABB
cancelRB
finish
[now >=creationTime+10 days]
unbid
C
F
withdraw

**FIGURE 11. An example of FSM for blind auctions from [46].**

can be error-prone. Moreover, the order of instruction also affects the correctness of the contract execution. Accordingly, recent works [46], [47] were proposed to formalize contract clauses to enhance security. We summarize them into the theme of *security modeling*. As shown in Figure 10, this theme spans two phases: *security design* and *security implementation*. In this theme, a high-level specification derived from business logic is characterized by rigorous and precise semantics, which facilitates formal verification while mitigating the vulnerabilities in the implementation.

FSolidM [46] allows developers to define smart contracts as finite state machines (FSMs) to help developers create more secure contracts by design. For instance, Figure 11 shows an example of FSMs for blind auctions [46], where *ABB* is the initial state of the FSM. Each transition (e.g., *bid* and *close*) is associated with an action that a user can perform and may have a corresponding guard for the action. FSolidM provides an easy-to-use editor and a code generator for creating FSMs and automatically generating Ethereum contract code. Additionally, FSolidM offers a plugin mechanism to enable developers to easily eliminate security vulnerabilities in contracts.

Recently, Idelberger *et al.* proposed the concept of logic-based smart contracts [47]. Compared with a smart contract based on a procedural language, such as Solidity, a logic-based smart contract has distinct technical advantages in bridging the gap between the contract implementation and the legal prose because a logic-based smart contract is easy to understand by involved entities during the process of negotiation and dispute resolution. With the logic-based approach, the programmer can write smart contracts only by describing contractual clauses (what has to be done) instead of writing

**TABLE 5.** Smart contract security solution in the phase of *security implementation*.

| Security Themes | Related Work | Platforms | Keywords | Open Source Tool |
|---|---|---|---|---|
| *Security Development* | OpenZeppelin [48] | Ethereum | Role-based Access control, Cryptography Utility, Node.js, Truffle | Yes |
| | Security EIPs [49] | Ethereum | EIP155, EIP214, … | No |
| | Upgradeable Contract [50] | Ethereum | Master-slave, Eternal storage, Upgradeable storage proxy, Unstructured upgradeable storage proxy | No |
| *Security Template* | Foundations and Design Landscape [51] | Δ[a] | Agreement, Template and parameters, Standardized code | No |
| | Essential Requirements and Design Options [52] | Δ | Essential requirement, Key design options | No |
| | Legal semantics and code validation [53] | Δ | Legal semantics, code validation | No |

[a] In this paper, symbol Δ denotes no specific platform or keywords for the corresponding related work.

down detailed steps (how to do it). These contractual clauses will then be encoded into logic rules, upon which a rule-based engine would reason and execute. Accordingly, many security issues could be avoided due to the absence of manual coding in this process.

## B. SECURITY IMPLEMENTATION

*Security implementation* includes three themes: *security development*, *security template*, and *security modeling*. We have explained *Security Modeling* previously. In this subsection, we mainly discuss *security development* and *security template*. We have summarized these two themes in Table 5.

### 1) SECURITY DEVELOPMENT

This theme concerns how developers of smart contracts can avoid security vulnerabilities during the process of contract implementation.

OpenZeppelin [48] is a security library for developing Ethereum smart contracts. It offers out-of-the-box role-based access control of the blockchain, as well as cryptography utilities such as SafeMath [35]. OpenZeppelin can be installed directly into Dapps using node.js and integrated with Truffle [54], an Ethereum development environment.

Security EIPs (Ethereum improvement proposals [49]) present appropriate proposals to avoid security vulnerabilities on the Ethereum platform. For example, EIP 155 provides a way to address simple replay attacks. Security EIPs are very significant for the implementation of secure smart contracts.

Some implementation skills are also exploited to enhance contract security. Reference [50] presents four standard techniques (see the keywords in Table 5) to develop upgradeable Ethereum smart contracts. This work shows that although the deployed contract code is immutable on Ethereum, a workaround still exists as long as we consider the security issues in advance. Moreover, fix schemes in section III aiming to address specific security vulnerabilities can also be seen as part of *security development* (we will not restate them here).

### 2) SECURITY TEMPLATE

Clack *et al.* [51] introduced smart contract templates, which connect a legal agreement to the final executable code via operational parameters. Figure 12 illustrates their work, where an agreement template often contains legal prose and parameters. The prose and parameters can be customized in the negotiation stage and then formed into a legal agreement. Correspondingly, the standardized smart contract code can be generated directly in terms of the agreement template and modified in negotiation. After the agreement is signed, the values of parameters are defined and then passed to the code template, which enables the final executable contract code. In their later work, Clark *et al.* [52] identified the essential requirement and design options of templates. Moreover, Clark proposed a series of approaches to verify whether the smart contract code faithfully performs the provisions of the legal contract [53]. Although they did not explicitly mention smart contract security in their works, we argue that standard code templates with security enhancements (see Figure 12), such as security patterns and security libraries, can be extracted from business logic to significantly reduce errors in manual coding. If such security templates are built, they could be tested and certified only once and then used by others while avoiding security issues.

## C. TESTING BEFORE DEPLOYMENT

Because smart contracts are difficult to patch after deployment, it is necessary to perform sufficient testing to ensure their security before their deployment. Table 6 classifies smart contract security solutions in the phase of *testing before deployment* into three security themes: *rigorous formal verification*, *code analysis tools*, and *security audit*.

### 1) RIGOROUS FORMAL VERIFICATION

Rigorous formal verification is a mathematical approach to verifying the correctness and security of a program. Smart contracts are very suitable for rigorous formal verification because the contract program is small and time-bounded.

**TABLE 6.** Smart contract security solutions in the phase of testing *before deployment*.

| Security Themes | Related Work | Platforms | Keywords | Open Source Tool |
|---|---|---|---|---|
| *Rigorous Formal Verification* | Behavior Model and Verification [55] | Ethereum | Formal modeling on source code, Behavior interaction priorities (BIP), Statistical model checking (SMC) | No |
| | Formalization Verification Based on F* [56] | Ethereum | EVM* and Solidity*, Program verification, Interactive proof assistant | No |
| | Formalization Based on Lem [57] | Ethereum | Formalization of EVM semantics, Lem language, Interactive theorem prover, | Yes |
| | Semantics Framework Based on F* [58] | Ethereum | Complete small-step semantics of EVM, F* proof assistant, Ethereum test suite | Yes |
| | Formalization Based on the K Framework [59] | Ethereum | Formalization of EVM semantics, K framework, Ethereum test suite, Reachability Logic prover | Yes |
| | Fabric ChainCode Formalization [60] | Fabric | Analysis on Java source code, Deductive verification, KeY prover, Serialization | No |
| *Code Analysis Tools* | Oyente [34] | Ethereum | Analysis on bytecode, Symbolic execution, Four common bugs, Four components | Yes |
| | Securify [18] | Ethereum | Analysis on bytecode, Semantics facts, The contract dependency graph, Compliance and violation patterns | No |
| | Zeus [17] | Ethereum, Fabric | Analysis on source code, Correctness and fairness, Abstract interpretation, Symbolic model checking, Constrained Horn clauses, LLVM bitcode | No |
| | Chaincode Scanner [61] | Fabric | Analysis on Go source code, Automated analysis of chaincodes, Nine vulnerability patterns | No |
| | Fabric Tool Based on Abstract Syntax Tree [38] | Fabric | Analysis on Go source code, Go tools, Abstract syntax tree (AST) | No |
| | Octopus [62] | Bitcoin, Ethereum, Eos | Analysis on bytecode or WebAssembly, Disassembler, Control flow analysis, Call flow analysis, Static single assignment, Symbolic execution | Yes |
| | TEETHER [63] | Ethereum | Automatic vulnerability discovery, Symbolic execution, Control flow graph Automatic exploit generation | Yes |
| | Mythril [64], SmartCheck [65], Manticore [66] | Ethereum | Δ | Yes |
| *Security Audit* | Zeppelin [67], SmartDec [68] | Ethereum | Δ | No |
| | ConsenSys Diligence [69] | Δ | Δ | No |
| | Erays [70] | Ethereum | Reverse engineering tool, Code complexity, Code reuse | Yes |

Rigorous formal verification is usually applied after the completion of the contract code, although more precisely, it can also be used to verify the correctness of the middle representation of smart contracts in the design or implementation phase (e.g., a high-level specification derived from business logic in *security modeling*).

There are several approaches to rigorously formalize and verify smart contracts. Among them, some are based on contract code [55], [56], whereas some are based on Ethereum virtual machine (EVM) semantics [57]–[59]. The latter can be viewed as tailored implementations of EVM that take into account the proof of security properties.

Abdellatif and Brousmich [55] proposed a novel approach for modeling smart contract behavior with strong semantics using a behavior interaction priorities (BIP) framework based on the source code of smart contracts. In addition, they simulated this behavior model with a series of runtime verification and simulation engines. The results from the simulation were further analyzed to verify security properties with a statistical model checking (SMC) tool, which revealed some malicious operations. However, under most circumstances, formalizing smart contract behavior based on existing source code is nontrivial. Using two prototype tools, Solidity$*$ and EVM$*$ (see Figure 13), Bhargavan *et al.* [56] translated Ethereum contracts into F$*$, a functional language equipped with an

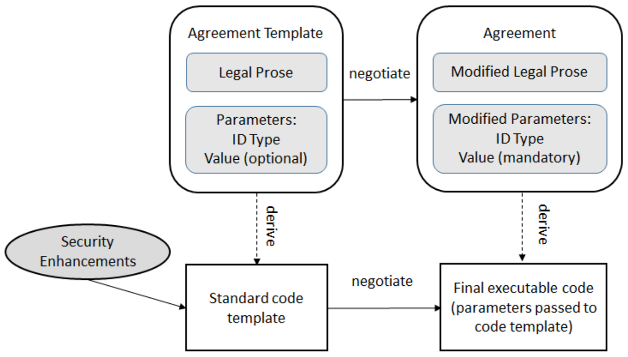

Agreement Template
Legal Prose
Parameters: ID Type Value (optional)
negotiate
Agreement
Modified Legal Prose
Modified Parameters: ID Type Value (mandatory)
derive
derive
Security Enhancements
Standard code template
negotiate
Final executable code (parameters passed to code template)

**FIGURE 12. Relations among the agreement template, agreement, standard code template, and final executable code.**

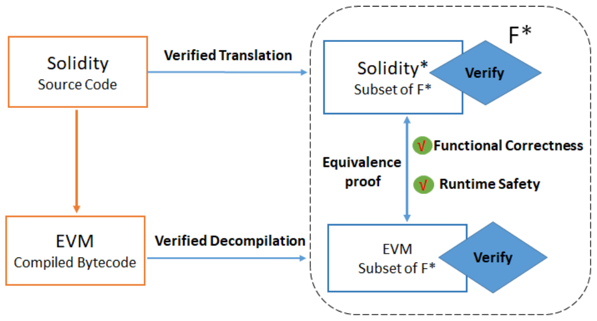


**FIGURE 13. Formalization verification based on contract code [56].**

interactive proof assistant for program verification. However, the translation only supports a fragment of EVM bytecode and leaves out many essential constructs.

To develop secure contracts, Hirai [57] formalized EVM semantics using Lem, from which the interactive theorem prover Isabelle/HOL can be extracted to prove security properties of smart contracts (see Figure 14). This work is the first formal EVM definition for smart contract verification. However, the semantics are only a sound overapproximation of the original semantics in [57] and thus cannot serve as a general-purpose basis for static analysis. To demonstrate the significance of rigorous semantic foundations for the design of security verification, Grishchenko *et al.* [58] presented the first complete small-step semantics of EVM bytecode, which they formalized in the F∗ proof assistant. Based on this formalization, program verifications, as well as proofs for fundamental security properties, were proposed. Moreover, Hildenbrandt *et al.* [59] presented fully executable rigorous formal semantics on EVM using the K framework [71]. They also showed how the existing language-independent tools, such as the reachability logic prover [72] for the K framework, can be applied to security analysis on EVM programs. Both [58] and [59] obtained executable code that was successfully validated against the Ethereum test suite to show the correctness of their semantic definitions. They successfully identified various mistakes and imprecisions in the existing semantics and enabled the verification of security properties for Ethereum smart contracts, thus offering formal verification and a precision guarantee for these smart contracts.

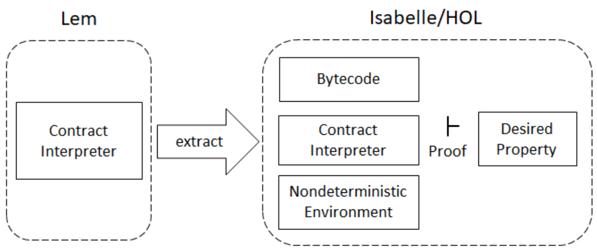


**FIGURE 14. Proof of security properties based on EVM semantics [57].**

Nevertheless, full automation of the verification of security properties still needs to be explored further.

Beckert *et al.* [60] proposed a formal specification and verification of smart contracts written in Java in Fabric using KeY [73], a semi-interactive deductive theorem prover for statically verifying sequential Java programs. Notably, they extended KeY to support the verification of smart contracts, and this extension operates on the Fabric ledger and handles serialization. As far as we know, the work of Beckert *et al.* is the only work on formal verification of Fabric smart contracts thus far.

### 2) CODE ANALYSIS TOOLS

Thus far, only a few static code analysis tools, whether they rely on a rigorous formalization or not, have been implemented to improve code quality and security. The tools conduct static code analysis on the source code or bytecode of smart contracts without executing the programs. The analysis may include some or all of the following steps. (I) Build an intermediate representation (IR), e.g., an abstract syntax tree (AST), for subsequent in-depth analysis. (II) Enrich the IR with some additional information, likely coming from static control or date flow analysis and formal verification techniques, such as *symbolic execution* [74], *abstract interpretation* [75], and *symbolic model checking* [76]. (III) Perform vulnerability detection according to a security pattern (anti-pattern or secure property) that defines the vulnerability criteria in IR terms.

Oyente [34], one of the most popular automatic security analysis tools for EVM smart contracts, leverages *symbolic execution* with the constraint solver Z3 [77] to find four common bugs (namely, transaction ordering dependence, timestamp dependence, mishandled exceptions, and reentrancy). Reference [11] uses Oyente to help eradicate these common bugs in smart contracts. Figure 15 shows the architecture of Oyente, which takes bytecode and the current Ethereum global state as input. Four components, namely CFG (Control Flow Graph) Builder, Explorer, Core Analysis, and Validator, form the core of Oyente. CFG Builder constructs a control flow graph of smart contracts, which feeds the Explorer and Graph visualizer. Explorer symbolically executes the contract, and the result is then fed to Core Analysis, where Oyente targets the common bugs. Finally, Validator filters out false positives before final reporting. Among 19,366 existing smart contracts, Oyente flags 8,833 as vulnerable, including the DAO bug. However, Oyente only detects particular

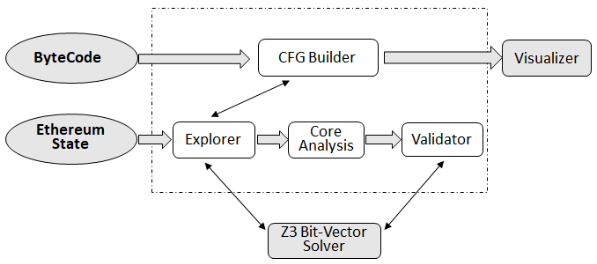


**FIGURE 15. The architecture of Oyente [34].**

vulnerabilities with false positives. Additionally, it only implements a lightweight semantics of the EVM bytecode, and thus misses a few critical commands concerning contract calls and creations.

*Symbolic execution* used by Oyente is a powerful technique for identifying vulnerabilities. However, it does not guarantee the exploration of all execution paths, which may result in false negatives. In contrast to Oyente, Securify [18] is an abstract interpreter that explores all possible paths. Additionally, Securify, though non-open-source, is a tool available for smart contract code analysis using a web page. Securify analyzes the contract dependency graph in stratified Datalog [78] to infer precise semantic information from EVM bytecode. Then, it checks compliance and violation patterns to verify whether a security property holds. Compared with Oyente, Securify's analysis is fully automated. Because patterns are expressed in a designated domain-specific language (DSL) in Securify, anyone can easily extend new patterns to address new vulnerabilities.

Kalra *et al.* proposed Zeus [17], a completely automated tool, to reason about the correctness and fairness of smart contracts using formal approaches including *abstract interpretation* and *symbolic model checking* along with constrained Horn clauses [79]. Zeus takes as input contract source code and security policy (written in XACML-styled templates). It performs static analysis atop the source code to determine the points at which the verification predicates must be asserted. Then, Zeus transforms the smart contract source code embedded with policy assertions into a low-level IR such as LLVM bitcode. Finally, Zeus feeds the bitcode to a verification engine to determine whether the security policy is satisfied. The evaluation shows that Zeus is sound with zero false negatives and a low false-positive rate, and it offers a significant improvement in analysis time over previous works, such as Oyente. Because most high-level languages already have mature LLVM bitcode translators, Zeus is scalable and is now available for the Fabric and Ethereum platforms. Unfortunately, Zeus is not open-source and is only used to analyze contracts whose source code is available.

Chaincode Scanner is a static security analyzer designed for Fabric smart contracts [61]. It often takes smart contracts written in the Go language as input. Some automated security analyses, such as control flow graph analysis and dependency graph analysis, are conducted to check for nine vulnerability patterns [37]. However, due to the lack of a detailed description of Chaincode Scanner, we cannot fully disclose the theory of the tool. Yamashita *et al.* [38] surveyed various artifacts and identified fourteen potential risks in Fabric. They argued that smart contracts written in general-purpose programming languages, such as Go and Java, lack restrictions and are more likely to cause nondeterministic risks. Some ready-made tools, such as Go tools [80], can be utilized to conduct static security analysis for some common risks, but this does not take into consideration some risks in the blockchain context. Therefore, Yamashita *et al.* implemented a prototype tool to cover these risks, which analyzed the AST of contracts. Moreover, they compared Chaincode Scanner and Go tools with their tool. Comparison results showed that their tool has more effective coverage and better performance.

Octopus [62] is a security analysis tool for smart contracts with EVM and WASM (WebAssembly [81]) support. To improve analysis efficiency, Octopus can translate bytecode into an assembly representation (another IR format) using a disassembler and then simplify the assembly into static single assignment (SSA) representation for further optimization. Moreover, Octopus utilizes control flow graphs and call flow graphs to analyze security vulnerabilities and exploits *symbolic execution* to find new paths into a program. Compared to other tools, Octopus can perform security analysis of smart contracts on multiple blockchain platforms, including BTC, ETH, NEO, and EOS.

Meanwhile, some other security analysis tools, such as TEETHER [63], Mythril [64], SmartCheck [65], and Manticore [66], also perform security analysis using formal verification approaches. References [19], [21] and [82] present empirical vulnerability analysis to compare the performances of these tools. Among these tools, TEETHER provides not only a methodology to find vulnerabilities but also end-to-end exploit generation. Moreover, an overview of foundations and tools for the static analysis of Ethereum smart contracts can be found in [83].

Another thing worth noting is that, unlike the approaches in security themes mentioned before, most code analysis tools, including Oyente, Securify, Zeus, Chaincode Scanner, Octopus, and TEETHER, analyze the control flow graph or contract dependency graph. Accordingly, with these tools, it is possible to address some security issues in the situation where multiple smart contracts interact. That is, although a smart contract is not vulnerable, it still may be dangerous because it calls a vulnerable one.

### 3) SECURITY AUDIT

Security issues and incorrectness in smart contracts can cause devastating financial consequences. Consequently, security audits are necessary to ensure the security and correctness of smart contracts. If developers are not confident enough in their professional ability to perform security audits, they can entrust the audit task to some professional constitutions [67]–[69], who will write a qualified auditing report for security issues they identify.

An ideal audit will be a combination of automatic and manual code analysis. Generally, the automatic code analysis uses

**TABLE 7.** Smart contract security solutions in the phase of *monitoring and analysis*.

| Security Themes | Related Work | Platforms | Keywords | Open Source Tool |
|---|---|---|---|---|
| *Bug Bounty* | EtherScan bug bounty [84] | Ethereum | Δ | No |
| | Rainden bug bounty [85] | Ethereum | Δ | No |
| | Hydra [86] | Ethereum | Exploit gap, N-of-N-version programming (NNVP), A fair exchange, Submarine commitments, Bug withholding | Yes |
| *Security Monitoring* | MAIAN [87] | Ethereum | Greedy contracts, Prodigal contracts, Suicidal contracts, Trace vulnerabilities, Symbolic analysis, Concrete validator | Yes |
| | ECF Online Detection [88] | Ethereum | Effectively callback free (ECF), Reentrancy, Execution environment | Yes |
| | DappGuard [89] | Ethereum | Active monitoring, Attack fingerprints, Gas usage, Strange message value, Suspicious fallback invocations, Transaction receipts | No |
| | Monitoring UI [90] | Fabric | React.js, Watson IoT platform, Monitor assets | Yes |
| *Post Hoc Analysis* | Graph Analysis [41] | Ethereum | Graph analysis, Money flow graph (MFG), Contract creation graph (CCG), Contract invocation graph (CIG), Attack forensics, Anomaly detection | Yes |
| | Ponzi Detection [28] | Ethereum | Ponzi scheme, Machine learning, Classifier model | Yes |

static code analysis tools, such as Oyente and Securify, to find vulnerabilities. These tools can save a significant amount of time for security auditors, but they may miss some critical security vulnerabilities. As a supplement, auditors can review every line of code and test them for different vulnerabilities by manual analysis.

Additionally, regulatory bodies also need to carry out public audits of smart contracts. Many smart contracts in public blockchains, such as Ethereum, do not have readily available source code, which makes audits challenging. To address this problem, Zhou *et al.* proposed a reverse- engineering tool entitled Erays [70] to reconstruct high-level pseudocode based on EVM bytecode. With Erays, security auditors can not only explore code complexity and code reuse of Ethereum smart contracts but also uncover contract intention and behavior. Moreover, any other approaches adopted by contract developers for security audit can still be used here.

## D. MONITORING AND ANALYSIS

Even if a contract has been deployed and run, users still need some measures to discover vulnerabilities that were missed in the previous phase for improvement in new releases. These measures include *bug bounty*, *security monitoring*, and *post hoc analysis*. We have summarized all of these measures in Table 7.

### 1) BUG BOUNTY

Although measures in the phase of *testing before deployment* can help us find most bugs, some bugs are still hard to identify. Bug bounty programs are often used to discover deeply hidden bugs. Unlike formal verification, a type of static analysis technique, bug bounty is a type of dynamic

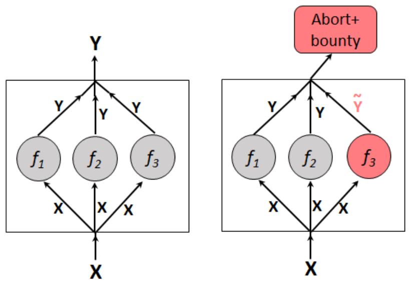


**FIGURE 16.** The Hydra framework with heads $f_1$, $f_2$ and $f_3$ [86].

analysis technique. In a bug bounty program, hackers expecting financial rewards generally find bugs when the system is running. Many Ethereum ecosystems, such as EtherScan [84] and Raiden [85], provide bug bounty programs to further improve system security.

However, bug bounties lack rigorous principles for pricing bounties appropriately. Because the financial value of critical bugs in gray markets may significantly exceed the bounty value [86], some hackers would rather sell or exploit bugs than claim them. Moreover, bounty payers often do not want to pay before confirming an exploit, whereas hackers worry that revealing exploits risks nonpayment. All of these things compromise the generalization of bug bounty programs. To address these challenges, Breidenbach *et al.* proposed Hydra [86], a bug bounty framework, and then applied it to smart contracts.

As the first general, principled bug bounty system on the blockchain, Hydra utilizes an exploit gap technology named N-of-N-version programming (NNVP). The rationale of Hydra is shown in Figure 16, where multiple versions of smart contract programs (Hydra heads) with the same functionality are independently developed. A hacker can obtain the bounty only when a head outputs a different result from the others, which shows that a bug exists in the head. In Hydra, a bug is exploitable only if it affects all heads similarly. That is, the same bug exists in all heads and causes the same exceptional output, which is almost impossible because those heads are independently developed. This property makes critical bugs detectable at runtime, but hard to exploit. In particular, with the Hydra smart contract, a fair exchange between bounty payers and hackers is also enabled. Bounty payers do not need to pay before confirming an exploit, whereas hackers no longer have to worry that revealing exploits risks nonpayment. Moreover, using an adjustable bounty value and submarine commitments, Hydra encourages economically rational hackers to disclose bugs while effectively preventing bug withholding. Although the authors designed the Hydra framework for the Ethereum platform, its fundamental methodology is easy to extend to any other blockchain, such as Fabric or EOS.

### 2) SECURITY MONITORING

Compared with static code analysis, which can reveal security vulnerabilities based on contract code before deployment, monitoring and analyzing transaction data on the blockchain can uncover many vulnerability exploits in real-time.

By analyzing multiple invocations of an Ethereum contract during its run-time, Nikolić *et al.* [87] systematically characterized a class of vulnerabilities that they call trace vulnerabilities. They focused their attention on three categories of contracts with trace vulnerabilities: greedy contracts (contracts that remain alive and lock Ether indefinitely), prodigal contracts (contracts that leak funds carelessly to arbitrary users), and suicidal contracts (contracts that can be killed by any arbitrary user). They implemented MAIAN [91] to specify and reason about trace properties. By employing symbolic analysis and a concrete validator, MAIAN finds exploits for the infamous Parity bug, which previous analyses failed to capture.

Grossman *et al.* [88] defined the notion of effectively callback free (ECF) objects. They argued that identifying vulnerabilities by monitoring ECF objects in the execution traces of Ethereum is feasible. Therefore, they proposed an efficient online algorithm for discovering reentrancy. Other known vulnerabilities can also be discovered similarly. To avoid unexpected consequences, they suggested that smart contracts should first be executed on the testnet before they are finally deployed on the mainnet [92].

Similar to ECF online detection, DappGuard [89] is an active monitoring and defense tool for Ethereum smart contracts that searches for attack fingerprints from transaction logs or receipts to identify attacks. For example, vulnerability exploits for smart contracts can be discovered through high gas usage, strange message values, and suspicious fallback invocations in transaction logs or receipts. Combined with code analysis tools such as Oyente, DappGuard can be extended to monitor the anomalous behavior of smart contracts in real-time.

Monitoring UI [90] is a blockchain monitoring platform that uses React.js, the Watson IoT platform [93] and the Fabric Node SDK to interact with a Fabric blockchain service. Operators can use this UI platform to monitor assets, perform transactions, and query the state of the blockchain. Unlike DappGuard [89], Monitoring UI does not provide secure monitoring of smart contracts. However, because all transaction data related to the contract are available on this platform, it can be extended to detect attacks in real-time by analyzing the transfer of assets.

### 3) POST HOC ANALYSIS

With the increasing volume of blockchain transaction data, some data analysis or machine learning approaches can be applied to discover attacks or abnormal activities on the blockchain.

Chen *et al.* [41] conducted the first systematic study of Ethereum transactions by graph analysis. They designed an approach to collect all transactions, including internal transactions resulting from the execution of smart contracts. From the transaction data, they built three graphs: a money flow graph (MFG), contract creation graph (CCG), and contract invocation graph (CIG). With these graphs, they characterized major activities on Ethereum and discovered many new insights, including degree distribution and node importance. According to contract invocation analysis, some security issues concerning multiple contracts interacting with each other may also be solved. For instance, two security issues, i.e., attack forensics and anomaly detection, were addressed. Attack forensics can be used to find all accounts controlled by an attacker. Anomaly detection can discover abnormal contract creation, which consumes many disks or network resources by creating many contracts that are rarely used. Figure 17 illustrates their approach.

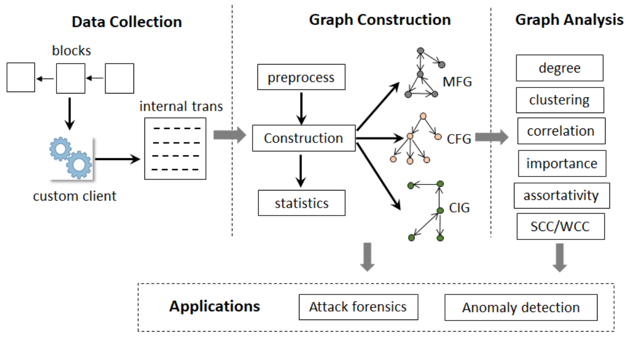


**FIGURE 17. An overview of graph analysis [41].**

The Ponzi scheme, a classic fraud, can acquire a large amount of money and have a very negative impact on Ethereum. Chen *et al.* [28] proposed an effective classifier to detect Ponzi schemes on Ethereum. Their approach utilizes data mining and machine learning to detect Ponzi contracts even if the source code for smart contracts is not available. By verifying smart contracts manually, account features and code features of Ponzi contracts were first

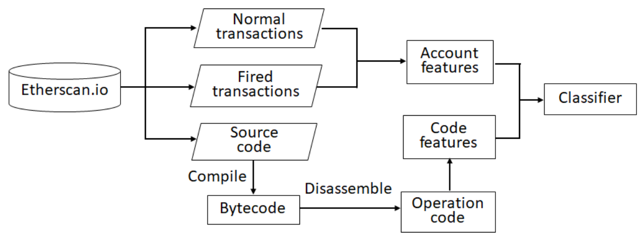


**FIGURE 18.** The framework of smart Ponzi scheme detection [28].

extracted from the transactions and operation codes, respectively. Then, the authors proposed a classifier model based on XGBoost [94], which is proven to be effective in detecting Ponzi schemes. Figure 18 illustrates the framework of their approach.

## V. CHALLENGES AND OPPORTUNITIES

### A. SECURITY DESIGN

With the application of smart contracts in different industries [95]–[97], more security patterns and security anti-patterns should be extracted from the emerging security vulnerabilities.

*Security Modeling* is a promising research direction. There is still considerable room for this direction. In FSMs-based approaches, the number of states and transitions grows exponentially with the number of contracts, which makes this modeling approach inappropriate for complex business logic. Moreover, a logic-based approach is still in the proof-of-concept stage even if this approach has distinct advantages in negotiation, notarizing, and enforcement of a contract. The algorithm for a logic approach is not efficient in blockchain scenarios. Moreover, how to effectively extract FSMs [46] and logic rules from legal agreements [47] to mitigate vulnerabilities is worth exploring. How to make the logic and procedural approaches compatible is also challenging.

With the popularity of blockchain, more domain-specific modeling approaches and logic languages with precise semantics should be proposed to avoid vulnerabilities. These models and languages should be easy to convert into code or real legal contracts adopted by the court directly.

### B. SECURITY IMPLEMENTATION

A promising research direction in security implementation of smart contracts is the integration of more security libraries, such as OpenZeppelin [48], into the contract development environment in the form of security plugins. These libraries will provide security enhancement for the development of smart contracts.

Another promising research direction in security implementation of smart contracts is the development of a formal high-level domain-specific language with explicit semantics and security enhancements. With this language, developers could write contract templates, from which legal agreements and executable smart contract code could be automatically extracted, thereby avoiding error-prone manual coding and eliminating major security issues. Moreover, to keep templates simple and semantics precise, developers could also integrate logic rules in a logic-based approach [47] into template technology.

### C. TESTING BEFORE DEPLOYMENT

Rigorous formal verification is an effective approach for detecting security vulnerabilities. For Ethereum, rigorous formal verification formalizes the EVM interpreter or smart contract code using a general-purpose intermediate-level language, such as Lem. This language serves as an intermediate between high-level languages and executable bytecode. That is, contract code in high-level languages will first be compiled into contract code in intermediate-level languages instead of EVM bytecode. Because these intermediate-level languages are well integrated with the program prover, it is amenable to the program verification. A promising research direction is to design new intermediate-level languages specifically for smart contracts, instead of these general-purpose intermediate-level languages, to facilitate the formal verification of smart contracts. Scilla [98] is such an attempt. With the popularity of the Ethereum platform, more and more such intermediate-level languages that are easy to understand but that have strong expressive power and precise operational semantics will appear. Different from Ethereum, Fabric uses a general-purpose programming language to write smart contracts [23]. Accordingly, the formalization of smart contracts for Fabric may be more complicated than that for Ethereum. Fortunately, some provers (e.g., KeY [73]) associated with general programming languages are available, and how to adapt these provers to blockchain scenarios is also worth exploring.

In addition to rigorous formal verification, code analysis tools are also exploited to find security vulnerabilities in smart contracts. However, these tools are still in the infancy stage. Their performance is not ideal. Many of them do not cover the full range of security vulnerabilities outlined in this paper. Moreover, there is still considerable room to improve their accuracy, false-positive rate, and false-negative rate in the process of vulnerability detection. For this aspect, deep learning may be useful. Moreover, because new security vulnerabilities continue to emerge, an excellent code analysis tool should be extensible. The tool should be able to identify new security vulnerabilities by defining new security properties. However, security properties are now often specified ad hoc and are mostly verified manually. A method for the unified and flexible specification of security properties to facilitate automatic verification is urgently needed.

In brief, because smart contracts are hard to patch once they are deployed on the blockchain, testing before deployment is significant. A convenient test platform for smart contracts, especially new test techniques such as mocking objects capable of effectively emulating the blockchain, are also worth exploring further. Additionally, the construction of standard test datasets for a specific blockchain platform to mitigate the difficulty of testing smart contracts is also a promising research direction.

**TABLE 8. Open-source websites of smart contract security solutions.**

| Related Work | Open Source Tool |
|---|---|
| FSolidM [46] | https://cps-vo.org/group/SmartContracts<br>https://github.com/anmavrid/smart-contracts |
| OpenZeppelin [48] | https://docs.openzeppelin.org<br>https://github.com/OpenZeppelin |
| Formalization based on Lem [57] | https://github.com/pirapira/eth-isabelle/tree/wtsc01 |
| Semantics Framework Based on F* [58] | https://secpriv.tuwien.ac.at/tools/ethsemantics |
| Formalization based on the K framework [59] | https://github.com/kframework/evm-semantics |
| Oyente [34] | https://github.com/melonproject/oyente |
| Octopus [62] | https://github.com/quoscient/octopus |
| TEETHER [63] | https://github.com/nescio007/teether |
| Mythril [64] | https://github.com/ConsenSys/mythril |
| SmartCheck [65] | https://github.com/smartdec/smartcheck |
| Manticore [66] | https://github.com/trailofbits/manticore |
| Erays [70] | https://github.com/teamnsrg/erays |
| Hydra [86] | https://thehydra.io/ |
| MAIAN [91] | https://github.com/MAIAN-tool/MAIAN |
| ECF Online Detection [88] | https://github.com/shellygr/ECFChecker |
| Monitoring UI [90] | https://github.com/IBM/monitoring_ui |
| Graph Analysis [41] | https://github.com/brokendragon/Ethereum_Graph_Analysis |
| Ponzi Detection [28] | http://ibase.site/scamedb/ |

### D. MONITORING AND ANALYSIS

Real-time monitoring and analysis are very significant for blockchain security. Blockchain platforms, such as Ethereum and Fabric, are expected to provide an internal framework for efficient monitoring of the execution of smart contracts in the future. Based on such a framework, organizations can expand the scope of security monitoring according to their business.

As more and more transactions of different businesses are dealt with on the blockchain, a variety of attacks and scams will emerge due to the enormous value transfer associated with these transactions, which provides a promising research direction for data mining on the blockchain. For instance, we may discover more new insights regarding anomalous behavior in smart contracts by in-depth graph analysis.

### E. OTHER DIRECTIONS

If the security of smart contracts is considered at the system level (e.g., EVM implementation), we will spend less on security in the contract development lifecycle. However, consideration of security at the system level is a more general problem and thus is also a significant challenge.

Corresponding to traditional software, some researchers advocate for a discipline of blockchain software engineering to address the issues posed by smart contract programming [99], [100]. We also hold this viewpoint. With the progress of blockchain security, blockchain engineers and researchers may propose more new best practices and security tools concerning blockchain software engineering.

## VI. CONCLUSION

As examined in this paper, the environment in which smart contracts are executed is decentralized and hard to patch for bugs. Consequently, smart contracts have many security issues in terms of security vulnerabilities and anomalous activities. Our focus on smart contract security from the perspective of the software lifecycle enabled us to reveal the practical and theoretical aspects of smart contract security to a greater degree than any previous study. We achieved this vital research outcome as follows. First, we illustrated key features of blockchains that cause security issues in smart contracts, including 1) decentralized, tamper-proof and public ledgers; 2) open-source smart contract code; 3) immature platforms and languages, and 4) pseudonymous transactions. We revealed the most common security vulnerabilities of smart contracts in Ethereum and Fabric. We also discussed the differences in vulnerabilities on different blockchain platforms. Then, we demystified the puzzles of security solutions for smart contracts in terms of numerous security themes spanning four contract development phases: 1) *security design*; 2) *security implementation*; 3) *testing before deployment*; and 4) *monitoring and analysis*. Finally, we summarized the challenges of security research for smart contracts as a software engineering problem and suggested several research directions to improve smart contract security. We hope that this paper builds on the current research outcomes of smart contract security and signifies a milestone in information security in the age of blockchain. We expect that many more research efforts will follow by expanding on the works classified in this paper and by applying the techniques outlined here to various business contexts.

## APPENDIX

For the convenience of reference, Table 8 shows the websites of all open-source tools in Section IV. These websites were available as of September 6, 2019.

## REFERENCES

[1] K. Salah, M. Rehman, N. Nizamuddin, and A. Al-Fuqaha, "Blockchain for AI: Review and open research challenges," *IEEE Access*, vol. 7, pp. 10127–10149, 2019.

[2] G. Zhang, T. Li, Y. Li, P. Hui, and D. Jin, "Blockchain-based data sharing system for ai-powered network operations," *J. Commun. Inf. Netw.*, vol. 3, no. 3, pp. 1–8, 2018.
[3] *AI & Blockchain: An Introduction*. Accessed: Sep. 8, 2019. [Online]. Available: https://mattturck.com/ai-blockchain/
[4] *Decentralized ai Blockchain Whitepaper*, Nebula AI Team, Montreal, QC, Canada, 2018.
[5] E. Karafiloski and A. Mishev, "Blockchain solutions for big data challenges: A literature review," in *Proc. IEEE EUROCON 17th Int. Conf. Smart Technol.*, Ohrid, Macedonia, Jul. 2017, pp. 763–768.
[6] P. Banerjee and S. Ruj, "Blockchain enabled data marketplace—Design and challenges," 2018, *arXiv:1811.11462*. [Online]. Available: https://arxiv.org/abs/1811.11462
[7] M. A. Khan and K. Salah, "IoT security: Review, blockchain solutions, and open challenges," *Future Gener. Comput. Syst.*, vol. 82, pp. 395–411, May 2018.
[8] A. Suliman, Z. Husain, M. Abououf, M. Alblooshi, and K. Salah, "Monetization of IoT data using smart contracts," *IET Netw.*, vol. 8, no. 1, pp. 32–37, Jan. 2019.
[9] R. Almadhoun, M. Kadadha, M. Alhemeiri, M. Alshehhi, and K. Salah, "A user authentication scheme of IoT devices using blockchain-enabled fog nodes," in *Proc. IEEE/ACS 15th Int. Conf. Comput. Syst. Appl. (AICCSA)*, Aqaba, Jordan, Oct./Nov. 2018, pp. 1–8.
[10] H. Hasan and K. Salah, "Combating deepfake videos using blockchain and smart contracts," *IEEE Access*, vol. 7, no. 1, pp. 41596–41606, Dec. 2019.
[11] H. Hasan and K. Salah, "Proof of delivery of digital assets using blockchain and smart contracts," *IEEE Access*, vol. 6, pp. 65439–65448, 2018.
[12] I. C. Lin and T. C. Liao, "A survey of blockchain security issues and challenges," *Int. J. Netw. Secur.*, vol. 19, no. 5, pp. 653–659, 2017.
[13] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen, "A survey on the security of blockchain systems," *Future Gener. Comput. Syst.*, to be published.
[14] N. Andola, M. Gogoi, S. Venkatesan, and S. Verma, "Vulnerabilities on hyperledger fabric," *Pervas. Mobile Comput.*, vol. 59, Oct. 2019, Art. no. 101050.
[15] *BCSEC Incorporation*. Accessed: Mar. 15, 2019. [Online]. Available: https://bcsec.org/analyse
[16] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts (SoK)," in *Proc. Int. Conf. Princ. Secur. Trust*, Uppsala, Sweden, Apr. 2017, pp. 164–186.
[17] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing safety of smart contracts," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, 2018, pp. 1–15.
[18] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Toronto, ON, Canada, 2018, pp. 67–82.
[19] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, "Security analysis methods on Ethereum smart contract vulnerabilities: A survey," 2019, *arXiv:1908.08605*. [Online]. Available: https://arxiv.org/abs/1908.08605
[20] *The DAO (Organization)*. Accessed: Sep. 6, 2019. [Online]. Available: https://en.wikipedia.org/wiki/The_DAO_(organization)
[21] D. Ardit, "Ethereum smart contracts: Security vulnerabilities and security tools," M.S. thesis, Dept. Comput. Sci., Norwegian Univ. Sci. Technol., Trondheim, Norway, 2017.
[22] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, Apr. 2014.
[23] E. Androulaki *et al.*, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proc. 13th ACM EuroSys Conf.*, Porto, Portugal, 2018, Art. no. 30.
[24] R. G. Brown *et al.*, "Corda: An introduction," R3 CEV, Aug. 2016. [Online]. Available: https://docs.corda.net/_static/corda-introductory-whitepaper.pdf
[25] *EOS White Paper*. Accessed: Jun. 8, 2019. [Online]. Available: https://github.com/EOSIO/Documentation/blob/master/Technical WhitePaper.md
[26] *Common weakness Enumeration*. Accessed: Jun. 8, 2019. [Online]. Available: https://cwe.mitre.org/data/index.html
[27] A. Juels, A. Kosba, and E. Shi, "The ring of Gyges: Investigating the future of criminal smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Vienna, Austria, 2016, pp. 283–295.
[28] W. Chen, Z. Zheng, J. Cui, E. Ngai, P. Zheng, and Y. Zhou, "Detecting Ponzi schemes on Ethereum: Towards healthier blockchain technology," in *Proc. World Wide Web Conf. World Wide Web*, Lyon, France, 2018, pp. 1409–1418.
[29] *SWC Registry*. Accessed: Jun. 8, 2019. [Online]. Available: https://smartcontractsecurity.github.io/SWC-registry/
[30] *Known Attacks*. Accessed: Jun. 8, 2019. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/known_attacks/
[31] *I Accidentally Killed It*. Accessed: Jun. 8, 2019. [Online]. Available: https://elementus.io/blog/which-icos-are-affected-by-the-parity-wallet-bug/
[32] *BeautyChain Token Bug*. Accessed: Jun. 8, 2019. [Online]. Available: https://blog.matryx.ai/batch-overflow-bug-on-ethereum-erc20-token-contracts-and-safemath-f9ebcc137434
[33] *Hundreds of Millions of Dollars Locked at* $0 \times 0$ *Address and Smart Contracts' Addresses*. Accessed: Jun. 8, 2019. [Online]. Available: https://medium.com/@maltabba/hundreds-of-millions-of-dollars-locked-at-ethereum-0x0-address-and-smart-contracts-addresses-how-4144dbe3458a
[34] L. Luu, Duc-Hiep Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Vienna, Austria, 2016, pp. 254–269.
[35] *SafeMath*. Accessed: Jun. 8, 2019. [Online]. Available: https://ethereumdev.io/safemath-protect-overflows/
[36] *Replay Attack*. Accessed: Jun. 8, 2019. [Online]. Available: https://medium.com/cypher-core/replay-attack-vulnerability-in-ethereum-smart-contracts-introduced-by-transferproxy-124bf3694e25
[37] T. Kaiser. Chaincode Scanner: Automated Security Analysis of Chaincode. ChainSecurity. Accessed: Sep. 6, 2019. [Online]. Available: https://static.sched.com/hosted_files/hgf18/18/GlobalForum-tobias-kaiser.pdf
[38] K. Yamashita, Y. Nomura, E. Zhou, B. Pi, and S. Jun, "Potential risks of Hyperledger Fabric smart contracts," in *Proc. IEEE Int. Workshop Blockchain Oriented Softw. Eng. (IWBOSE)*, Hangzhou, China, Feb. 2019, pp. 1–10.
[39] *Chaincode Scanner*. Accessed: Sep. 6, 2019. [Online]. Available: https://chaincode.chainsecurity.com/
[40] V. Saini. *ContractPedia: An Encyclopedia of 40+ Smart Contract Platforms*. Accessed: Sep. 6, 2019. [Online]. Available: https://hackernoon.com/contractpedia-an-encyclopedia-of-40-smart-contract-platforms-4867f66da1e5
[41] T. Chen, Y. Zhu, Z. Li, J. Chen, X. Li, X. Luo, X. Lin, and X. Zhange, "Understanding Ethereum via graph analysis," in *Proc. IEEE Int. Conf. Comput. Commun.*, Honolulu, HI, USA, Apr. 2018, pp. 1484–1492.
[42] *General Philosophy*. Accessed: Jun. 8, 2019. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/general_philosophy/
[43] *Security Guidelines*. Accessed: Sep. 6, 2019. [Online]. Available: https://github.com/slowmist/eos-smart-contract-security-best-practices/blob/master/README_EN.md#security-guidelines
[44] M. Wohrer and U. Zdun, "Smart contracts: Security patterns in the Ethereum ecosystem and solidity," in *Proc. IEEE Int. Workshop Blockchain Oriented Softw. Eng. (IWBOSE)*, Mar. 2018, pp. 2–8.
[45] M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: Platforms, applications, and design patterns," in *Financial Cryptography and Data Security*. Cham, Switzerland: Springer, 2017, pp. 494–509.
[46] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "VeriSolid: Correct-by-design smart contracts for Ethereum," 2019, *arXiv:1901.01292*. [Online]. Available: https://arxiv.org/abs/1901.01292
[47] F. Idelberger, G. Governatori, R. Riveret, and G. Sartor, "Evaluation of logic-based smart contracts for blockchain systems," in *Proc. Int. Symp. Rules Rule Markup Lang. Semantic Web*, Stony Brook, NY, USA, 2016, pp. 167–183.
[48] *OpenZeppelin*. Accessed: Sep. 6, 2019. [Online]. Available: https://openzeppelin.org
[49] *Security EIPs*. Accessed: Sep. 6, 2019. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/security_eips
[50] *Upgradable Contract*. Accessed: Sep. 6, 2019. [Online]. Available: https://hackernoon.com/how-to-make-smart-contracts-upgradable-2612e771d5a2
[51] C. D. Clack, V. A. Bakshi, and L. Braine, "Smart contract templates: Foundations, design landscape and research directions," 2016, *arXiv:1608.00771*. [Online]. Available: https://arxiv.org/abs/1608.00771

[52] C. D. Clack, V. A. Bakshi, and L. Braine, "Smart contract templates: Essential requirements and design options," 2016, *arXiv:1612.04496*. [Online]. Available: https://arxiv.org/abs/1612.04496

[53] C. D. Clack, "Smart contract templates: Legal semantics and code validation," *J. Digit. Banking*, vol. 2, no. 4, pp. 338–352, 2018.

[54] *Truffle*. Accessed: Sep. 6, 2019. [Online]. Available: https://truffleframework.com/

[55] T. Abdellatif and K.-L. Brousmiche, "Formal verification of smart contracts based on users and blockchain behaviors models," in *Proc. IEEE 9th IFIP Int. Conf. New Technol., Mobility Secur. (NTMS)*, Paris, France, Feb. 2018, pp. 1–5.

[56] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, "Formal verification of smart contracts: Short paper," in *Proc. ACM Workshop Program. Lang. Anal. Secur.*, Vienna, Austria, 2016, pp. 91–96.

[57] Y. Hirai, "Defining the Ethereum virtual machine for interactive theorem provers," in *Proc. Int. Conf. Financial Cryptogr. Data Secur. (FC)*, Sliema, Malta, 2017, pp. 520–535.

[58] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of Ethereum smart contracts," in *Proc. Int. Conf. Princ. Secur. Trust*, Thessaloniki, Greece, 2018, pp. 243–269.

[59] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu, "KEVM: A complete formal semantics of the Ethereum virtual machine," in *Proc. IEEE 31st Comput. Secur. Found. Symp. (CSF)*, vol. 1, Aug. 2018, pp. 204–217.

[60] B. Beckert, M. Herda, M. Kirsten, and J. Schiffl, "Formal specification and verification of hyperledger fabric chaincode," in *Proc. Int. Conf. Formal Eng. Methods*, Gold Coast, QLD, Australia, 2018, pp. 44–48.

[61] *ChainSecurity*. Accessed: Sep. 6, 2019. [Online]. Available: https://medium.com/chainsecurity/release-of-hyperchecker-2dff2ebe30cc

[62] *Octopus*. Accessed: Jun. 8, 2019. [Online]. Available: https://github.com/quoscient/octopus

[63] J. Krupp and C. Rossow, "teEther: Gnawing at Ethereum to automatically exploit smart contract," in *Proc. 27th USENIX Secur. Symp.*, Baltimore, MD, USA, Aug. 2018, pp. 1317–1333.

[64] *Mythril*. Accessed: Jun. 8, 2019. [Online]. Available: https://github.com/ConsenSys/mythril

[65] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of Ethereum smart contracts," in *Proc. IEEE/ACM 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, Gothenburg, Sweden, May/Jun. 2018, pp. 9–16.

[66] *Manticore*. Accessed: Sep. 6, 2019. [Online]. Available: https://github.com/trailofbits/manticore

[67] *Security Audits for High-Impact Projects*. Accessed: Jun. 8, 2019. [Online]. Available: https://zeppelin.solutions

[68] *Smart Contract Security Audit*. Accessed: Sep. 6, 2019.[Online]. Available: http://smartcontracts.smartdec.net/

[69] *ConsenSys Diligence*. Accessed: Sep. 6, 2019. [Online]. Available: https://consensys.net/diligence/

[70] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey, "Erays: Reverse engineering Ethereum's opaque smart contracts," in *Proc. 27th USENIX Secur. Symp.*, Baltimore, MD, USA, Aug. 2018, pp. 1371–1385.

[71] G. Roşu and T. F. Şerbănută, "An overview of the K semantic framework," *J. Logic Algebr. Program.*, vol. 79, no. 6, pp. 397–434, 2010.

[72] A. Stefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu, "Semantics-based program verifiers for all languages," *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 74–91, 2016.

[73] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, *Deductive Software Verification—The KeY Book* (Lecture Notes in Computer Science). Basel, Switzerland: Springer, 2016.

[74] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[75] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. 4th ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang.*, Los Angeles, CA, USA, 1977, pp. 238–252.

[76] Y. Vizel, G. Weissenbacher, and S. Malik, "Boolean satisfiability solvers and their applications in model checking," *Proc. IEEE*, vol. 103, no. 11, pp. 2021–2035, Nov. 2015.

[77] Microsoft Corporation. *The $Z_3$ Theorem Prover*. Accessed: Jun. 8, 2019. [Online]. Available: https://github.com/Z3Prover/z3

[78] J. D. Ullman, *Principles of Database and Knowledge-Base Systems*, vol. 1. New York, NY, USA: Computer Science Press, 1989.

[79] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The SeaHorn verification framework," in *Proc. Int. Conf. Comput. Aided Verification*, San Francisco, CA, USA, 2015, pp. 343–361.

[80] *Go Code Analysis*. Accessed: Jun. 8, 2019. [Online]. Available: https://github.com/dominikh/go-tools

[81] *WebAssembly*. Accessed: Jun. 8, 2019. [Online]. Available: https://webassembly.org/

[82] R. M. Parizi, A. Dehghantanha, K. K. R. Choo, and A. Singh, "Empirical vulnerability analysis of automated smart contracts security testing on blockchains," in *Proc. 28th Annu. Int. Conf. Comput. Sci. Softw. Eng.*, Toronto, ON, Canada, 2018, pp. 103–113.

[83] I. Grishchenko, M. Maffei, and C. Schneidewind, "Foundations and tools for the static analysis of Ethereum smart contracts," in *Proc. Int. Conf. Comput. Aided Verification*, Oxford, U.K., 2018, pp. 51–78.

[84] *EtherScan Bug Bounty*. Accessed: Jun. 8, 2019. [Online]. Available: https://etherscan.io/bugbounty

[85] *Raiden Bug Bounty*. Accessed: Jun. 8, 2019. [Online]. Available: https://raiden.network/bug-bounty.html

[86] L. Breidenbach, P. Daian, F. Tramèr, and A. Juels, "Enter the hydra: Towards principled bug bounties and exploit-resistant smart contracts," in *Proc. 27th USENIX Secur. Symp.*, Baltimore, MD, USA, Aug. 2018, pp. 1335–1352.

[87] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proc. ACM 34th Annu. Comput. Secur. Appl. Conf.*, New York, NY, USA, 2018, pp. 653–663.

[88] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online detection of effectively callback free objects with applications to smart contracts," in *Proc. ACM Program. Lang.*, vol. 2, 2017, Art. no. 48.

[89] T. Cook, A. Latham, and J. H. Lee. (2017). DappGuard: Active Monitoring and Defense for Solidity Smart Contracts. MIT. [Online]. Available: https://courses.csail.mit.edu/6.857/2017/project/23.pdf

[90] *Monitoring UI*. Accessed: Jun. 8, 2019. [Online]. Available: https://github.com/IBM/monitoring_ui

[91] *MAIAN*. Accessed: Jun. 8, 2019. [Online]. Available: https://github.com/MAIAN-tool/MAIAN

[92] *Testnet and Mainnet—What Role Do They Play*. Accessed: Jun. 8, 2019. [Online]. Available: https://www.coinsuggest.com/testnet-mainnet

[93] *Watson IoT Platform*. Accessed: Jun. 8, 2019. [Online]. Available: https://www.ibm.com/cloud/watson-iot-platform

[94] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, San Francisco, CA, USA, 2016, pp. 785–794.

[95] J. L. Zhao, S. Fan, and J. Yan, "Overview of business innovations and research opportunities in blockchain and introduction to the special issue," *Financial Innov.*, vol. 2, no. 1, 2016, Art. no. 28.

[96] J. Leng, P. Jiang, K. Xu, Q. Liu, J. L. Zhao, Y. Bian, and R. Shi, "Makerchain: A blockchain with chemical signature for self-organizing process in social manufacturing," *J. Cleaner Prod.*, vol. 234, pp. 767–778, Oct. 2019.

[97] J. Leng, D. Yan, Q. Liu, K. Xu, J. L. Zhao, R. Shi, L. Wei, D. Zhang, and X. Chen, "ManuChain: Combining permissioned blockchain with a holistic optimization model as bi-level intelligence for smart manufacturing," *IEEE Trans. Syst., Man, Cybern., Syst.*, to be published. doi: 10.1109/TSMC.2019.2930418.

[98] I. Sergey, A. Kumar, and A. Hobor, "Scilla: A smart contract intermediate-level language," 2018, *arXiv:1801.00687*. [Online]. Available: https://arxiv.org/abs/1801.00687

[99] S. Porru, A. Pinna, M. Marchesi, and R. Tonelli, "Blockchain-oriented software engineering: Challenges and new directions," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. Companion*, Buenos Aires, Argentina, May 2017, pp. 169–171.

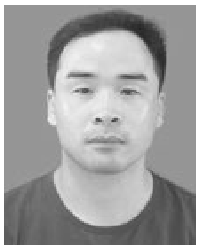
[100] G. Destefanis, "Smart contracts vulnerabilities: A call for blockchain software engineering?" in *Proc. IEEE Int. Workshop Blockchain Oriented Softw. Eng.*, Campobasso, Italy, Mar. 2018, pp. 19–25.

**YONGFENG HUANG** received the Ph.D. degree in computer applications from Southeast University, China, in 2016. He has been with the School of Computer Engineering, Jiangsu University of Technology, China, since 2016. He is currently a Visiting Scholar with the City University of Hong Kong. His work has appeared in *Computer Communications*, *Transaction on Emerging Telecommunication Technologies*, the *Journal on Communications*. His work has appeared in several conferences such as the IEEE Consumer Communications and Networking. His research interests include mobile opportunistic networks, blockchain, big data analytics, and big data privacy protection.

**YIYANG BIAN** received the Ph.D. degree from the City University of Hong Kong and the Ph.D. degree from the University of Science and Technology of China. He is currently an Assistant Professor with the School of Information Management, Nanjing University. His work has appeared in *Transportation Research Part A*, *Environment and Planning A*, and *Industrial Management & Data Systems*. His work has appeared in several conferences of information systems such as the International Conference on Information Systems, the Hawaii International Conference on System Sciences, and the Pacific Asia Conference on Information Systems. His research interests include IT switching, cloud computing, and data analytics.

**RENPU LI** received the Ph.D. degree in management science and engineering from Tianjin University, China, in 2003. He has been a Professor with the School of Computer Engineering, Jiangsu University of Technology, Changzhou, since 2015. His work has appeared in the *European Journal of Operational Research* and *Lecture Notes in Artificial Intelligence*. His work has appeared in several conferences such as the International Conference on Fuzzy Systems and Knowledge Discovery. His research interests include data mining, rough sets, and big data.

**J. LEON ZHAO** is a Chair Professor and the former Head of the Department of Information Systems with the City University of Hong Kong. He was the Interim Head and a Eller Professor in MIS with the University of Arizona. His research has appeared in *MIS Quarterly*, *Information Systems Research*, the *Journal of Management Information Systems*, *Management Sciences*, and *INFORMS Journal on Computing*. His research areas include big data analytics, financial information services, workflow modeling and design, and blockchain.



**PEIZHONG SHI** received the Ph.D. degree from the School of Computer Science and Engineering, Southeast University, China, in 2014. He is currently with the School of Computer Engineering, Jiangsu University of Technology, China. He was invited as a Visiting Scholar with the Center on Global Internet Finance, City University of Hong Kong, from October 2018 to March 2019 (Six months). He is also the in charge of the National Natural Science Foundation of China (NSFC). His current research interests include wireless sensor networks, QoS guarantee based on cross-layer approach, and Blockchain security and its applications based on the consortium Blockchain architecture.