Empirical Software Engineering (2021) 26: 117
https://doi.org/10.1007/s10664-021-10018-0

Maintenance-related concerns for post-deployed Ethereum smart contract development: issues, techniques, and future challenges

Jiachi Chen[1] · **Xin Xia**[1] · **David Lo**[2] · **John Grundy**[1] · **Xiaohu Yang**[3]

Accepted: 6 July 2021 / Published online: 25 August 2021

Abstract
Software development is a very broad activity that captures the entire life cycle of a software, which includes designing, programming, maintenance and so on. In this study, we focus on the maintenance-related concerns of the post-deployment of smart contracts. Smart contracts are self-executed programs that run on a blockchain. They cannot be modified once deployed and hence they bring unique maintenance challenges compared to conventional software. According to the definition of ISO/IEC 14764, there are four kinds of software maintenance, i.e., corrective, adaptive, perfective, and preventive maintenance. This study aims to answer (i) What kinds of issues will smart contract developers encounter for corrective, adaptive, perfective, and preventive maintenance after they are deployed to the Ethereum? (ii) What are the current maintenance-related methods used for smart contracts? To obtain the answers to these research questions, we first conducted a systematic literature review to analyze 131 smart contract related research papers published from 2014 to 2020. Since the Ethereum ecosystem is fast-growing, some results from previous publications might be out-of-date and there may be a gap between academia and industry. To address this, we performed an online survey of smart contract developers on Github to validate our findings and received 165 useful responses. Based on the survey feedback and literature review, we present the first empirical study on smart contract maintenance-related concerns. Our study can help smart contract developers better maintain their smart contract-based projects, and we highlight some key future research directions to improve the Ethereum ecosystem.

Keywords Empirical study · Literature review · Smart contracts · Ethereum · Smart contracts maintenance

1 Introduction

With the great success of Bitcoin (Nakamoto 2008), considerable attention has been paid to the emerging concepts of blockchain technology (Blockchain 2019). However, the usage

Communicated by: Daniel Méndez

✉ Xin Xia
xin.xia@monash.edu

Extended author information available on the last page of the article.

scenario of Bitcoin is limited, as the main application of Bitcoin is storing and transferring monetary values (Efanov and Roschin 2018). The appearance of Ethereum (2019) at the end of 2015 removed many of the limitations of blockchain-based systems. Ethereum leverages a technology named *smart contracts*, which are Turing-complete programs that run on the blockchain (Wood 2014). Blockchain technology gives immutable, self-executed, and decentralized features to these smart contracts. This in turn means that smart contracts *cannot be modified once deployed to the blockchain*, and all of their execution depends on this immutable code. Running these smart contracts across highly distributed servers costs *"gas"*, which in turn costs money. These features ensure the trustworthiness of smart contracts and make the technology attractive to developers and users. By utilizing smart contracts, developers can easily develop Decentralized Applications (*DApps*) (2019), which have been applied to different areas, such as IoT (Chen et al. 2018b), financial (Fabian and Vitalik 2018), gaming (Cryptokitties 2019), and data security domain (Velner et al. 2017).

Like all computer code, smart contracts may have errors or developers might want to extend their features in the future. However, some features of Ethereum – like the gas system and smart contract immutability – make smart contracts much harder to maintain than conventional software (Bosu et al. 2019). Ethereum is a permission-less network and sensitive information – transactions, bytecode and balance of smart contracts – are visible to everyone, and everyone can call the contract by sending transactions (Wood 2014). These features increase possible security threats and counter-actions needed. Smart contracts on Ethereum have several other unique characteristics – the use of the *"gas"* system to fund running of transactions; relatively few patterns and standards for structuring smart contract code; lack of source code available for most deployed smart contracts; and relative lack of tools to check smart contracts for errors, compared to conventional software. All of these features increase the difficulty of smart contract maintenance.

In software engineering, the term *software maintenance* refers to the modification of a software product after delivery to correct faults and to improve performance or other attributes (Pigoski 1996). It is a very broad activity according to the definition of ISO/IEC 14764 (ISO/IEC 2006). There are four main kinds of maintenance, i.e., adaptive, perfective, corrective, and preventive maintenance. In the context of the four categories of maintenance, the following illustrate the potential impact of such factors on smart contract maintenance:

- ***Adaptive maintenance*** aims to keep software usable in a changed or changing environment. However, the running environment of smart contracts is often unpredictable. For example, smart contracts usually call other contracts. However, the callee contracts might crash and cannot work anymore. Since the callee contracts are immutable, the crash of the callee contract can lead to serious consequences of the caller contract. The unpredictable environment makes it very difficult to conduct adaptive maintenance for smart contracts.
- ***Perfective maintenance*** is used to improve the performance or maintainability by adding new requirements and functionalities newly elicited from users. However, the scalability issues and the gas system of Ethereum make smart contracts difficult to add too many functionalities, else they become very costly to run and unwieldy.
- ***Corrective maintenance*** focuses on fixing discovered bugs and errors in a program. The lack of tools and community support due to the relative newness of smart contracts makes it hard to detect and remove smart contract bugs.
- ***Preventive maintenance*** aims to remove latent faults of programs before they become operational faults. For example, a code smell is a characteristic in the source code that possibly indicates a deeper problem (Fowler and Beck 1999). Refactoring the code

to remove code smells to increase software robustness is a typical preventive maintenance method. However, due to the immature ecosystem of smart contracts, it is not easy to find appropriate advanced methods to conduct preventive maintenance for smart contracts.

In this paper, we focus on the maintenance-related concerns of post-deployment smart contracts. Unlike traditional programs that can be upgraded directly, **to maintain a smart contract, developers usually need to redeploy a smart contract and discard the old version.** Although maintaining smart contracts is not easy, it is still important to find methods to maintain them. For example, in 2016, attackers found the DAO (Decentralized Autonomous Organization) smart contract contains a vulnerability named Reentrancy (Chen et al. 2020b; Luu et al. 2016). This vulnerability was then utilized by attackers and led to the famous DAO attack (Siegel D 2018), which made the DAO lose 3.6 million Ethers (about $20/Ether when the attack happened). According to recent research (Kalra et al. 2018; Liu et al. 2018a), a similar vulnerability is prevalent in Ethereum smart contracts; all of these contracts can be attacked and lead to financial loss. Thus, it is important to conduct corrective maintenance for these contracts to remove issues like the Reentrancy vulnerability to ensure the contracts are bug-free and robust.

Many previous works (Zou et al. 2019; Parizi et al. 2018a; Bosu et al. 2019; Chakraborty et al. 2018; Li et al. 2017) conduct empirical studies to investigate the challenges to the entire software development life cycle of smart contracts. This includes smart contract design, programming, security, maintenance, documentation and so on. However, none focus exclusively on smart contract maintenance. To fill this gap, we provide a comprehensive empirical study on smart contract maintenance based on a systematic literature review that covers 131 smart-contract-related papers selected from a collection of 946 papers to find maintenance-related challenges, and methods for smart contracts. Our study aims to answer the following two key research questions:

RQ1: What kinds of maintenance issues will smart contract developers encounter? We identify 9 issues related to corrective, adaptive, perfective, and preventive maintenance, and another 4 issues corresponding to the overall maintenance process for smart contracts. These maintenance issues are extracted from previous publications. Since Ethereum and smart contracts are fast-evolving, some results from previous works might be outdated. There might be a gap between academia and industry. For example, Zhou (2019) mentioned that smart contracts miss the support of exception handling, e.g., the *try...catch*. However, Solidity adds the exception handling in v6.0 (Solidity 2020b). To make our results more reliable, we use an online survey to validate our findings. We sent the survey to 1,500 smart contract developers on Github, and received 165 useful responses. The feedback from the survey can also be a supplement to our findings. We analyze the reasons for smart contract maintenance issues according to the survey results.

RQ2: What are the current maintenance methods for smart contracts? To help developers maintain smart contracts, we summarize four kinds of current maintenance methods from 41 publications. 31 publications introduce offline checking methods to help developers maintain smart contracts. They can help maintain smart contracts before they are deployed/redeployed to Ethereum. Seven publications introduced online checking methods, which can help maintain deployed smart contracts by detecting malicious input or automatically upgrading smart contracts. Two previous works suggested developers to use the *Selfdestruct* function to undo contracts when emergencies happen. Another work describes how smart contract can be upgraded by using *DELEGATECALL* instruction.

The main contributions of this paper are:

- To the best of our knowledge, this is the first in-depth empirical study that focuses on the maintenance issues of smart contracts on Ethereum, and we divide the issues into four categories.
- Our study identifies the key current maintenance methods used for smart contracts, which gives guidance for smart contract developers to better maintain their contracts.
- Our study highlights the limitations and possible future work related to smart contracts on Ethereum. This gives directions for smart contract developers and researchers to develop improved tools and focus future research.

The remainder of this paper is organized as follows. In Section 2, we provide background knowledge of smart contracts and Ethereum. In Section 3, we introduce the methodology to conduct the literature reviews and the survey. After that, we present the answers to the two research questions in Sections 4 and 5, respectively. In Section 6, we highlight key threats to validity. We discuss what should be done in the future to improve the Ethereum ecosystem in Section 7 and review related work in Section 8. Finally, we conclude the whole study in Section 9.

2 Background

2.1 Ethereum

In 2008, the first blockchain-based cryptocurrency named Bitcoin was introduced and demonstrated the enormous potential of blockchain to the world. However, the biggest limitation of Bitcoin is that it only allows users to encode non-Turing-complete scripts to process transactions, which greatly limits its capability. To address this limitation, Ethereum was born at the end of 2015 and brought a revolutionary technology named smart contracts. Nowadays, Ethereum has become the second most popular blockchain system and the most popular platform on which to run smart contracts. Similar to Bitcoin, Ethereum also provides its cryptocurrency and names it as Ether. In Jan. 2018, Ether reached its highest value to $1389 / Ether (Marketcap 2020). Unlike Bitcoin, which has a fixed number of coins (21 million in total), 18 million Ethers are created every year (Wood 2014) (and 72 million Ether were generated at its launch). Currently, two new Ethers are created with each block, and it requires about 14-15s to create a new block; the average Ethereum block size is between 20 to 30 KB, and the biggest Ethereum block size is around 2MB (Ethstates 2020). Ethereum does not support concurrency, and all transactions need to be executed by all nodes, which leads to a low throughput of Ethereum. Ethereum only allows about 15 transactions per second on average (EtherScan 2018), which has become one of its biggest limitations. At the end of 2017, there is a famous smart-contract-based game named CryptoKitties (Cryptokitties 2019) published in the Ethereum. However, the popularity of the game slowed down all transactions as too many players sent transactions to the Ethereum blockchain.

2.2 Hard Fork and Soft Fork

Any software or operating system needs periodic upgrades to fix errors or add new functionalities. For the blockchain system, those updates are called a “fork”. There are two kinds of forks, i.e., hard fork and soft fork.

Hard Fork Figure 1 shows an example of a hard fork. The blockchain system is a decentralized network. All the nodes on the network need to follow the same rules. The set of rules is known as the protocol. In Fig. 1, the blue block is called a divergence block, where the blockchain system updates its protocol. When a protocol is updated, and the new protocol is not backwards-compatible. Some nodes on the blockchain do not accept the new protocol, and they choose to use the old version. Thus, the blockchain forks into 2 incompatible blockchains, which run the new and old protocol, respectively.

Soft Fork Updates of protocols by soft fork are backwards-compatible. Nodes that did not upgrade to the new version will still be able to participate in validating and verifying transactions. In this case, there is only one chain on the blockchain when using a soft fork. Notice that the functionality of a node with the old protocol is also affected. As the example in Fig. 2 shows, the maximum block size allowed by the old protocol is 3MB, and the new protocol limits the block size to 2MB. The non-upgraded nodes can still process transactions and push new blocks that are 2MB or less. However, if a non-upgraded node tries to push a block that is greater than 2MB, the upgraded nodes will reject to broadcast the block, which encourages the non-upgraded nodes to update the new protocols.

2.3 Smart Contracts

Smart contracts can be regarded as Turing-complete programs that run on the blockchain (Wood 2014). They are usually developed in a high-level language, e.g., Solidity, Vyper (2020). Solidity is the most popular programming language with which to develop smart contracts on Ethereum. Based on the immutable blockchain technology concept, smart contracts cannot be modified once added to the blockchain. Once started, all running of the contract is based on its code. No one can affect it, not even the creator. Ethereum uses EVM (Ethereum Virtual Machine) to execute smart contracts. When developers deploy a smart contract to Ethereum, the contract will be compiled into EVM bytecode, and the bytecode will be stored on the blockchain forever. The only way to remove the bytecode from Ethereum is by using the Selfdestruct function (Solidity 2020b). There is a unique 40 bytes hexadecimal hash value to identify a contract address. Since Ethereum is a permission-less network; every one can send a transaction and invoke contract functions if they know the function signatures, which includes its function id and parameter types (Solidity 2020b).

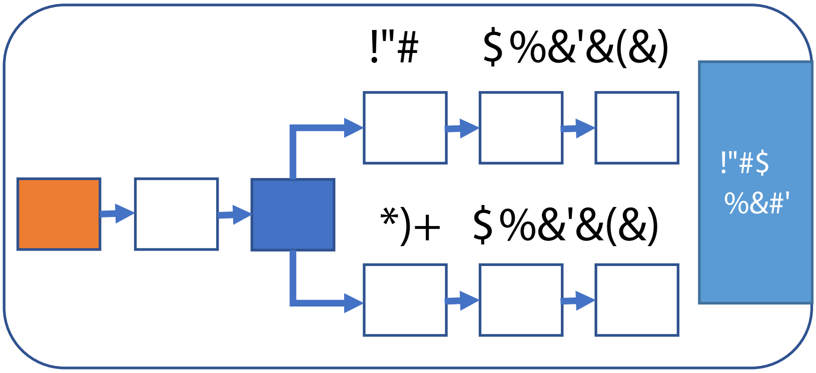

Fig. 1 An Example of Hard Fork. The blue block called divergence block, where the blockchain system updates its protocol. The new protocol for hard fork is not backward-compatible

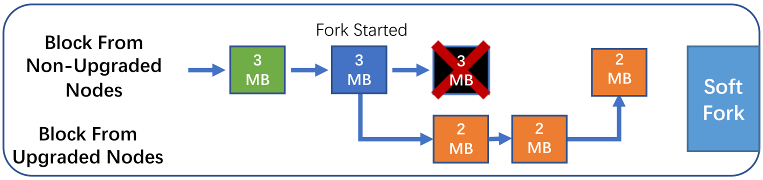

Fig. 2 An Example of Soft Fork. The blue block called divergence block, where the blockchain system updates its protocol. The new protocol for soft fork is backward-compatible

Even worse, all the transactions, bytecode, invocation parameters are visible to everyone, which makes smart contracts face major security challenges.

2.4 The Gas System

In Ethereum, transactions are executed by *miners*. To incentivize the execution of smart contracts by miners, transaction senders need to pay an amount of Ether to the miner, the so-called *gas mechanism*. For each transaction, the EVM will calculate its gas cost, and the transaction sender is required to define a gas price, e.g., 20 Gwei / gas unit ($1Ether = 10^9 Gwei$). The final transaction fee is calculated by $gas\_cost \times gas\_price$. Miners have the right to decide whether or not execute a transaction. Thus, higher gas prices can lead to faster execution, and lower gas prices can lead to a transaction that is never added to a block. According to the *ETH Gas Station* (2020), in May 2020, if the gas price is higher than 40 Gwei, the transaction can be executed within 2 minutes. If the gas price is lower than 25 Gwei, the execution time can exceed half an hour.

Another function the *"gas"* system is to ensure that the execution of smart contracts can be eventually terminated. In Ethereum, the transaction caller is required to set a *gas limit*, which refers to the maximum gas cost of a transaction. If the gas cost of a transaction exceeds the gas limit, the execution will be terminated with an exception thrown by EVM named *out-of-gas error*.

The gas system ensures the normal running of the Ethereum. However, it also increases the difficulty of smart contract development, as developers need to estimate the maximum gas cost of the contracts. Ethereum block has a maximum size, which limits the amount of data that can be included. The current maximum block size limits the maximum gas limit to 12.5 million gas units (Ethstates 2020). When the maximum gas cost of a transaction exceeds the 12.5 million, it will be reverted forever.

2.5 Upgradeable Smart Contracts

Even though smart contracts cannot be changed once deployed to the blockchain, there is a method to develop "upgradeable" contracts. Ethereum provides a function named *DelegateCall*, which allows a contract to use code in other contracts, and all storage changes are made in the caller's value. Specifically, *DelegateCall* can be implemented by *addr.delegatecall(bytes memory)*. *addr* is the address of the callee contract (The value of *addr* can be changed by sending a transaction to the contract). The function selector and input value are encoded as *bytes memory*, and will be sent to the callee contract when *DelegateCall* is executed. Once the execution of the function on the callee contract is finished, the return value will be transferred back to the caller contracts. When bugs are found at the callee contract, the proxy contract can redirect the *addr* to a new contract.

Figure 3 is an example of the upgradeable contract, which contains three contracts. The proxy contract holds the data of a contract, and all the storage changes are made in the proxy contract. The proxy contract uses *DelegateCall* to call the functions *f()* and *g()*. These functions are implemented in contract A and B, respectively. Once errors are found or new functionalities need to be added, contract A and B can be discarded directly. The proxy contract can call the code of the new contract by using *DelegateCall*. Based on this approach, *OpenZeppelin*, a famous smart contract organization, has provided a library (OpenZeppelin 2020) to help developers develop upgradeable smart contracts in just a few lines. *EIP 2535* (Mudge 2021) (the Diamond Standard) also defines the standard to help developers design upgradeable smart contracts.

2.6 Software Development and Maintenance

Software development refers to a set of activities that throughout the entire life cycle of software, which includes the process of designing, creating, deploying and supporting software (Bourque et al. 2014). Thus, software maintenance is an important and inevitable part of the software development life cycle. According to previous work (Boehm and Basili 2005), software maintenance can lead to 60% of software cost. Besides, in many software development models, e.g., Spiral model (Boehm 1988), Agile development (Beck et al. 2001), it is not easy to split the process of development and maintenance. For example, Agile software development refers to software development methodologies based on iterative development. In each iteration, new requirements and solutions will be added to improve the software. According to the definition of ISO/IEC 14764 (ISO/IEC 2006), there are four kinds of software maintenance, i.e., corrective, adaptive, perfective, and preventive maintenance. Among them, *Perfective maintenance* is used to improve the performance or maintainability by adding new requirements and functionalities newly elicited from users, which is similar to the steps of Agile development. Thus, there are many overlaps between the software maintenance and development.

2.7 Card Sorting

Card sorting is a method to organize data into logical groups (Spencer 2009), which is widely used to help users organize and structure data. To conduct a card sorting, we first need to identify the key concepts and write them into labeled cards. A card can be everything that helps the discussion, e.g., a piece of paper or a virtual card on a laptop. After that, we are required to group cards into different categories that make sense to them. Due to the low-tech and inexpensive nature of card sorting, it is usually used to design workflow, architecture, category tree, or folksonomy.

There are three kinds of card sorting, i.e., open card sorting, closed card sorting, and hybrid card sorting. Open card sorting is used for organizing data with no predefined groups.

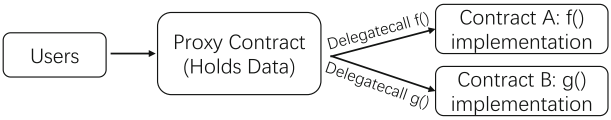

Fig. 3 An Example of the upgradeable contract

Specifically, each card will be clustered into a group with a certain topic or meaning first. If there is no appropriate group, a new group will be generated. All the groups are low-level subcategories and will be evolved into high-level subcategories further. Closed card sorting is used for organizing data with predefined groups. Each card is required to be clustered into one of the groups. Hybrid card sorting combines open card sorting and closed card sorting. Hybrid card sorting has predefined groups but allows to create new groups during the process.

3 Methodology

Figure 4 shows the overview of our methodology, which contains two phases, i.e., literature review and survey. In phase 1, we perform a systematic literature review, which aims to find the answers to research questions from prior smart contract related papers. After obtaining the answers, we use an online survey to validate whether smart contract developers agree with our findings. In the following subsections, we present the detailed steps of our literature review and survey.

3.1 Literature Review

In this paper, we follow the method provided by Kitchenham and Charters (2007) to perform the literature review. There are three steps in phase 1, i.e., literature search, literature selection, and data analysis.

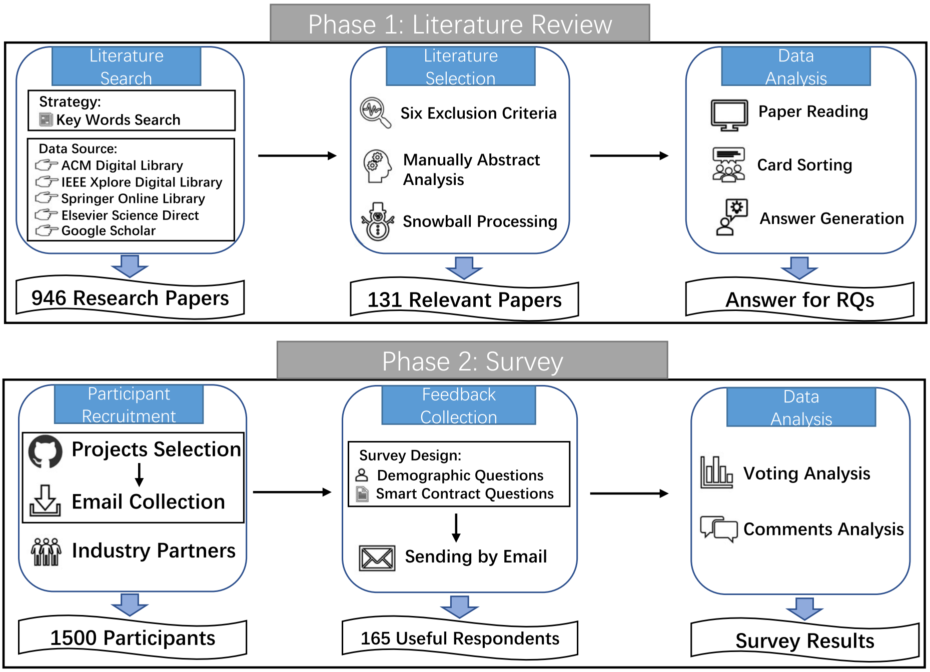

Fig. 4 Overview of methodology design

3.1.1 Literature Search

Guided by prior literature reviews (Conoscenti et al. 2016; Segura et al. 2016; Huang et al. 2019), we select five search engines, i.e., ACM Digital Library, IEEE Xplore Digital Library, Springer Online Library, Elsevier Science Direct, and Google Scholar. From these search engines, we can find peer reviewed research papers published in journals, conferences, workshops, and symposia.

We used keyword search to obtain 946 initial smart contract related papers. The detailed numbers of the research papers returned by different search engines are shown in Table 1. (The duplicated papers are removed.) All of these 946 research papers contain at least one of the keywords “smart contracts” , “smart contract”, “Ethereum”, “blockchain”, “DApps” in their title. Since there are many other blockchain platforms supporting smart contracts, and our focus is Ethereum, all the selected papers should contain the keyword ”Ethereum” or “smart contract” in their abstract.

3.1.2 Literature Selection

Although all the papers that we find in our literature search contain the keywords “smart contract” or “Ethereum” in their abstract, some of them are still irrelevant to our study. For example, some research related to other smart contract platforms might also contain the keyword “Ethereum” in their abstracts. We applied the following five exclusion criteria to remove irrelevant papers:

Exclusion Criteria

(1) Studies are not written in English.
(2) Master or Ph.D. theses.
(3) Keynote papers.
(4) Studies not related to Ethereum.
(5) Studies not related to smart contracts.

In this study we only focus on maintenance-related concerns for post-deployed Ethereum smart contract development issues. Thus, research based on underlying blockchain technology, e.g., consensus algorithms, are excluded. We only focus on the following topics:

Inclusion Topics

(1) Smart contract empirical studies.
(2) Smart contract security / reliability Analysis.

Table 1 Initial number of smart contract related research papers returned by each search engine

Search engine	Papers
ACM Digital Library	73
IEEE Xplore Digital Library	177
Springer Online Library	54
Elsevier Science Direct	11
Google Scholar	631
Total	**946**

(3) Smart contract standards.
(4) Smart contract optimization, e.g., gas optimization.
(5) Other smart contract technologies, e.g, smart contract generation, decompilers.

To reduce errors, we conducted close card sorting (Spencer 2009) to check the collected data. Card sorting is a common method used to evaluate and derive categories from the data (Kim et al. 2016). There are three types of card sort, i.e., open card sort, closed card sort, and hybrid card sorting. Among these three kinds of card sort, closed card sort has predefined categories. We apply closed card sort to select relevant papers, as there only two categories, e.g., relevant or irrelevant. For each card, it has a title (the name of the paper) and description (abstract of the papers). Two experienced researchers with four-year smart contract related experience (including a non-coauthor) carefully read the abstract of the initial 946 research papers independently, and then compare their results after finishing the reading. If there are some differences, they discussed to decide the whether the papers should be excluded. Finally, 112 relevant papers are selected from initial 946 papers. After that, we followed the prior study (Huang et al. 2019) to conduct a snowballing step to enlarge the paper list. We manually checked the references of the identified 112 papers and from these found another 19 papers. All of these 19 papers are selected from the reference of the 946 papers with the same selection method. Specifically, we first check whether the title of the paper on the reference contains the keywords, e.g., “smart contracts”, “Ethereum”, “blockchain”. Then, the two researchers use open card sorting to analyze the abstract of the paper to finally decide whether the paper should be included or not. Thus, we finally selected 131 papers for analysis. The paper list can be found at: https://github.com/Jiachi-Chen/Maintenance.

3.1.3 Data Analysis

The Ethereum proposal was presented in late 2013, and the system went live at the end of 2015. All of the 131 selected papers were published between 2014 to 2020 (for details see Fig. 5), and the full papers were carefully read by the same two researchers. Considering our study aims to find answers with categories being unknown in advance (different kinds of maintenance issues and methods), we decided to adopt an open card sorting approach to help find the answers of these two RQs. The detailed steps used for this are described in Fig. 6. The two researchers first read the paper carefully and were required to collect the answers to the two RQs shown in Table 2, i.e., (1). What are the reported challenges / issues of smart contract maintenance? (2). What are the used maintenance methods? If we could not find any answers from a paper, the paper is omitted from our list. For the answers of

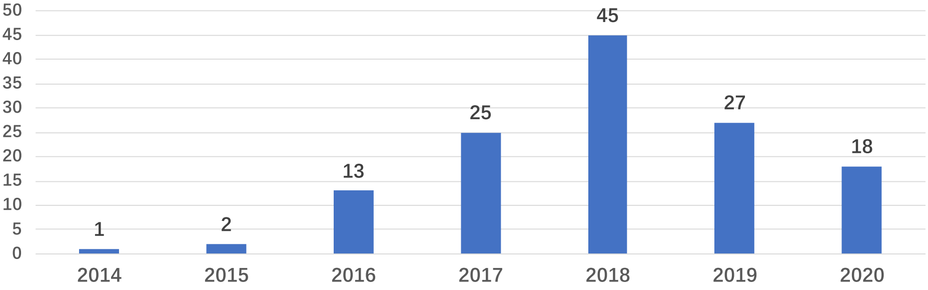

Fig. 5 The number of papers published between 2014 to 2020

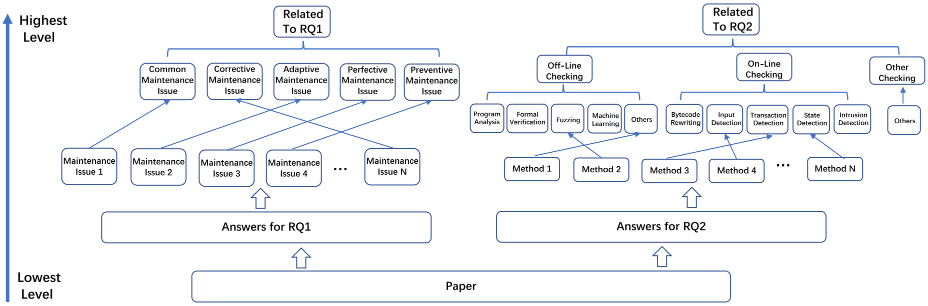

Fig. 6 The steps of open card sorting

(1), the data collected from papers were first summarized into detailed maintenance issues. For example, previous works (Chen et al. 2018a, 2020c) mentioned that "..over 90% of real smart contracts suffer from gas-costly patterns in Ethereum...", which will be summarized into a detailed maintenance issue, i.e., *The Difficulty of Handling the Gas System*. The detailed maintenance issues were then clustered according to their maintenance types, e.g., corrective, adaptive, perfective maintenance, and common maintenance. For the answers of (2), they were first grouped according to the technique they used, e.g., programming analysis or fuzzing. After that, they will be clustered into a higher level according to their checking types, e.g., off-line / on-line checking.

3.2 Survey

3.2.1 Survey Design

Our smart contract developer survey contains three parts, i.e., demographic questions, smart contract maintenance related questions, and suggestion related questions. We follow the previous smart contract related work (Chen et al. 2020b) to design the following five demographic questions in our survey. Since our survey is based on Google Form, and Google cannot be accessed in China, we also designed a Chinese version to receive responses from Chinese developers. The translated version was double-checked to ensure consistence with the English version.

Demographics

- Professional smart contract developer? : Yes / No
- Involved in open source software development? : Smart Contract Projects only / Traditional Projects Only / Both / None

Table 2 Data collection for each RQ

RQs	Type of data we collected
RQ1	What are the challenges / issues of smart contract maintenance? The data is classified by corrective, adaptive, perfective, and preventive maintenance.
RQ2	What are the used maintenance methods? e.g., off-line / on-line security checking methods, other methods.

- Main role in developing smart contract.
- Experience in years
- Current country of residence

These questions aim to understand the background and experience of the respondents, which allows us to remove some feedback that we wish to exclude, e.g., feedback provided by very inexperienced respondents.

In the second part of the survey, we designed 15 questions to help provide answers to the same two research questions that we found from the literature survey. The details of the survey can be found at: https://github.com/Jiachi-Chen/Maintenance. The list of the questions included in our survey can be found in Table 3. For questions 1, 3-6, 8-9, 11, we give the participants several choices that are obtained by literature review. Besides, for these questions, we give a textbox to allow participants to write comments. For questions 10 and 12, we follow the previous survey (Chen et al. 2020b) to give five scores to participants from score 1 (lowest agreement) to score 5 (highest agreement), and score 3 refers to “neutral”.

In the third part of the survey, we give a text box to respondents to allow them to give us final comments or questions.

3.2.2 Survey Design Explanation

In this subsection, we explain how we designed the survey by answering two questions, i.e., (1). How we obtain the choices for questions 1, 3-6, 8-9, and 11. (2). Where do the other questions come from?

Table 3 List of questions included in the survey

ID	Question
Q1	How do you obtain your required knowledge about smart contracts?
Q2	Do you believe smart contracts have higher security requirements than traditional, centralized apps, e.g., mobile apps, web apps?
Q3	How do you test / debug your smart contracts for security and scalability?
Q4	How do you maintain smart contracts after deployment?
Q5-6	Have you developed an upgradeable smart contract before? If not, why?
Q7	Do you believe smart contracts are harder to maintain than traditional centralized apps, e.g., mobile apps, web apps? Why?
Q8	What maintenance issues do your smart contracts have?
Q9	Which features / limitations of Ethereum can increase the difficulty of maintenance?
Q10	Are you satisfied with the current ecosystem for smart contracts, e.g., platforms for sharing data?
Q11	Have you ever used the code of smart contracts from the following platforms, e.g., Github, Stack Overflow, Etherscan?
Q12	Give a score for IDE, testing tools, security audit tools, smart contract explorer, Q&A site, Comments from Public (Github, DApp Store), community support, Solidity and Ethereum document, respectively.
Q13	Do you think smart contracts are suitable for developing a large scale project?
Q14	Do you think it is necessary to have an app store like IOS Store for smart contracts?
Q15	Currently, there are many technologies that can improve the security of smart contracts. Do you think it is important to merge them into EVM / Ethereum / IDE?

Below we list how we obtain the choices for questions 1, 3-6, 8-9, and 11

Question 1: Many previous works have mentioned that smart contract development lacks appropriate tools / techniques to verify code correctness (See Section 4.2.1). However, our literature review showed that there are many tools to check the vulnerabilities of smart contracts. Question 1 was included to validate our hypothesis that practitioners do not consult academic literature. Only asking developers whether they read academic papers might lead them to make a binary choice. Thus, we added some sources like "Books, Blogs, Video Tutorials" to make the choices more representative.

Question 3: Previous works (Zou et al. 2019; Chakraborty et al. 2018) investigated how developers test a smart contract. All the choices are according to the result of their work.

Question 4: All the choices were selected from our literature reviews. From the literature, off-line checking is the most common way to maintain a smart contract. However, this kind of method only works before deploying smart contracts to blockchain, which refer to the second choice. Online-Checking cannot be used directly (See Section 7.2). Thus, we didn't include choices for this method. Besides, the selfdestruct function and upgradeable function can be used to maintain smart contracts, which refers to the third and the fourth choices. Also, we added a choice for developers that never maintain a smart contract, as literature shows that most contracts are never called or used.

Question 5-6: Previous work (Chen et al. 2020a) investigated why developers do not use selfdestruct function. Based on their results, we design the options to collect the answer "why developers do not develop upgradeable contracts".

Question 8-9: All the choices are selected from literature reviews. (All of them can be found at "Answer to RQ1" , see Section 3.1.3)

Question 11: From our literature reviews, we found that the source code used to evaluate smart contract tools are from Q&A websites, Github and Etherscan. Besides, according to authors' experience in developing smart contracts, we also add choices "Solidity Documents", "Code from Google Search or other search engines" and "Other" to make the result more reliable.

Below we answer where our other survey questions come from

Question 2: Literature shows that smart contracts have higher security requirements than traditional apps (See Section 4.1.2). We wanted to investigate whether developers agree with this opinion.

Question 7: Similar to Q2, the literature mentions that the immutability of smart contracts makes them hard to be modified once deployed, which makes smart contracts hard to be maintained (See Section 4.1.2). We wanted to investigate whether developers agree with this opinion.

Question 10 and 12: Literature mentions that smart contracts lack tools to check security (See Section 4.2.1), lack community support (See Section 4.2.2), high-quality reference code (See Section 4.5.2), standards (See Section 4.5.3). We wanted to investigate the attitude of developers about these findings from the literature, and in question 12, we wanted developers to give a detailed score about these findings.

Question 13: Literature shows that smart contracts have scalability issues that cannot support a large-scale project. (See Section 4.4.1) We wanted to investigate whether developers agree with this opinion.

Question 14: In Section 7.1, we discussed that having a DApp store and comment system can help to improve the smart contract system. This question is used to investigate developers' attitudes about this.

Question 15: In Section 7.2, we discussed that merging cutting-edge technologies can help to improve Ethereum and Solidity. This question is used to investigate developers' attitudes about this.

3.2.3 Survey Validation

Guided by Kitchenham and Pfleeger (2008), we utilized an anonymous survey (Tyagi 1989) to collect personal opinions. To increase response rates, we offered a raffle to respondents so that they can choose to leave an email to take part in the raffle to win two $50 Amazon gift cards. We first sent our survey to our research partners to conduct a small scale test to refine the survey. They were asked to tell us (1) Whether the expressions used in the survey are clear and easy to understand, (2) How many minutes were needed to complete the whole survey. The only modifications from this survey validation were the expression of some questions in the survey to make them clearer/more consistent terminology usage. We only changed their grammar or rephrased the sentence to make it easier to understand without adding or deleting questions. All of our research partners said that the survey could be conducted within 15 minutes. Thus, we didn't make any other modifications to the survey.

3.2.4 Recruitment of Respondents

The ideal respondents of our survey are smart contract developers. We aimed to send our survey to Github developers who contributed to smart contract related projects. We first searched for projects on Github by using keywords “Smart Contract”, “Ethereum”, “Blockchain”, and ranked the projects by the most stars. Then, to increase the response rate and exclude non-smart-contract developers, we manually selected relevant projects by reading the descriptions of the projects. After that, we crawled the emails and names of contributors of the selected projects by using Github Developer API[1]. We finally obtained 1,500 emails of developers and sent an email to invite them to participate in our survey. We also have some industry partners working in well-known companies, e.g, Alibaba, Facebook, and sent our survey to them (The number of industry partners is 20). Since some developers might not be familiar with “software maintenance”, we inform the concept in the email to reduce the misleading.

3.2.5 Data Analysis

We received a total of 178 valid responses from 32 different countries (The response rate is about 11.87%), which is a good response number and rate compared to previous smart contract related surveys (Chen et al. 2020b; Zou et al. 2019; Bosu et al. 2019; Chakraborty et al. 2018; Chen et al. 2020a). Among these 178 respondents, 13 of them claim that they do not have any experience in smart contract development. Thus, we removed them from our dataset and used the remaining 165 for further analysis. The top three countries in which respondents reside are China (35.76%), USA (15.15%) and UK (9.09%). The average years of experience in developing smart contracts of our respondents are 2.31 years. Among these respondents, 106 (64.24%) of them claim their main role is development, 42 (25.45%) indicate testing/maintenance/evolution, 29 (17.58%) indicate project management, 6 (3.64%)

---

[1] https://developer.github.com/v3/

indicate risk analysis, 4 (2.42%) indicate research. (Some respondents have multiple job roles; thus the total number exceeds 165.)

4 RQ1: What are the maintenance issues of smart contracts?

There are four broad kinds of maintenance, i.e., corrective, adaptive, perfective, and preventive maintenance. In this section, we identify the key maintenance issues for smart contracts considering these four aspects. We also introduce some common maintenance issues (CMI), which appear in all kinds of maintenance. All the findings are obtained by literature reviews (the source are cited), and we give survey results to cross-validate each finding. It should be noted that software maintenance is a very broad activity. Some kind of maintenance, e.g., perfective maintenance also requires developers to develop new functionalities as well as change old. Thus, some of the challenges we discuss can be encountered in both smart contract development and maintenance phases. We use Tables 4 and 5 to help readers better understand the relation between the survey results and the findings collected from literature. The first column of the tables is the survey ID (detailed information can be found at Table 3). The survey results shown in the third column are used to validate the findings we collected by literature review that are listed in the second column of the table. For example, many literature mentioned that smart contracts have high requirements for security. Thus, in Q2 of the survey, we found that 78.18% of the respondents agree with this result, which shows its correctness.

4.1 Common Maintenance Issues

4.1.1 No Ideal Deployed Contract Modification Methods

Immutability is an important feature of smart contracts, which makes smart contracts distinct from traditional apps in their stability. However, this feature also leads – intentionally – to great difficulty for their modification.

From our survey, we received four answers [2] for the question "How do you maintain your smart contracts" (Q4 in Table 3). The four answers are:

1. I never maintained a contract (18.79%)
2. I discard the old contract directly and deploy a new one (39.39%)
3. I use *Selfdestruct* function to destroy the old contract and deploy a new one (38.79%)
4. I develop upgradeable contracts. (35.76%).

However, all of these four answers are imperfect and can lead to high financial loss in some situations.

For answer (1), this method is very inadvisable as some bugs are usually inevitable. Without maintenance, the usefulness life of the programs will be much shortened and attackers can freely attack existing contracts that contain vulnerabilities.

For answer (2), this method can lead to enormous financial loss for the contract owners, as the Ethers cannot be transferred unless a specific code is included in the contract. Although the contract owners find there is a bug like the reentrancy (Liu et al. 2018a; Rodler

---

[2] The questions are multi-choice. Thus the sum of each options can exceed 100%. The same with the other questions.

Table 4 Part 1 - The mapping between survey results and our findings collected from the literature

Survey ID	Findings and Related Section	Survey Result
Q1	Inconsistent: Previous works reports smart contract development lacks appropriate tools to verify code correctness v.s. Academia proposed many tools in recent years. (S4.2.1)	52.1% respondents obtain knowledge from journal and conference papers → The methods to require knowledge is not the main reason for the inconsistent.
Q2	Smart contracts have high requirements for security (S4.1.2)	Smart contracts have higher security requirements (78.18%)
Q3	Inconsistent: Previous works reports smart contract development lacks appropriate tools to verify code correctness v.s. Academia proposed many tools in recent years. (S4.2.1)	Respondents use program analysis (28.48%), formal verification(9.09%), unit testing(80.61%), code reviews(73.94%), functional and integration testing (70.91%) to test smart contracts → Most of tools proposed by academia are hard to used and not user friendly
Q4	The immutable of smart contracts lead to the great difficulty for their modification. (S4.1.1)	Four methods to maintain a smart contract, and all of them are imperfect.
Q5-6	Developing upgradeable contracts is also not a ideal method to maintain smart contracts (S4.1.1)	Developing upgradeable contracts can increase development cost and security risks. (32.17% and 33.04%)
Q7	Smart contracts have high requirement for security (S4.1.2)	Smart contracts are harder to maintain compared to traditional apps (64.85%)
Q8	Smart contract development lacks appropriate tools to verify code correctness (S4.2.1)	Lack of tools / techniques to audit code. (66.2%)
	The grammar of Solidity is too simple to support large projects, which lead to the scalability issues (S4.4.1)	There are not enough useful libraries and APIs (49.7%); not easy to handle the memory and storage in Solidity programming (38.79%)
	Gas system is also not easy to use, especially when the scale of the project becomes larger. (S4.4.2)	It is not easy to handle the gas system when maintaining smart contracts (38.79%)
	The qualities of open-source smart contracts are poor in Ethereum (S4.5.2)	Solidity lacks useful reference code. (38.18%)
	There are only limited numbers of smart contract related standards (S4.5.3)	Ethereum lacks standards (49.7%)

et al. 2018) in their smart contracts, there was no way to modify the contract, as the contract did not contain a Selfdestruct function and was not develop as an upgradeable contract, which might lead to an enormous financial loss for the organization.

For answer (3), adding a *Selfdestruct* function can reduce the financial loss when emergencies happen. Using the DAO attack as an example, if the DAO contract had this function, the DAO organization could use it to destruct the contract and transfer all the Ethers when the attack was detected. After fixing the bugs, they can deploy a new contract, and transfer the Ethers to the new contract. However, this method is still harmful to both contract owners and users in some situations. Our previous work (Chen et al. 2020a) investigated the reasons why developers do not add Selfdestruct functions in their contracts. Developer feedback showed the following reasons. *First*, adding a Selfdestruct function also opens an attack vector to the attackers. Thus, developers need to pay more effort to test smart

Table 5 Part 2 - The mapping between survey results and our findings collected from the literature

Survey ID	Findings and related section	Survey result
Q9	There is more financially attractive for attacking smart contracts compared to traditional software, thus leading to more attack (S4.1.2)	There is more financially attractive for attacking smart contracts (49.09%)
	Ethereum smart contracts run on a permissionless network, which lead to higher requirement for security (S4.1.2)	The permission-less feature could increase the difficulty of maintenance. (55.76%)
	Making smart contracts readable is a challenge (S4.1.3)	89.1% respondents use the source code of smart contracts (Q11), and 57.14% of them said the poor readability of smart contracts increases the difficulty of code reuse.
	Some unplanned forks can increase the difficulty of smart contract maintenance.(S4.3.1)	Ethereum might add new functions through hard fork, which might affect the currents contracts running on the blockchain. (50.3%)
	Many callee contracts on Ethereum contain vulnerabilities, which might lead to the crash and make the contracts cannot work anymore.(S4.3.2)	It would make their contracts hard to be maintained if the callee contracts crashed or be destructed. (62.42%)
Q10	Ethereum lacks advanced software engineering theories to perform preventive maintenance. (S4.5)	Only 7.88% and 16.97% respondents said they are very satisfied or satisfied with the current ecosystems of smart contracts.
Q11	Making smart contracts readable is a challenge (S4.1.3)	89.1% respondents use the source code of smart contracts, and 57.14% of them said the poor readability of smart contracts increases the difficulty of code reuse. (Q9)
Q12	Community support is not enough for smart contract developers. (S4.2.2)	The community support receives an average score of 3.03
Q13	The Scalability Issues of Smart contracts cannot support large scale projects (S4.4.1)	Only 14.55% respondents believe smart contracts are suitable for developing a large scale project
Q14	DApp Store and Comment System can improve the smart contract ecosystem. (S7.1)	Having positive opinions about the need for a DApp store like the Android Google Play Store (84.24%)
Q15	Merging Cutting-Edge technologies can improve the performance of Ethereum and Solidity (S7.2).	90.9% respondents hold positive opinions about merging cutting-edge technologies into the EVM and updated by nodes on Ethereum.

contract security and permissions. The testing can add additional complexity to the development, which can increase the development cost. *Second*, adding a Selfdestruct function can also lead to a trust concern for the smart contract users. This is because many users trust Ethereum because of the immutability of smart contracts. All the execution of the contract depends on its code; even the owner cannot transfer Ethers on the contract balance. This feature is important in financial applications as it ensure the asset safety of contract users. However, the *Selfdestruct* function breaks the immutability of the contracts. It gives power to the contract owners to transfer all the Ethers of the contracts. Thus, this method can lead to the reduction of the number of users of the smart contract using it. *Finally*, the *Selfdestruct* function can also lead to a financial loss in some situations, as the Ethers that were sent to the contract after destroying it will be lost. Thus, this method is still not a perfect method to maintain smart contracts.

For answer (4), still raises the same trust concern similar to answer (3), as the smart contract immutability features are also be broken. According to our survey (Q5-6 in Table 3), we found that only 29.70% of the respondents have developed upgradeable smart contracts. There are three reasons why developers do not develop upgradeable contracts. 41.74% of the respondents claim that they do not know how to develop upgradeable smart contracts. Thus, to develop upgradeable smart contracts, they need to pay a learning cost. 32.17% and 33.04% of the respondents said developing upgradeable contracts can increase the development cost and security risks. Thus, this method still incurs a high cost for maintenance.

To summarize, all of these four methods have disadvantages or limitations, and can lead to a high cost of smart contract maintenance.

4.1.2 High Requirement for Security

Unlike traditional programs that can be upgraded directly, developers need to redeploy a new smart contract to the blockchain. Ensuring the security of the contract before redeploying it to the blockchain is important, as each the modification can cost a lot (see Setion 4.1.1). According to our survey (Q2 and Q7), 129 (78.18%) respondents believe smart contracts have higher security requirements. 107 (64.85%) respondents said smart contracts are harder to maintain compared to traditional apps. The reasons introduced below lead to the high-security requirement of the smart contracts.

1. **The immutability Features**. All the transactions and the code of smart contracts are immutable, which means that developers need to ensure the security of the code and each transaction. Once any bugs are detected, there is no direct way to patch them. Attackers can utilize the errors / bugs to steal Ethers or lock the balance maliciously (Atzei et al. 2017). Thus, immutability raises a high security requirement for the smart contracts.
2. **Financial Attractiveness**. Financial profit is an important motivation for attackers. According to our survey (Q9), about 81 (49.09%) respondents believe that there is more financially attractive for attacking smart contracts compared to traditional software, thus leading to more attack (Torres et al. 2019). Since many contracts hold Ethers, attackers can earn profits through their attacks. Even worse, the sensitive information of smart contracts are visible to anyone, e.g., bytecode, Ethers on the balance. Attackers can launch precision strikes to the vulnerable contracts. Thus, developers need to pay more efforts to ensure the security of smart contracts.
3. **Permission-less Network**. Ethereum smart contracts run on a permission-less network; everyone can execute the smart contracts by sending a transaction. 92 (55.76%) respondents (Q9) mentioned that the permission-less feature could increase the difficulty of the maintenance. They need to pay more effort to test the permission of the contracts. Previous work (Chen et al. 2020a) introduced a security issue named *Limits of Permissions*. Some contracts do not check the permission of their sensitive functions. Attackers can utilize the vulnerabilities of the permission check to steal Ethers.

4.1.3 Low Readability

Readability is important to help developers understand the smart contracts and maintain their smart contracts (Zou et al. 2019). According to our survey, 147 (89.1%) respondents

(Q11) claim that they use the source code of other smart contracts from open sourced platforms, e.g., Etherscan, Github to help author and maintain their smart contracts. 57.14% of the respondents (Q9) also said the poor readability of smart contracts increases the difficulty of code reuse. Making smart contracts readable is a challenge, as developers need to balance the readability with gas consumption. For example, optimizing code is a common method to reduce gas consumption. The more gas-efficient code usually corresponds to shorter code. However, this shorter code can lead to poorer readability.

4.1.4 The Lack of Experienced Developers and Researchers.

Experienced developers and researchers are the main inventors of new advanced SE methods to address the limitation of smart contracts, e.g., developing tools, improving ecosystem. However, our survey results and literature review shows that less experienced people programming in Ethereum compared to traditional development.

Ethereum is a young system, which was published in 2016. The most experienced developers and researchers of the respondents of the survey have 4 years experience (22 respondents) in smart contracts development, the minimum, average, and median numbers are 0.2, 2.31, and 2.5 years, respectively. Compared to the experiences of the respondents (including developers and researchers) of previous works, e.g., in machine learning (Wan et al. 2019) (min: 3, max: 16, median: 6, avg: 7.6 years), in desktop software development (Wan et al. 2018) (min: 3, max: 12, avg: 6.5 years), the smart contract developers and researchers seem less experienced.

4.2 Corrective Maintenance Issues

It is not easy to discover all potential bugs before deploying smart contracts to the blockchain. Some bugs / errors of the contracts might be exposed to the public under certain situations. Corrective maintenance is the modification of a smart contract after deployment to the blockchain to correct discovered bugs / errors. Diagnosing errors of smart contracts is the major task in corrective maintenance. However, it is painful and difficult to diagnose errors in a smart contract. According to our survey, 96 (66.2%) respondents (Q8) complain that debugging and testing is not easy. There are two main reasons that lead to the difficulty of the diagnosing errors, i.e., the lack of mature tools and community support.

4.2.1 The Lack of Mature Tools

Many previous works (Zou et al. 2019; Norvill et al. 2017; Bosu et al. 2019) mentioned that smart contract development lacks appropriate tools / techniques to verify code correctness. Thus, it is not easy to fix bugs in smart contracts. A similar theme is also received in our survey. 96 (66.2%) respondents (Q8) claim that they cannot find useful tools to debug / test / audit their contracts. However, with the development of smart contract ecosystems, a large number of tools have been developed. For example, tools based on static analysis (Luu et al. 2016; Liu et al. 2018a; Tikhomirov et al. 2018) and formal verification (Bhargavan et al. 2016; Bigi et al. 2015; Hildenbrandt et al. 2018) have been proposed. Some tools have excellent performance and speed in detecting common security issues. Thus, “lack of tools” seems to be addressed with the effort of researchers and developers. There is a gap between academia and industry, as many tools developed in academia are not yet known about and used in industry.

To find the reason, we asked how developers obtain their required knowledge about smart contracts. The Solidity documentation, blogs, and Q&A website are the top three most popular sources to acquire knowledge; the numbers are 149 (90.3%), 114 (69.1%), and 88 (53.3%), respectively (Q1). The state-of-art tools usually published in academic journal and conference papers, and 86 (52.1%) respondents (Q1) said journal and conference papers are an important approach to require knowledge. Thus, the methods to require knowledge is not the main reason why developers think that there are not enough tools.

We also investigated the usage conditions for different kinds of tools and how developers test their contracts. We found that only 47 (28.48%) and 15 (9.09%) respondents (Q3) use static analysis tools and formal verification tools to test their smart contracts. Unit testing, code reviews, functional and integration testing are still the most popular methods to test smart contracts. About 80.61%, 73.94%, and 70.91% of respondents (Q3) choose these methods to test their contracts. Developer comments said that "although there are many tools that can be chosen, most of them are hard to use and not user friendly". Thus, although there is a large number of tools that have been developed, developers still complain there are only a few tools they think can be used in practice.

4.2.2 The Lack of Community Support

Community support is a primary source of knowledge for blockchain software projects (Chakraborty et al. 2018). Community support consists of many parts. For example, when developers encounter technical problems, a Q&A website such as Stack Overflow is an important source to help them address the problems. Developers can open source their projects to Github. Other developers can submit issue reports to help them polish the projects. The App store is also an important place to receive reviews. Reviews might contain feature requests, user feedbacks, issue reports that can help developers upgrade their software.

However, community support is not enough for smart contract developers. Previous works (Zou et al. 2019; Hegedűs 2019) found that smart contract developers lack community support as the blockchain technology is new and there are not enough smart contract developers to answer their questions. Since more and more developers take part in smart contract development, we used our survey to investigate whether community support is still lacking in Ethereum.

In our survey, we asked respondents to give a score for the community support (Q12). Score 1 refers to 'very unsatisfied', 3 refers to 'neutrality', and 5 means 'very satisfied'. The community support receives an average score of 3.03, while the score for other comparative items e.g., Solidity document, and Smart contract Explorer receive scores of 3.53 and 3.52, respectively. Thus developers still believe that community support is not sufficient compared to other resources. Surprisingly, the score for the Q&A website, e.g, Stack Overflow, is 3.43, which can show that the Q&A website is not the culprit for the lack of community support. We found that the score for the "Comments from public (E.g., DApp, Github)" is only 2.57, which is the lowest score among all the comparative items.

Previous works (Zou et al. 2019; Hegedűs 2019) claimed that smart contract developers lack community support because there are not enough smart contract developers to answer technical questions. However, our survey shows a different answer. The culprit for the lack of community support is not the Q&A website, but the comments from the public, e.g., issue reports from Github, comments from App Store.

4.3 Adaptive Maintenance Issues

Adaptive maintenance aims to keep a software product usable in a changed or changing environment. In traditional software, the environment changes are usually reflected in the upgrading of the operating systems, the hardware, or software, e.g., database. Conducting adaptive maintenance for the traditional environment changing is not difficult, as these kinds of environment changes are predictable. For example, the updated operating systems usually will give a specific date and detailed API documents.

However, the environment of smart contracts is more unpredictable. In this subsection, we highlight two challenges, which makes it is not easy to conduct adaptive maintenance for smart contracts.

4.3.1 Unpredictable Fork Problems

Ethereum uses soft forks and hard forks (See Section 2.2) to update the blockchain system. Some forks are planned, while some are controversial unpredictable forks, which might result in smart contract maintenance needs.

In a planned fork, developers are informed in advance, and they usually do not need to update the code of smart contracts. For example, in 2017, a hard fork named “Byzantium” of Ethereum added a ‘REVERT’ opcode, which permits error handling without consuming all gas (Mushegian 2020). The function *revert()* in smart contract code will refer to the new opcode automatically. Thus, the planned forks are more likely to be accepted by miners and developers.

However, unplanned forks are also common in Ethereum, which can increase the difficulty of smart contract maintenance. The first unplanned fork happened in July 2016 and was the result of the DAO attack (Siegel D 2018). The DAO attack made the DAO (Decentralized Autonomous Organization) lose 3.6 million Ethers. To retrieve the loss, the DAO appealed for a hard fork. The hard fork reversed all the transactions to the block before the attack. This hard fork is controversial, as many miners believe it breaks the law of Ethereum. The opposition miners did not take part in the fork, and a new blockchain was generated, named Ethereum Classic (ETC) (2018). After the hard fork, both ETC and Ethereum contain the same smart contracts. Thus, which contracts to maintain might be a problem for some developers. The same situation also happened to their callee contracts. For example, contract A has two callee contracts, i.e., contract B and C. Unfortunately, contract B chooses to maintain the contract on ETC, while contract C chooses to maintain the contract on Ethereum. Thus, contract A will always have a unmaintained callee contract.

In Oct. 2016 and Nov. 2016, two unpredictable hard forks were launched to address different problems that have arisen from the DoS attacks. These two hard forks named “EIP-150 Hard Fork” (2020) and “Spurious Dragon” (2020), respectively. In “EIP-150 Hard Fork”, Ethereum increased the gas cost of every type of call from 40 to 700 unit. The “Spurious Dragon” also increases the gas cost of the “*EXP*” opcode. This increased gas cost might increase the risk of “out-of-gas error”. Thus, some contracts need to refactor their code to handle these gas cost changes.

According to our survey, 83 (50.30%) respondents (Q9) are afraid that the forks of Ethereum might result in various potential problems for their smart contracts. Moreover, the unpredictable forks make it difficult for developers to perform adaptive maintenance.

4.3.2 Unpredictable Callee Contracts

Ethereum is a permission-less network; everyone can call the function of the smart contract by sending a transaction. Michael et al. (Frowis and Bohme 2017) investigated the call relations of smart contracts on Ethereum by checking the hard code address on their bytecode. They found that it is very common for smart contracts to call each other in Ethereum. However, they also found that many callee contracts on Ethereum contain vulnerabilities. These vulnerabilities might lead to the crash and make the contracts cannot work anymore. Beside, many callee contracts also contain *selfdestruct* function, which allow their contract owners to destruct the contracts. Once a contract is destructed, the contract cannot be called anymore, and all the Ethers sent to the destructed contract will be locked forever.

According to our survey (Q9), 103 (62.42%) respondents said it would make their contracts hard to be maintained if the callee contracts crashed or be destructed.

4.4 Perfective Maintenance Issues

As long-lived software (Lohr and Peldszus 2020), users are likely to elicit new requirements during the entire smart contract life cycle. Thus, adding additional functionalities, performance enhancement, and efficiency and maintainability improvements for smart contracts are necessary to respond to the new requirements. This is called the perfective maintenance of smart contracts. Thus, there is an overlap between perfective maintenance issues with development issues, as some new functionalities are required to be developed during this maintenance process.

However, due to the scalability issues of Solidity and EVM, it is not easy to add too many functionalities to smart contract-based projects. The Gas system also increases the difficulty of perfective maintenance. Due to these issues, we find that only 24 (14.55%) of the respondents (Q13) of our survey believe smart contracts are suitable for developing a large scale project.

4.4.1 The Scalability Issues

Solidity Solidity is the most popular programming language for smart contract development, which is an object-oriented language and a bit like JavaScript. However, the grammar of Solidity is too simple to support large projects, which lead to the scalability issues of smart contracts (Zou et al. 2019). First, 82 (49.70%) respondents (Q8) to our survey said there are not enough useful libraries and APIs. Thus, developers need to develop various kinds of APIs and libraries which increases the difficulty of implementing new requirements. Besides, 62 (37.58%) and 64 (38.79%) respondents (Q8) also said it is also not easy to handle the memory and storage in Solidity programming, respectively. For example, Solidity only allows creating 16 local variables in a function. Thus, developers have to use storage variables instead of local variables. Peter et al. (Hegedűs 2019) investigated more than 40,000 smart contracts on Ethereum using 16 metrics, e.g., LOC, nesting level. They found the smart contracts are neither overly complex nor coupled much, and do not rely heavily on inheritance. Their results also prove that real-world smart contracts are small-scale programs and do not contain too many functionalities.

EVM The Ethereum Virtual Machine (EVM) is the runtime environment for smart contracts in Ethereum. Some features of EVM make it scale poorly to support large-scale projects. First, EVM does not support multi-thread execution, which makes the execution of smart contracts inefficient. In some large-scale projects, it is important to execute multiple functionalities in parallel to increase execution speed (Zou et al. 2019). Second, EVM limits the maximum size of stack to 1024 items with 256 bits for each item. The limited stack sizes can easily lead to vulnerabilities and increase the difficulty of developing complex applications (Luu et al. 2016). Finally, EVM uses a key-value store, which is a very simplistic database and can lead to low efficiency (Grech et al. 2019).

Ethereum Ethereum does not support concurrency. To construct the blockchain and ensure security, each node on Ethereum stores the entire transaction history and current state of Ethereum, e.g., account balance, contract variables. Thus, all transactions must be executed and verified by all the nodes. This mechanism makes Ethereum support only around 15 transactions per second, leading to serious scalability issues of smart contract applications. (Bez et al. 2019)

4.4.2 The Difficulty of Handling the Gas System

Ethereum adopts a unique gas system to execute the computational cost of each transaction. The gas system ensures the normal running of the Ethereum system, e.g., giving rewards for miners, avoiding DoS Attack. However, this gas system is also not easy to use, especially when the scale of the project becomes larger. According to our survey, 64 (38.79%) respondents (Q8) claim that it is not easy to handle the gas system when maintaining their smart contracts.

First, users need to pay Ethers for the gas cost, and the gas cost depends on the computational cost of the code. Thus, it is important for developers to reduce the gas cost. As we discussed in Section 4.1.3, there is a trade-off between the gas cost and the readability, and readability is very important for maintenance and large-scale projects. According to previous works (Chen et al. 2018a, 2020c), over 90% of real smart contracts suffer from gas-costly patterns in Ethereum. However, fixing these gas-costly patterns reduce the readability of smart contracts.

4.5 Preventive Maintenance Issues

Preventive maintenance aims to lessen the likelihood of a sudden breakdown of the programs (Tai and Alkalai 1998). Guided by advanced software engineering theories, preventive maintenance usually involves some form of redesign or refactor of a smart contract to remove latent faults / errors/ bugs. For example, a code smell is not a bug but are any characteristics in the source code that possibly indicates a deeper problem (Fowler and Beck 1999). Refactoring the code to remove code smells in software to increase its robustness is a typical preventive maintenance method. However, due to the immature ecosystem of smart contracts, it is not easy to find appropriate advanced software engineering (SE) methods, e.g., code smells for smart contracts, to perform preventive maintenance. According to our survey (Q15), only 13 (7.88%) and 28 (16.97%) respondents said they are very satisfied or satisfied with the current ecosystems of smart contracts.

4.5.1 The Lack of Advanced SE Approach and Research Data

During our literature review, we found that there are only a small number of works that propose advanced SE methods to help conduct the preventive maintenance of smart contracts. Most of these works aim to improve the reliability of smart contracts, e.g., security check tools (detailed introduced in Section 5). Compared to traditional software, the maintenance methods of smart contracts to remove latent errors are much less, e.g., code smell removal (Fontana et al. 2016), bug prediction (Giger et al. 2012), self-admitted technical debt determination (Yan et al. 2018). The lack of research data is an important issue.

In traditional software maintenance, a large number of MSR (Mining Software Repository) methods have been developed to help conduct preventive maintenance. For example, history bug reports can be utilized to predict whether a source code file contains latent errors (Zhang et al. 2019). User reviewers can provide feature requests to help developers improve the programs (Maalej and Nabil 2015; Grano et al. 2017). Comments in source code can be used to detect self-admitted technical debate, which can be used to signal future errors (Yan et al. 2018). Privacy policies, Stack Overflow (SO) posts, error messages, and commit messages are wildly used to help maintain traditional apps. These methods are not difficult to be applied to smart contract projects. However, the lack of related research data makes it is not easy to develop advanced SE methods for smart contracts.

4.5.2 The Lack of High Quality Reference Code

High-quality reference source code can be a good example when developers conduct preventive maintenance. However, the qualities of open-source smart contracts are poor in Ethereum, and 63 (38.18%) respondents (Q8) of our survey mentioned that Solidity lacks useful reference code.

He et al. (2019) found that the copy-paste vulnerabilities were prevalent in Ethereum, and over 96% of smart contracts have duplicates, which means the ecosystem of smart contracts on Ethereum is highly homogeneous. Among these contracts, 9.7% of them have similar vulnerabilities. Similar findings are reported by Kiffer et al. (Kiffer et al. 2018); they investigated 1.2 million contracts, and they can be reduced to 5,877 contract “clusters” that have highly-similar bytecode. The highly homogeneous nature of smart contracts show that only a limited number of contracts can be referenced during maintenance and development.

Kiffer et al. (2018) also found that more than 60% of smart contracts are never actually called. Most of these contracts are useless and hard to be reused. Similar findings were also reported by Di and Salzer (2019). They analyzed the bytecode of smart contracts on Ethereum and found 44,883 are useless and hard to be reused. Only 0.6% of the contracts have more than 1,000 transactions, while most of the active contracts are similar ERC20 contracts (Fabian and Vitalik 2018), which are used to make tokens. Thus, the active contracts also cannot provide too much reference value.

Hegedűs (2019) analyzed more than 40 thousand Solidity source files. They found that the open sourced smart contract code either quite well-commented or not commented at all. Without comments in the source code, it is not easy for developers to understand and reuse the reference code.

4.5.3 The Lack of Standards

Standards can give guidance for developers to increase the maintainability and reliability of their smart contracts, which is the main motivation for preventive maintenance. For example, the ERC 20 (Fabian and Vitalik 2018) standard defines some rules for token-related contracts. The rules contain 9 functions (3 are optional) and 2 events. This standard allows any tokens on Ethereum to be re-used by other applications, e.g., wallets, decentralized exchanges. At the end of 2017, the Cryptokitties (2019) was published and swept the globe.

To help other developers develop similar applications, ERC 721 was published in Jan. 2018. ERC 721 is a standard that describes how to build non-fungible tokens (NFTs) on Ethereum, and a NFT is a unit of data on blockchain that represents an unique digital asset, e.g., a photo or a game. Developers can conduct preventive maintenance to make their contracts follow the ERC 721 standard. Thus, their applications can much more easily interact with other similar applications.

However, there are only limited numbers of smart contract related standards (EIP 2020). According to our survey (Q8), 82 (49.70%) respondents said Ethereum lacks standards, which increases the difficulty of the maintenance of smart contracts.

5 RQ2: What are the current maintenance methods for smart contracts?

We discuss answers found for our second Research Question, and introduce the current smart contract maintenance methods identified from 41 analysed research papers.

5.1 Distribution

Among our 131 smart contract selected papers, 41 papers proposed methods that can be used to maintain smart contracts. Unlike traditional software where programs can be upgraded directly, smart contracts need to redeploy new versions to the blockchain and discard old versions. Most maintenance methods check security issues of smart contracts before redeploying them to the blockchain, which are so-called *offline checking methods*. There are 31 papers related to this topic. 7 research papers propose methods that can help maintain a deployed smart contracts. This kind of method is called an *online checking method*. The final three papers introduce a method that uses *DELEGATECALL* to upgrade a smart contract, and a method that redeploys smart contracts by using *Selfdestruct* function, respectively. The distribution of these methods is shown in Fig. 7.

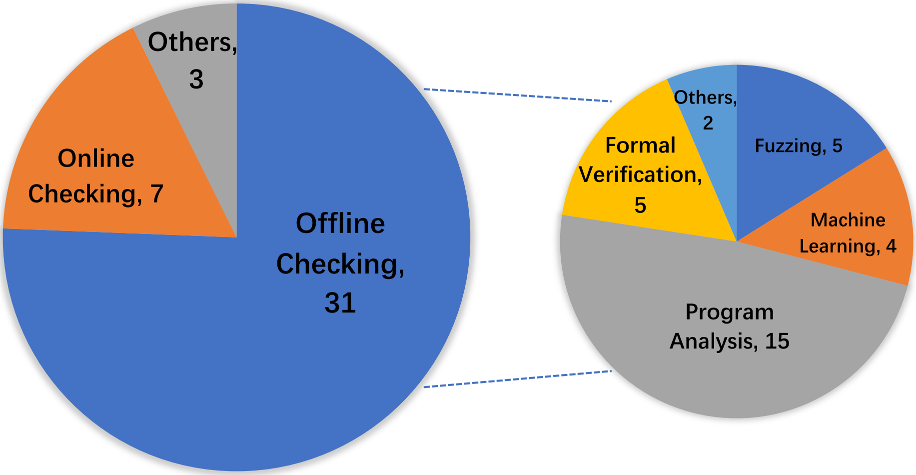

Fig. 7 Distribution of maintenance methods

5.2 Offline Checking Methods

Table 6 summarises the 31 publications which use offline checking methods to help maintain smart contracts. Developers can use the proposed methods to check for security vulnerabilities to help them to maintain smart contracts. For example, using the proposed methods to locate bugs during corrective maintenance, and checking for vulnerabilities of the update versions before redeploying them to Ethereum. We divide the methods presented in these papers into five categories – program analysis, fuzzing, formal verification, machine learning, and others. In the following subsections, we discuss some key examples.

Table 6 Literature of offline checking methods

Category	Name of Publications	Years
Program Analysis	OSIRIS: Hunting for Integer Bugs in Ethereum Smart Contracts (Torres et al. 2018)	2018
	The art of the scam: Demystifying honeypots in Ethereum smart contracts (Torres et al. 2019)	2019
	Security Assurance for Smart Contract (Zhou et al. 2018)	2018
	Vandal: A Scalable Security Analysis Framework for Smart Contracts (Brent et al. 2018)	2018
	MadMax: surviving out-of-gas conditions in Ethereum smart contracts (Grech et al. 2018)	2018
	Finding The Greedy, Prodigal, and Suicidal Contracts at Scale (Nikolić et al. 2018)	2018
	sCompile: Critical Path Identification and Analysis for Smart Contracts (Chang et al. 2019)	2019
	teether: Gnawing at Ethereum to Automatically Exploit Smart Contracts (Krupp and Rossow 2018)	2018
	Making Smart Contracts Smarter (Luu et al. 2016)	2016
	Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contract (Mossberg et al. 2019)	2019
	SmartCheck: Static Analysis of Ethereum Smart Contracts (Tikhomirov et al. 2018)	2018
	TokenScope: Automatically Detecting Inconsistent Behaviors of Cryptocurrency Tokens in Ethereum (Chen et al. 2019a)	2019
	Towards saving money in using smart contracts (Chen et al. 2018a)	2018
	GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts (Chen et al. 2020c)	2020
	Securify: Practical Security Analysis of Smart Contracts (Tsankov et al. 2018)	2018
Formal Verification	Formal Verification of Smart Contracts (Bhargavan et al. 2016)	2016
	A formal verification tool for Ethereum VM bytecode (Park et al. 2018)	2018
	Kevm: A complete formal semantics of the Ethereum virtual machine (Hildenbrandt et al. 2018)	2018
	Towards verifying Ethereum smart contract bytecode in Isabelle/HOL (Amani et al. 2018)	2018
	ZEUS: Analyzing Safety of Smart Contracts (Kalra et al. 2018)	2018

Table 6 (continued)

Category	Name of Publications	Years
Fuzzing	ContractFuzzer: fuzzing smart contracts for vulnerability detection (Jiang et al. 2018)	2018
	ReGuard: Finding Reentrancy Bugs in Smart Contracts (Liu et al. 2018a)	2018
	EVMFuzz: Differential Fuzz Testing of Ethereum Virtual Machine (Fu et al. 2019)	2019
	sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts (Nguyen et al. 2020)	2020
	Exploiting the Laws of Order in Smart Contracts (Kolluri et al. 2019)	2019
Machine Learning	S-gram: Towards Semantic-Aware Security Auditing for Ethereum Smart Contracts (Liu et al. 2018b)	2018
	Hunting the Ethereum Smart Contract: Color-inspired Inspection of Potential Attacks (Huang 2018)	2018
	Towards Safer Smart Contracts: A Sequence Learning Approach to Detecting Security Threats (Tann et al. 2018)	2019
	Checking Smart Contracts with Structural Code Embedding (Gao et al. 2020)	2020
Others	Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach (Mavridou and Laszka 2018)	2018
	Mutation Testing for Ethereum Smart Contract (Li et al. 2019)	2019

5.2.1 Program Analysis

CFG (Control Flow Graph) Based Tools In 2016, Luu et al. (2016) identified four kinds of new security issues of smart contracts and proposed the first tool, named *Oyente*, to detect them through Ethereum bytecode. Although EVM is a stack-based machine, similar to JVM, Ethereum bytecode has many differences compared to the Java bytecode. For example, Java bytecode has a clearly-defined set of targets for every jump, but the jump position of Ethereum bytecode needs to be calculated during symbolic execution. Thus, *Oyente* first splits opcodes into several blocks and then uses symbolic execution to build CFG (Control Flow Graph). CFG stores the relationship between blocks, e.g., jump, conditional jump. Based on the CFG, *Oyente* defines several rules to detect related security issues.

A similar method to that of *Oyente* has been widely applied by other tools. For instance, *GasReducer* (Chen et al. 2018a) and *GasChecker* (Chen et al. 2020c) are tools used to detect some gas-inefficient patterns. They use the CFG generated by *Oyente*, and design patterns to detect related security vulnerability patterns. Besides, Torres et al. (2019), Chang et al. (2019), Nikolić et al. (2018), Zhou et al. (2018), Krupp and Rossow (2018), and Mossberg et al. (2019) also use similar methods that design rules based on the CFG to detect other smart contract vulnerabilities.

Some works make optimizations, e.g., Maian (Nikolić et al. 2018) validate the results of the symbolic execution by using a concrete validation step. In the concrete validation, they create a private fork of Ethereum and then run the result generated by the symbolic execution to check its correctness. Since the results are generated by symbolic execution, and concrete validation is used to increase performance, we also classify Maian in this category.

Decompilers *Vandal* (Brent et al. 2018) is a decompiler for smart contract bytecode. Its output includes a control-flow graph, three-address code for all operations, and function boundaries. Based on *Vandal*, developers and researchers can develop other tools to maintain their smart contracts. For example, *MadMax* (Grech et al. 2018) uses logic-based specifications to detect gas-focused vulnerabilities of smart contracts based on the output of *Vandal*. Tsankov et al. (2018) proposed a tool named *Securify*, which uses semantic information to detect vulnerabilities of smart contracts bytecode. *Securify* first decompiles the EVM bytecode. It then analyzes the data flow and control flow dependencies. Finally, it uses several patterns to check related vulnerabilities.

Transaction-based Tools *TokenScope* (Chen et al. 2019a) is the first tool that uses transaction histories to detect inconsistent behaviors of ERC20 Tokens. By using the stored Ethereum transaction records, *TokenScope* identifies three key information of contract bytecode, i.e., core data structures, standard interfaces, and standard events. It then compares the key information with the standard to find any inconsistent tokens.

Source Code Level Static Analysis Detecting vulnerabilities through bytecode is not easy as EVM removes some key information while compiles source code to bytecode. *SmartCheck* (Tikhomirov et al. 2018) takes smart contract source code as input, and converts the code to the AST (abstract syntax tree) (AST 2020). Based on the AST, *SmartCheck* uses several patterns to detect 21 kinds of smart contract issues.

5.2.2 Formal Verification

Formal verification is a method that uses formal methods of mathematics to prove or disprove the correctness of a system (Drechsler and et al 2004). This method usually uses a formal proof on an abstract mathematical model to make the verification.

Bhargavan et al. (2016) proposed the first formal verification tool for smart contracts based on the F* proof assistant (Swamy et al. 2016), and Amani et al. (2018) presented a tool based on Isabelle/HOL (Nipkow et al. 2002). However, both of these the tools only use incomplete semantics of EVM, which might lead to errors. Thus, Park et al. (2018) use a complete and thoroughly tested formal semantics of EVM to enhance the efficacy of their tool.

Kalra et al. (2018) introduced 11 kinds of vulnerabilities of smart contracts and proposed a tool named *Zeus* to detect seven of them. *Zeus* takes source code as input and translates the Solidity source code to LLVM bytecode (2021). Based on the LLVM bytecode, *Zeus* designs several policy violations and uses a verifier to determine assertion violations.

5.2.3 Fuzzing

Fuzzing for smart contracts is an automated testing technique which uses random, unexpected, or invalid data as the input to the contract. Such input data is expected to lead to detecting some unwanted behaviors, e.g., crashes, failure of some functions, permission errors.

Jiang et al. (2018) proposed the first fuzzing tool named *ContractFuzzer*, which applies fuzzing to detect seven kinds of security issues. *ContractFuzzer* utilizes smart contract ABI (Solidity 2020b) to generate fuzzing inputs. Then, they define test oracles and use static analysis to log smart contracts runtime behaviors. Finally, *ContractFuzzer* analyzes the logs to find security issues. The following works make some optimization. For example, *sFuzz* (Nguyen et al. 2020) can cover more branches to find more security issues. *EthRacer* (Kolluri et al. 2019) can run directly on Ethereum bytecode and without the need of ABI, which enlarges the usage scenario. ReGuard (Liu et al. 2018a) provides a web service for developers to make it is easy to use. EVMFuzz (Fu et al. 2019) designs a differential fuzz testing framework, which supports different programming languages for EVM smart contracts.

5.2.4 Machine Learning

With the development of the Ethereum ecosystem, some developers have used machine learning to help maintain smart contracts. Machine learning related methods need a ground truth to train the model. *S-gram* (Liu et al. 2018b) uses *Oyente* to obtain the ground truth and utilizes a combination of N-gram language modeling and lightweight static semantic labeling to predict potential vulnerabilities. *SmartEmbed* (Gao et al. 2020) uses *SmartCheck* to label the vulnerabilities and utilizes deep-learning to train the model to predict smart contract vulnerabilities. Tann et al. (2018) use *MAIAN* to label the security issues and use LSTM to predict potential issues. Huang (2018) first translate the bytecode into RGB color. Based on a manually labeled dataset, they use a convolutional neural network to train the model and predict the security issues.

5.2.5 Other Approaches

Mavridou and Laszka (2018) proposed a tool, named *FSolidM*, to automatically generate smart contracts. They claim that the generated contracts are bug-free and can reduce development efforts. *FSolidM* regards smart contracts as finite state machines (FSMs). Based on FSMs, they provide a set of plugins that contain common contract design patterns and a graphical interface. Developers can add plugins to the contracts to improve security and functionalities.

Wu et al. (Li et al. 2019) use mutation testing to enhance the security of smart contracts. Mutation testing is a type of white-box software testing technique that changes some statements of the code and check if the test cases can find some errors. This method is based on well-defined mutation operators, and the mutation operators only make minor changes to the programs. Wu et al. designed 15 mutation operators, e.g., variable units, keywords, and use them to find bugs on smart contracts.

5.3 Online Checking Methods

Online checking methods can help smart contract developers defend their contracts against attacks even after they have been deployed. Table 7 introduces seven publications that use online checking methods to help maintain smart contracts. However, most of the online checking methods cannot be used directly and need to be merged into the EVM if an EIP[3] (2020) adopts any of those in a new version.

Ayoade et al. (Ayoade et al. 2019) proposed a method that can automatically detect vulnerable EVM bytecode segments and uses a guarded bytecode segment to replace it. Their

---

[3] Ethereum Improvement Proposals (EIPs) describe standards for the Ethereum platform, including core protocol specifications, client APIs, and contract standards.

Table 7 Literatures of online checking methods

Methodology	Name of Publications	Years
Bytecode Rewriting	Smart Contract Defense through Bytecode Rewriting (Ayoade et al. 2019)	2019
Bytecode Rewriting	Monitoring smart contracts: ContractLarva and open challenges beyond (Azzopardi et al. 2018)	2018
Input Detection	Town Crier: An Authenticated Data Feed for Smart Contracts (Zhang et al. 2016)	2016
Input Detection	FSFC: An input filter-based secure framework for smart contract (Wang et al. 2020)	2020
Transactions Detection	ÆGIS: Smart Shielding of Smart Contracts (Ferreira et al. 2019)	2019
Transactions Detection	VULTRON: Catching Vulnerable Smart Contracts Once and for All (Wang et al. 2019a)	2019
State Detection	Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks (Rodler et al. 2018)	2018
Intrusion Detection	ContractGuard: Defend Ethereum Smart Contracts with Embedded Intrusion Detection (Wang et al. 2019b)	2019

tool is based on predefined policy rules and can only support a limited number of simple rules. Similarly, *ContractLarva* (Azzopardi et al. 2018) insert protection code into the source code of smart contracts. This updated bytecode can defend against related attacks.

TownCrier (Zhang et al. 2016) and *FSFC* (Wang et al. 2020) provide approaches to detect malicious input to protect smart contracts. *TownCrier* can be regarded as a bridge between the smart contracts and front-end programs, e.g., websites. When a frond-end program sends transactions to smart contracts, *TownCrier* uses a combination of Software Guard Extensions (Costan and Devadas 2016) and Intel's recently released trusted hardware capability (Intel 2015) to check whether the input data can be trusted. *FSFC* is a filter-based security framework for smart contracts. It uses several firewall rules and uses a monitor to identify malicious input.

ÆGIS (Ferreira et al. 2019) and *VULTRON* (Wang et al. 2019a) detect and reverse malicious transactions to protect smart contracts. *ÆGIS* uses predefined patters to identify malicious transactions. *VULTRON* compares the actual transferred Ethers and the normal transfered Ethers to find malicious transactions.

Sereum (Rodler et al. 2018) monitors state updates of smart contracts, such as changes to storage variables, to detect re-entrancy attacks. There are two components of *Sereum*, i.e., a taint engine and an attack detector. *Sereum* focuses on conditional jumps and the data that influences the conditional jumps. The taint engine is used to detect the change of state update, which loads to conditional jumps. When a re-entrancy attack happens, the state will be updated multiple times. Once the attack detector detects such malicious behaviors, the transaction will be reversed.

ContractGuard (Wang et al. 2019b) is the first intrusion detection system for smart contracts against attacks. It monitors the network for abnormal behaviors. To detect abnormal behaviors, *ContractGuard* deploys smart contracts on a testbed and trains a model. When malicious activities are detected, *ContractGuard* will reverse the transactions to recover the contract states and raise an alarm to the contract owner.

5.4 Other Methods

Colombo et al. (2018) introduced a specification-driven method that uses the *DELEGATECALL* instruction to upgrade smart contracts when unwanted behaviors are detected. To detect unwanted behaviors, they predefined several checkpoints for smart contracts. The checkpoints monitor the important state of smart contracts, e.g., its balance. When an unexpected behavior is detected, the checkpoints will revert the transactions to ensure the safety of the contracts. Finally, developers are required to upgrade contracts by using the *DELEGATECALL* instruction.

Marino and Juels (2016) defined several standards for smart contracts and suggested developers add a *Selfdestruct* function in the contracts. When the contract is attacked, the developers can undo the contracts. A similar suggestion is given by Chen et al. (2020b). They suggest developers add an interrupter in the contracts. Interrupter is a mechanism to stop the contract when unwanted behaviors are detected, and *Selfdestruct* function is an easy way to stop the contract.

6 Threats To Validity

6.1 Internal Validity

In this paper, we answered two research questions by performing a literature review. Most of the papers (74.05%) are published between 2017 to 2019, and their findings and studies may be outdated as the Ethereum ecosystem is fast-evolving. For example, Solidity, the most popular programming language for smart contracts, has 80 versions from Jan. 2016 to Jun. 2020 (Solidity 2020a). Thus, it is likely that some findings and results in the publications are out-of-date. To reduce this threat, we used an online survey to collect the opinion from many real-world smart contract developers. We compared our literature review findings with the feedback from developers to help ensure the overall validity of our findings.

It is possible that the respondents to our survey may provide some dishonest or unprofessional answers. To reduce this influence, we first informed developers that we will not collect personal information when sending the invitation emails. The survey is anonymous and we cannot trace their information if they do not leave their email address. All questions are optional, which means developers can choose to answer a part of the questions. According to Ong and Weiss (2000) work, confidentiality and anonymity are useful to obtain un-biased data from survey respondents.

To collect more responses, we translated our survey into a Chinese version to address the language barrier and as Google cannot be visited in China. There might be inconsistency between the Chinese and English versions of our surveys. Besides, all the respondents are written in Chinese, which needs to translate to English when analyzing the data. This process also might lead to some errors. To reduce this risk, two Chinese authors with good English skills read the survey and responded several times to ensure the correctness of the translation.

6.2 External Validity

We collected responses to our survey by sending emails to Github developers. However, we might have missed some other developers who might have different opinions. Fortunately, the survey results show that the respondents to our survey have a wide variety of backgrounds in terms of experience in developing smart contracts, job roles, and open source projects they contribute to. Thus, the diversity of backgrounds help us to trust the survey results and can reflect real-world situations of Ethereum smart contract development.

In the future, new functionalities will be added to Ethereum and Solidity. They might also be updated to help better address some smart contract maintenance issues. Thus, some findings and results in this paper might be out-of-date in the future. This is an inevitable trend for smart contract related empirical studies. While the methods we have identified are still working, our findings can help developers and researchers.

7 Discussion

In this section, we discuss some future research directions and give suggestions for both developers and researchers according to our RQ1 and RQ2 findings presented in Sections 4 and 5.

7.1 Improving the Smart Contract Ecosystem

DApp Store and Comment System Although there are some DApp stores for smart contracts, none of them have a smart contract verification system. They neither reject cloned contracts, nor have a rating system. As we discussed in RQ1, many copy-paste vulnerabilities are prevalent in the Ethereum blockchain's deployed smart contracts. There are also many useless smart contracts i.e. "dead" contracts in Ethereum. These contracts are the noisy data on the blockchain and increase the difficulty of finding useful smart contracts. According to our survey, 139 (84.24%, Q14) developers have positive opinions about the need for a DApp store like the Android Google Play Store. Such a DApp store could regulate the behaviors of smart contracts. For example, rejecting copied contracts, rating useful contracts, giving various classifications for contracts. Thus, developers could more easily find high quality contracts for reference or for use as callee contracts. A review system would allow smart contract users to submit reviews when they find bugs or suggest features that need to be improved. Such comments can help developers better maintain their contracts. It could also be a valuable research dataset. Based on such a dataset, many traditional MSR methods can be applied to help improve and maintain smart contracts. For example, as we introduced in the previous section, there are five machine learning-based methods to help maintain smart contracts. However, four of them use other tools to label the ground truth, and there are many false positives / negatives of the tools were used to label the ground truths. Thus, the performance of these tools is not very good. Real-world produced data, e.g., review comments, could substantially improve the performance of these machine learning tools, just as it has for many traditional software maintenance activities and tools.

Call for High-Quality Standards, Libraries and Reference Code Although Ethereum has had a rapid improvement in its ecosystem, developers still claim there is a lack of standards, libraries, and useful reference code. Currently, most of the standards are published on EIPs (2020), and many teams provide libraries and referee code, e.g., OpenZepplelin Contracts (2020), Smart contract best practice (ConsenSys 2020). However, the number is still small and not enough for the vast Ethereum ecosystem.

More User Friendly Tools In previous sections, we introduced 41 works which can help maintain smart contracts. However, according to our survey, 96 (66.2%, Q8) respondents claim they cannot find useful tools to debug / test / audit their contracts, or such tools are too hard to use or deploy in real-world smart contract development. An important reason for this inconsistency is that most current tools are not easy to use for practitioners. Thus, making these tools easier to deploy and use is an important task for the future. For example, merging some tools into smart contract IDEs, or adding a user interface to the tools.

7.2 Improving Ethereum and Solidity

Merging Cutting-Edge Technologies The previous section introduced eight online checking methods that could improve the security and maintainability of smart contracts after they have been deployed. However, most of these online checking methods cannot be used directly. Specifically, transaction detection methods can revert malicious transactions only if they were merged into the EVM and updated by nodes on Ethereum. Then, a node (miner) could revert malicious transactions instead of broadcasting to the whole Ethereum network. Similar to bytecode rewriting tools, these methods can fix a buggy bytecode snippet after they are deployed. However, this kind of method requires modification of the code stored on the blockchain, which cannot be done directly. To use such a method, there should be a well-thought-out plan to ensure the correctness of smart contracts and the concerns of breaking the immutability (discussed in Section 4.1.1). For example, there could be a DAO (Decentralized Autonomous Organization) responsible for updating code periodically by using the bytecode rewriting tools. When the DAO detects a smart contract needs to modify its bytecode, the DAO should inform the contract users / owners and allow them to vote to decide whether the code should be updated. According to our survey (Q15), 150 (90.9%) respondents hold positive opinions about merging cutting-edge technologies into the EVM and updated by nodes on Ethereum.

Mitigating Scalability Issues The scalability issue is one of the main challenges for smart contract maintenance. Several methods have been proposed to help redesign Ethereum to mitigate this issue. First, the *sharding technology* is a future direction for Ethereum to address the scalability issues. Currently, all the nodes on Ethereum need to process every transaction, which leads to low throughput. By applying sharding to Ethereum, the whole network can be split into several smaller parts, called shards. A subset of the total miner nodes would only process transactions on a certain shard. Thus, it can improve the throughput of Ethereum multiple times. Such sharding technology can also enable a smart contract to be executed by multiple threads. A contract could then be split into several parts and executed by different nodes. Enlarging the maximum stack sizes and reduce the gas cost of the storage can also mitigate the scalability issues. This mechanism aims to reduce the bulky problems of Ethereum, where all the nodes store the whole blockchain data. If the bulky problem is addressed, it is not difficult to make an optimization for stack size, database performance, and price for storage. Bruce (2014) proposed a new data structure named an account tree. The account tree holds the balance of all non-empty addresses, which enables us to remove old transactions. Thus, new nodes do not need to store all transactions and can reduce the total bulk of the blockchain.

Trusted Modification Methods In Section 4.1.1, we introduced four modification methods for smart contracts. Among them, using the *Selfdestruct* function and developing *upgradeable contracts* cost the least. However, these two methods can lead to a major trust concern from the users and other security issues. Previous work (Chen et al. 2020a) introduced a method to reduce the trust and security concern for the usage of the *Selfdestruct* function, which can also be applied to upgradeable contracts. This method suggests that developers should distribute the rights to the users of the contracts. They could vote to decide whether the contracts should be destructed or upgraded. Using consensus protocols, such as PoS (2019) and DPoS (2019) are examples of such voting. For example, if a user invests 100 Ethers to the contract, the user has 100 score to vote. The more Ethers users invest contracts, the more rights they have. When the voting process finished, users who do not agree can transfer their Ethers to other accounts. Also, the delay can reduce the risk of the Ethers locking, as Ethers transferred to the destructed address will be locked forever. During the voting and delaying steps, developers should suspend the function of the contracts to prevent attacks or other unwanted behaviors.

8 Related Work

We review previous key empirical studies on smart contracts, and highlight the difference between our work at the end of the section.

8.1 Survey Based Smart Contract Empirical Studies

Bosu et al. (2019) pre-designed some questions and used an online survey to collect the opinions from developers on Github. Their work aimed to answer who contributes to smart contracts and their motivation for development, what is the difference between smart contract development and traditional software development, the challenges of smart contract development, and what kinds of tools that developers feel they need.

Chakraborty et al. (2018) sent an online survey to 1,604 developers on Github and received 145 responses. Their survey aimed to find the best current software development practices for smart contracts. Their findings suggest that some traditional software engineering practices are still working for blockchain projects. They identified that the smart contract ecosystem is immature and needs more SE methods, resources, and tools.

Chen et al. (2020b) defined 20 contract defects by analyzing posts on Stack Exchange. They divided the defects into five categories, i.e., security, availability, performance, maintainability, and reusability defects. They claimed that removing these contract defects can improve the robustness and enhance development efficiency. To validate whether real-world developers regard these contracts as harmful, they use an online survey to collect developers' opinions. The results show that all the 20 contract defects are potential harmful to smart contracts.

Novelty and Differences of this work Both our work and Bosu et al.'s work (2019) investigated the challenges of smart contract development. Our work investigated the maintenance-related challenges for post-deployed Ethereum Smart Contract development, which is much more comprehensive than Bosu et al.'s work. The only similarity between the two works is that we both reported a lack of tools as one of the challenges for smart contract development / maintenance. Our work has a deeper analysis for the reasons why the academia proposed many tools with excellent performance but the smart contract developers also feel they lack tools to check smart contract security. (See Section 4.2.1).

There is a big difference between Chakraborty et al.'s work (2018) and our work. Both works used surveys to collect developers' opinions; their work used surveys to find the answers of pre-defined research questions, while our survey aimed to validate the findings that we collected from our literature review. Their work aims to understand the software development practices of smart contract projects. For example, how smart contract developers test their code, e.g., using unit testing or code review; what's the requirement during the development, e.g., the needs for community discussion, while our work focuses on the challenges during smart contract maintenance.

Chen et al.'s work (2020b) reported detailed patterns / code that are harmful for smart contract development / maintenance, while our work is at a higher level that reports the challenges of smart contract maintenance instead of specific code patterns.

8.2 Literature Review Based Smart Contract Empirical Studies.

Conoscenti et al. (2016) proposed an empirical study to help developers understand how to use smart contracts and blockchain technology to build a decentralized and private-by-design IoT system. To obtain key related information they conducted a systematic literature review based on 18 publications. Their work introduced several use cases of blockchain in the IoT domain and the factors affect integrity, anonymity, and adaptability of blockchain technology.

Udokwu et al. (2018) selected 48 publications from 496 papers. Based on the selected papers, they described the key current usages of smart contract technology and challenges in adopting smart contracts to other applications. Their analysis showed that the most popular applications of smart contracts are supply chain management, finance, healthcare, information security, smart city, and IoT. They also identified 18 limitations of blockchain technology that affects the adoption of smart contracts for other applications.

Macrinici et al. (2018) pre-defined seven research questions and selected 64 publications to find answers. Their results show that the most popular topic in smart contract research is offering solutions to address related problems, e.g, developing tools, proof-of-concepts, and designing protocols. They also summarized 16 smart contract related problems and divided them into three categories, i.e., blockchain mechanism, contract source code, and EVM problems.

Novelty and Differences of this work Our work is the most comprehensive literature review based on smart contract empirical study (our 131 publications v.s. Conoscenti et al. 's 18 publications v.s. Udokwu et al. 's 48 publications v.s. Macrinici et al.'s 64 publications). There might be a gap between academia and industry knowledge, usage, practices, and desired outcomes. Thus, findings based on previous published literature might be out-of-date. Ours is the only work that uses an online survey to validate our findings from the literature review. Also, the fast-growing ecosystem of Ethereum can make even recent findings quickly out of date. Thus, the findings based exclusively on literature reviews might not be reliable. For example, Zou et al. (2019) mention that Solidity lacks the support of *try-catch*, which increases the difficulty of the development. However, Solidity added this support from version 0.6.0 (Solidity 2020b). Also, our work is the only one that focuses on smart contract maintenance issues, while the mentioned three works focus on IoT, adopting smart contracts to other applications, and the most popular topic in smart contract research, respectively.

8.3 Security Related Smart Contract Empirical Studies.

Li et al. (2017) reviewed security issues for the blockchain systems from 2015 to 2017. They classified these issues into nine categories and introduced the related causes. For example, one of the categories is the “51% vulnerability” and the cause is the consensus mechanism. To help developers understand such attacks better, they also gave example real attacks as case studies and analyzed the vulnerabilities utilized by the attackers.

Bartoletti et al. (2020) found that the infamous Ponzi scheme has migrated to Ethereum. Misbehaving developers use smart contracts to design a Ponzi scheme to make money. Bartoletti et al. manually checked real-world smart contracts and summarized four kinds of Ponzi smart contracts, i.e., tree-shaped, chain-shaped, waterfall, handover Ponzi scheme. To help further research on Ponzi scheme detection, they manually labeled a dataset that contains 184 schemes. A follow-up work (Weili et al. 2018; Chen et al. 2019b) used this dataset to design machine learning methods to detect Ponzi smart contracts.

Delmolino et al. (2016) are the lectures of a university who teach smart contract programming. They documented the pitfalls of smart contracts according to their teaching experiences. The pitfalls include errors in encoding state machines, failing to use cryptography, misaligned incentives, and Ethereum-specific mistakes.

Atzei et al. (2017) studied attacks on smart contracts on Ethereum between 2015 to 2017, and provided a classification of programming pitfalls which might lead to the security issues of smart contracts. Their work introduced six vulnerabilities in the Solidity level, three vulnerabilities in the EVM level, and three vulnerabilities in the blockchain level. For most of the vulnerabilities introduced in the paper, a detailed introduction, code examples, and attack examples are given to help readers better understand.

Novelty and Differences of this work The motivation between our work and these security-related smart contract empirical studies have big differences. Our work aims to highlight the maintenance-related concerns for post-deployed Ethereum smart contract development, and security concerns is only a very small part of our work. These works focus on only security issues with more detailed information, e.g., the specific code patterns and attack examples.

8.4 Other Smart Contract Empirical Studies

Zheng et al. (2020) described the challenges of developing smart contracts in the whole life cycle, including creation challenges, deployment challenges, execution challenges, and completion challenges. Their work not only focused on the Ethereum platform, but is also more narrow in other ways. Thus, they also analysed some differences between six smart contract platforms. Another work (Zheng et al. 2018) discussed the challenges of the blockchain system, and the opportunities of blockchain technology. For the challenge, they mainly focused on the architecture of blockchain and consensus algorithms. For the opportunities, they introduced the applications of blockchain, e.g., IoT, Finance. Reyna et al. (2018) investigated the challenges of applying blockchain technology to the IoT to increase the security and reliability. Mohanta et al. (2018) introduced seven uses cases for smart contracts, including supply chain, IoT, and healthcare systems. Many empirical studies also focus on the performance of smart contract tools (Perez and Livshits 2019; Parizi et al. 2018a), programming languages (Harz and Knottenbelt 2018; Schrans et al. 2018; Parizi et al. 2018b), ecosystem (Kiffer et al. 2018; He et al. 2019; Hegedűs 2019), permissions (Vukolić 2017), design patterns (Bartoletti and Pompianu 2017), life cycle (Di and

Salzer 2019), call relations (Bistarelli et al. 2019). Durieux et al. (2020) presented an empirical study of 9 state-of-art smart contract vulnerability analysis tools. To evaluate these tools, they use two datasets, i.e., a small-scale dataset consists of 69 vulnerable smart contracts and a large-scale dataset with all verified smart contracts (47, 518 contracts) on Etherscan. They found that only 42% of vulnerable smart contracts in small-scale dataset can be detected by all the 9 tools. About 97% of smart contracts are labeled as vulnerable by at least one tool. According to their analysis result, Mythril (Software C 2019) has the highest accuracy (27%) in detecting smart contract vulnerabilities.

Novelty and Differences of this work In this paper, we summarized the key maintenance issues and current maintenance methods for smart contracts as evidence from our literature review, which has a different topic with the smart contract empirical studies mentioned above. Ours is also the only work to date that has conducted a literature review to collect maintenance issues of smart contracts and used an online survey to validate these findings with practitioners.

9 Conclusion

In this paper, we conducted the first empirical study on the Ethereum smart contract maintenance issues. We performed a systematic literature review to obtain related information and used an online survey to validate our findings with practitioners. Our study contains two research questions. In RQ1, we identified 9 kinds of issues related to corrective, adaptive, perfective, and preventive maintenance of smart contacts, and another 4 issues corresponding to the overall maintenance process for smart contracts. In RQ2, we summarized current maintenance methods used for smart contracts from 41 publications and divided them into three categories, offline checking methods, online checking methods, and other methods. We also highlighted two kinds of future research directions and discussed some suggestions for both smart contract developers and researchers according to the previous RQ answers and our survey results.

References

Amani S, Bégel M, Bortin M, Staples M (2018) Towards verifying Ethereum smart contract bytecode in Isabelle/HOL

AST (2020) Abstract syntax tree. https://en.wikipedia.org/wiki/Abstract_syntax_tree

Atzei N, Bartoletti M, Cimoli T (2017) A survey of attacks on Ethereum smart contracts (sok). In: International conference on principles of security and trust. Springer, pp 164–186

Ayoade G, Bauman E, Khan L, Hamlen K (2019) Smart contract defense through bytecode rewriting, IEEE

Azzopardi S, Ellul J, Pace GJ (2018) Monitoring smart contracts: Contractlarva and open challenges beyond. In: International conference on runtime verification. Springer, pp 113–137

Bartoletti M, Pompianu L (2017) An empirical analysis of smart contracts: platforms, applications, and design patterns. In: International conference on financial cryptography and data security. Springer, pp 494–509

Bartoletti M, Carta S, Cimoli T, Saia R (2020) Dissecting Ponzi schemes on Ethereum: identification, analysis, and impact. Futur. Gener. Comput. Syst. 102:259–277

Beck K, Beedle M, Van BennekumA, Cockburn A, Cunningham W, Fowler M, Grenning J, Highsmith J, Hunt A, Jeffries R et al (2001) Manifesto for agile software development

Bez M, Fornari G, Vardanega T (2019) The scalability challenge of Ethereum: An initial quantitative analysis. In: 2019 IEEE international conference on service-oriented system engineering (SOSE). IEEE, pp 167–176

Bhargavan K, Delignat-Lavaud A, Fournet C, Gollamudi A, Gonthier G, Kobeissi N, Kulatova N, Rastogi A, Sibut-Pinote T, Swamy N et al (2016) Formal verification of smart contracts: Short paper. In: Proceedings of the 2016 ACM workshop on programming languages and analysis for security. pp 91–96

Bigi G, Bracciali A, Meacci G, Tuosto E (2015) Validation of decentralised smart contracts through game theory and formal methods. In: Programming languages with applications to biology and security. Springer, pp 142–161

Bistarelli S, Mazzante G, Micheletti M, Mostarda L, Tiezzi F (2019) Analysis of Ethereum smart contracts and opcodes. In: International conference on advanced information networking and applications. Springer, pp 546–558

Blockchain (2019) What is blockchain. https://en.wikipedia.org/wiki/Blockchain

Boehm B, Basili VR (2005) Software defect reduction top 10 list. Found Empir Softw Eng 426(37):426–431

Boehm BW (1988) A spiral model of software development and enhancement. Computer 21(5):61–72

Bosu A, Iqbal A, Shahriyar R, Chakraborty P (2019) Understanding the motivations, challenges and needs of Blockchain software developers: a survey. Empir. Softw. Eng. 24(4):2636–2673

Bourque P, Fairley RE et al (2014) Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0. IEEE Computer Society Press, Washington

Brent L, Jurisevic A, Kong M, Liu E, Gauthier F, Gramoli V, Holz R, Scholz B (2018) Vandal: A scalable security analysis framework for smart contracts. arXiv:1809.03981

Bruce J (2014) The mini-blockchain scheme. White paper

Chakraborty P, Shahriyar R, Iqbal A, Bosu A (2018) Understanding the software development practices of blockchain projects: a survey. In: Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement. pp 1–10

Chang J, Gao B, Xiao H, Sun J, Cai Y, Yang Z (2019) sCompile: Critical path identification and analysis for smart contracts. In: International conference on formal engineering methods. Springer, pp 286–304

Chen J, Xia X, David L, John G (2020a) Why do smart contracts self-destruct? investigating the selfdestruct function on ethereum. arXiv:2005.07908

Chen J, Xia X, Lo D, Grundy J, Luo X, Chen T (2020b) Defining smart contract defects on ethereum. IEEE Trans Softw Eng

Chen T, Li Z, Zhou H, Chen J, Luo X, Li X, Zhang X (2018a) Towards saving money in using smart contracts. In: 2018 IEEE/ACM 40th International conference on software engineering: new ideas and emerging technologies results (ICSE-NIER). IEEE, pp 81–84

Chen T, Zhang Y, Li Z, Luo X, Wang T, Cao R, Xiao X, Zhang X (2019a) TokenScope: automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum. In: Proceedings of the 2019 ACM SIGSAC conference on computer and communications security. pp 1503–1520

Chen T, Feng Y, Li Z, Zhou H, Luo X, Li X, Xiao X, Chen J, Zhang X (2020c) GasChecker: scalable analysis for discovering gas-inefficient smart contracts. IEEE Trans Emerg Topics Comput

Chen W, Ma M, Ye Y, Zheng Z, Zhou Y (2018b) IoT service based on jointcloud blockchain: The case study of smart traveling. In: 2018 IEEE symposium on service-oriented system engineering (SOSE), IEEE, pp 216-221

Chen W, Zheng Z, Ngai ECH, Zheng P, Zhou Y (2019b) Exploiting blockchain data to detect smart Ponzi schemes on Ethereum. IEEE Access 7:37575–37586

Colombo C, Ellul J, Pace GJ (2018) Contracts over smart contracts: Recovering from violations dynamically. In: International symposium on leveraging applications of formal methods. Springer, pp 300–315

Conoscenti M, Vetro A, De Martin JC (2016) Blockchain for the internet of things: a systematic literature review. In: 2016 IEEE/ACS 13th International conference of computer systems and applications (AICCSA). IEEE, pp 1–6

ConsenSys (2020) Smart contract best practices. https://github.com/ConsenSys/smart-contract-best-practices

Costan V, Devadas S (2016) Intel SGX explained. IACR Cryptology ePrint Archive 2016(086):1–118

Cryptokitties (2019) https://www.cryptokitties.co/

DApp (2019) Decentralized application. https://en.wikipedia.org/wiki/Decentralized_application

Delmolino K, Arnett M, Kosba A, Miller A, Shi E (2016) Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In: International conference on financial cryptography and data security. Springer, pp 79–94

Di AngeloM, Salzer G (2019) Mayflies, breeders, and busy bees in Ethereum: smart contracts over time. In: Proceedings of the third ACM workshop on blockchains, cryptocurrencies and contracts. pp 1–10

DPoS (2019) Delegated proof of stake. https://lisk.io/academy/blockchain-basics/how-does-blockchain-work/delegated-proof-of-stake

Drechsler R et al (2004) Advanced formal verification, vol 122. Springer, Berlin

Durieux T, Ferreira JF, Abreu R, Cruz P (2020) Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In: Proceedings of the ACM/IEEE 42nd International conference on software engineering. pp 530–541

Efanov D, Roschin P (2018) The all-pervasiveness of the blockchain technology. Procedia Comput Sci 123:116–121

EIP (2020) The ethereum improvement proposal repository. https://github.com/Ethereum/EIPs

EIP150 (2020) EIP-150. https://blog.Ethereum.org/2016/10/13/announcement-imminent-hard-fork-eip150-gas-cost-changes/

ETC (2018) Ethereum classic. https://Ethereumclassic.github.io/

Ethereum (2019) Ethereum.org. https://www.Ethereum.org/

EtherScan (2018) https://etherscan.io/

Ethstates (2020) Ethereum network status. https://ethstats.net/

Fabian V, Vitalik B (2018) ERC20. https://github.com/Ethereum/EIPs/blob/master/EIPS/eip-20.md

Ferreira TorresC, Baden M, Norvill R, Jonker H (2019) ÆGIS: smart shielding of smart contracts. In: Proceedings of the 2019 ACM SIGSAC conference on computer and communications security. pp 2589–2591

Fontana FA, Mäntylä MV, Zanoni M, Marino A (2016) Comparing and experimenting machine learning techniques for code smell detection. Empir. Softw. Eng. 21(3):1143–1191

Fowler M, Beck K (1999) Refactoring: improving the design of existing code. Addison-Wesley Professional, Boston

Frowis M, Bohme R (2017) In code we trust? Measuring the control flow immutability of all smart contracts deployed on Ethereum. LNCS 10436:357–372

Fu Y, Ren M, Ma F, Jiang Y, Shi H, Sun J (2019) Evmfuzz: Differential fuzz testing of Ethereum virtual machine. arXiv:1903.08483

Gao Z, Jiang L, Xia X, Lo D, Grundy J (2020) Checking smart contracts with structural code embedding. IEEE Trans Softw Eng

GasStation (2020) ETH gas station. https://ethgasstation.info/

Giger E, D'Ambros M, Pinzger M, Gall HC (2012) Method-level bug prediction. In: Proceedings of the 2012 ACM-IEEE International symposium on empirical software engineering and measurement, IEEE, pp 171–180

Grano G, Di Sorbo A, Mercaldo F, Visaggio CA, Canfora G, Panichella S (2017) Android apps and user feedback: a dataset for software evolution and quality improvement. In: Proceedings of the 2nd ACM SIGSOFT international workshop on app market analytics. pp 8–11

Grech N, Kong M, Jurisevic A, Brent L, Scholz B, Smaragdakis Y (2018) Madmax: Surviving out-of-gas conditions in Ethereum smart contracts. Proceedings of the ACM on programming languages 2(OOPSLA):1–27

Grech N, Brent L, Scholz B, Smaragdakis Y (2019) Gigahorse: thorough, declarative decompilation of smart contracts, IEEE

Harz D, Knottenbelt W (2018) Towards safer smart contracts: A survey of languages and verification methods. arXiv:1809.0980

He N, Wu L, Wang H, Guo Y, Jiang X (2019) Characterizing code clones in the Ethereum smart contract ecosystem. arXiv:1905.00272

Hegedűs P (2019) Towards analyzing the complexity landscape of solidity based Ethereum smart contracts. Technologies 7(1):6

Hildenbrandt E, Saxena M, Rodrigues N, Zhu X, Daian P, Guth D, Moore B, Park D, Zhang Y, Stefanescu A et al (2018) Kevm: A complete formal semantics of the Ethereum virtual machine, IEEE

Huang R, Sun W, Xu Y, Chen H (2019) Towey D, A survey on adaptive random testing. IEEE Trans Softw Eng, Xia X

Huang THD (2018) Hunting the Ethereum smart contract: Color-inspired inspection of potential attacks. arXiv:1807.01868

Intel (2015) Intel corporation. Intelà software guard extensions evaluation SDK user's guide for windows* OS. https://software.intel.com/sites/products/sgx-sdk-users-guide-windows

ISO/IEC (2006) ISO/IEC/IEEE international standard for software engineering - software life cycle processes - maintenance. ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998), pp 1–58

Jiang B, Liu Y, Chan W (2018) Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering. pp 259–269

Kalra S, Goel S, Dhawan M, Sharma S (2018) ZEUS: analyzing safety of smart contracts. In: The network and distributed system security symposium (NDSS). pp 1–12

Kiffer L, Levin D, Mislove A (2018) Analyzing ethereum's contract topology. In: Proceedings of the internet measurement conference, vol 2018, pp 494–499

Kim M, Zimmermann T, DeLine R, Begel A (2016) The emerging role of data scientists on software development teams, IEEE

Kitchenham B, Charters S (2007) Guidelines for performing systematic literature reviews in software engineering. EBSE Technical Report

Kitchenham BA, Pfleeger SL (2008) Personal opinion surveys. In: Guide to advanced empirical software engineering. Springer, pp 63–92

Kolluri A, Nikolic I, Sergey I, Hobor A, Saxena P (2019) Exploiting the laws of order in smart contracts. In: Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis. pp 363–373

Krupp J, Rossow C (2018) Teether: Gnawing at Ethereum to automatically exploit smart contracts. In: 27th USENIX security symposium. pp 1317–1333

Li X, Jiang P, Chen T, Luo X, Wen Q (2017), A survey on the security of blockchain systems. Future Gener Comput Syst

Li Z, Wu H, Xu J, Wang X, Zhang L, Chen Z (2019) MuSC: A tool for mutation testing of Ethereum smart contract. In: 2019 34th IEEE/ACM International conference on automated software engineering (ASE). IEEE, pp 1198–1201

Liu C, Liu H, Cao Z, Chen Z, Chen B, Roscoe B (2018a) Reguard: finding reentrancy bugs in smart contracts. In: 2018 IEEE/ACM 40th international conference on software engineering: companion (ICSE-Companion). IEEE, pp 65–68

Liu H, Liu C, Zhao W, Jiang Y, Sun J (2018b) S-gram: towards semantic-aware security auditing for Ethereum smart contracts. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering. pp 814–819

LLVM (2021) The llvm project. https://llvm.org/

Lohr M, Peldszus S (2020) Maintenance of long-living smart contracts. In: CEUR workshop proceedings

Luu L, Chu DH, Olickel H, Saxena P, Hobor A (2016) Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. ACM, pp 254–269

Maalej W, Nabil H (2015) Bug report, feature request, or simply praise? on automatically classifying app reviews, IEEE

Macrinici D, Cartofeanu C, Gao S (2018) Smart contract applications within blockchain technology: A systematic mapping study. Telematics Inform. 35(8):2337–2354

Marino B, Juels A (2016) Setting standards for altering and undoing smart contracts. In: International symposium on rules and rule markup languages for the semantic web. Springer, pp 151–166

Marketcap (2020) https://www.ccn.com/marketcap/

Mavridou A, Laszka A (2018) Designing secure Ethereum smart contracts: A finite state machine based approach. In: International conference on financial cryptography and data security. Springer, pp 523–540

Mohanta BK, Panda SS, Jena D (2018) An overview of smart contract and use cases in blockchain technology. In: 2018 9th international conference on computing, communication and networking technologies (ICCCNT). IEEE, pp 1–4

Mossberg M, Manzano F, Hennenfent E, Groce A, Grieco G, Feist J, Brunson T, Dinaburg A (2019) Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In: 2019 34th IEEE/ACM international conference on automated software engineering (ASE). IEEE, pp 1186–1189

Mudge N (2021) Eip2535: diamond standard. https://eips.ethereum.org/EIPS/eip-2535

Mushegian N (2020) EIP-140. https://github.com/Ethereum/EIPs/issues/140

Nakamoto S (2008) Bitcoin: A peer-to-peer electronic cash system

Nguyen TD, Pham LH, Sun J, Lin Y, Minh QT (2020) sFuzz: an efficient adaptive fuzzer for solidity smart contracts. ICSE

Nikolić I, Kolluri A, Sergey I, Saxena P, Hobor A (2018) Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th annual computer security applications conference. pp 653–663

Nipkow T, Paulson LC, Wenzel M (2002) Isabelle/HOL: a proof assistant for higher-order logic, vol 2283. Springer Science & Business Media, Berlin

Norvill R, Pontiveros BBF, State R, Awan I, Cullen A (2017) Automated labeling of unknown contracts in Ethereum. In: 2017 26th international conference on computer communication and networks (ICCCN). IEEE, pp 1–6

Ong AD, Weiss DJ (2000) The impact of anonymity on responses to sensitive questions 1. J. Appl. Soc. Psychol. 30(8):1691–1708

OpenZeppelin (2020) OpenZeppelin upgradeable smart contract document. https://docs.openzeppelin.com/learn/upgrading-smart-contracts

Openzepplelin (2020) Openzepplelin contracts. https://github.com/OpenZeppelin/openzeppelin-contracts

Parizi RM, Dehghantanha A, Choo KKR, Singh A (2018a) Empirical vulnerability analysis of automated smart contracts security testing on blockchains. In: Proceedings of the 28th annual international conference on computer science and software engineering. IBM Corp., pp 103–113

Parizi RM, Dehghantanha A, et al. (2018b) Smart contract programming languages on blockchains: An empirical evaluation of usability and security. In: International conference on blockchain. Springer, pp 75–91

Park D, Zhang Y, Saxena M, Daian P, Roşu G (2018) A formal verification tool for Ethereum VM bytecode. In: Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering. pp 912–915

Perez D, Livshits B (2019) Smart contract vulnerabilities: Does anyone care? arXiv:1902.06710

Pigoski TM (1996) Practical software maintenance: best practices for managing your software investment. Wiley, Hoboken

PoS (2019) Proof of stake. https://en.wikipedia.org/wiki/Proof_of_stake

Reyna A, Martín C, Chen J, Soler E, Díaz M (2018) On blockchain and its integration with IoT. Challenges and opportunities. Future Gener Comput Syst 88:173–190

Rodler M, Li W, Karame GO, Davi L (2018) Sereum: Protecting existing smart contracts against re-entrancy attacks. arXiv:1812.05934

Schrans F, Eisenbach S, Drossopoulou S (2018) Writing safe smart contracts in Flint. In: Conference companion of the 2nd international conference on art, science, and engineering of programming. pp 218–219

SDHardFork (2020) Spurious dragon hard fork. https://blog.Ethereum.org/2016/11/18/hard-fork-no-4-spurious-dragon/

Segura S, Fraser G, Sanchez AB, Ruiz-Cortés A (2016) A survey on metamorphic testing. IEEE Trans Softw Eng 42(9):805–824

Siegel D (2018) Understanding the DAO attack. https://www.coindesk.com/understanding-dao-hack-journalists/

Software C (2019) Mythril: Security analysis tool for evm bytecode. https://github.com/ConsenSys/mythril

Solidity (2020) Releases of solidity. https://github.com/Ethereum/solidity/releases

Solidity (2020) Solidity document. http://solidity.readthedocs.io

Spencer D (2009) Card sorting: Designing usable categories, Rosenfeld Media, New York

Swamy N, Hriţcu C, Keller C, Rastogi A, Delignat-Lavaud A, Forest S, Bhargavan K, Fournet C, Strub PY, Kohlweiss M et al (2016) Dependent types and multi-monadic effects in F. In: Proceedings of the 43rd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages. pp 256–270

Tai AT, Alkalai L (1998) On-board maintenance for long-life systems. In: Proceedings. 1998 IEEE workshop on application-specific software engineering and technology. ASSET-98 (Cat. No. 98EX183). IEEE, pp 69–74

Tann A, Han XJ, Gupta SS, Ong YS (2018) Towards safer smart contracts: A sequence learning approach to detecting vulnerabilities. arXiv:1811.06632. pp 1371–1385

Tikhomirov S, Voskresenskaya E, Ivanitskiy I, Takhaviev R, Marchenko E, Alexandrov Y (2018) Smartcheck: Static analysis of Ethereum smart contracts. In: Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain. pp 9–16

Torres CF, Schütte J, State R (2018) Osiris: Hunting for integer bugs in Ethereum smart contracts. In: Proceedings of the 34th Annual computer security applications conference. pp 664–676

Torres CF, Steichen M et al (2019) The art of the scam: Demystifying honeypots in Ethereum smart contracts. In: 28th {USENIX} security symposium ({USENIX} security, vol 19, pp 1591–1607

Tsankov P, Dan A, Drachsler-Cohen D, Gervais A, Buenzli F, Vechev M (2018) Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security. ACM, pp 67–82

Tyagi PK (1989) The effects of appeals, anonymity, and feedback on mail survey response patterns from salespeople. J. Acad. Mark. Sci. 17(3):235–241

Udokwu C, Kormiltsyn A, Thangalimodzi K, Norta A (2018) The state of the art for blockchain-enabled smart-contract applications in the organization. In: 2018 Ivannikov Ispras Open Conference (ISPRAS). IEEE, pp 137-144

Velner Y, Teutsch J, Luu L (2017) Smart contracts make Bitcoin mining pools vulnerable. In: International conference on financial cryptography and data security. Springer, pp 298–316

Vukolić M (2017) Rethinking permissioned blockchains. In: Proceedings of the ACM workshop on blockchain, cryptocurrencies and contracts. pp 3–7

Vyper (2020) Vyper document. https://vyper.readthedocs.io

Wan Z, Xia X, Hassan AE, Lo D, Yin J, Yang X (2018) Perceptions, expectations, and challenges in defect prediction. IEEE Trans Softw Eng

Wan Z, Xia X, Lo D, Murphy GC (2019) How does machine learning change software development practices? IEEE Trans Softw Eng

Wang H, Li Y, Lin SW, Ma L, Liu Y (2019a) Vultron: catching vulnerable smart contracts once and for all. In: 2019 IEEE/ACM 41st International conference on software engineering: new ideas and emerging results (ICSE-NIER). IEEE, pp 1–4

Wang X, He J, Xie Z, Zhao G, Cheung SC (2019b) ContractGuard: Defend ethereum smart contracts with embedded intrusion detection. IEEE Trans Serv Comput

Wang Z, Dai W, Choo KKR, Jin H, Zou D (2020) FSFC: An input filter-based secure framework for smart contract. J Netw Comput Appl :102530

Weili C, Zibin Z, Jiahui C, Edith N, Peilin Z, Yuren Z (2018) Detecting ponzi schemes on ethereum: towards healthier blockchain technology. In: Proceedings of the 2018 world wide web conference on world wide web, international world wide web conferences steering committee, pp 1409–1418

Wood G (2014) Ethereum: A secure decentralised generalised transaction ledger. Project Yellow Paper

Yan M, Xia X, Shihab E, Lo D, Yin J, Yang X (2018) Automating change-level self-admitted technical debt determination. IEEE Trans. Softw. Eng. 45(12):1211–1229

Zhang F, Cecchetti E, Croman K, Juels A, Shi E (2016) Town crier: An authenticated data feed for smart contracts. In: Proceedings of the 2016 aCM sIGSAC conference on computer and communications security. pp 270–282

Zhang T, Chen J, Zhan X, Luo X, Lo D, Jiang H (2019) Where2Change: Change request localization for app reviews. IEEE Trans Softw Eng

Zheng Z, Xie S, Dai HN, Chen X, Wang H (2018) Blockchain challenges and opportunities: A survey. Int J Web Grid Servi 14(4):352–375

Zheng Z, Xie S, Dai HN, Chen W, Chen X, Weng J, Imran M (2020) An overview on smart contracts: Challenges, advances and platforms. Futur. Gener. Comput. Syst. 105:475–491

Zhou E, Hua S, Pi B, Sun J, Nomura Y, Yamashita K, Kurihara H (2018) Security assurance for smart contract, IEEE

Zou W, Lo D, Kochhar PS, Le XBD, Xia X, Feng Y, Chen Z, Xu B (2019) Smart contract development: Challenges and opportunities. IEEE Trans Softw Eng

Jiachi Chen is currently a Ph.D student at the Faculty of Information Technology, Monash University, Australia. Prior to join Monash University, he spent two years at the Hong Kong Polytechnic University and half a year at Zhejiang University in China both as a research assistant. His research interests include mining software repository and smart contract analysis.

Xin Xia is the director of the software engineering application technology lab, Huawei, China. Prior to joining Huawei, he was an ARC DECRA Fellow and a lecturer at Monash University, Australia. Xin received his Ph.D in computer science from Zhejiang University in 2014. To help developers and testers improve their productivity, his current research focuses on mining and analyzing rich data in software repositories to uncover interesting and actionable information.More information at: https://xin-xia.github.io/

David Lo is a ACM Distinguished Member and a Professor of Information Systems at Singapore Management University. He received his PhD in Computer Science from National University of Singapore in 2008. His research interest is in the intersection of software engineering and data science, encompassing socio-technical aspects and analysis of different kinds of software artefacts, with the goal of improving software quality and developer productivity. His work has been published in premier and major conferences and journals in the area of software engineering, AI, and cybersecurity.

John Grundy is Australian Laureate Fellow and Professor of Software Engineering at Monash University, Australia. He has published widely in automated software engineering, domain-specific visual languages, model-driven engineering, software architecture, and empirical software engineering, amoung many other areas. He is Fellow of Automated Software Engineering and Fellow of Engineers Australia.

Xiaohu Yang is a professor at College of Computer Science & Technology, Zhejiang University. He is the Director of Blockchain Research Center and Vice Director of Computer Software Institute at Zhejiang University. His research interests include software engineering, blockchain, and cloud computing. He received the B.S. degree, the M.S. degree and the Ph.D. degree all in computer science at Zhejiang University in 1988, 1990, and 1993, respectively.

Affiliations

Jiachi Chen[1] · **Xin Xia**[1] · **David Lo**[2] · **John Grundy**[1] · **Xiaohu Yang**[3]

Jiachi Chen
jiachi.chen@Monash.edu

David Lo
davidlo@smu.edu.sg

John Grundy
John.Grundy@monash.edu

Xiaohu Yang
yangxh@zju.edu.cn

[1] Faculty of Information Technology, Monash University, Melbourne, Australia
[2] School of Information Systems, Singapore Management University, Singapore, Singapore
[3] College of Computer Science and Technology, Zhejiang University, Hangzhou, China