



MONASH University

***Maintaining Ethereum Smart Contracts
by Finding and Detecting Defects***

Jiachi Chen

Doctor of Philosophy

A thesis submitted for the degree of *Doctor of Philosophy* at

Monash University in 2021

Faculty of Information Technology

Copyright notice

© Jiachi Chen (2021) Except as provided in the Copyright Act 1968, this thesis may not be reproduced in any form without the written permission of the author.

I certify that I have made all reasonable efforts to secure copyright permissions for third-party content included in this thesis and have not knowingly added copyright content to my work without the owner's permission.

Abstract

Ethereum smart contracts are Turing-complete programs deployed on a blockchain. The features of blockchain make smart contracts immutable to change. Like all the other computer code, smart contracts also need maintenance. However, the immutability of smart contracts makes them much harder to be maintained compared to traditional programs.

This thesis aims to help developers maintain smart contracts developed by Solidity on Ethereum. We first conducted an empirical study to investigate what kinds of maintenance issues will smart contract developers encounter, and how do developers maintain smart contracts? We totally found 13 maintenance issues and divided them into five groups. We also found that most developers choose to discard the old contract and redeploy a new contract when they need to patch or add new features on contracts. Before redeploying the new contract to Ethereum, an important step is checking its robustness and security.

A contract defect is an error, flaw or fault in a contract that causes it to produce an incorrect or unexpected result, or to behave in unintended ways. Detecting and removing contract defects help increase software robustness and enhance development efficiency, which are widely used to maintain smart contracts. Most of the existing works first introduced some defects and then developed tools to detect them based on predefined patterns. However, smart contracts ecosystem is fast-evolving, and many new kinds of defects might come out with the new features of the Solidity. Thus, it is important to propose methods that could be used to find contract defects. In this thesis, we introduce two methods that use online posts and historical destructed smart contracts to find new smart contract defects.

For the first method, we crawled all 17,128 posts from StackExchange. Then, we used key-words filtering to select Solidity defects-related posts. Then, we followed the card sorting approach to analyze and categorize the filtered contract defects-related posts. We totally summarized 20 contract defects from five aspects: security, availability, performance, maintainability, and reusability. For the second method, we first collected all the verified (open-sourced) smart contracts from Etherscan, and 756 of them are self-destructed. We then proposed an approach to find the upgrade version of the self-destructed contracts. By analyzing the difference between the self-destructed contracts and their upgrade versions, we found five reasons that led to the death of the contracts; two of them (i.e., Unmatched

ERC20 Token and Limits of Permission) are contract defects that might affect the life span of contracts.

Finally, we proposed *DefectChecker*, a symbolic execution-based tool to detect the defined contract defects. *DefectChecker* can detect contract defects from contracts bytecode. During the symbolic execution, *DefectChecker* generates the control flow graph (CFG) of smart contracts, as well as the “stack event”, and identifies three features, i.e., “Money Call”, “Loop Block”, and “Payable Function”. By using the CFG, stack event, and the three features, *DefectChecker* is extendable by designing different rules to detect contract defects. Our evaluation results show that *DefectChecker* obtains 88.8% of F-score in the whole dataset and only requires 0.15s on average to analyze one smart contract.

Publications during enrolment

Publications included in this thesis

1. **Jiachi Chen**, Xin Xia, David Lo, John Grundy, and Xiaohu Yang, 2021. Maintenance-Related Concerns for Post-deployed Ethereum Smart Contract Development: Issues, Techniques, and Future Challenges. *Empirical Software Engineering (EMSE'21)*, 52 pages, accepted, <https://doi.org/10.1007/s10664-021-10018-0>. (Chapter 3)
2. **Jiachi Chen**, Xin Xia, John Grundy, David Lo, John Grundy, Xiapu Luo, and Ting Chen, 2020. Defining Smart Contract Defects on Ethereum. *IEEE Transactions on Software Engineering (TSE'20)*, 29, 4, Article 26 (October 2020), 17 pages, accepted, <https://doi.org/10.1109/TSE.2020.2989002>. (Chapter 4)
3. **Jiachi Chen**, Xin Xia, David Lo, and John Grundy, 2021. Why Do Smart Contracts Self-Destruct? Investigating the Selfdestruct Function on Ethereum. *ACM Transactions on Software Engineering and Methodology (TOSEM'21)*, 37 pages, accepted. (Chapter 5)
4. **Jiachi Chen**, Xin Xia, John Grundy, David Lo, John Grundy, Xiapu Luo, and Ting Chen, 2021. DEFECTCHECKER: Automated Smart Contract Defect Detection by Analyzing EVM Bytecode. *IEEE Transactions on Software Engineering (TSE'20)*, 19 pages, accepted, <https://doi.org/10.1109/TSE.2021.3054928>. (Chapter 6)

Submitted manuscripts not included in this thesis

5. **Jiachi Chen**, Xin Xia, David Lo, John Grundy, Ting Chen and Zhipeng Gao, 2021. DeFiDefender: Investigating and Detecting Decentralized Financial Traps on Ethereum Smart Contracts. *International Conference on Software Engineering (ICSE'21)*, 11 pages, under review.

Other publications during candidature

6. Zhiyuan Wan, Xin Xia, David Lo, **Jiachi Chen**, Xiapu Luo, and Xiaohu Yang, 2020, Smart Contract Security: a Practitioners' Perspective, *43rd ACM/IEEE International Conference on Software Engineering (ICSE'21)*, 11 pages, accepted, <https://doi.org/10.1109/ICSE43902.2021.00127>.
7. **Jiachi Chen**, Finding Ethereum Smart Contracts Security Issues by Comparing History Versions, 2020, *IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*, 3 pages, accepted, <https://doi.org/10.1145/3324884.3418923>.

Thesis including published works declaration

I hereby declare that this thesis contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

This thesis includes 4 original papers published in peer reviewed journals and 0 unpublished publications. The core theme of the thesis is smart contract maintenance. The ideas, development and writing up of all the papers in the thesis were the principal responsibility of myself, the student, working within the software engineering under the supervision of Prof. John Grundy, Dr. Jiangshan Yu and Dr. Xin Xia.

The inclusion of co-authors reflects the fact that the work came from active collaboration between researchers and acknowledges input into team-based research.

In the case of Chapter 2-5, my contribution to the work involved the following:

Thesis Chapter	Publication Title	Status (published, in press, accepted or returned for revision)	Nature and % of student contribution	Co-author name(s) Nature and % of Co-author's contribution*	Co-author(s), Monash student Y/N*
Chapter 2	Maintenance-Related Concerns for Post-deployed Ethereum Smart Contract Development: Issues, Techniques, and Future Challenges	Published	68%. Concept, All of the experiment, writing the manuscript and revisions for the paper	1) Xin Xia, Concept, 10% and discuss with key challenges 2) David Lo, input into manuscript and discuss with key challenges, 10% 3) John Grundy, input into manuscript and discuss with key challenges, 10% 4) Xiaohu Yang, discuss with key challenges, 2%	None of them
Chapter 3	Defining Smart Contract Defects on Ethereum. IEEE Transactions on Software Engineering	Published	60%. Concept, All of the experiment, writing the manuscript and revisions for the paper	1) Xin Xia, Concept, input into manuscript and discuss with key Challenges, 10% 2) David Lo, input into manuscript and discuss with key challenges, 10%	None of them

					3) John Grundy, input into manuscript and discuss with key challenges, 10% 4) Xiapu Luo, Idea and Concept, 5% 5) Ting Chen, discuss with key Challenges, 5%	
Chapter 4	Why Do Smart Contracts Self-Destruct? Investigating Selfdestruct Function Ethereum	the on	Accepted	70%. Concept, All of the experiment, writing the manuscript and revisions for the paper	1) Xin Xia, Concept, input into manuscript and discuss with key Challenges, 10% 2) David Lo, input into manuscript and discuss with key challenges, 10% 3) John Grundy, input into manuscript and discuss with key challenges, 10%	None of them
Chapter 5	DEFECTCHECKER: Automated Contract Detection Analyzing Bytecode	Smart Defect by EVM	Published	60%. Concept, All of the experiment, writing the manuscript and revisions for the paper	1) Xin Xia, Concept, input into manuscript and discuss with key Challenges, 10% 2) David Lo, input into manuscript and discuss with key challenges, 10% 3) John Grundy, input into manuscript and discuss with key challenges, 10% 4) Xiapu Luo, Idea and Concept, 5% 5) Ting Chen, discuss with key Challenges, 5%	None of them

I have renumbered sections of submitted or published papers in order to generate a consistent presentation within the thesis.

Student name: Jiachi Chen

Jiachi Chen

Student signature:

Date: 16-Nov-2021

The undersigned hereby certify that the above declaration correctly reflects the nature and extent of the student's and co-authors' contributions to this work. In instances where I am not the responsible author I have consulted with the responsible author to agree on the respective contributions of the authors.

Main Supervisor name: John Grundy

Main Supervisor signature:

Date: 17-Nov-2021

Acknowledgements

First, I would like to express my sincere gratitude and appreciation to my supervisors: Prof. John Grundy, Dr. Jiangshan Yu, and Dr. Xin Xia, for their patient and thoughtful guidance thought my PhD study. Although Dr. Xin Xia has left Monash for several months and Dr. Jiangshan Yu became my supervisor in the last year of my PhD study, they still provide lots of help for my research. I feel very lucky to have had Prof. John Grundy as my supervisor, who is a top-notch researcher, kind and patient human being. I still remember the first draft of my paper was full of English grammar and logic errors. Prof. John Grundy helped me a lot to polish the paper and made it finally be accepted by a top journal. Without his help and instruction, I believe that my PhD study would be much harder.

I also appreciate the help from Prof. David Lo, who helped me a lot with all of my publications during the PhD period. His valuable comments and edits contribute a lot to the acceptance of my papers.

Furthermore, I would extend my thanks to all academic staff and administrative staff from the Faculty of Information Technology, and also the Australian government, as this research was supported by an Australian Government Research Training Program (RTP) Scholarship.

Finally, I would express my gratitude to my family. Due to the COVID-19, I was stuck at home for a long time. Their unconditional support keeps me motivated all the time. Especially for my loving wife, who brings a lovely little daughter during my PhD period, make my life much more wonderful.

Contents

1	Introduction	1
1.1	Background	1
1.1.1	Blockchain and Smart Contracts	1
1.1.2	Smart Contract Example with Defects	2
1.1.3	Smart Contracts Maintenance	4
1.2	Research Questions and Thesis Statement	5
1.3	Thesis Contribution	10
1.4	Structure of the thesis	13
2	Maintenance-Related Concerns for Post-deployed Ethereum Smart Contract Development: Issues, Techniques, and Future Challenges	14
2.1	Introduction	15
2.2	Background	19
2.2.1	Ethereum	19
2.2.2	Hard Fork and Soft Fork	20
2.2.3	Smart Contracts	22
2.2.4	The Gas System	22
2.2.5	Upgradeable Smart Contracts	23
2.2.6	Software Development and Maintenance	24
2.2.7	Card Sorting	25

2.3	Methodology	25
2.3.1	Literature Review	26
2.3.1.1	Literature Search	26
2.3.1.2	Literature Selection	27
2.3.1.3	Data Analysis	29
2.3.2	Survey	30
2.3.2.1	Survey Design	30
2.3.2.2	Survey Design Explanation	31
2.3.2.3	Survey Validation	35
2.3.2.4	Recruitment of Respondents	35
2.3.2.5	Data Analysis	36
2.4	RQ1: What are the maintenance issues of smart contracts?	36
2.4.1	Common Maintenance Issues	39
2.4.1.1	No Ideal Deployed Contract Modification Methods	39
2.4.1.2	High Requirement for Security	41
2.4.1.3	Low Readability	42
2.4.1.4	The Lack of Experienced Developers and Researchers.	43
2.4.2	Corrective Maintenance Issues	43
2.4.2.1	The Lack of Mature Tools	44
2.4.2.2	The Lack of Community Support	45
2.4.3	Adaptive Maintenance Issues	46
2.4.3.1	Unpredictable Fork Problems	46
2.4.3.2	Unpredictable Callee Contracts	47
2.4.4	Perfective Maintenance Issues	48
2.4.4.1	The Scalability Issues	48
2.4.4.2	The Difficulty of Handling the Gas System	50

2.4.5	Preventive Maintenance Issues	50
2.4.5.1	The Lack of Advanced SE Approach and Research Data .	51
2.4.5.2	The Lack of High Quality Reference Code	51
2.4.5.3	The Lack of Standards	52
2.5	RQ2: What are the current maintenance methods for smart contracts? . . .	53
2.5.1	Distribution	53
2.5.2	Offline Checking Methods	54
2.5.2.1	Program Analysis	54
2.5.2.2	Formal Verification	57
2.5.2.3	Fuzzing	58
2.5.2.4	Machine Learning	58
2.5.2.5	Other Approaches	59
2.5.3	Online Checking Methods	59
2.5.4	Other Methods	61
2.6	Threats To Validity	61
2.6.1	Internal Validity	61
2.6.2	External Validity	62
2.7	Discussion	63
2.7.1	Improving the Smart Contract Ecosystem	63
2.7.2	Improving Ethereum and Solidity	64
2.8	Related Work	66
2.8.1	Survey Based Smart Contract Empirical Studies	67
2.8.2	Literature Review Based Smart Contract Empirical Studies.	68
2.8.3	Security Related Smart Contract Empirical Studies.	69
2.8.4	Other Smart Contract Empirical Studies.	71
2.9	Conclusion	72

3 Defining Smart Contract Defects on Ethereum	73
3.1 Introduction	74
3.2 Background	77
3.2.1 Smart Contracts - A Decentralized Program	77
3.2.2 Features of Smart Contracts	78
3.2.3 Solidity	79
3.2.4 ERC-20 Token	82
3.3 RQ1: Contract defects in Smart Contracts	83
3.3.1 Motivation	83
3.3.2 Approach	83
3.3.3 Results	88
3.3.3.1 Security Defects	88
3.3.3.2 Availability Defects	95
3.3.3.3 Performance Defects	97
3.3.3.4 Maintainability Defects	98
3.3.3.5 Reusability Defects	100
3.4 RQ2: Practitioners' Perspective	101
3.4.1 Motivation	101
3.4.2 Approach	101
3.4.2.1 Validation Survey	101
3.4.2.2 Survey Design	102
3.4.2.3 Recruitment of Respondents	103
3.4.3 Results	103
3.5 RQ3: Distribution and Impact of Contract Defects	106
3.5.1 Motivation	106
3.5.2 Approach	107

3.5.3	Results	108
3.6	Discussion	112
3.6.1	Implications	112
3.6.2	Challenge in Detection Contract Defects	115
3.6.3	Possible Detection Methods	116
3.6.3.1	Bytecode Level Detection	116
3.6.3.2	Source Code Level Detection	118
3.6.4	Code Smells in Ethereum	120
3.7	Threats to Validity	120
3.7.1	Internal Validity	120
3.7.2	External Validity	121
3.8	Related Work	122
3.9	Conclusion and Future work	124

4	DEFECTCHECKER: Automated Smart Contract Defect Detection by Analyzing EVM Bytecode	126
4.1	Introduction	127
4.2	Background and Motivation	129
4.2.1	Smart Contracts	130
4.2.2	Contract Defects in Smart Contracts	134
4.2.2.1	Definition of Impact Levels	136
4.2.2.2	Examples of Smart Contract Defects	136
4.3	The <i>DefectChecker</i> Approach	142
4.3.1	Design Overview	142
4.3.2	Basic Block Builder	143
4.3.3	Symbolic Execution	144
4.3.4	Feature Detector	148

4.3.4.1	Money Call	148
4.3.4.2	Loop Block	149
4.3.4.3	Payable Function	149
4.3.5	DefectChecker	150
4.3.5.1	Transaction State Dependency	150
4.3.5.2	DoS Under External Influence	151
4.3.5.3	Strict Balance Equality	151
4.3.5.4	Reentrancy	151
4.3.5.5	Nested Call	152
4.3.5.6	Greedy Contract	152
4.3.5.7	Unchecked External Calls	152
4.3.5.8	Block Info Dependency	153
4.4	Evaluation	153
4.4.1	Experimental Setup	153
4.4.2	Dataset	153
4.4.3	Evaluation Methods and Metrics	155
4.4.4	Experimental Results and Analysis	156
4.4.5	Comparison with state-of-the-art tools	161
4.4.6	Threats to Validity	166
4.5	A Large Scale Evaluation	167
4.5.1	Dataset	167
4.5.2	Contract Defects on Ethereum	167
4.5.3	Case Study	169
4.5.4	Threats to Validity	173
4.6	Related Work	174
4.7	Conclusion and Future Work	178

5 Why Do Smart Contracts Self-Destruct? Investigating the Selfdestruct Function on Ethereum	180
5.1 Introduction	181
5.2 Background	185
5.2.1 Smart Contracts	185
5.2.2 Function Modifier	187
5.2.3 Selfdestruct Function	188
5.2.4 The DAO Attack - A Motivation Example of the <i>selfdestruct</i> Function	189
5.2.5 ERC20 Standard	190
5.2.6 Card Sorting	191
5.3 RQ1: Developer's Perspective about selfdestruct Function	192
5.3.1 Motivation	192
5.3.2 Approach	192
5.3.2.1 Validation Survey	192
5.3.2.2 Survey Design	193
5.3.2.3 Recruitment of Respondents	194
5.3.3 Result	194
5.3.3.1 Reasons for including the <i>selfdestruct</i> function	196
5.3.3.2 Reasons for excluding the <i>selfdestruct</i> function	197
5.4 RQ2: Reasons for Self-destruct	198
5.4.1 Motivation	198
5.4.2 Approach	200
5.4.2.1 Stage 1. Data Collection:	201
5.4.2.2 Stage 2. PS Pairs Generation:	203
5.4.2.3 Stage 3. Reason Generation:	205
5.4.3 Reasons for Self-destruct	207

5.4.3.1	Definitions	207
5.4.3.2	Distribution	211
5.5	RQ3: LifeScope: A Self-destruct Issues Detection Tool for Smart Contracts	212
5.5.1	Motivation	212
5.5.2	Approach	212
5.5.2.1	Unmatched ERC20 token	213
5.5.2.2	Limits of Permission	216
5.5.3	Evaluation For Unmatched ERC20 Token	218
5.5.3.1	Dataset	218
5.5.3.2	Result	219
5.5.3.3	Comparison:	219
5.5.4	Evaluation For Limits of Permission	221
5.5.4.1	Dataset	221
5.5.4.2	Evaluation Methods and Metrics	221
5.5.4.3	Result	221
5.5.4.4	Comparison	222
5.6	Discussion	225
5.6.1	Implications	225
5.6.1.1	For Researchers	225
5.6.1.2	For Practitioners	226
5.6.1.3	For Educators	226
5.6.2	Towards More Secure selfdestruct Functions	227
5.6.3	Tokens of Destructed Unmatched ERC20 Contracts	229
5.6.4	Inconsistency	230
5.6.4.1	<i>selfdestruct</i> functions on Etherscan vs. Survey Results. .	230
5.6.4.2	Self-destructed Contracts on Blockchain vs. Etherscan. .	230

5.6.4.3	<i>GasToken</i> Contracts on Blockchain vs. Survey Results. 232
5.6.5	Threats to Validity	234
5.7	Related Work	236
5.8	Conclusions and Future Work	239
6	Conclusion and Future Work	243
6.1	Key Findings and Contributions	243
6.2	Limitations and Future Works	245
6.2.1	Limitations and Future Works for Chapter 2	245
6.2.2	Limitations and Future Works for Chapter 3	246
6.2.3	Limitations and Future Works for Chapter 4	246
6.2.4	Limitations and Future Works for Chapter 5	249
Bibliography		251

List of Figures

2.1	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.	20
2.2	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.	21
2.3	An Example of the upgradeable contract.	23
2.4	Overview of methodology design	26
2.5	The number of papers published between 2014 to 2020.	29
2.6	The steps of open card sorting.	29
2.7	Distribution of Maintenance Methods	53
3.1	Example of a Card	85
3.2	Example of a Card for "Block Info Dependency"	87
4.1	Overview architecture of <i>DefectChecker</i>	143
4.2	Example of Symbolic Execution	145
4.3	The relationship between the number of contract defects and number of Instructions & Cyclomatic Complexity	170
4.4	Transaction Detail of Case Study 1	171
4.5	Transaction Lists of Case Study 2	173

5.1	Overview architecture of finding the self-destructed reasons	200
5.2	A smart contract on Etherscan	202
5.3	Transactions of a Self-destructed Contract	202
5.4	An example considered in our card sort that contain a predecessor contract (0x3b96990a8ef293cdd37c8e1ad3d210a0166f40e1) and successor contract (0xedd7c94fd7b4971b916d15067bc454b9e1bad980).	207
5.5	A real diff example of Limits of permission in our dataset. Predecessor Address: 0xfa1d63b87f40c92d27fb255419c1ea8c49086de; Successor Address: 0x64b09d1a4b01db659fc36b72de0361f2c6c521b1	209
5.6	A real diff example of Unsafe Contract in our dataset. Predecessor Address: 0x8b099bdcfea93faecfac13d0dbc1d08c4e1ec595; Successor Address: 0x17683235257f2089e3e4acc9497f25386a529507	209
5.7	A real diff example of Unmatched ERC20 token in our dataset. Predecessor Address: 0x848217a9569ca64ffba9d000cda05f9d2fa97f5; Successor Address: 0xf42230a7e21375c29648ae9544f7da394e20ead3	209
5.8	A real diff example of Setting Changes in our dataset. Predecessor Address: 0xa41aa09607ca80ee60d2ce166d4c02a71860e5c5 ; Successor Address: 0x41c6af7b388e80030e63f2686dc2ff9bfd1267c9	210
5.9	Approach to Check Permissions	215

List of Tables

2.1	Initial Number of Smart Contract Related Research Papers Returned by Each Search Engine	27
2.2	Data Collection for Each RQ.	30
2.3	List of questions included in the survey.	32
2.4	Part 1 - The mapping between survey results and our findings collected from the literature.	37
2.5	Part 2 - The mapping between survey results and our findings collected from the literature.	38
2.6	Literature of Offline Checking Methods	55
2.7	Literatures of Online Checking Methods.	56
3.1	Classification scheme.	85
3.2	Definitions of the 20 contract defects.	89
3.3	Survey results, distributions, and impacts of the 20 contract defects.	101
3.4	Features of Each Contract Defects	109
3.5	Features of Each Impact Level	109
3.6	Tools that detect some contract defects identified by our study.	114
4.1	The Definitions of contract defects with Impact level 1-3. The first eight contract defects can be detected by <i>DefectChecker</i>	135
4.2	The Information Required to Detect Each Contract Defect	147

4.3	Some Features of Dataset	154
4.4	Experimental results for <i>DefectChecker</i>	156
4.5	Input and Defects Detected of Each Tool	160
4.6	Experiment result of <i>Oyente</i>	161
4.7	Experiment result of <i>Mythril</i>	161
4.8	Experiment result of <i>Securify</i>	161
4.9	Result Comparison(F-Measure) between Four Tools	162
4.10	Overall Precision, Recall, and F-Measure of Each Tool	164
4.11	Time Consumption of Each Tool	164
4.12	Contract Defects in Ethereum	168
5.1	Information for the 756 self-destructed contracts	199
5.2	Reasons of Self-destruct and their distributions among 340 self-destructed smart contracts.	208
5.3	Results of Predicting <i>Limits of Permission</i> by using Five Machine Learning Algorithms	222
5.4	Results of Predicting <i>Limits of Permission</i> for 10-fold cross-validation by using Decision Tree	222
5.5	Top 50 words with highest information gain score.	225
5.6	self-destructed-related Info. on Ethereum blockchain	232
5.7	Open Interview Questions (Excerpt)	232

Chapter 1

Introduction

1.1 Background

1.1.1 Blockchain and Smart Contracts

With the great success of Bitcoin [141], its underlying blockchain system [18] attracts the attention of both academia and industry. A blockchain is a decentralized network that consists of distributed nodes. Each node runs a consensus protocol to maintain a shared ledger to secure the data on the blockchain. By Oct. 2021, the global market cap of Bitcoin reached 1.03 trillion dollars [26], which makes it become the most popular cryptocurrency in the world.

However, Bitcoin only allows users to encode non-Turing-complete scripts to process simple logic, which limits its usage scenario. In 2015, Ethereum [68] brought a revolutionary technology named smart contracts [214]. Smart contracts can be regarded as Turing-complete programs deployed on the blockchain, in which consensus protocol ensures their correct execution [32]. By utilizing smart contracts, developers can easily develop their decentralized applications (DApp), and apply blockchain techniques to different fields like gaming [53] and finance [72].

Smart contracts are usually developed using a high-level programming language, such as Solidity [173]. When developers deploy smart contracts to Ethereum, the source code of contracts will be compiled into bytecode. Since Ethereum is an add-only distributed ledger,

once smart contracts are deployed to a blockchain, they are immutable to be modified even when bugs are detected. Each smart contract is identified by a unique 160-bit hexadecimal string referred to as its contact address. Anyone can invoke this smart contract by sending transactions to the corresponding contract address. Their bytecode and transactions are all stored on the blockchain and visible to all users. The Ethereum Virtual Machine (EVM) is used to run smart contracts. The execution of smart contracts depends on their code. For example, if a contract does not contain functions that can transfer Ethers, even the creator can not withdraw the Ethers. Once smart contracts are deployed, they will exist as long as the whole network exists unless they execute the *selfdestruct* function [173]. This is the only way to remove a smart contract from Ethereum.

1.1.2 Smart Contract Example with Defects

```

1 pragma solidity ^0.4.25;
2 contract Example{
3     address[] participants;
4     uint participantID = 0;
5     function() payable{
6         require(msg.value == 1 Ether);
7         participants[participantID] = msg.sender;
8         participantID++;
9         if(this.balance == 10 Ether) // Strict Balance Equality
10            getWinner();
11    }
12    function getWinner(){
13        uint random = uint(block.blockhash(block.number)) % participants
14            .length; // Block Info Dependency
15        participants[random].transfer(9 Ether);
16        participantID = 0;
17    }
}

```

Listing 1.1: A simple contract

A contract defect is an error, flaw or fault in a smart contract that causes it to produce an incorrect or unexpected result, or to behave in unintended ways [209, 46]. Contract defects are related to not only security issues but also design flaws which might slow down development or increase the risk of bugs or failures in the future.

Listing 1.1 is an example of a smart contract with three contract defects in lines 9 and 14. The contract implements a simple gambling game by using Solidity [173], which is the most popular smart contract programming language on the Ethereum platform. Users can send 1 Ether to the contract. Once the contract receives 10 Ethers, the contract will choose 1 user as the winner randomly and send 9 Ethers to him/her.

The first line is called the version pragma, which is used to identify the compiler version of the contract. Lines 3-4 are the global parameters. Function on line 5 named fallback function, which is the only unnamed function of the smart contract. This function will be executed automatically when an error function call happens. For example, a user calls function “ δ ”, but there is no function named “ δ ” in the contract. In this situation, a fallback function will be executed to handle the error call. If the fallback function is marked by a keyword named *payable*, the fallback function will also be executed automatically when the contract receives Ethers. Lines 6 guarantee that each user sends 1 Ether to the contract. If the user sends other amounts of Ethers, the transaction will be reverted. When the contract receives 10 Ethers (line 9), the contract will choose one user to send 9 Ethers by using function *getWinner* (line 12). The contract generates a random number by using the block info related functions¹ in line 13. Then, the contract sends 9 Ether to the winner in line 14 and clears the storage in line 15.

However, there are two defects in this contract. The first defect is named *Strict Balance Equality*. In Ethereum, attackers can send Ethers to any contracts forcibly by utilizing the selfdestruct function [173]. This method will not trigger the fallback function, which means the victim contract cannot reject the Ethers. Attackers can send 1 Wei (1 Ether = 10^{18} Wei) to the contract, and the balance on the contract will never equal to 10 ETH in line 9. Thus, the logic of an equal balance check will fail to work due to the unexpected ethers sent by attackers. A possible solution for this defect is using a “range” to replace

¹block.blockhash and block.number are methods provided by Solidity to obtain block related information.

“equal”. Specifically, we can modify the code in line 9 to “*if(this.balance) >= 10 Ether && this.balance < 11 Ether*”

The second defect is *Block Info Dependency* in line 13. Ethereum provides a set of APIs (e.g., `block.blockhash`, `block.timestamp`) to help smart contracts obtain block-related information, like timestamps or hash numbers. Many contracts use these pieces of block information to execute some operations. However, the miner can influence block information; for example, miners can vary block time stamp by roughly 900 seconds. In other words, block info dependency operation can be controlled by miners to some extent. A possible solution for this is using the APIs provided by a trusted third-party organization. For example, Chainlink is a decentralized blockchain oracle network built on Ethereum, which also provide an API to generate random numbers [27].

1.1.3 Smart Contracts Maintenance

In software engineering, software maintenance are very broad activities that throughout the whole life cycle of a program. It is not easy to split the process of development and maintenance in many software development models. For example, Agile software development [12] refers to software development methodologies based on iterative development. In each iteration, new requirements and solutions will be added to improve the software. While according to the definition of ISO/IEC 14764 [108], there are four kinds of maintenance, i.e., corrective, adaptive, perfective, and preventive maintenance. *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.

Smart contracts also need maintenance, as they might have errors, or developers want to extend their features in the future. ***However, the immutability of smart contracts makes them almost impossible to be modified after they are deployed to Ethereum.*** According to

our empirical study (see Chapter 3) [33], in which we analyzed 131 smart contract related research papers published from 2014 to 2020, the only method for maintaining smart contracts is discarding the old contract and redeploying a contract. Fortunately, there are still two special methods that could make this process more smoothly.

The first method is using the *Selfdestruct* function. Developers could add a *Selfdestruct* function in the smart contract. When they want to upgrade the contract, they can call the *Selfdestruct* function to destruct the contract and transfer all the balance to another account. This method is easy to use, as it only needs to add a few lines of code in the contract. However, after executing the *Selfdestruct* function, all the data stored on the contract will be removed, which might lead to serious problems.

Another method named *Upgradeable Contracts* was also introduced to address this concern. Ethereum provides a function named *DelegateCall*, which allows a contract to use code in other contracts (implementation contract), and all storage changes are made in the caller's value (proxy contract). Once the function execution on the implementation contract is finished, the return value will be transferred back to the proxy contracts. When bugs are found or some new features need to be added at the implementation contract, the proxy contract can redirect to a new implementation contract, and the old one will be discarded.

1.2 Research Questions and Thesis Statement

Like all the other computer codes, smart contracts also need maintenance. In this thesis, we first give a big picture review of smart contract maintenance in RQ1, which aims to highlight the key issues of smart contract maintenance and current maintenance methods. According to the results of RQ1, we found that detecting and removing contract defects is one of the most popular reasons to maintain a smart contract. However, the revolutionary features make smart contract defects very different from traditional programs. Thus, we

aim to find as many defects as possible by using different methods. In RQ2, we introduced a method that is able to find contract defects from online Q&A posts. Then, we proposed a tool to detect the defined defects through EVM bytecode in RQ3. In RQ4, we presented a method to find defects by analyzing historical destructed smart contracts. A tool to detect the found defects is also introduced in the same RQ. The details of the research questions are as follows:

RQ1: What kinds of maintenance issues will smart contract developers encounter, and how do developers maintain smart contracts?

To answer this research question, we conducted 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. We totally found 13 maintenance issues and divided them into five groups. We also found that the most effective and popular method to maintain / upgrade / patch a smart contract is discarding the old contract and redeploying a new contract. Before redeploying the new smart contract to Ethereum, an important step is checking its robustness. According to our study, 31 publications proposed tools to analyze whether a smart contract contains specific contract defects.

Detecting and removing contract defects helps increase software robustness and enhance development efficiency [190, 115], which is widely used to maintain smart contracts. For example, Luu et al. [131] proposed a tool named *Oyente* to detect four security defects, i.e., mishandled exception, transaction-ordering dependence, timestamp dependence, and reentrancy attack. Kalra et al. [113] developed a tool named *Zeus* to detect seven security defects of smart contracts. Nikolic et al. [146] designed a tool named *Maian*, which focused on defects that lead to a contract not able to release Ethers, can transfer Ethers to arbitrary addresses, or can be killed by anybody. All of these methods designed tools to detect predefined defects. Developers use these tools to check whether a smart contract contains related

issues and then remove them to make smart contracts safer and more robust.

All of these tools first give some predefined defects and then develop tools to detect them. Although detecting and removing predefined defects can make smart contracts safer, one of the challenge is finding predefined defects. Besides, smart contracts ecosystem is fast-evolving. For example, Solidity, the most popular programming language for smart contracts, has 80 versions from Jan. 2016 to Jun. 2020 [172]. Thus, some predefined patterns might out-of-date, and many new kinds of defects might come out with the new features of the Solidity. In this case, it is very important to propose methods that can be used to find smart contract defects. In this thesis, we introduce two methods that use online posts and historical destructed smart contracts to find new smart contract defects, which highlight the second and fourth research questions:

RQ2: How to find smart contract defects by analyzing online posts?

To define defects for smart contracts, we need to collect issues that developers encountered. Developers often collaborate and share experience over question and answer (Q&A) sites like Ethereum StackExchange [175], the most popular and widely-used Q&A site for users of Ethereum. By analyzing posts on Ethereum StackExchange, we can identify and define a set of contract defects on Ethereum. To answer this research question, we first crawled all 17,128 posts from Ethereum StackExchange. Then, we used key words filtering to select Solidity defects-related posts. After that, we followed the card sorting [174] approach to analyze and categorize the filtered contract defects-related posts. We totally summarized 20 contract defects from five aspects: security, availability, performance, maintainability, and reusability. To validate the acceptance of our newly defined smart contract defects, we conducted an online survey and received 138 responses and 84 comments from developers in 32 countries. The feedback and comments show that developers believe removing the defined contract defects can improve the quality and robustness of smart contracts.

RQ3: How to detect the defined contract defects on RQ2 through contract bytecode ?

To answer this research question, we propose *DefectChecker*, a symbolic execution-based tool to detect contract defects. *DefectChecker* can detect contract defects from smart contracts bytecode without the need for source code. During the symbolic execution, *DefectChecker* generates the control flow graph (CFG) of smart contracts, as well as the “stack event”, and identifies three features, i.e., “Money Call”, “Loop Block”, and “Payable Function”. By using the CFG, stack event, and the three features, *DefectChecker* is extendable by designing different rules to detect contract defects. To increase the speed to analyze smart contracts, we remove the traditional SMT (satisfiability modulo theories) solver, e.g., Z3 [55]. Specifically, when executing a conditional jump, we should determine the satisfiability of the conditional expression by invoking an SMT solver. If the SMT solver cannot find a solution, we consider the corresponding program path as infeasible. Therefore, symbolic execution can be used to discover dead code. However, there may be little dead code in EVM bytecode, because the compiler can eliminate dead code during the compilation of smart contracts. To accelerate our analysis, we consider the conditional expression, which is equal to “0” as unsatisfiable and all other conditional expressions as satisfiable, without checking their satisfiability. Our evaluation results show that *DefectChecker* obtains 88.8% of F-score in the whole dataset and only requires 0.15s on average to analyze one smart contract.

RQ4: How to find smart contract defects by analyzing historical destructed smart contracts?

According to our empirical study (Chapter 2 - Answer for RQ1), many developers choose to add a *selfdestruct* function in their smart contracts. In this case, they can easily destruct the contracts and transfer all the balance to reduce the impact of financial loss when emergency situations happen, e.g., a contract being attacked. However, destructing smart contracts is also harmful. For example, Ethers sent to a self-destructed contract will

be locked forever, which increases the risk of using *selfdestruct* function. Thus, it is interesting to know why smart contract developers use *selfdestruct* function to destruct smart contracts. To find the reason, we conducted an online survey to collect feedback from real-world smart contract developers and summarize the key reasons. Their feedback shows that 66.67% of the developers will deploy an updated contract to the Ethereum after destructing the old contract. According to this information, we propose a method to find the self-destructed contracts (also called predecessor contracts) and their updated version (successor contracts) by computing the code similarity. By analyzing the difference between the predecessor contracts and their successor contracts, we found five reasons that led to the death of the contracts; two of them (i.e., Unmatched ERC20 Token and Limits of Permission) are contract defects that might affect the life span of contracts, and a tool to detect these two defects are also given in this RQ.

In this thesis, we totally proposed two methods to find contract defects from different perspectives, i.e., RQ2 (Chapter 3) and RQ4 (Chapter 5). Both of them have related advantages and limitations. Specifically, RQ2 introduced a breadth-first method by analyzing online Q&A posts, which could cover a high range of defects from a different aspect of smart contracts, e.g., security, performance. However, the biggest limitation is that we cannot dig into much depth for a specific issue. RQ4 focused on a specific feature of smart contracts, i.e., the Selfdestruct function. Although we only found a limited number of defects in RQ4, we still found many important selfdestruct-related features. For example, we found that Selfdestruct could be used to design Gastoken, and it was being utilized to launch a DDoS attack and had other harmful impacts to Ethereum (S5.6.4). Besides, Proxy contracts, e.g., EIP-2535, 1822, 1967, etc, were better ways to upgrade contracts, and we could also use a similar method to investigate the proxy contracts to find more defects. (S5.8) 3). Also, we found most of the selfdestructed-ERC20 token contracts will not return tokens back (S5.6.3), which could lead to a financial loss of users. All the above

selfdestruct-related features could guide further research on smart contracts.

1.3 Thesis Contribution

Chapter 2 conducts an empirical study on smart contract maintenance-related concerns, and makes the following key contributions:

1. 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.
 2. Our study identifies the key current maintenance methods used for smart contracts, which gives guidance for smart contract developers to better maintain their contracts.
 3. 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.
- *This work has led to a research paper published on the Empirical Software Engineering (EMSE) in 2021.*

Chapter 3 conducts an empirical study to find smart contract defects by analyzing online posts. We make the following key contributions in this work:

1. We define 20 contract defects for smart contracts considering five aspects: *security, availability, performance, maintainability* and *reusability*. We list symptoms and give a code example of each contract defects, which can help developers better understand the defined contract defects. To help further researches, we also give possible solution and possible tools for the contract defects.

2. We manually identify whether the defined 20 defects exist in real-life smart contracts.

Our dataset² contains a collection of 587 smart contracts, which can assist future studies on smart contract analysis and testing. Also, we analyze the impacts of the defined contract defects and summarize 5 common impacts. These impacts can help developers decide the priority of defects removal.

3. Our work is the first empirical study on contract defects for smart contracts. We aim to identify their importance, and gather inputs from practitioners. This work is a requirement engineering step for a practical contract defects detection tool, which is an important first step that can lead to the development of practical and impactful tools to practitioners.

- *This work has led to a research paper published on the IEEE Transactions on Software Engineering (TSE) in 2020.*

Chapter 4 introduces a tool named DEFECTCHECKER, an automated smart contract defect detection tool by analyzing EVM bytecode. In this work, we make the following contributions:

1. To the best of our knowledge, *DefectChecker* is the most **accurate** and the **fastest** symbolic execution-based model for smart contract defects detection.
2. We systematically evaluated our tool using an open source dataset to test its performance. In addition, we crawled all of the bytecode (165,621) on the Ethereum platform by the time of writing the paper and identified 25,815 smart contracts that contain at least one contract defect. Using these results, we find some real-world attacks, and give examples to show the importance of detecting contract defects.
3. Our datasets, tool and analysis results have been released to the community at <https://github.com/Jiachi-Chen/DefectChecker/>.

²The dataset can be found at <https://github.com/Jiachi-Chen/TSE-ContractDefects>

- This work has led to a research paper published on the IEEE Transactions on Software Engineering (TSE) in 2021.

Chapter 5 first uses an online survey to investigate why developers include or exclude *selfdestruct* function. Based on the finding, i.e., 66.67% of developers will deploy an updated contract to the Ethereum after destructing the old contract, we propose a method to find contract defects through historical destructed smart contracts. We make the following key contributions in this work:

1. To the best of our knowledge, this is the most comprehensive empirical work that investigates the *selfdestruct* function of smart contracts in Ethereum. We conduct an online survey to collect feedback from developers. According to this survey feedback, we summarize 6 reasons why developers add *selfdestruct* functions and 6 reasons why they do not add them to their smart contracts.
2. We design an approach to find 5 reasons why smart contracts self-destructed. These self-destruct reasons can be used as a guidance when practitioners develop their contracts. Also, our approach gives inspiration for researchers. They can use the same approach to find more self-destruct reasons and apply the method to other smart contract platforms, e.g., Ethereum Classic³ [206].
3. We propose a tool named LIFESCOPE to detect two problems that might shorten the life span of smart contracts. LIFESCOPE obtains 100% of F-measure in detecting *Unmatched ERC20 Token*. And it achieves an F-measure and AUC of 77.89% and 0.8673, respectively in detecting *Limits of Permission*.
4. According to the feedback from our survey, there are six common reasons why some developers do not use *selfdestruct* function. We give five suggestions for developers

³Ethereum Classic is another popular blockchain platform which support the running of smart contracts.

to address these issues and to help them better use the `selfdestruct` function in their smart contacts.

- *This work has led to a research paper published on the ACM Transactions on Software Engineering and Methodology (TOSEM) in 2021, as well as a short paper on the IEEE/ACM International Conference on Automated Software Engineering(ASE'20-SRC) in 2020.*

1.4 Structure of the thesis

In this thesis, we introduce the background and research questions in Chapter 1. In Chapter 2, we conduct an empirical study on investigating the maintenance issues and current maintenance methods for smart contracts. In Chapter 3 and 4, we introduce a methods that can be used to find contract defects by online posts and a related tool named DEFECTCHECKER to detect the defined smart contract defects, respectively. In Chapter 5, we present a method to find smart contract defects by analyzing historical destructed smart contracts. Finally, we give the conclusion and future research directions in Chapter 6.

Chapter 2

Maintenance-Related Concerns for Post-deployed Ethereum Smart Contract Development: Issues, Techniques, and Future Challenges

Chen, J., Xia, X., Lo, D., Grundy, J.C., Yang X. Maintenance-Related Concerns for Post-deployed Ethereum Smart Contract Development: Issues, Techniques, and Future Challenges, *Empirical Software Engineering*, 26, 117, 2021.
<https://doi.org/10.1007/s10664-021-10018-0>

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.

2.1 Introduction

With the great success of Bitcoin [141], considerable attention has been paid to the emerging concepts of blockchain technology [18]. However, the usage scenario of Bitcoin is limited, as the main application of Bitcoin is storing and transferring monetary values [62]. The appearance of Ethereum [68] 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 [212]. 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*) [205], which have been applied to different areas, such as IoT [44], financial [78], gaming [53], and data security domain [191].

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 sys-

tem and smart contract immutability – make smart contracts much harder to maintain than conventional software [22]. 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 [212]. 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 [160]. It is a very broad activity according to the definition of ISO/IEC 14764 [108]. 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 [85]. 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 [32, 131]. This vulnerability was then utilized by attackers and led to the famous DAO attack [54], which made the DAO lose 3.6 million Ethers (about \$20/Ether when the attack happened). According to recent research [114, 127], 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 [224, 156, 22, 124] 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 [224] mentioned that smart contracts miss the support of exception handling, e.g., the *try...catch*. However, Solidity adds the exception handling in v6.0 [173]. 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 2.2, we provide background knowledge of smart contracts and Ethereum. In Section 2.3, 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 2.4 and 2.5, respectively. In Section 2.6, we highlight key threats to validity. We discuss what should be done in the future to improve the Ethereum ecosystem in Section 2.7 and review related work in Section 2.8. Finally, we conclude the whole study in Section 2.6.

2.2 Background

2.2.1 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

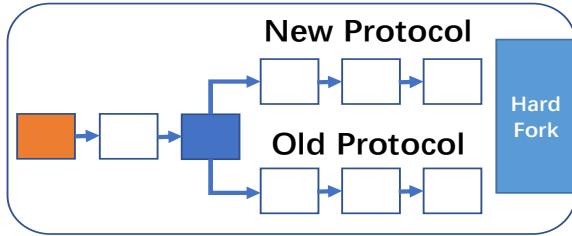


Figure 2.1: An Example of Hard Fork. The blue block called divergence block, where the blockchain system updates its protocol. The new protocol for hard fork is not backward-compatible.

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 [26]. Unlike Bitcoin, which has a fixed number of coins (21 million in total), 18 million Ethers are created every year [212] (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 [75]. 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 [74], which has become one of its biggest limitations. At the end of 2017, there is a famous smart-contract-based game named CryptoKitties [53] published in the Ethereum. However, the popularity of the game slowed down all transactions as too many players sent transactions to the Ethereum blockchain.

2.2.2 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

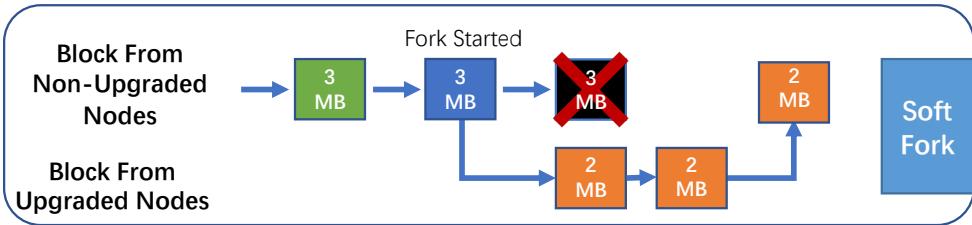


Figure 2.2: An Example of Soft Fork. The blue block called divergence block, where the blockchain system updates its protocol. The new protocol for soft fork is backward-compatible.

kinds of forks, i.e., hard fork and soft fork.

Hard Fork. Figure 2.1 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 Figure 2.1, 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 Figure 2.2 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.

2.2.3 Smart Contracts

Smart contracts can be regarded as Turing-complete programs that run on the blockchain. They are usually developed in a high-level language, e.g., Solidity, Vyper [196]. 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 [173]. 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 [173]. Even worse, all the transactions, bytecode, invocation parameters are visible to everyone, which makes smart contracts face major security challenges.

2.2.4 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 ($1\text{Ether} = 10^9\text{Gwei}$). The final transaction fee is calculated by $\text{gas_cost} \times \text{gas_price}$. 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* [91], in May 2020, if the gas price is higher than

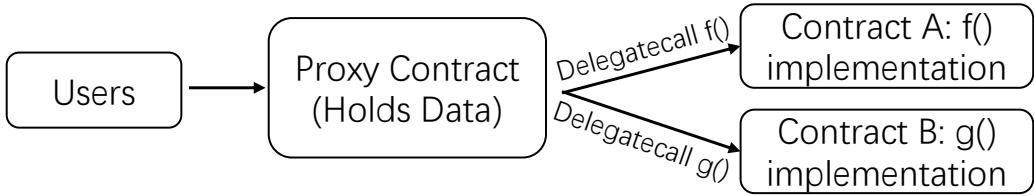


Figure 2.3: An Example of the upgradeable contract.

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 [75]. When the maximum gas cost of a transaction exceeds the 12.5 million, it will be reverted forever.

2.2.5 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 2.3 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 [153] to help developers develop upgradeable smart contracts in just a few lines. *EIP 2535* [144] (the Diamond Standard) also defines the standard to help developers design upgradeable smart contracts.

2.2.6 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 [23]. Thus, software maintenance is an important and inevitable part of the software development life cycle. According to previous work [19], software maintenance can lead to 60% of software cost. Besides, in many software development models, e.g., Spiral model [20], Agile development [12], 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 [108], there are four kinds of software maintenance, i.e., corrective, adaptive, perfec-

tive, 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.

2.2.7 Card Sorting

Card sorting is a method to organize data into logical groups [174], 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. 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.

2.3 Methodology

Figure 2.4 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

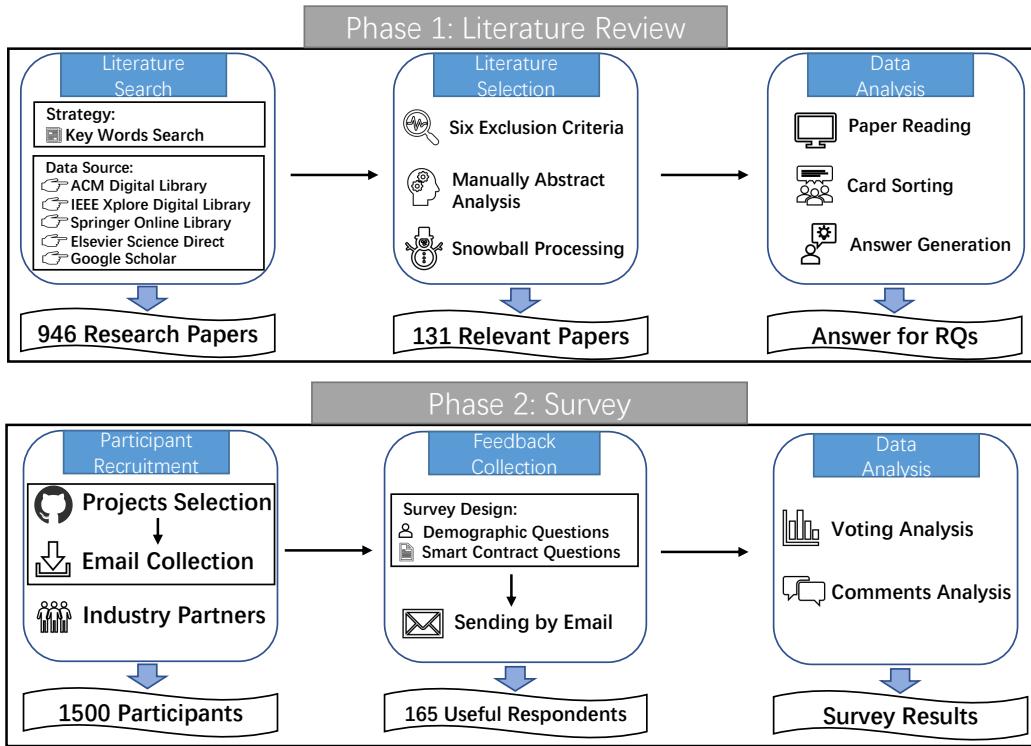


Figure 2.4: Overview of methodology design

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.

2.3.1 Literature Review

We follow the method provided by Kitchenham et al. [119] to perform the literature review. There are three steps in phase 1, i.e., literature search, literature selection, and data analysis.

2.3.1.1 Literature Search

Guided by prior works [49, 169, 103], 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

Table 2.1: 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

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 2.1. (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.

2.3.1.2 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.
- (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 [174] to check the collected data. Card sorting is a common method used to evaluate and derive categories from the data [117]. 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 pre-defined 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 [103] 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 con-

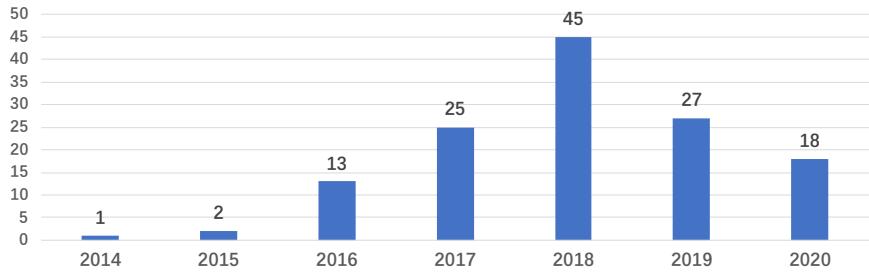


Figure 2.5: The number of papers published between 2014 to 2020.

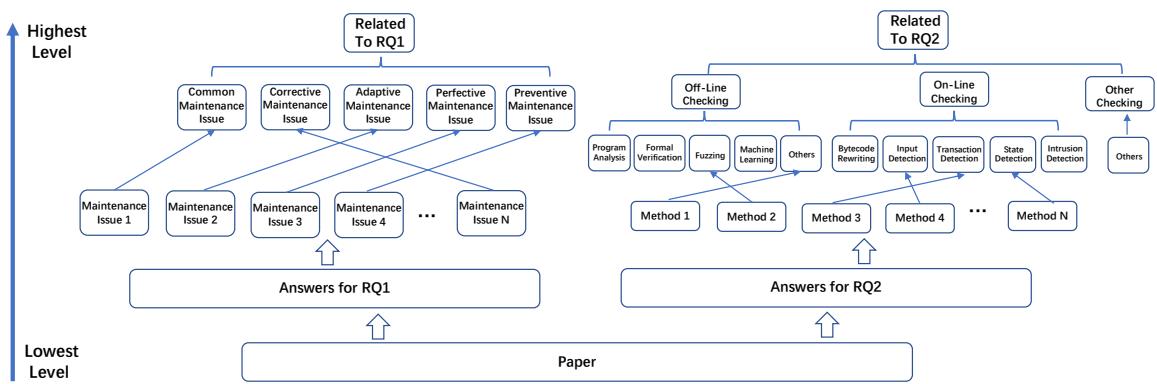


Figure 2.6: The steps of open card sorting.

tracts”, “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>.

2.3.1.3 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 Figure 2.5), 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 de-

Table 2.2: 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.

scribed in Figure 2.6. The two researchers first read the paper carefully and were required to collect the answers to the two RQs shown in Table 2.2, 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 (1), the data collected from papers were first summarized into detailed maintenance issues. For example, previous works [40, 36] 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.

2.3.2 Survey

2.3.2.1 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 [32] 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
- 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 2.3. 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 [32] 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.

2.3.2.2 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

Table 2.3: 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?

other questions come from?

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 3.4.2.1). 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 [224, 28] 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 3.7.2). 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 [30] 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 3.3.1.3)

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 3.4.1.2). 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 3.4.1.2). 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 3.4.2.1), lack community support (See Section 3.4.2.2), high-quality reference code (See Section 3.4.5.2), standards (See Section 3.4.5.3). 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 3.4.4.1) We wanted to investigate whether developers agree with this opinion.

Question 14: In Section 3.7.1, 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 3.7.2, we discussed that merging cutting-edge technologies can help to improve Ethereum and Solidity. This question is used to investigate developers' attitudes about this.

2.3.2.3 Survey Validation

Guided by Kitchenham et al. [120], we utilized an anonymous survey [187] 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.

2.3.2.4 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 API¹. 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

¹<https://developer.github.com/v3/>

the email to reduce the misleading.

2.3.2.5 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 [32, 224, 22, 28, 30]. 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%) indicate risk analysis, 4 (2.42%) indicate research. (Some respondents have multiple job roles; thus the total number exceeds 165.)

2.4 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 Table 2.4 and 2.5 to help readers better understand the relation between the survey results and the findings collected

Table 2.4: 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%)

Table 2.5: 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 permission-less 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 de-structured. (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.

from literature. The first column of the tables is the survey ID (detailed information can be found at Table 2.3). 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.

2.4.1 Common Maintenance Issues

2.4.1.1 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 answers² for the question “How do you maintain your smart contracts” (Q4 in Table 2.3). 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.

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

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 [127, 165] in their smart contracts, there was no way to modify the contract, as the contract did not contain a `Selfdestruct` function and was not developed 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 [30] 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 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 ensures 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 2.3), 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.

2.4.1.2 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 2.4.1.1). 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 [4]. 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 [184]. 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 [30] 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.

2.4.1.3 Low Readability

Readability is important to help developers understand the smart contracts and maintain their smart contracts [224]. 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.

2.4.1.4 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 [198] (min: 3, max: 16, median: 6, avg: 7.6 years), in desktop software development [197] (min: 3, max: 12, avg: 6.5 years), the smart contract developers and researchers seem less experienced.

2.4.2 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.

2.4.2.1 The Lack of Mature Tools

Many previous works [224, 149, 22] 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 [131, 127, 181] and formal verification [15, 16, 101] 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 com-

plain there are only a few tools they think can be used in practice.

2.4.2.2 The Lack of Community Support

Community support is a primary source of knowledge for blockchain software projects. 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 [224, 100] 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 [224, 100] 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.

2.4.3 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.

2.4.3.1 Unpredictable Fork Problems

Ethereum uses soft forks and hard forks (See Section 2.2.2) 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 [140]. 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 [54]. 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) [206]. 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" [64] and "Spurious Dragon" [167], 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.

2.4.3.2 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. [86] 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.

2.4.4 Perfective Maintenance Issues

As long-lived software [130], 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.

2.4.4.1 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 [224]. 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. [100] 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 [224]. 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 [131]. Finally, EVM uses a key-value store, which is a very simplistic database and can lead to low efficiency [95].

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. [14]

2.4.4.2 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 2.4.1.3, 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 [40, 36], 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.

2.4.5 Preventive Maintenance Issues

Preventive maintenance aims to lessen the likelihood of a sudden breakdown of the programs [177]. 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 [85]. 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.

2.4.5.1 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 2.5). Compared to traditional software, the maintenance methods of smart contracts to remove latent errors are much less, e.g., code smell removal [83], bug prediction [92], self-admitted technical debt determination [217]. 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 [220]. User reviewers can provide feature requests to help developers improve the programs [133, 94]. Comments in source code can be used to detect self-admitted technical debate, which can be used to signal future errors [217]. 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.

2.4.5.2 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. [99] 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. [116]; 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. [116] 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 [58]. 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 [78], which are used to make tokens. Thus, the active contracts also cannot provide too much reference value.

Hegedűs et al. [100] 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.

2.4.5.3 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 [78] 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 [53] was published and swept the globe. To help other developers develop similar applications, ERC 721 was published in Jan. 2018. ERC 721

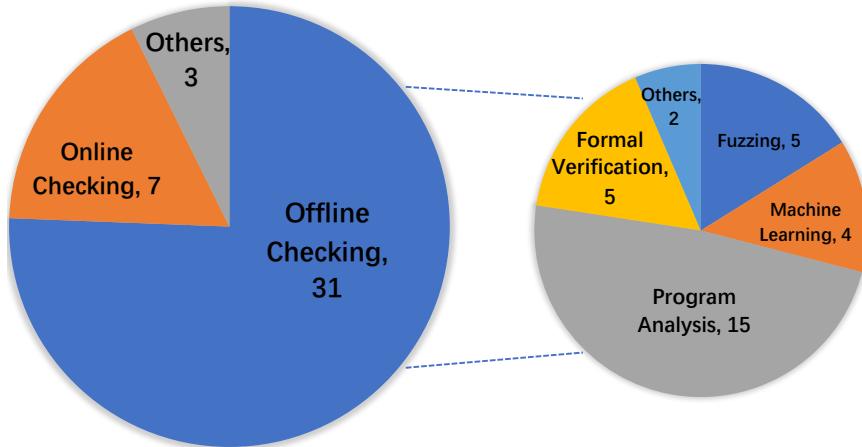


Figure 2.7: Distribution of Maintenance Methods

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 [63]. According to our survey (Q8), 82 (49.70%) respondents said Ethereum lacks standards, which increases the difficulty of the maintenance of smart contracts.

2.5 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.

2.5.1 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 up-

graded 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 redeloys smart contracts by using *Selfdestruct* function, respectively. The distribution of these methods is shown in Figure 2.7.

2.5.2 Offline Checking Methods

Table 2.6 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 (PA), fuzzing, formal verification (FV), machine learning (ML), and others. In the following subsections, we discuss some key examples.

2.5.2.1 Program Analysis

CFG (Control Flow Graph) Based Tools. In 2016, Luu et al. [131] 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

Table 2.6: Literature of Offline Checking Methods

Category	Name of Publications	Years
PA	OSIRIS: Hunting for Integer Bugs in Ethereum Smart Contracts [183]	2018
	The art of the scam: Demystifying honeypots in Ethereum smart contracts [184]	2019
	Security Assurance for Smart Contract [223]	2018
	Vandal: A Scalable Security Analysis Framework for Smart Contracts [24]	2018
	MadMax: surviving out-of-gas conditions in Ethereum smart contracts [96]	2018
	Finding The Greedy, Prodigal, and Suicidal Contracts at Scale [147]	2018
	sCompile: Critical Path Identification and Analysis for Smart Contracts [29]	2019
	teether: Gnawing at Ethereum to Automatically Exploit Smart Contracts [122]	2018
	Making Smart Contracts Smarter [131]	2016
	Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contract [139]	2019
	SmartCheck: Static Analysis of Ethereum Smart Contracts [181]	2018
	TokenScope: Automatically Detecting Inconsistent Behaviors of Cryptocurrency Tokens in Ethereum [42]	2019
	Towards saving money in using smart contracts [40]	2018
	GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts [36]	2020
	Securify: Practical Security Analysis of Smart Contracts [186]	2018
FV	Formal Verification of Smart Contracts [15]	2016
	A formal verification tool for Ethereum VM bytecode [158]	2018
	Kevm: A complete formal semantics of the Ethereum virtual machine [101]	2018
	Towards verifying Ethereum smart contract bytecode in Isabelle/HOL [2]	2018
	ZEUS: Analyzing Safety of Smart Contracts [114]	2018
Fuzzing	ContractFuzzer: fuzzing smart contracts for vulnerability detection [110]	2018
	ReGuard: Finding Reentrancy Bugs in Smart Contracts [127]	2018
	EVMFuzz: Differential Fuzz Testing of Ethereum Virtual Machine [88]	2019
	sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts [142]	2020
	Exploiting the Laws of Order in Smart Contracts [121]	2019
ML	S-gram: Towards Semantic-Aware Security Auditing for Ethereum Smart Contracts [128]	2018
	Hunting the Ethereum Smart Contract: Color-inspired Inspection of Potential Attacks [104]	2018
	Towards Safer Smart Contracts: A Sequence Learning Approach to Detecting Security Threats [178]	2019
	Checking Smart Contracts with Structural Code Embedding [90]	2020
Others	Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach [136]	2018
	Mutation Testing for Ethereum Smart Contract [125]	2019

Table 2.7: Literatures of Online Checking Methods.

Methodology	Name of Publications	Years
Bytecode Rewriting	Smart Contract Defense through Bytecode Rewriting [5]	2019
Bytecode Rewriting	Monitoring smart contracts: ContractLarva and open challenges beyond [6]	2018
Input Detection	Town Crier: An Authenticated Data Feed for Smart Contracts [219]	2016
Input Detection	FSFC: An input filter-based secure framework for smart contract [201]	2020
Transactions Detection	ÆGIS: Smart Shielding of Smart Contracts [81]	2019
Transactions Detection	VULTRON: Catching Vulnerable Smart Contracts Once and for All [199]	2019
State Detection	Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks [165]	2018
Intrusion Detection	ContractGuard: Defend Ethereum Smart Contracts with Embedded Intrusion Detection [200]	2019

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* [40] and *GasChecker* [36] 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. [184], Chang [29], Nikolic et al. [147], Zhou et al. [223], Krupp et al. [122], Mossberg et al. [139] also use similar methods that design rules based on the CFG to detect other smart contract vulnerabilities.

Some works make optimizations, e.g., Maian [147] 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* [24] 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 con-

tracts. For example, *MadMax* [96] uses logic-based specifications to detect gas-focused vulnerabilities of smart contracts based on the output of *Vandal*. Tsankov et al. [186] 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* [42] 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* [181] takes smart contract source code as input, and converts the code to the AST (abstract syntax tree) [3]. Based on the AST, *SmartCheck* uses several patterns to detect 21 kinds of smart contract issues.

2.5.2.2 Formal Verification

Formal verification is a method that uses formal methods of mathematics to prove or disprove the correctness of a system [60]. This method usually uses a formal proof on an abstract mathematical model to make the verification.

Bhargavan et al. [15] proposed the first formal verification tool for smart contracts based on the F* proof assistant [176], and Amani et al. [2] presented a tool based on Isabelle/HOL [148]. However, both of these the tools only use incomplete semantics of EVM, which might lead to errors. Thus, Park et al. [158] use a complete and thoroughly tested formal semantics of EVM to enhance the efficacy of their tool.

Kalra et al. [114] 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 [129]. Based on the LLVM bytecode, *Zeus* designs several policy violations and uses a verifier to determine assertion violations.

2.5.2.3 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. [110] proposed the first fuzzing tool named *ContractFuzzer*, which applies fuzzing to detect seven kinds of security issues. *ContractFuzzer* utilizes smart contract ABI [173] 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* [142] can cover more branches to find more security issues. *EthRacer* [121] can run directly on Ethereum bytecode and without the need of ABI, which enlarges the usage scenario. Re-Guard [127] provides a web service for developers to make it is easy to use. EVMFuzz [88] designs a differential fuzz testing framework, which supports different programming languages for EVM smart contracts.

2.5.2.4 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* [128] 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* [90] uses *SmartCheck* to label the vulnerabilities

and utilizes deep-learning to train the model to predict smart contract vulnerabilities. Tann et al. [178] use *MAIAN* to label the security issues and use LSTM to predict potential issues. Huang et al. [104] 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.

2.5.2.5 Other Approaches

Mavridou et al. [136] 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. [125] 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.

2.5.3 Online Checking Methods

Online checking methods can help smart contract developers defend their contracts against attacks even after they have been deployed. Table 2.7 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

EIP³ [63] adopts any of those in a new version.

Ayoade et al. [5] proposed a method that can automatically detect vulnerable EVM bytecode segments and uses a guarded bytecode segment to replace it. Their tool is based on predefined policy rules and can only support a limited number of simple rules. Similarly, *ContractLarva* [6] insert protection code into the source code of smart contracts. This updated bytecode can defend against related attacks.

TownCrier [219] and *FSFC* [201] 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 front-end program sends transactions to smart contracts, *TownCrier* uses a combination of Software Guard Extensions [52] and Intel’s recently released trusted hardware capability [106] 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 [81] and *VULTRON* [199] detect and reverse malicious transactions to protect smart contracts. *ÆGIS* uses predefined patterns to identify malicious transactions. *VULTRON* compares the actual transferred Ethers and the normal transferred Ethers to find malicious transactions.

Sereum [165] 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 leads 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 [200] is the first intrusion detection system for smart contracts against

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

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.

2.5.4 Other Methods

Colombo et al. [48] 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 et al. [135] 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. [32]. 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.

2.6 Threats To Validity

2.6.1 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 [172]. 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 et al.’s [150] 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.

2.6.2 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.

2.7 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 Section 2.4 and 2.5.

2.7.1 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 [63], and many teams provide libraries and referee code, e.g., OpenZeppelin Contracts [154], Smart contract best practice [50]. 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.

2.7.2 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 2.4.1.1). 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 et al. [25] 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 2.4.1.1, 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 [30] 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 [211], DPoS [126] 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.

2.8 Related Work

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

2.8.1 Survey Based Smart Contract Empirical Studies

Bosu et al. [22] 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. [28] 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. [32] 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 [22] 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 3.4.2.1).

There is a big difference between Chakraborty et al.'s work [28] 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 [32] 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.

2.8.2 Literature Review Based Smart Contract Empirical Studies.

Conoscenti et al. [49] 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. [188] 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. [134] 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. Udoekwu 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, Zhou et al. [224] 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 [173]. 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.

2.8.3 Security Related Smart Contract Empirical Studies.

Li et al. [124] 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 [9] 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 [204, 45] used this dataset to design machine learning methods to detect Ponzi smart contracts.

Delmolino et al. [56] 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. [4] 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.

2.8.4 Other Smart Contract Empirical Studies.

Zheng et al. [221] 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 [222] 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. [163] investigated the challenges of applying blockchain technology to the IoT to increase the security and reliability. Mohanta [138] 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 [159, 156], programming languages [98, 166, 157], ecosystem [116, 99, 100], permissions [195], design patterns [11], life cycle [58], call relations [17]. Durieux et al. [61] 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 [51] 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.

2.9 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.

Chapter 3

Defining Smart Contract Defects on Ethereum

Chen, J., Xia, X., Lo, D., Grundy, J.C., Luo, X. and Chen, T. Defining Smart Contract Defects on Ethereum, *IEEE Transactions on Software Engineering*.
<https://doi.org/10.1109/TSE.2020.2989002>

Abstract: *Smart contracts* are programs running on a blockchain. They are immutable to change, and hence can not be patched for bugs once deployed. Thus it is critical to ensure they are bug-free and well-designed before deployment. A *Contract defect* is an error, flaw or fault in a smart contract that causes it to produce an incorrect or unexpected result, or to behave in unintended ways. The detection of contract defects is a method to avoid potential bugs and improve the design of existing code. Since smart contracts contain numerous distinctive features, such as the *gas system*. *decentralized*, it is important to find smart contract specified defects. To fill this gap, we collected smart-contract-related posts from Ethereum StackExchange, as well as real-world smart contracts. We manually analyzed these posts and contracts; using them to define 20 kinds of *contract defects*. We categorized them into indicating potential security, availability, performance, maintainability and reusability problems. To validate if practitioners consider these contract as harmful, we created an online survey and received 138 responses from 32 different countries. Feedback showed these contract defects are harmful and removing them would improve the quality and robustness of smart contracts. We manually identified our defined contract defects in 587 real world smart contract and publicly released our dataset. Finally, we summarized 5

impacts caused by contract defects. These help developers better understand the symptoms of the defects and removal priority.

3.1 Introduction

The considerable success of decentralized cryptocurrencies has attracted great attention from both industry and academia. Bitcoin [141] and Ethereum [84, 213] are the two most popular cryptocurrencies whose global market cap reached \$162 billion by April 2018 [26].

A *Blockchain* is the underlying technology of cryptocurrencies, which runs a consensus protocol to maintain a shared ledger to secure the data on the blockchain. Both Bitcoin and Ethereum allow users to encode rules or scripts for processing transactions. However, scripts on Bitcoin are not Turing-complete, which restrict the scenarios of its usage. Unlike Bitcoin, Ethereum provides a more advanced technology named *Smart Contracts*.

Smart contracts are Turing-complete programs that run on the blockchain, in which consensus protocol ensures their correct execution [84]. With the assistance of smart contracts, developers can apply blockchain techniques to different fields like gaming and finance. When developers deploy smart contracts to Ethereum, the source code of contracts will be compiled into *bytecode* and reside on the blockchain. Once a smart contract is created, it is identified by a 160-bit hexadecimal address, and anyone can invoke this smart contract by sending transactions to the corresponding contract address. Ethereum uses *Ethereum Virtual Machine (EVM)* to execute smart contracts and transaction are stored on its blockchain.

A blockchain ensures that all data on it is immutable, i.e., cannot be modified, which means that smart contracts cannot be patched when bugs are detected or feature additions are desired. The only way to remove a smart contract from blockchain is by adding a *self-destruct* [173] function in their code. Even worse, smart contracts on Ethereum operate on a permission-less network. Arbitrary developers, including attackers, can call the methods

to execute the contracts. For example, the famous ***DAO attack*** [54] made the DAO (Decentralized Autonomous Organization) lose 3.6 million Ethers (\$150/Ether on Feb 2019), which then caused a controversial hard fork [109, 206] of Ethereum.

It is thus critical to ensure that smart contracts are bug-free and well-designed before deploying them to the blockchain. In software engineering, a software defect is an error, flaw or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways [209, 46]. Contract defects are related to not only security issues but also design flaws which might slow down development or increase the risk of bugs or failures in the future. Detecting and removing contract defects helps increase software robustness and enhance development efficiency [190, 115]. Since the revolutionary changes of smart contracts compared to traditional softwares, e.g., the gas system, decentralized features, smart contracts contain many specific defects.

In this paper, we conduct an empirical study on defining smart contracts defects on Ethereum platform, the most popular decentralized platform that runs smart contracts. Please note that some previous works [131, 146, 114] focus on improving the quality of smart contracts from the security aspect. However, this is the first paper that aims to provide a systematic study of contract defects from five aspects: security, availability, performance, maintainability and reusability. These previous works were not comprehensive and did not validate whether practitioners consider these contract defects as harmful. To address these limitations, we conducted our results from 17,128 *Ethereum.StackExchange*¹ posts and validated it by an online survey. To help developers better understand the symptoms and distribution of smart contract defects, we manually labeled a dataset and released it publicly to help further study. In this paper, we address the following key research questions:

RQ1: What are the smart contract defects in Ethereum?

¹<https://ethereum.stackexchange.com/>

We identified and defined 20 smart contract defects from *StackExchange* posts and real-world smart contracts. These 20 contract defects are considered from security, availability, performance, maintainability and reusability aspects. By removing the defined defects from the contracts, it is likely to improve the quality and robustness of the programs.

RQ2: How do practitioners perceive the contract defects we identify?

To validate the acceptance of our newly defined smart contract defects, we conducted an online survey and received 138 responses and 84 comments from developers in 32 countries. The options in the survey are from 'Very important' to 'Very unimportant' and we give each option a score from 5 to 1, respectively. The average score of each contract defect is 4.22. The feedbacks and comments show that developers believe removing the defined contract defects can improve the quality and robustness of smart contracts.

RQ3: What are the distributions and impacts of the defects in real-world smart contracts?

We manually labeled 587 smart contracts and found that more than 99% of smart contracts contain at least one of our defined defects. We then summarized 5 impacts that can help researchers and developers better understand the symptoms of these contract defects.

The main contributions of this paper are:

- We define 20 contract defects for smart contracts considering five aspects: *security, availability, performance, maintainability and reusability*. We list symptoms and give a code example of each contract defects, which can help developers better understand the defined contract defects. To help further researches, we also give possible solution and possible tools for the contract defects.
- We manually identify whether the defined 20 defects exist in real-life smart contracts.

Our dataset² contains a collection of 587 smart contracts, which can assist future studies on smart contract analysis and testing. Also, we analyze the impacts of the

²The dataset can be found at <https://github.com/Jiachi-Chen/TSE-ContractDefects>

defined contract defects and summarize 5 common impacts. These impacts can help developers decide the priority of defects removal.

- Our work is the first empirical study on contract defects for smart contracts. We aim to identify their importance, and gather inputs from practitioners. This work is a requirement engineering step for a practical contract defects detection tool, which is an important first step that can lead to the development of practical and impactful tools to practitioners.

The remainder of this paper is organized as follows. In Section 4.2, we provide background knowledge of smart contracts. In Sections 4.3-4.5, we present the answers to the three research questions, respectively. We discuss the implications, and challenge in automatic contract defects detection in Section 4.6. In Section 4.7, we introduce threats to validity. Finally, we elaborate the related work in Section 4.8, and conclude the whole study and mention future work in Section 4.9.

3.2 Background

In this section, we briefly introduce background knowledge about smart contracts as well as the Solidity programming language for smart contract definition.

3.2.1 Smart Contracts - A Decentralized Program

A smart contract is “*a computerized transaction protocol that executes the terms of contract*” [179]. Their bytecode and transactions are all stored on the blockchain and visible to all users. Since Ethereum is an add-only distributed ledger, once smart contracts are deployed to a blockchain, they are immutable to be modified even when bugs are detected. Once a smart contract is created, it is identified by a unique 160-bit hexadecimal string referred to as its contact address. The Ethereum Virtual Machine (EVM) is used to run smart

contracts. The executions of smart contracts depend on their code. For example, if a contract does not contain functions that can transfer Ethers, even the creator can not withdraw the Ethers. Once smart contracts are deployed, they will exist as long as the whole network exists unless they execute *selfdestruct* function [173]. *selfdestruct* is a function that if it is executed, the contract will disappear and its balance will transfer to a specific address. In this paper, we describe smart contracts developed using *Solidity* [173], the most popular smart contract programming language in Ethereum.

3.2.2 Features of Smart Contracts

The Gas System. In Ethereum, miners run smart contracts on their machines. As compensation for miners who contribute their computing resources, the creators and users of smart contracts will pay a certain amount of Ethers to the miners. The Ethers that are paid to miners are computed by: *gas cost * gas price*. Gas cost depends on the computational resource the transaction will take and gas price is offered by the transaction creators. The minimum unit of gas price is *Wei* (1 Ether = 10^{18} Wei). The miners have the right to choose which transaction can be executed and broadcasted to the other nodes on the blockchain [213]. Therefore, if the gas price is too low, the transactions may not be executed. To limit the gas cost, when a user sends a transaction to invoke a contract, there will be a limit (*Gas Limit*) that determines the maximum gas cost. If the gas cost exceeds the *Gas Limit*, the execution is terminated with an exception often referred to as *out-of-gas error*.

Data location. In smart contracts, data can be stored in *storage*, *memory* or *calldata* [173]. *storage* is a persistent memory area to store data. For each *storage variable*, EVM will assign a storage slot ID to identify it. Writing and reading *storage variable* is the most expensive operation as compared with reading from the other two locations. The second memory area is named *memory*. The data of the *memory variables* will be released after their life cycle finished. Writing and reading to *memory* is cheaper than *storage*. *Calldata*

is only valid for parameters of external contract functions. Reading data from the *Calldata* is much cheaper than *memory* or *storage*.

3.2.3 Solidity

Solidity is the most popular programming language that is used to program smart contracts on the Ethereum platform. In this subsection, we give a basic overview of Solidity programming as well as a Solidity example.

Fallback Function. The fallback function [173] is the only unnamed function in Solidity programming. This function does not have arguments or return values. It is only executed when an error function call happens. For example, a user calls function “ δ ” but the callee contract does not contain this function. The fallback function will be executed to handle the error. Also, if a fallback function is marked by *payable*³, e.g., line 13 in listing 4.1, it will be executed automatically when the contract receives Ethers.

Ether Transfer and Receive. Solidity provides three APIs to transfer Ethers between accounts, i.e., *address.transfer(amount)*, *address.send(amount)*, and *address.call.value(amount)()*. *transfer* and *send* will limit the gas of fallback function in callee contracts to 2300 gas [173]. This gas is not enough to write to storage, call functions, or send Ethers. Therefore, *transfer* and *send* functions can only be used to send Ethers to *External Owned Accounts* (EOA).⁴ *call* will not limit the gas of fallback function. Therefore, *call* can be used to send Ethers to either contract or EOA. The difference between *transfer* and *send* is that *transfer* will throw an exception and terminate the transaction if the Ether fails to send, while *send* will return a boolean value instead of throwing an exception.

```
1 pragma solidity ^0.4.25;
2 contract Gamble{
3     address owner;
4     address [] members;
5     address [] participators;
```

³If a function wants to receive Ethers, it has to add *payable*

⁴There are two types of accounts on Ethereum: externally owned accounts which controlled by private keys, and contract accounts which controlled by their contract code.

```

6   uint participantID = 0;
7   modifier onlyOwner /* Transaction State Dependency */
8     require(tx.origin==owner);
9   }
10  function constructor() // constructor_function
11    owner = // this is the address of tx.origin
12    0xdCad...d1D3AD; /* Hard Code Address */
13  function() payable // Executed when receiving Ethers
14    ReceiveEth();
15  function ReceiveEth() payable{
16    if(msg.value!=1 ether){
17      revert(); //msg.value is the number of received ETHs
18    members.push(msg.sender);
19    participants[participantID] = msg.sender;
20    participantID++;
21    if(this.balance==10 ether) /* Strict Balance Equality*/
22      getWinner();
23  function getWinner() //choose a member to be the winner
24    /* Block Info Dependency */
25    uint winnerID = uint(block.blockhash(block.number)) %
26      participants.length;
27    participants[winnerID].send(8 ether);
28    participantID = 0;
29  function giveBonus() returns(bool) //send 0.1 ETH to all members as
30    bonus
31  /* Unmatched Type Assignment , Nested Call */
32  for(var i = 0;i < members.length; i++){
33    if(this.balance > 0.1 ether)
34      /*DoS Under External Influence */
35      members[i].transfer(0.1 ether); }
36    /* Missing Return Statement */
37  function suicide(address addr) onlyOwner //Remove the contract from
38    blockchain
39    selfdestruct(addr); }
40  function withDraw(uint amount) onlyOwner //withdraw certain Ethers
41    to owner account
42    address receiver = 0x05f4...d27;
43    receiver.call.value(amount); }}
```

Listing 3.1: A “Gamble” smart contract. However, this contract contains several contract defects.

Version Controller. Ethereum supports multiple versions of Solidity. When deploying a smart contract to the Ethereum, developers need to choose a specific Solidity compiler version to compile the contract. Solidity is a young and evolving programming language. There are more than 20 versions released up to 2019. Different versions might have several significant language changes. If developers do not choose the correct version of Solidity, the smart contract compilation might fail. To make code reuse easier, a contract can be an-

notated with *version pragma* that indicates the version that supported. The *version pragma* is used as: “*pragma solidity^version*” or “*pragma solidity version*”. For example, “*pragma solidity 0.4.1*” means that this contract supports compile version 0.4.1 and above (except for v0.5.0) while “*pragma solidity 0.4.1*” means that the contract only supports compile version 0.4.1.

Permission Check. Smart contracts on Ethereum run in a permission-less network; everyone can call methods to execute the contracts. Developers usually add permission checks for permission-sensitive functions. For example, the contract will record the owner’s address in its constructor function as the constructor function can only be executed once when deploying the contract to the blockchain. In each transaction, the contract compares whether the caller’s address is the same as the owner’s address. Solidity provides *msg* related APIs to receive caller information. For example, contracts can get the caller address from *msg.sender*. Besides, Solidity also provides function modifiers to add prerequisite checks to a function call. A function with a function modifier can be executed if it passes the check of the modifier.

Solidity Example. Listing 4.1 is a simple example of a smart contract which is developed in Solidity. The contract is a gambling contract, each gambler sends 1 Ether to this contract. When the contract receives 10 Ethers, it will choose one gambler as the winner and sends 8 Ethers to him.

The first line indicates the contract supports compiler version 0.4.25 to 0.5.0 (not included). The *modifier* on line 7 is used to restrict the behavior of functions. For example, *onlyOwner* requires the *tx.origin* equals to the *owner*, and *tx.origin* is used to get the original address that kicked off the transaction, otherwise, the transaction will be roll back. If a function contains modifiers the function will first execute the modifiers. Line 10 is the constructor function of the contract. This function can only be executed once when deploying the contract to Ethereum. In the constructor function, the contract assigns a hard-coded

address to the *owner* variable to restore the owner address. Fallback function (L13) is a specific feature in smart contract as we introduced in Section 4.2.2. When receiving Ethers, *ReceiveEth* will be activated and the contract uses *msg.value* to check the amount of Ethers they received (L16). If the amount that they received not equal to 1 Ether, the transaction will be reverted. Otherwise, the contract records the address of those who send the Ethers (L18). When the balance equals to 10 Ethers, the contract will execute the *getWinner* function and choose one gambler as the winner (L21-22). The function uses *block.hash* and *block.number* to generate a random number. This two block-related APIs is used to obtain block related information. After getting the winner, the contract uses *address.send()* to send Ethers to the winer (L26). *address.send()* is one method to send Ethers. This method will return a boolean value to inform the caller whether the money is successfully sent but do not throw an exception. *address.transfer()* can also be used to send Ethers, but this function will throw an exception when errors happen. Note that, these two functions have *gas limitation* of 2300 if the recipient is a contract account (See Section 4.2.2). *address.call.value()* in Line 39 can be used to send Ethers to a smart contract, similar to *address.send()*. This method also returns a boolean value to inform the caller whether the money is successfully sent but does not throw an exception.

3.2.4 ERC-20 Token

In recent years, thousands of cryptocurrencies have been created. However, most of them are implemented by smart contracts that run on the Ethereum (also called tokens) rather than having their own blockchain system. Ethereum provides several token standards to standardize tokens' behaviors. In this case, different tokens can interact accurately and be reused by other applications (e.g., wallets and exchange markets). The ERC-20 standard [78] is the most popular token standard used on Ethereum. It defines 9 standard interfaces (3 are optional) and 2 standard events. To design ERC-20 compliant tokens, de-

velopers must strictly follow this standard. For example, the standard method *transfer* is declared as “function transfer (address *_to*, uint256 *_value*) public returns (bool *success*)”, which is used to transfer a number of tokens to address *_to*. The function should fire the TRANSFER event to inform whether the tokens are transferred successfully. The function also should *throw* an exception if the message caller’s account balance does not have enough tokens to spend.

3.3 RQ1: Contract defects in Smart Contracts

3.3.1 Motivation

Smart contracts cannot be patched after deploying them to the blockchain. Detecting and removing contract defects is a good way to ensure contacts’ robustness. Since the revolutionary changes of smart contracts compared to traditional softwares, e.g., the gas system, decentralized features, smart contracts might contain many specific defects compared to traditional programs, e.g., Android Apps. To fill this gap, we try to define a set of new smart contract defects from *StackExchange* posts in this section. We give definitions, examples and possible solutions of our defined contract defects specialized for Ethereum smart contracts.

3.3.2 Approach

3.2.1. StackExchange Posts: To define defects for smart contracts, we need to collect issues that developers encountered. Programmers often collaborate and share experience over Q&A site like *Ethereum StackExchange* [175], the most popular and widely-used question and answer site for users of Ethereum. By analyzing posts on *Ethereum StackExchange*, we can identify and define a set of contract defects on Ethereum. In this paper, we crawled 17,128 *StackExchange* posts and analyzed them further.

3.2.2. Key Words Filtering: It is time-consuming to find important information from thou-

sands of Q&A posts. Therefore, we utilized keywords to filter important information from StackExchange posts. To ensure the completeness of our keywords list, two authors of this paper read the solidity documents [173] carefully and recorded the keywords they think are important. After that, they merged the keywords list and used these keywords to filter StackExchange posts. When reading the posts, we added new keywords to enrich our list and filter new posts. We finally used 66 keywords to filter 4,141 posts.

3.2.3. *Manually Filtering*: In this paper, we aim to find Solidity-related smart contract defects. However, the filtered 4,141 posts which contain the keywords might not related to Solidity-related contract defects. Many posts are about the web3 [203], development environment (Remix [162], Truffle [185]), wallet or functionality. We need to remove them from the dataset and only retain posts that are related to contract defects. For example, the title of a post is “Transfer ERC20 token from one account to another using web3”. Although the post contains key words “ERC20”, the posts are related to web3, not Solidity related contract defects. Therefore, we emit it from our dataset. Two authors of this paper, who both have rich experience in smart contract development, manually analysed all of the posts and finally found that a total of 393 posts are related to Solidity-related smart contract defects. The detailed analysis results of these 4,141 posts can be found at: <https://github.com/Jiachi-Chen/TSE-ContractDefects>

3.2.4. *Open Card Sorting*: We followed the card sorting [174] approach to analyze and categorize the filtered contract defects-related posts. We created one card for each post. The card contains the information of defect title, description, and comments. The same two authors worked together to determine the labels of each post. The detailed steps are:

Iteration 1: We randomly chose 20% of the cards. The same two authors first read the title and description of the card to understand the defects that the post discussed. Then, they read the comments to understand how to solve the defects. After that, they discussed the root cause of the defect. If the root cause of the card were unclear, we omitted it from

Table 3.1: Classification scheme.

Category	Description
Gas Limitation	Bugs caused by gas limitation.
Permission Check	Bugs caused by permission check failure.
Inappropriate Logic	There are inappropriate logics inside a contract, which can be utilized by attackers.
Ethereum Features	Ethereum has many new features, e.g., Solidity, Gas System. Developers do not familiar with the differences which might lead to mistakes.
Version Gaps	Errors due to the update of Ethereum or Solidity.
Inappropriate Standard	Ethereum provides several standards, but many contracts do not follow them.

How do I fix this compile error: “throw” is deprecated in favour of “revert()”, “require()” and “assert()”?

I've just downloaded **Mist 0.9.0** and I tried to compile a new contract but it does not let me do it with this error:

1 **“throw” is deprecated in favour of “revert()”, “require()” and “assert()”.**

 throw;

In this code caused by the modifier onlyOwner:

```
/*
 * Ownable
 *
 * Base contract with an owner.
 * Provides onlyOwner modifier, which prevents function from running if it is called!
 */
contract Ownable {
    address public owner;

    function Ownable() {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        if (msg.sender != owner) {
            throw;
        }
    }

    function transferOwnership(address newOwner) onlyOwner {
        if (newOwner != address(0)) {
            owner = newOwner;
        }
    }
}
```

1 Possible duplicate of [Difference between require and assert and the difference between revert and throw](#) – Ismael Sep 10 '17 at 17:33

1 The equivalent code is `require(msg.sender == owner);` – Ismael Sep 10 '17 at 17:36

Figure 3.1: Example of a Card

our card sort. All of the themes are generated during the sorting. After this iteration, the first five categories shown in Table 3.1 are found.

Example of categorizing a card: Fig. 3.1 is an example of a card for a defect reporting post. The card contains three parts, i.e., title, description, and two comments. The two authors first read the title and description of the card to understand the contract defect(s) described by the posts. After that, they read the comments. The first comment gives a link to a previous similar post, and the second comment introduces ideas on how to fix this particular error. From the link, we can determine that the root cause of the error is because “*throw*” is deprecated since Solidity version 0.4.5. Therefore, the defect category for this card is “*Version Gaps*”.

Iteration 2: Two authors independently categorized the remaining 80% of the cards into the initial classification scheme by following the same method, described in iteration 1. During the categorizing process, they found another category named “*Inappropriate Standard*”, which is common in the remaining cards. After that, they compared their results and discussed any differences. Finally, they categorized the defects into 6 themes; the detailed information is shown in Table 3.1. We used Cohen’s Kappa [47] to measure the agreement between the two authors. Their overall Kappa value is 0.82, indicating a strong agreement.

3.2.5. Defining Contract Defects From Posts: After categorizing the filtered posts, we summarized 6 high-level root causes from *StackExchange* posts. Then, the same two authors read the cards again, with the aim to find more detail behaviors for the definition of the contract defects. Finally, we summarized 16 contract defects. Following are two examples:

Example 1. Deprecated APIs: The error described in the Fig. 3.1 is classified into “*Version Gaps*”, which shows the high-level root cause. It is not difficult to find the reason of the error as the user has made use of a deprecated API, i.e., *throw*. We thus conclude

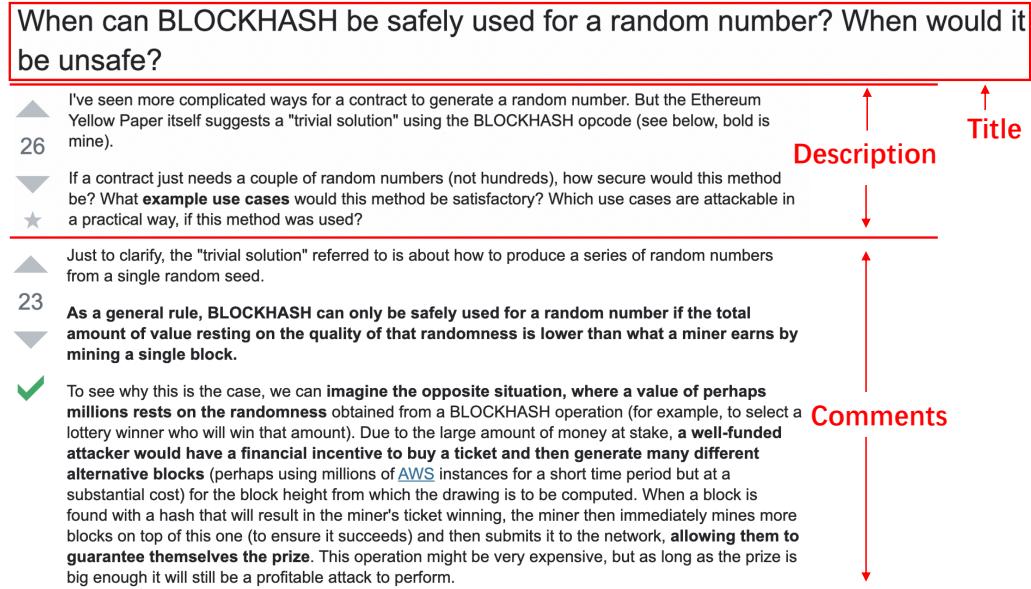


Figure 3.2: Example of a Card for "Block Info Dependency"

that we obtain a contract defect category named “Deprecated APIs”.

Example 2. Block Info Dependency: Fig. 3.2 is another example that belongs to the defect category “Ethereum Features”. From the post, we can determine that if the profit of the controlling contract is higher than what a miner earns by mining a single block (5 ETH), there is a high probability that the contracts will be controlled by the miner. Therefore, using **BLOCKHASH** to generate random numbers is not safe. Finally, we infer a contract defect named “Block Info Dependency” from this card.

3.2.6. Dataset Labeling: In order to assist future studies on smart contract analysis and testing, we manually identified whether the defined contract defects exist in our dataset, which consists of 578 real-world smart contracts. To build this dataset, we first crawled all 17,013 verified smart contracts from Etherscan. Then, for the scalability reasons, we randomly chose 600 smart contracts from these 17,013 contracts. We filtered out 13 smart contracts as they do not contain any functions in their contracts. Finally, we obtained 587 smart contracts with 231,098 lines of code. The total amount of Ethers in these accounts are more than 4 million Ethers.

3.2.7. Defining Contract Defects From Code: During the process of labeling, we found some smart contracts have high similarity but also have some small differences. For example, there are two functions; the only differences for these two functions is the first function denotes a return value but does not return anything. The second function denotes the return value and correctly returns the statement. Therefore, from the difference, we defined a contract defect named “*Missing Return Statement*”. We totally defined 4 contract defects from real-world smart contracts. i.e., *Missing Return Statement*, *Strict Balance Equality*, *Missing Reminder*, and *Greedy Contract*. Finally, we defined 20 contract defects.

3.3.3 Results

In this part, we define and give examples of each defects. We divide these defects to five categories according to their consequences, i.e., *Security defects*, *Performance defects*, *Availability defects*, *Maintainability defects*, and *Reusability defects*. We first give a brief definition of each contract defects in Table 3.2. Then, we give detailed definitions and code examples in the followed paragraphs:

3.3.3.1 Security Defects

In this subsection, we define 9 contract defects that can lead to security issues. These may be exploited by attackers to gain financial benefits or attack vulnerable contracts.

(1) Unchecked External Calls: To transfer Ethers or call functions of other smart contracts, *Solidity* provides a series of external call functions for raw addresses, i.e., *address.send()*, *address.call()*, *address.delegatecall()* [173]. Unfortunately, these methods may fail due to network errors or out-of-gas error, e.g., the 2300 gas limitation of fallback function introduced in Section 3.2. When errors happen, these methods will return a *boolean* value (*False*), but never throw an exception. If callers do not check return values of external calls, they cannot ensure whether code logic is correct.

Example: An example of this defect is given in *Listing 4.1*. In function *getWinner*

Table 3.2: Definitions of the 20 contract defects.

Contract Defect	Definition
<i>Unchecked External Calls</i>	Do not check the return value of external call functions.
<i>DoS Under External Influence</i>	Throwing exceptions inside a loop which can be influenced by external users
<i>Strict Balance Equality</i>	Using strict balance quality to determine the execute logic.
<i>Unmatched Type Assignment</i>	Assigning unmatched type to a value, which can lead to integer overflow
<i>Transaction State Dependency</i>	Using tx.origin to check the permission.
<i>Re-entrancy</i>	The re-entrancy bugs.
<i>Hard Code Address</i>	Using hard code address inside smart contracts.
<i>Block Info Dependency</i>	Using block information related APIs to determine the execute logic.
<i>Nested Call</i>	Executing CALL instruction inside an unlimited-length loop.
<i>Deprecated APIs</i>	Using discarded or unrecommended AIPs or instructions.
<i>Unspecified Compiler Version</i>	Do not fix the smart contract to a specific version.
<i>Misleading Data Location</i>	Do not clarify the reference types of local variables of <i>struct</i> , <i>array</i> or <i>mapping</i> .
<i>Unused Statement</i>	Creating values which never be used.
<i>Unmatched ERC-20 standard</i>	Do not follow the ERC-20 standard for ICO contracts.
<i>Missing Return Statement</i>	A function denote the type of return values but do not return anything.
<i>Missing Interrupter</i>	Missing backdoor mechanism in order to handle emergencies.
<i>Missing Reminder</i>	Missing events to notify caller whether some functions are successfully executed.
<i>Greedy Contract</i>	A contract can receive Ethers but can not withdraw Ethers.
<i>High Gas Consumption Function Type</i>	Using inappropriate function type which can increase gas consumption.
<i>High Gas Consumption Data Type</i>	Using inappropriate data type which can increase gas consumption.

(L23), the contract does not check the return value of *send* (L26), but the array *participants* is emptied by assigning *participatorID* to 0 (L25). In this case, if the *send* method failed, the winner will lose 8 Ethers.

Possible Solution: Using *address.transfer()* to instead *address.send()* and *address.call.value()* if possible, or Checking the return value of *send* and *call*.

(2) DoS Under External Influence: When an exception is detected, the smart contract will rollback the transaction. However, throwing exceptions inside a loop is dangerous.

Example: In line 33 of *Listing 4.1*, the contract uses *transfer* to send Ethers. However, In Solidity, *transfer* and *send* will limit the gas of fallback function in callee contracts to 2,300 gas [173]. This gas is not enough to write to *storage*, call functions or send Ethers. If one of *member[i]* is an attacker's smart contract and the *transfer* function (L33) can trigger an out-of-gas exception due to the 2,300 gas limitation. Then, the contract state will rollback. Since the code cannot be modified, the contract can not remove the attacker from *members* list, which means that if the attacker does not stop attacking, no one can get bonus anymore.

Possible Solution: Avoid throwing exceptions in the body of a loop. We can return a boolean value instead of throwing an exception. For example, using “*if(msg.send(...)* == *false*) *break*;” instead of using “*msg.transfer(...)*”.

(3) Strict Balance Equality: Attackers can send Ethers to any contracts forcibly by utilizing *selfdestruct(victim_address)* API [173]. This way will not trigger the fallback function, meaning the victim contract cannot reject the Ethers. Therefore, the logic of equal balance check will fail to work due to the unexpected ethers send by attackers.

Example: Attackers can send 1 Wei (1 Ether = 10^{18} Wei) to *Contract Gamble* in *Listing 4.1* by utilizing *selfdestruct* method. This method will not trigger fallback function (L13). Thus, the Ethers will not be thrown by *ReceiveEth* (L16). If this attack happens, the *getWinner()* (L23) would never be executed, because the *getWinner* can only be executed

when the balance of the contract is strictly equal to 10 Ethers (L21).

Possible Solution: Since the attackers can only add the amount of the balance, we can use a range to replace “==”. In this case, attackers cannot affect the logic of the programs. Using the defect in Listing 4.1 as an example, we can modify the code in L21 to “*if (this.balance ≥ 10 ether && this.balance < 11 ether)*”

(4) Unmatched Type Assignment: Solidity supports different types of integers (e.g., *uint8*, *uint256*). The default type of integer is *uint256* which supports a range from 0 to 2^{256} . *uint8* takes less memory, but only supports numbers from 0 to 2^8 . Solidity will not throw an exception when a value exceeds its maximum value. The progressive increase is a common operation in programming, and performing an increment operation without checking the maximum value may lead to overflow.

Example: The variable *i* in line 30 of *Listing 4.1* is assigned to *uint8*, because 0 is in range of *uint8* (0-255). If the *members.length* is larger than 255, the value of *i* after 255 is 0. Thus, the loop will not stop until running out of gas or balance of account is less than 0.1.

Possible Solution: Using *uint* or *uint256* if we are not sure of the maximum number of loop iterations.

(5) Transaction State Dependency: Contracts need to check whether the caller has permissions in some functions like *suicide* (L33 in *Listing 4.1*). The failure of permission checks can cause serious consequences. For example, if someone passes the permission check of *suicide* function, he/she can destroy the contract and stole all the Ethers. *tx.origin* can get the original address that kicked off the transaction, but this method is not reliable since the address returned by this method depends on the transaction state.

Example: We can find this defect in line 8 of *Listing 4.1*. The contract uses *tx.origin* to check whether the caller has permission to execute function *suicide* (L35). However, if an attacker uses function *attack* in *Listing 4.4* to call *suicide* function (L35 in *Listing 4.1*), the

permission check will fail. *suicide* function will check whether the sender has permission to execute this function. However, the address obtained by *tx.origin* is always the address who creates this contract (0xdCad...d1D3AD *L12 in Listing 4.1*). Therefore, anyone can execute the *suicide* function and withdraw all of the Ethers in the contract.

Possible Solution: Using *msg.sender* to check the permission instead of using *tx.origin*.

(6) Block Info Dependency: Ethereum provides a set of APIs (e.g., *block.blockhash*, *block.timestamp*) to help smart contracts obtain block related information, like timestamps or hash number. Many contracts use these pieces of block information to execute some operations. However, the miner can influence block information; for example, miners can vary block time stamp by roughly 900 seconds [71]. In other words, block info dependency operation can be controlled by miners to some extent.

Example: In Listing 4.1 line 25, the contract uses *blockhash* to generate which member is the winner. However, the gamble is not fair because miners can manipulate this operation.

Possible Solution: To generate a safe random number in Solidity, we should ensure the random number cannot be controlled by a single person, e.g., a miner. We can use the information of users like their addresses as their input numbers, as their distributions are completely random. Also, to avoid attacks, we need to hide the values we used from other players. Since we cannot hide the address of users and their submitted values, a possible solution to generate a random number without using block related APIs is using a hash number. The algorithm has three rounds:

Round 1: Users obtain a random number and generate a hash value in their local machine. The hash value can be obtained by *keccak256*, which is provided by Solidity. After obtaining the random number, users submit the hash number.

Round 2: After all users submit their hash number, users are required to submit their original random number. The contract checks whether the original number can generate the same hash number.

Round 3: If all users submit the correct original numbers, the contract can use the original numbers to generate a random number.

(7) Re-entrancy: Concurrency is an important feature of traditional software. However, Solidity does not support it, and the functions of a smart contract can be interrupted while running. Solidity allows parallel external invocations using *call* method. If the callee contract does not correctly manage the global state, the callee contract will be attacked – called a re-entrancy attack.

Example: Listing 4.2 shows an example of re-entrancy. The *Attacker* contract invokes *Victim* contract's *withDraw()* function in Line 11. However, *Victim* contract sends Ethers to *attacker* contract (L6) before resetting the balance (L7). Line 6 will invoke the fallback function (L9) of *attacker* contract and lead to repeated invocation.

Possible Solution: Using *send()* or *transfer* to transfer Ethers. *send()* and *transfer* have *gas limitation* of 2300 if the recipient is a contract account, which are not enough to transfer Ethers. Therefore, these two functions will not cause Re-entrancy.

```

1 contract Victim {
2     mapping(address => uint) public userBalannce;
3     function withDraw() {
4         uint amount = userBalannce[msg.sender];
5         if(amount > 0){
6             msg.sender.call.value(amount)();
7             userBalannce[msg.sender] = 0; } } ... }
8 contract Attacker{
9     function() payable{
10        Victim(msg.sender).withDraw(); }
11     function reentrancy(address addr){
12        Victim(addr).withDraw(); } ... }
```

Listing 3.2: Attacker contract can attack Victim contract by utilizing Re-entrancy

(8) Nested Call: Instruction *CALL* is very expensive (9000 gas paid for a non-zero value transfer as part of the CALL operation [213]). If a loop body contains CALL operation but does not limit the number of times the loop is executed, the total gas cost would have a high probability of exceeding the gas limitation because the number of iterations may be high and it is hard to know its upper limit.

Example: In *Listing 4.1*, the function *giveBonus* (line 28) uses *transfer* (L33) which generates *CALL* to send Ethers. Since the *members.length* (L30) does not limit its size, *giveBonus* has a probability to cause out of gas error. When this error happens, this function can not be called anymore because there is no way to reduce the *members.length*.

Possible Solution: The developers should estimate the maximum number of loop iterations that can be supported by the contract and limit these loop iterations.

(9) Misleading Data Location: In traditional programming languages like *Java* or *C*, variables created inside a function are local variables. Data is stored in *memory* and the *memory* will be released after the function exits. In Solidity, the data of *struct*, *mapping*, *arrays* are stored in *storage* even they are created inside a function. However, since *storage* in solidity is not dynamically allocated, storage variables created inside a function will point to the *storage slot*⁵ 0 by default [173]. This can cause unpredictable bugs.

Example: Function *reAssignArray* (L6) in *Listing 4.3* creates a local variable *tmp*. The default data location of *tmp* is **storage**, but EVM cannot allocate storage dynamically. There is no space for *tmp*, but instead, it will point to the storage slot 0 (*variable* in L3 of *Listing 4.3*). For the result, once function *reAssignArray* is called, the variable *variable* will add 1, which can cause bugs for the contract.

Possible Solution: Clarifying the data location of *struct*, *mapping*, and *arrays* if they are created inside a function.

```

1 pragma solidity ^0.4.25; /* Unspecified Compiler Version */
2 contract DefectExample{
3     uint variable;
4     uint[] investList;
5     function() payable{}
6     function reAssignArray(){
7         /* Misleading Data Location */
8         uint[] tmp;
9         tmp.push(0);
10        investList = tmp;
11        function changeVariable(uint value1, uint value2){
12            /* Unused Statement */
13            uint newValue = value1;

```

⁵Each storage variables has its own storage slot to identify its position.

```

14     variable = value2;
15     /* High Gas Consumption Function Type */
16     function highGas(uint[20] a) public returns (uint){
17         return a[10]*2;
18     function lowGas(uint[20] a) external returns (uint){
19         return a[10]*2; }
```

Listing 3.3: DefectExample

```

1 contract attacker{
2     ...
3     function attack(address addr, address myAddr){
4         Gamble gamble = Gamble(addr);
5         gamble.suicide(myAddr); }
```

Listing 3.4: An attacker contract by utilizing Transaction State Dependency.

3.3.3.2 Availability Defects

We define 4 contract defects related to availability. These may not be utilized by attackers but are bad designs for contracts that can lead to potential errors or financial loss for the caller.

(1) Unmatched ERC-20 Standard: ERC-20 Token Standard [78] is a technical standard on Ethereum for implementing tokens of cryptocurrencies. It defines a standard list of rules for Ethereum tokens to follow within the larger Ethereum ecosystem, allowing developers to predict the interaction between tokens accurately. These rules include how the tokens are transferred between addresses and how data within each token is accessed. The function name, parameter types and return value should strictly follow the ERC20 standard. ERC-20 defines 9 different functions and 2 events to ensure the tokens based on ERC20 can easily be exchanged with other ERC20 tokens. However, we find that many smart contracts miss return values or miss some functions.

Example: *transfer* and *transferFrom* are two functions defined by ERC20. They are used to transfer tokens from one account to another. ERC20 defines that these two functions have to return a *boolean* value, but many smart contracts miss this return value, leading to errors when transferring tokens.

Possible Solution: Checking that the contract has strictly followed the ERC20 standard.

(2) Missing Reminder: Other programs can call smart contracts through the contracts' *Application Binary Interface (ABI)*. ABI is the standard way to interact with contracts in the Ethereum ecosystem, both from outside the blockchain and for contract-to-contract interaction. However, the ABIs can only tell the caller what the inputs and outputs of a function are, but it will not inform them whether the function call is successful or not. Throwing an event to notify a caller whether the function is successfully executed can reduce unnecessary errors and gas waste.

Example: A typical scenario of this contract defect is missing reminders when receiving Ethers. In *Listing 4.1*, users may not understand the game rules clearly, and send Ethers which not equal to 1 Ether (line 16-17). However, the smart contract will check whether the received Ether is equal to 1 Ether, then the Ether will return back. There are several reasons for invoking failures. For example, the user may mistakenly believe the error is caused by network and resend the Ethers, which can lead to gas waste. Adding reminders (throwing events) to notify caller whether some functions are successfully executed can avoid unnecessary failure.

Possible Solution: Adding reminders for functions that are interacting with the outside.

(3) Missing Return Statement: Some functions denote return values but do not return anything. For these, EVM will add a default return value when compiling the code to bytecode. Since the callers may not know the source code of the callee contract, they may use the return value to handle code execution and lead to unpredictable bugs.

Example: Function *giveBonus* (L28) in *Listing 4.1* declares the return type *bool*, but the function does not return *true* or *false*. Then, EVM will assign the default return value as *false*. If developers call this function, the return value will always be the *false* and some functions in the caller contracts may never be executed.

Possible Solution: Adding the return statements for each function.

(4) Greedy Contract: A contract can withdraw Ethers by sending Ethers to another address or using `selfdestruct` function. Without these withdraw-related functions, Ethers in contracts can never be withdrawn and will be locked forever. We define a contract to be a greedy contract if the contract can receive ethers (contains payable fallback function) but there is no way to withdraw them.

Example: In *Listing 4.3*, the contract has a *payable fallback function* in line 5, which means this contracts can receive Ethers. However, the contracts cannot send Ethers to other contracts or addresses. Therefore, the Ethers in this contract will be locked forever.

Possible Solution: Adding withdraw method if the contract can receive Ethers.

3.3.3.3 Performance Defects

We define 3 contract defects related to performance. The contracts with these defects can increase their gas cost.

(1) Unused Statement: If function parameters or local variables do not affect any contract statements nor return a value, it is better to remove these to improve code readability.

Example: function parameter `value1` and local variable `newValue` in function `changeVariable` (L11 of *Listing 4.3*) are useless, because they never affect contract statements nor return values. Although the compiler will remove these useless statements when compiling source code to binary code, these can reduce contract readability.

Possible Solution: Removing all unused statements in the contract to make it easier to read.

(2) High Gas Consumption Function Type: For *public* functions, Solidity immediately copies function arguments (*Arrays*) to *memory*, while *external* functions can read directly from *calldata* [213]. Memory allocation is expensive, whereas reading from *calldata* is cheap. To lower gas consumption, if there are no internal functions call this function and

the function parameters contain *array*, it is recommended to use *external* instead of *public*.

Example: In *Listing 4.3*, function *highGas* (L16) and function *lowGas* (L18) have the same capabilities. The only difference is that *highGas* is modified by *public* which can be called by external and internal functions. *lowGas* is modified by *external* which can only be called by external. Calling function *highGas* costs 496 gas while calling *lowGas* only costs 261 gas.

Possible Solution: Using *external* instead of *public* if the function can only be called by external.

(3) High Gas Consumption Data Type: *bytes* is dynamically-sized byte array in Solidity, *byte[]* is similar with *bytes*, but *bytes* cost less gas than *byte[]* because it is packed tightly in *calldata*. EVM operates on 32 bytes a time, *byte[]* always occupy multiples of 32 bytes which means great space is wasted but not for *bytes*. Therefore, *bytes* takes less storage and costs less gas. To lower gas consumption, it is recommended to use *bytes* instead of *byte[]*.

Example: Replacing *byte[]* by *bytes* can save a small amount of gas for each function call. However, as the contract is called more times, a large amount of gas can potentially be saved.

Possible Solution: Using *bytes* instead of *byte[]*.

3.3.3.4 Maintainability Defects

We define 2 contract defects related to maintainability. These contract defects can shorten the life cycle of the contract.

(1) Hard Coded Address: Since we cannot modify smart contracts after deploying them, hard coded addresses can lead to vulnerabilities.

Example: There are two main kinds of errors this contract defect can lead to. The first is *Illegal Address*. Ethereum uses a mixed-case address checksum to verify whether

an address is legal or not. The rule is defined in EIP-55 [192]. There is an error address in line 12 of Listing 4.1. The owner address is an illegal address, the last bit of the address should be ‘F’, but by mistake, it becomes ‘D’. The illegal address makes no one that can withdraw the amount of this contract. The second is *Suicide Address*. *selfdestruct* function (L36) can remove the code from the blockchain and make the contract become a suicide contract, but it is potentially dangerous. If someone sends Ether to suicide contracts, the Ether will forever be lost. *receiver* (L38) is a smart contract who contains *selfdestruct* function. Its address is hardcoded in line 38 of Listing 4.1 and cannot be modified. If the *receiver* performed the *selfdestruct* function, it will become a suicide contract. All the Ethers sent to *receiver* will be lost forever.

Possible Solution: Removing the hard coded addresses and inputting the addresses as function parameters.

(2) Missing Interrupter: When bugs are detected by attackers, they can attack the contracts and steal their Ethers. The DAO lost \$50 million Ethers due to a bug in the code that allowed an attacker to draw off the Ethers [54] repeatedly. The interrupter is a mechanism to stop the contract when bugs are detected. We cannot modify contracts after deploying them to the blockchain. However, if a contract contains interrupter, the owner of the victim contract can reduce their losses.

Example: When bugs are found in *Listing 4.1*, the Ethers on the contract can be stolen by attackers. Fortunately, the contract contains an interrupter on *suicide function* (L35). So, the owner of the contract can call *suicide*. Then, the remain Ethers will be sent to the given address. After fixing the bugs, the contracts can be redeployed.

Possible Solution: The easiest interrupter is adding a *selfdestruct* function [173], Ethers on the contracts can be withdrawn and the contracts destroyed when attacks happen. Adding an interrupter to the contracts, if the contract holds a large amount of Ethers.

3.3.3.5 Reusability Defects

We define 2 contract defects related to reusability. These contract defects can increase the difficulty of code reuse.

(1) Deprecated APIs: Solidity is a young and evolving programming language. Some APIs will be discarded or updated in the future. In this case, Solidity documentation usually uses warning to inform developers that some APIs will be deprecated in the future. These APIs might still be supported by the current compiler version. However, if developers use these APIs, they might need to refactor the code for the code reuse, which leads to resource waste.

Example: *CALLCODE* operation will be discarded in the future [173], *throw*, *suicide*, *sha3* are replaced by *revert*, *selfdestruct*, *keccak256* respectively in the recent version.

Possible Solution: Following the latest Solidity document and using the latest APIs.

(2) Unspecified Compiler Version: Different versions of Solidity may contain different APIs/instructions. In Solidity programming, multiple APIs only be supported in some specific versions. If a contract do not specify a compiler version, developers might encounter compile errors in the future code reuse because of the version gap.

Example: In the first line of *Listing 4.3*, *pragma solidity^0.4.25* means that this contract supports compile version 0.4.25 and above (except for v0.5.0) while *pragma solidity 0.4.25* means that the contract only supports compile version 0.4.25. Since it is hard to foresee the language constructions in the future version, it is recommended to indicate a specific compiler version to avoid unnecessary bugs.

Possible Solution: Fixing the compiler version used by the contract.

Table 3.3: Survey results, distributions, and impacts of the 20 contract defects.

Contract Defect	Distribution	Score	#Defects	Impacts
Unchecked External Calls		4.50	25 (4.26%)	IP3
DoS Under External Influence		4.31	6 (1.02%)	IP2
Strict Balance Equality		4.28	5 (0.85%)	IP2
Unmatched Type Assignment		4.42	22 (3.75%)	IP2
Transaction State Dependency		4.54	5 (0.85%)	IP1
Reentrancy		4.66	12 (2.04%)	IP1
Hard Code Address		4.10	84 (14.31%)	IP3
Block Info Dependency		4.05	42 (7.16%)	IP3
Nested Call		4.45	13 (2.21%)	IP2
Deprecated APIs		4.06	247 (42.08%)	IP5
Unspecified Compiler Version		3.84	532 (90.63%)	IP5
Misleading Data Location		4.28	1 (0.17%)	IP2
Unused Statement		4.04	10 (1.70%)	IP5
Unmatched ERC-20 standard		4.29	45 (7.67%)	IP4
Missing Return Statement		4.16	263 (44.80%)	IP4
Missing Interrupter		4.06	523 (89.10%)	IP4
Missing Reminder		4.06	27 (4.60%)	IP4
Greedy Contract		4.25	6 (1.02%)	IP3
High Gas Consumption Function Type		4.08	422 (71.89%)	IP5
High Gas Consumption Data Type		4.07	0 (0%)	IP5

3.4 RQ2: Practitioners' Perspective

3.4.1 Motivation

To validate whether our defined contract defects are harmful, we created an online survey to collect opinions from real-world smart contract developers.

3.4.2 Approach

3.4.2.1 Validation Survey

We followed the instructions of Kitchenham et al. [120] for personal opinion surveys and utilized an anonymous survey [187] to increase response rates. Respondents can choose to leave an email address, as all respondents could choose to take part in a raffle to win

two \$50 Amazon gift cards. We first conducted a small scale survey to test and refine our questions. These participants give feedback about: (1) whether the expression of the contract defects is clear and easy to understand, and (2) whether the length of each question is suitable. Finally, we modified our survey based on the feedback we collected.

3.4.2.2 Survey Design

To help respondents better understanding the aim of our survey, we explained what is contract defect at the beginning of the survey and gave detailed definitions and examples of the 20 contract defects in related questions. We first captured the following pieces of information to collect demographic information about the respondents:

Demographics:

- Professional smart contract developer? : Yes / No
- Involved in open source software development? : Yes / No
- Main role in developing smart contract.
- Experience in years
- Current country of residence
- Highest educational qualification

Examples of Contract Defects: Next, we gave detailed definitions and examples of the 20 contract defects. We asked respondents to rate the importance of these contract defects, i.e., removing them can improve the security, reliability, or usability of a project. Since some of the defined contract defects are not easy to understand, we added an option "*I don't understand*" to ensure results are reliable. Finally, we give each question six options (i.e., Very important, Important, Neutral, Unimportant, Very unimportant and I don't understand). We also give each question a textbox to enable respondents to give their opinions.

Other Questions: We give a textbox so respondents can tell us if they have any other comments, questions, or concerns.

3.4.2.3 Recruitment of Respondents

In order to get a sufficient number of respondents from different backgrounds, we first sent our survey to our partners who are working or study in world-famous companies or academic institutions. We sent our email to 1489 practitioners who contribute to open source smart contract related projects on GitHub. All respondents could enter their email to take part in a raffle to win two \$50 Amazon gift cards.

3.4.3 Results

We totally received 138 responses (The response rate is about 9.27%) from 32 different countries, and we received 84 comments on our defined contract defects. 113 (81.88%) of these respondents are involved in open source software development efforts. The top two countries in which the respondents reside are China (38.41%) and USA (7.97%). The average years of experience in developing smart contracts are 1.95 years. Since the Ethereum was published only in late 2015, we believe the average year of 1.95 years shows that the respondents have good experience in developing smart contracts. We do not remove the feedback from developers with little experience as their feedback is also very useful as they might be the ones actually authoring the contracts with defects. Among these respondents, 89 (64.49%), 17 (12.32%), 16 (11.59%), 7 (5.07%) described their job roles as development, testing, management and security audit respectively. The other 9 responses said they have multiple roles.

Table 3.3 shows the results of our survey. The first column indicates each contract defect and the second column illustrates the distribution of respondents' choice. The distribution is from "Very unimportant" (left-most red bar) to "Very important" (right-most green bar). To clearly show the result, we give each option a score and count the weighted average

score which is shown in the third column. To be specific, we give “very important” a score 5 and give “very unimportant” a score 1.

We received very positive feedback from developers with **almost all contract defects’ scores are larger than 4, and the average score is 4.22**. The score of “Unspecified Compiler Version” is 3.84 but it is also a positive score. To understand the reasons, we reviewed comments about this defect. We found that many developers who voted “unimportant” mentioned the difference among different minor versions in the same major version (e.g., 0.4.19 and 0.4.20) is small. However, they admitted that the difference among different major versions (e.g., 0.4.0 and 0.5.0) is significant. Some developers gave comments that removing this contract defect is very important when they want to reuse code in the future. Besides, we also found many examples from StackExachange posts that many developers failed to compile the contracts because these contracts do not specify their compiler versions. Therefore, we believe this defect is important on code reuse.

“Missing Interrupter”, “Missing Reminder”, and “Unspecified Compiler Version” received the top three most negative feedbacks (“Unimportant” and “Very unimportant”). For **“Missing Interrupter”**, 5 developers mentioned that adding interrupters in smart contracts will ensure the benefits for the smart contract owners. However, such a back-door mechanism may cause users to distrust the contracts. This worry makes sense, but we believe it can be fixed if the contract owners add some insurance mechanism to the contracts. For example, they can define rules to detect abnormal states, and the back-door mechanism can only be executed when the abnormal state is detected. For **“Missing Reminder”**, we did not receive comments from respondents who chose negative options. We sent emails to the developers who gave their email address and received three feedbacks. All mentioned that the smart contracts they developed are used inside their companies. They will write a detailed document of each function. If other developers in their companies have problems, then they fix the problems using face to face discussion. Therefore, this contract defect is

not important for them. However, we believe that if the smart contracts are deployed on Ethereum and other developers can call the functions, removing this contract defect can reduce potential problems. For “**Unspecified Compiler Version**”, we found 4 developers who gave negative feedback mention that there are only very few differences between the versions under the same large version, e.g., between 0.4.21 and 0.4.22. However, we do not agree with this observation. As we have mentioned, even if two versions only have a small difference, but it is hard to foresee language constructions in the future version. Thus, it is possible that there might be two versions that contain a big difference in the future. Besides, refuting this feedback, version 0.4.0 (the first version of 0.4+) and 0.4.25 (the latest version of 0.4+) do indeed have big differences, as many APIs like *throw* have been deprecated.

We also received 18 negative comments for the other 7 smart contract defects. The negative comments of “**Unmatched type assignment**”, “**Re-entrancy**”, “**Hard Code Address**”, “**Misleading Data location**”, and “**High Gas Consumption Function Type**” all mentioned that these contract defects have been removed in the latest version of Solidity. However, when developers deploy smart contracts to Ethereum, they need to choose a Solidity version by themselves. Most developers choose old versions of Solidity instead of the latest version [73]. This means that these defects are still potentially harmful. “**Strict Balance Equality**” received 3 negative comments. Two developers said this is not a common case, and another developer said receiving Ether cannot be prevented. Thus, it might be hard to avoid exact balance checks in some situations. We admit that defect is not common in Ethereum smart contracts. However, this defect is still harmful and can open up another attack vector to attackers. Developers can use other logic, such as “ $\geq \&& <$ ” to avoid “ $=$ ” (see possible solution for this defect introduced in Section 4.3.3.1). “**Unmatched ERC-20 standard**” received 2 negative comments. These comments mentioned that this contract defect could only be used for ICO smart contracts, which limits its usage scenario. How-

ever, ICO smart contracts are very popular in Ethereum, and they hold a large amount of Ethers. Thus, we believe this defect category is still useful.

Certainly, We receive many positive comments. Some positive comments we received included:

- *You provide a very good summary of some very important security checkpoints.*
- *Those controls and warnings should be integrated into the Solidity compiler, and displayed in common development tools like Remix and Truffle.*
- *It is nice to have such a summary of these vulnerabilities among smart contracts, I think it would be very helpful for the blockchain practitioners as well as the researchers.*
- *These suggestions above are very useful to avoid various kinds of flaws.*
- *Generally speaking, all of these contract defects can lead to serious problems. I learned a lot from this survey.*

3.5 RQ3: Distribution and Impact of Contract Defects

3.5.1 Motivation

To help developers and researchers better understand the impacts of our defined smart contract defects, we summarized 5 impacts and manually label 587 smart contracts to show their distribution in the real-world smart contracts. Our labeling results provided ground truth for future studies on smart contract defects detection. As it is not easy to remove all contract defects due to tight project schedules or financial reasons, the impacts and distributions of different contract defects can help developers decide which defect should be fixed first.

3.5.2 Approach

Distribution: We obtained 587 smart contracts from real-world Ethereum accounts. The first and last authors of this paper independently read these smart contracts and determined whether the contracts contained our defined contract defects. They each have three-year experience on smart-contract-based development and have published three smart-contract-related papers together. Their overall Kappa value was 0.71, which indicates substantial agreement between them. After completing the labeling process, they discussed their disagreement and gave a final result. Finally, we generate a dataset which shows the distribution of the contract defects we defined.

Impact Level (IL) Definition: To summarize the impacts of each contract defect, we consider from three dimensions, i.e., contract dimension (unwanted behavior), attacker dimension (attack vector), and user dimension (usability), which can be found on Table 3.5.

The contract dimension focuses on the severity level of the contract defect. From our survey, 27 developers claimed that defects, e.g., *Reentrancy*, *Dos Under External Influence*, might enable attackers to attack the contracts, and 9 of them mentioned that attackers can utilize defects like *Reentrancy* to stole all the Ethers on the contract. Also, 16 developers agree that defects, e.g., *High Gas Consumption Function Type*, *Deprecated APIs*, will not affect the normal running of the contract, but have bad effects for the users or callers. From the *StackExchange* posts, we can also find the comments of the posts mentioned that the defects could lead to the crashing, losing all Ethers, and losing a part of the Ethers. Finally, we totally find the defects can lead to 5 common consequences to the contracts. They are crashing, being controlled by attackers, losing all Ethers, losing a part of the Ethers, normal running but have bad effects for the users or caller. We have split the 5 common consequences into three severity levels, i.e., *critical*, *major*, and *trivial*. *Critical* represents contract defects, which can lead to the crashing, being controlled by attackers, or can lose all Ethers. *Major* represents the contract defects that can lead to the loss of a

part of the Ethers. Contracts with *trivial* severity level will not affect the normal running of the contract.

The attacker dimension focuses on attackers' behaviors. Since financial services are the most attractive targets for attackers, we believe that if attackers can use the defects to steal Ethers, the impact level should be higher. Whether the defect can be triggered by attackers is also an important aspect.

The users dimension focuses on the external influence of the defects. This dimension contains three aspects, i.e., potential errors for caller, gas waste, and mistakes on code reuse. Some defects do not affect the normal running of the contracts. However, they can lead to the errors of the caller programs. Some defects can also increase the gas costs of the callers and users. As code reuse is important in software engineering, some defects can make the contracts hard to be understand and reuse.

We only consider the worst-case scenario outcome for each contract defect, even though some defects will have different impact levels under different application scenario. We use *Hard Code Address* as an example. In most situations, Hard Code Address will not lead to the loss of Ethers. However, if the hard-coded address is a self-destructed contract, a contract with this defect can lose a part of its Ethers. Thus we consider *Hard Code Address* can lead to major unwanted behavior.

After defining the three dimensions, we map each contract defect onto one or more. The detailed results are shown in Table. We found there are 5 common types of distribution. According to the distribution, we summarized 5 impact levels and assigned each contract defect to have one impact level.

3.5.3 Results

We use Table 3.5 to clarify the difference between each impact level. IP1 is the highest, and IP5 is the lowest. Contract defects with impact level 1-2 can lead to critical unwanted

Table 3.4: Features of Each Contract Defects

Contract Defects	Unwanted Behavior			Attack Vector		Usability		
	Critical	Major	Trivial	Triggered by External	Stolen Ethers	Potential Errors for Callers	Gas Waste	Mistakes on Code Reuse
Unchecked External Calls		✓						
Dos Under External Influence	✓			✓				
Strict Balance Equality	✓			✓				
Unmatched Type Assignment	✓			✓				
Transaction State Dependency	✓			✓	✓			
Reentrancy	✓			✓	✓			
Hard Code Address		✓						
Block Info Dependency		✓		✓				
Nested Call	✓			✓				
Deprecated APIs			✓				✓	✓
Unspecified Compiler Version			✓				✓	✓
Misleading Data Location	✓			✓				
Unused Statement			✓				✓	✓
Unmatched ERC-20 standard			✓			✓		
Missing Return Statement			✓				✓	✓
Missing Interrupter			✓			✓		
Missing Reminder			✓			✓		
Greedy Contract	✓							
High Gas Consumption Function Type			✓				✓	✓
High Gas Consumption Data Type			✓				✓	✓

Table 3.5: Features of Each Impact Level

Impact Level	Unwanted Behavior			Attack Vector		Usability		
	Critical	Major	Trivial	Triggered by External	Stolen Ethers	Potential Errors for Callers	Gas Waste	Mistakes on Code Reuse
IP1	✓			✓	✓			
IP2	✓			✓				
IP3	T1	T2		T2				
IP4			✓			✓		
IP5			✓				✓	✓

behaviors, like crashing or a contract being controlled by attackers. Contract defects with impact level 3 can lead to major unwanted behaviors, like lost ethers. Impact level 4-5 can lead to trivial problems, e.g., low readability, which would not affect the normal running of the contract.

The detailed definition of the five impact levels are as follows:

Impact 1 (IP1): The smart contracts containing the related contract defects can lead to critical unwanted behaviors. Unwanted behaviors can be triggered by attackers, and they can make profits by utilizing the defects.

Impact 2 (IP2): The smart contracts containing the related contract defects can lead to critical unwanted behaviors. Unwanted behaviors can be triggered by attackers, but they cannot make profits by utilizing the defects.

Impact 3 (IP3): There are two types of IP3. *Type 1*: The smart contracts containing the related contract defects can lead to critical unwanted behaviors, but unwanted behaviors cannot be triggered externally. *Type 2*: The smart contracts containing the related contract defects can lead to major unwanted behaviors. The unwanted behaviors can be triggered by attackers, but they cannot make profits by utilizing the defects.

Impact 4 (IP4): The smart contracts containing the related contract defects can work normally. However, the contract defects can lead to potential risks of errors when outside programs call the contracts.

Impact 5 (IP5): The smart contracts containing the related contract defects can work normally and will not lead to the errors for the callers. However, the contract defects can lead to gas waste, and make the contracts hard to understand and reuse.

Table 3.3 lists the detailed distribution of each contract defect (the fourth column) in our dataset and its related impact (the last column). We find the distribution for Impacts 1 – 5 to be 2.90%, 7.16%, 27.09%, 93.86%, 99.14%, respectively. Note that one smart contract can have multiple defects of different impacts simultaneously.

“Unspecified Compiler Version” is the most common contract defect in our dataset (90.63% contracts contain this defect). We also found that this contract defect is the most popular one among the 20 defects when we analyze StackExchange posts. Many developers want to reuse the contracts but encounter compiler errors. These contracts usually do not specify a compiler version. In this case, developers have to try different compiler versions or refactor the code, which increases the workload for code reuse.

“Missing Interrupter” is also very popular in our dataset (89.1% contracts contain this defect). This defect receives the greatest number of comments in our survey. On the one hand, developers admit that adding interrupter is important for contracts when emergencies happen. On the other hand, some developers also worried that the interrupter could lead to distrust by the contract users. Better understanding attitudes to this defect may need further research effort. For example, researchers can design a survey for developers to investigate the reasons why they add or do not add interrupters. By knowing the reason why developers do not add it, researchers might design a better method to implement interrupter. By knowing the reason why developers add interrupters, researchers can investigate whether contracts with interrupters in our dataset are consistent with these reasons, and what are the most popular reasons.

99.82% of smart contracts in our dataset contain at least one contract defect of the impact 4 or impact 5. These contract defects will not affect the normal running of the contracts, but it may have unpredictable impacts to the caller or code reuse. The distribution may illustrate that the developers focus more on the functionality but do not consider the code reuse or handle unpredictable behaviors caused by attackers. This finding is similar to Chen et. al [43]. They found that 96% of smart contracts are involved in no more than 5 transactions, and they are not be used anymore, indicating that many developers do not consider future reuse of these contracts.

About 32.03% of smart contracts contain contract defects at levels 1-3, which can lead

to unwanted behaviors. However, we found that only 7.33% of smart contract contains defects that can lead to critical unwanted behaviors, e.g., crashing or being controlled by attackers.

We also found that ERC-20 related smart contracts are the most popular (36.11%) in Ethereum. However, 21.22% of them do not strictly follow the ERC-20 standards. We did not find any smart contracts which contain *High Gas Consumption Data Type*. Since the size of our dataset is limited, and this contract defect has related posts on *StackExchange*, this contract defect might exist if we investigate more contracts. In summary, our findings showed that defined contract defects are very common in real-world smart contracts.

3.6 Discussion

In this section, we first give the implications of our work for researchers, practitioners and educators. Then, we list three challenges for future research on automatic contract defect detection.

3.6.1 Implications

For Researchers: *Research Guidance.* In this paper, we defined 20 contract defects. Several previous studies analyzed some of them. We have investigated whether there are existing tools that can detect some of the contract defects identified by our work. We show the results in Table 3.6. We first collected the titles of papers which were published at CCS, S&P, USENIX Security, NDSS, ACSAC, ASE, FSE, ICSE, TSE, TIFS, and TOSEM from 2016 to 2019, since Ethereum went live on July 30, 2015 [208]. Then, we used the keywords “smart contract”, “Ethereum”, “blockchain”, “Contracts” to search for papers which are related to the smart contract technology. After that, we read the abstract of each paper to verify its relevance. Finally, we found a total of 4 related papers (i.e., Oyente [131, 132], Zeus [113], Maian [146] and Contractfuzzer [111]). We provide a description of these

four tools in Section 3.8. We find that 7 contract defects can be detected by these existing tools and most of them are security related defects. These tools focus more on the security aspects but do not consider the other two aspects considered as equally important by practitioners. Therefore, researchers can pay more attention to developing tools that can detect the other 13 contract defects.

Behavior vs. Perception [57]. The belief of whether a contract defect is important or not may result in prioritizing testing effort. The survey results and contract defect distribution shown in Table 3.3 can help us investigate whether the practitioners' perception is consistent with their behavior. We find that the top two most frequent contract defects are 'Unspecified Compiler Version' and 'Missing Interrupter' (according to the column *No. Defects* in Table 3.3). Their survey scores are also the lowest (3.92 and 4.0 according to the column *Score* in Table 3.3), indicating that practitioners do not perceive them as important as other defects, and thus they pay less attention to them in practice which causes them to appear more than other contract defects. The appearance of these two contract defects is consistent with practitioners' perception. However, there are many inconsistent examples. According to the definition of 5 impacts introduced in Section 4.5.3, it is clear that IP1 can cause the most serious problems compared to other impacts. We find the 'Unchecked External Calls' has the second highest survey score (4.64), which shows that developers think this defect is very important. However, its impact is IP3, which shows that there is an inconsistency between the practitioners' perception (high survey score) and their behavior (medium impact to the project). Future contract defect detection tools should provide rationales that explicitly describe the connection between contract defects and its impact. This could assist developers better prioritize testing efforts, and understand the detection results well.

Contract Defects in Other Smart Contract Platforms. We propose a method which summarizes contract defects from online posts. Our study focused on defining contract defects for

Table 3.6: Tools that detect some contract defects identified by our study.

Contract Defects	Tools
Unchecked External Calls	Oyente, Zeus, Contractfuzzer
Reentrancy	Oyente, Zeus, Contractfuzzer
Block Info Dependency	Oyente, Zeus, Contractfuzzer
Transaction State Dependency	Zeus
DoS Under External Influence	Zeus
Unmatched Type Assignment	Zeus
Greedy Contract	Maian

Ethereum smart contracts, but the same method can be applied to other popular blockchain platforms, e.g., EOS [65], Hyperledger [105]. These blockchain platforms also support the running of smart contracts and have their unique features. There are thousands of posts on *StackExchange* related to these platforms. Researchers can analyze the related posts and find specific features and contract defects of these smart contract platforms. Our work defined 20 contract defects and provide a dataset which identifies these contract defects on 587 contract accounts, which point out a new direction for future research. For example, researchers can develop automatic contract defect detection tools, and our dataset can be used as ground truth to validate the performance of these tools.

For Practitioners: We are the first to conduct an empirical study by analyzing many online StackExchange posts to understand and define contract defects for smart contracts, and utilize an online survey to validate the acceptance of the defined contract defects among real-world developers. Our results showed that most of the smart contracts in our dataset contained at least one of the defined contract defects. The results may indicate that developers do not consider future use and handle unpredictable attacks. However, since the smart contracts are immutable to patch, the consideration of future use and unpredictable attacks is very important. We also concluded 5 impacts of the defined contract defects to help practitioners better understand the consequences. The defined contract defects can be regarded as a coding guidance for practitioners when they develop smart contracts. By removing the defined contract defects, they can develop robust and well-designed smart

contracts.

Developing contract defect detection tools is also a good direction. Our online survey received many comments from managers of smart-contract-related companies, some listed in Section 4.4.3. They showed much interest in developing and using related tools and highlighted that such detection tools should be integrated into Solidity compiler and development tools.

For Educators: Educators should emphasize the importance of removing contract defects before deploying smart contracts to blockchain. A survey [1] shows that more than 20% of top 50 universities are offering blockchain courses until Oct. 2018. However, most courses focus on teaching basic grammar rule of Solidity programming or blockchain related knowledge but ignore other concerns (security, architecture, usability). The distribution of the defined contract defects also indicates that many developers do not realize the importance for the reuse of smart contracts and handling unpredictable attacks. Educators can improve such conditions by helping students to better understand the impacts of the contract defects. Thus, it is highly recommended that educators pay more attention to teaching contract defect related problems for smart contract development.

3.6.2 Challenge in Detection Contract Defects

We point out three challenges to give a guideline for future research on automatic contract defect detection.

(1) Program Understanding. Some contract defects do not have a specific pattern, which increase the difficulty of automatic detection. For example, there are multiple methods to implement interrupter for the contracts. Developers can use *selfdestruct* function to kill the contract. They can also write a method to stop the contract when attack happens. To detect these kinds of contract defects, we need to understand the smart contracts. However, automatically understanding code is not easy.

(2) Bytecode Level Detection. When deploying a smart contract to Ethereum, EVM will compile the source code to the bytecode and the bytecode will be stored on the blockchain. Everyone can check the bytecode of the smart contracts, but source code may not be visible to the public. Smart contracts usually call other contracts, but the callee contracts may not open their source code to inspection. In other words, they do not know whether the smart contract they called is safe or not. Therefore, detecting contract defects through bytecode is very important because each smart contract's bytecode can be found on Ethereum but only around 0.45% of smart contracts have opened up their source code by Jan. 2019 [74]. However, it is not easy to detect contract defects from bytecode level as it loses the most semantic information.

(3) EVM Operation. When compiling a smart contract to bytecode, EVM will optimize the source code, which means some information will be removed or optimized, so it is hard to know the original information on the source code. For example, detecting whether a function has return value on source code level is straightforward. However, it is not easy to detect it at bytecode level as even we do not add a return value for a function, the EVM will add a default value for it. Therefore, we cannot know whether the return value is added by EVM or developers.

3.6.3 Possible Detection Methods

In this section, we discuss possible detection methods for each of the contract defects that we have defined. Since 7 defects shown in Table 3.6 have already been detected by previous tools, we only discuss the remaining 13 defects.

3.6.3.1 Bytecode Level Detection

Detecting contract defects by bytecode is important for smart contracts in Ethereum, as all the bytecode of the contracts can be found on the Ethereum, but only less than 1% contracts have open source code. To detect contract defects by bytecode, the defects should have

regular patterns. For example, *Nested Call* can be found in a loop which does not limit its loop times and contains the *CALL* instruction. *Missing Interrupter* does not have a regular pattern, as there are multiple ways to realize interrupter. To the best of our knowledge, we have found 6 contract defects that can be detected by bytecode among our 13 smart contract defects. A common method to detect defects by bytecode is using symbolic execution as it can statically reason about a program path-by-path [131].. The method usually converts bytecode to the opcode and splits them into several blocks.⁶ A basic block is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit. Then, we can symbolically execute the instruction and construct a control flow graph (CFG) for each contract, which can be used to detect the contract defects.

(1) Nested Call: After obtaining the CFG, we can identify which blocks belong to loops. If the loop body contains *CALL* instructions and does not limit its loop iterations, the loop contains a *Nested CALL* defect.

(2) Strict Balance Equality: To get the balance of the contract, the contract will generate a *BALANCE* instruction. We can start from this instruction; If a *BALANCE* instruction is read by *EQ* (the *EQ* instruction is used to compare whether two values are equal), it means there is a strict balance equality check. If this check happens at a conditional jump expression, it means this contract contains a Strict Balance Equality defect.

(3) Hard Code Address: Addresses of Ethereum strictly follow the EIP55 [192] standard. We need to identify whether the opcode contains a 20-byte-value and follow the EIP55 standard. The default bytecode stored on Ethereum is called runtime bytecode, which does not contain the constructor function. However, many hard code addresses are stored in the constructor function. To obtain the constructor function, we can check the value of the first transaction of the contract.

(4) Unmatched ERC-20 standard: The ERC-20 standard contains 9 functions (3 are

⁶A basic block is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit.

optional). From bytecode, we can get the hash value of each function. The hash value is obtained from its function name and parameter types. For example, the hash value of “transfer (address, uint256)” is “A9059CBB”. Therefore, we can identify whether a contract is an ERC20 token contract by comparing the hash value of each function. Then, we need to check whether each function strictly follows the ERC-20 standard.

(5) High Gas Consumption Function Type: We can identify the public functions through CFG. If a public function is not be called by any other function this means the function can be changed to an “external” function.

(6) High Gas Consumption Data Type: To detect this defect, we need to identify the pattern of byte[] from opcode. byte[] is easy to identify as it always occupies multiples of 32 bytes.

3.6.3.2 Source Code Level Detection

As we introduced in Section 4.6.2 (3), a part of the information will be removed or optimized when compiling the source code to the bytecode. Therefore, the remaining 7 contract defects need to be detected from smart contract source code.

(1) Deprecated APIs: Solidity document does not suggest using some APIs in the latest version, as they will be deprecated in the future. However, these APIs can still be compiled. When compiling to the bytecode, their instructions might be the same as the recommended APIs. To detect deprecated APIs, we need to use the latest version of Solidity and detect which APIs are deprecated.

(2) Unused Statement: Since some unused statements will be optimized by the EVM, this defect should be detected from source code. To detect this defect, we can compile the contract by using the Solidity compiler [214] and compare it to the original contract. There might be some unused statements that cannot be optimized by EVM. To detect these unused statements, we can utilize the CFG and detect whether all the paths can be executed.

(3) Unspecified Compiler Version: When compiling source code to bytecode, developers need to choose a specific version of the Solidity compiler. In this case, we cannot detect the defect from its bytecode. To detect this defect, we need to check its pragma solidity. [173]

(4) Misleading Data Location: If a smart contract has a *Misleading Data Location*, we will find that the contract modified the value on a specific storage position. However, we cannot know whether this operation is due to the contract defect. In this case, we need to detect the defect from source code. To detect it, we first need to check whether there is an array, struct, or mapping created in a function. Then, if the contract pushes a value before assigning to a storage value, this defect is detected.

(5) Missing Return Statement: The reason for this has been introduced in Section 4.6.2.3. To detect this defect, we can split source code into functions by using AST (abstract syntax tree), and check whether a function is missing a return statement.

(6) Missing Interrupter: There are multiple ways to realize interrupters, so we cannot find a method to detect this defect from bytecode. To detect the defect, we first need to summarize common methods of realizing interrupters. Then, we detect each kind of interrupter. For example, adding a *selfdestruct* function is one of the interrupters. In this case, we just need to detect whether a contract contains a *selfdestruct* function.

(7) Missing Reminder: There are also many kinds of functions that need to add reminders. To detect this defect, we all need to summarize what kind of functions need to add a reminder, then detect the defect one by one. For example, when receiving Ethers, we might use a reminder to throw an event to inform the user. In this case, we first need to locate function that can receiving Ethers. Then, verifying whether the function throws an event to inform users.

3.6.4 Code Smells in Ethereum

In software engineering, code smells are the symptoms in the source code that possibly indicate deeper problems [85]. Code smells are related to not only security issues but also design flaws, which might slow down development or increase the risk of bugs or failures in the future. Detecting and refactoring out code smells helps increase software robustness and enhance development efficiency [190]. In this paper, we defined 20 contract defects. There are many similarities between code smells and contract defects. According to Martin Fowler’s book [85], code smells do not directly trigger bugs but can lead to “potential” program faults. This definition is similar to the definition of Impact level 4 and 5. According to our definition, the contracts containing contract defects with impact level 4 and 5 can work normally, but they can lead to potential risks of errors when outside programs call the contracts, or increase the difficulty of code reuse. In this case, the contract defects with IP4 and IP5, e.g., “*Unused Statements*”, “*Unspecified Compiler Version*”, can also be considered as smart contract code smells.

3.7 Threats to Validity

3.7.1 Internal Validity

We used keywords to filter StackExchange posts. The scale of our keywords dataset determines how much manual effort we need to pay. It is not easy to cover all keywords, which means we may not cover all contract defects. Due to the time and human resource limitation, we defined 20 contract defects in this study, but researchers can define more contract defects by using our methods. To reduce this threat, we manually labeled 587 smart contracts to validate the existing of these contract defects. To provide a more stable labeling process, we followed the card sorting process, and two authors labeled the smart contracts independently. However, it is still possible that some errors exist in our dataset because of misunderstanding of smart contracts. To reduce the errors, we choose the most

experienced authors to label the contracts. They each have three-year experience on smart contract based development and have published several smart contract related papers.

The impact of smart contract defects depend on our understanding of each contract defect. However, different researchers and developers may have different understandings. To minimize this threat, we read the related posts and real-world examples and discussed with several smart contracts developers to help improve the correctness. We also considered feedback and comments from our survey.

It is difficult to ensure that all developers have a good understanding about all of the contract defects and are indeed paying attention when doing our survey. It is possible that some feedback might contain incorrect information. For example, some survey respondents give "very important" or "very unimportant" feedback to all defects. To reduce the influence of this situation, we first added an option "I don't understand" to each question and removed these responses when analyzing our survey data. We also made each question optional. Therefore, if developers find that a question is hard to understand or they lose their patience, they can skip the question instead of giving incorrect answers. Finally, we remove feedbacks given by developers whose answers are all the same when analyzing the survey data, e.g., all "very important", all "very unimportant". In addition, to help Chinese developers better understand our contract defects, three Chinese authors of this paper translated the survey into Chinese and reviewed the translated version to make sure the translation is correct.

3.7.2 External Validity

Solidity is a fast-growing programming language. In 2018, 9 versions were updated and released [69], which means many features may be added or removed in the future. Ethereum can also be updated through hard fork [109]. The latest hard fork named Constantinople will happen on the first half of 2019 [68]. Constantinople will add five new Ethereum Im-

provement Proposals (EIPs) to ensure proof-of-work more energy efficient. Some new opcodes will be added (e.g., CREATE2) and some opcodes will be modified (e.g., SSTORE). This means some new contract defects may be created, or existing contract defects will be modified. Thousands of new smart contracts may quickly be deployed to the blockchain. The distribution of the contract defects on real-world smart contracts may change with new developments of smart contract technology. Many new posts are uploaded to the *StackExchange*, and these posts can expose new contract defects. Our method can also be applied to this situation, but it needs further effort.

3.8 Related Work

Atzei et al. [4] proposed the first systematic exposition survey on attacks on Ethereum smart contracts. They introduce 12 kinds of security vulnerabilities from Solidity, EVM, and Blockchain level. Besides, they also introduce some attacks, which can be used by the attackers to make profits. The work claims that security vulnerabilities introduced in the paper are obtained from academic literature, Internet blogs, discussion forums, and based on authors' practical experience on programming smart contracts. However, the paper does not introduce the detailed steps of finding the vulnerability and does not validate whether developers consider these vulnerabilities as harmful. Another difference with our work is that our work does not only focus on the security aspect. Instead, we consider from security, availability, performance, maintainability and reusability aspects.

Oyente [131, 132] is the first bug detection tool of smart contracts, which utilizes symbolic execution to detect four security issues, i.e., mishandled exception, transaction-ordering dependence, timestamp dependence and reentrancy attack. First, Oyente builds a skeletal control flow graph for the input contracts. Then, they faithfully simulate EVM code and execute the instructions to produce a set of symbolic traces. After that, Oyente defines different patterns to check whether the tested contracts contain the security prob-

lems or not. Oyente measured 19,366 existing Ethereum contracts and found 8,519 of them contain the defined security problems.

Kalra et al. [113] found many false positives and false negatives in Oyente’s results. They developed a tool called Zeus, an upgraded version of Oyente. Their tool feeds Solidity source code as input and translates them to LLVM bitcode. Zeus detects 7 security issues, 4 of them are the same as Oyente and other 3 problems are *unchecked send*, *Failed send*, *Integer overflow/underflow*. To evaluate their tool, Kalra crawled 1524 distinct smart contracts from Etherscan [74], Etherchain [67] and EtherCamp [66] explorers. The result indicates about 94.6% of contracts contain at least one security problem.

Jiang et al. [111] focus on 7 security vulnerabilities, i.e., Gasless Send, Exception Disorder, Reentrancy, Timestamp Dependency, Block Number Dependency, Dangerous DelegateCall and Freezing Ether. They also developed a tool named ContractFuzzer to detect these issues. Their tool consists of an offline EVM instrumentation tool and an online fuzzing tool. Based on smart contract ABI, ContractFuzzer can automatically generate fuzzing inputs to test the defined security issues. They tested 6,991 smart contracts and found that 459 of them have vulnerabilities.

Nikolic [146] et al. focus on security issues that can lead to a contract not able to release Ethers, can transfer Ethers to arbitrary addresses, or can be killed by anybody. Their tool, MAIAN, takes as input data either Bytecode or source code. MAIAN contains two major parts: symbolic analysis and concrete validation. Like Oyente, simulates an Ethereum Virtual Machine, utilizes symbolic execution, and defines several execution rules to detect these security issues. Their results were deduced from 970,898 smart contracts and found that a total of 34,200 (2,365 distinct) contracts contain at least one of these three security issues.

Gao [89] et al. designed a tool named SMARTEMBED, which detect bugs in smart contracts by using a clone detection method. SMARTEMBED contains a training phase and a

prediction phase. In the training phase, there are two kinds of dataset, i.e., source code database and bug database. Source code database contains all the verified (open sourced) smart contracts in the Etherscan. The bug database records the bugs of each smart contract in their source code database. To build the prediction module, SMARTEMBED first converts each smart contract to an AST(Abstract syntax tree). After normalizing the parameters and irrelevant information on the AST, SMARTEMBED transfers the tree structure to a sequence representation. Then, they use *Fasttext* [21] to transfer code to embedding matrices. Finally, they compute the similarity between the given smart contracts with contracts in their database to find the clone contracts and clone related bugs.

We defined 20 contract defects from three different aspects. The above four papers introduce some security problems while we focus on a broader problem coverage. We do not just focus on security problems but help developers build better smart contracts. We also define patterns to help developers increase software usability and architecture. While these works show several security problems, but did not validate whether practitioners consider these problems as harmful. Our work not only validated our defined defects by an online survey, but also analysis their impacts and distribution, which can give a clear guidances for developers.

3.9 Conclusion and Future work

We conducted the first empirical study to understand and characterize smart contract contract defects. We first selected 4,141 warning related StackExchange posts from 17,128 posts. Then we manually analyzed these posts and defined 20 smart contract defects from five aspects – security, availability, performance, maintainability and reusability problems. To validate our defined contract defects, we created an online survey. The feedback from our survey indicates our contract defects are important and addressing them can help developers improve the quality of their smart contracts. We analyzed the impacts for each

contract defect and labeled 587 real-world smart contracts from Ethereum platform.

Two groups can benefit from this study. For smart contract developers, they can develop more robust and better-designed smart contracts. The 5 impacts could help developers decide the priority of removal. For software engineering researchers, our dataset can provide ground truth for them to develop smart contract defect detection tools. We plan to develop automated contract defect detection tools to detect these defined contract defects. We also plan to extend our contract defect list and dataset, when more posts will be published in StackExchange, and more features will be added into *Solidity* in the future.

Chapter 4

DEFECTCHECKER: Automated Smart Contract Defect Detection by Analyzing EVM Bytecode

Chen, J., Xia, X., Lo, D., Grundy, J.C., Luo, X., Chen, T. DEFECTCHECKER: Automated Smart Contract Defect Detection by Analyzing EVM Bytecode, to appear in *IEEE Transactions on Software Engineering*. <https://doi.org/10.1109/TSE.2021.3054928>

Abstract: Smart contracts are Turing-complete programs running on the blockchain. They are immutable and cannot be modified, even when bugs are detected. Therefore, ensuring smart contracts are bug-free and well-designed before deploying them to the blockchain is extremely important. A contract defect is an error, flaw or fault in a smart contract that causes it to produce an incorrect or unexpected result, or to behave in unintended ways. Detecting and removing contract defects can avoid potential bugs and make programs more robust. Our previous work defined 20 contract defects for smart contracts and divided them into five impact levels. According to our classification, contract defects with seriousness level between 1-3 can lead to unwanted behaviors, e.g., a contract being controlled by attackers. In this paper, we propose *DefectChecker*, a symbolic execution-based approach and tool to detect eight contract defects that can cause unwanted behaviors of smart contracts on the Ethereum blockchain platform. *DefectChecker* can detect contract defects from smart contracts' bytecode. We verify the performance of *DefectChecker* by applying it to an open-source dataset. Our evaluation results show that *DefectChecker* obtains

a high F-score (88.8% in the whole dataset) and only requires 0.15s to analyze one smart contract on average. We also applied *DefectChecker* to 165,621 distinct smart contracts on the Ethereum platform. We found that 25,815 of these smart contracts contain at least one of the contract defects that belongs to impact level 1-3, including some real-world attacks.

4.1 Introduction

In recent years, decentralized cryptocurrencies have attracted considerable interest. To ensure these systems are scalable and secure without the governance of a centralized organization, decentralized cryptocurrencies adopt the *blockchain* concept as their underlying technology. Bitcoin[141] was the first digital currency, and it allows users to encode scripts for processing transactions automatically. However, scripts in Bitcoin are not Turing-complete, which restricts their application to currencies, such as money transfer or payment. To address this limitation, Ethereum [212] leverages a technology named *Smart Contracts*, which are Turing-complete programs that run on the blockchain. By utilizing this technology, practitioners can develop decentralized applications (DApps) [205] and apply blockchain techniques to different fields such as gaming [53] and finance [72].

Smart contracts are usually developed using a high-level programming language, such as Solidity [173]. When developers deploy a smart contract to Ethereum, the contract will first be compiled into Ethereum Virtual Machine (EVM) *bytecode*. Then, each node on the Ethereum system will receive the smart contract bytecode and have a copy in their ledger. Anyone, even attackers, can invoke the smart contract by sending transactions to the corresponding contract address.

Key features of smart contracts make them become attractive targets for hackers. On the one hand, many smart contracts hold valuable Ethers, and they cannot hide their balance, which gives financial motivation for attacks by hackers [43, 41]. On the other hand, smart contracts run in a permission-less network, which means hackers can check all the transac-

tions and bytecode freely, and try to find bugs on the contracts. Even worse, smart contracts cannot be modified, even when bugs are detected. Therefore, ensuring smart contracts are bug-free and well-designed before deploying them to Ethereum is extremely important.

A contract defect [107, 31] is an error, flaw, or fault in a smart contract that causes it to produce an incorrect or unexpected result, or to behave in unintended ways [46]. The detection and removal of contract defects is a method to avoid potential bugs and improve the design of existing code. In our previous work [31], we first defined 20 contract defects by analyzing StackExchange [175] posts. It is also the first work that used an online survey to validate whether smart contract developers consider these contract defects as harmful, which make the definitions more persuasive. The work divided the defined 20 contract defects into five impact levels and showed that smart contracts contain defects with impact levels 1 to 3 can lead to unwanted behaviors, e.g., contracts being controlled by attackers.

However, our previous work did not propose a suitable tool that could detect these contract defects. To address this limitation, in this paper, we propose *DefectChecker* to detect eight contract defects defined in our previous work that belong to serious impact level 1 (high) to level 3 (medium), by using the bytecode of smart contracts. *DefectChecker* symbolically executes the smart contract through bytecode, and without the needs of source code. During the symbolic execution, *DefectChecker* generates the CFG of smart contracts, as well as the “stack event”, and identifies three features, i.e., “Money Call”, “Loop Block”, and “Payable Function”. By using the CFG, stack event, and the three features, we design eight rules to detect each contract defect.

We verify the performance of *DefectChecker* by applying it to an open-source dataset developed in our previous work [31]. We also compare its results with those of three state-of-the-art tools, i.e., *Oyente*, *Mythril* and *Securify*. Our evaluation results show that *DefectChecker* obtains the highest F-score (88.8% in the whole dataset) and requires the least time (0.15s per contract) to analyze one smart contract compared to these other baseline

tools. We also crawled all of the bytecode of smart contracts deployed on Ethereum by Jan. 2019 and applied *DefectChecker* to these 165,621 distinct bytecode smart contracts. We found that 15.9% of smart contracts on Ethereum contain at least one of contract defects (the severity level 1 to 3) using *DefectChecker*.

The main contributions of this work are:

- To the best of our knowledge, *DefectChecker* is the most **accurate** and the **fastest** symbolic execution-based model for smart contract defects detection.
- We systematically evaluated our tool using an open source dataset to test its performance. In addition, we crawled all of the bytecode (165,621) on the Ethereum platform by the time of writing the paper and identified 25,815 smart contracts that contain at least one contract defect. Using these results, we find some real-world attacks, and give examples to show the importance of detecting contract defects.
- Our datasets, tool and analysis results have been released to the community at <https://github.com/Jiachi-Chen/DefectChecker/>.

The organization of the rest of this paper is as follows. In Section 5.2, we provide background knowledge of smart contracts and introduce eight contract defects with code examples. Then, we introduce the architecture of DEFECTCHECKER in Section 5.3 and present its evaluation in Section 5.4. We conduct a large scale evaluation based on Ethereum smart contracts in Section 5.5 and give two real-world attacks as case studies. In section 5.6, we introduce the related works. Finally, we conclude the study and discuss possible future work in Section 5.7.

4.2 Background and Motivation

In this section, we briefly introduce key background information about smart contracts and their contract defects.

4.2.1 Smart Contracts

Contracts. Leveraging blockchain techniques, smart contracts are autonomous protocols stored on the blockchain. Once started, the running of a contract is automatic and it runs according to the program logic defined beforehand [40]. When developers deploy a smart contract to Ethereum, the contract will be compiled to EVM bytecode and identified by a unique 160-bit hexadecimal hash contract address. The smart contract execution depends on their code, and even the creator cannot affect its running or state. For example, if a contract does not contain functions for Ether transfer, even the creator cannot withdraw the Ethers. Smart contracts run on a permission-less network. Anyone can invoke the methods of smart contracts through ABI (Application Binary Interface) [173]. The contract bytecode, transactions, and invocation parameters are visible to everyone.

Gas System. To ensure the security of smart contracts, each transaction of a smart contract will be run by all miners. Ethereum uses the *gas system* [84] to measure its computational effort, and the developers who send transactions to invoke smart contracts need to pay an execution fee. The execution fee is computed by: $gas_cost \times gas_price$. Gas cost depends on the computational resource that takes by the execution and gas price is offered by the transaction creators. To limit gas cost, when developers send their transactions to invoke contracts, they will set the *Gas Limit* which determines the maximum gas cost. If the gas cost of a transaction exceeds its *Gas Limit*, the execution will fail and throw an *out-of-gas error* [212]. There are some special operations which will limit the *Gas Limit* to a specific value. For example, *address.transfer()* and *address.send()* are two methods provided by Ethereum that are used to send Ethers. If a smart contract uses these methods to send Ethers to another smart contract, the *Gas Limit* will be restricted to 2300 gas units [173]. 2300 gas units are not enough to write to storage, call functions or send Ethers, which can lead to the failure of transactions. Therefore, *address.transfer()* and *address.send()* can only be used to send Ethers to *external owned accounts* (EOA). (There are two types of

accounts on Ethereum: externally owned accounts which controlled by private keys, and contract accounts which controlled by their contract code [212].)

Ethereum Virtual Machine (EVM). To deploy a smart contract to Ethereum, its source code needs to be compiled to bytecode and stored on the blockchain. EVM is a stack-based machine; when a transaction needs to be executed, EVM will first split bytecode into bytes; each byte represents a unique instruction called opcode. There are 140 unique opcodes by April 2019 [212], and each opcode is represented by a hexadecimal number [212]. EVM uses these opcodes to execute the task. For example, consider a bytecode 0x6070604001. EVM first splits this bytecode into bytes (0x60, 0x70, 0x60, 0x40, 0x01), and executes the first byte 0x60, which refers to opcode *PUSH1*. *PUSH1* pushes one byte data to EVM stack. Therefore, 0x70 is pushed to the stack. Then, EVM reads the next 0x60 and push 0x40 into the stack. Finally, EVM executes 0x01, which refers to opcode *ADD*. *ADD* obtains the next two values from the top of the stack, i.e., 0x70 and 0x40, and put their sum (B0), a hex result into the stack.

EVM Bytecode v.s. JVM Bytecode in Control Flow Analysis. Control flow analysis methods have been widely used in other stack-based machines, e.g., JVM [189]. However, there are many differences in analyzing the control flow of Java bytecode and EVM bytecode. These differences present some new challenges in analyzing EVM bytecode. We highlight the key differences between EVM bytecode analysis method we used in this paper and JVM bytecode analysis. These include:

(1) JVM bytecode has a fixed stack depth under different control-flow paths. The execution of JVM cannot reach the same program point with different stack sizes [95]. There are no such constraints for EVM bytecode, which greatly increases the difficulty of identifying the control-flow constructs in EVM bytecode. For example, for a simple recursive code “function f(int a)f(a);”. The code will be compiled in EVM as:

¹ Block 1:
² JUMPDEST

```

3 PUSH Block1' ID
4 DUP2
5 PUSH Block2' ID
6 JUMP
7 Block 2:
8 JUMPDEST

```

There are two blocks; two block identifiers are pushed in the same block (block 1) and will be read by the same instruction (JUMP). The difference between the JVM and EVM is that the JVM creates a new frame [155] with a new operand stack for each method call, whereas the EVM just has one global operand stack. (A frame is used to store data and partial results, as well as to perform dynamic linking, return values for methods, and dispatch exceptions.)

(2) JVM bytecode has a clearly defined set of targets for each jump [39]. In contrast, the jump target for EVM bytecode is read from the EVM stack. When a conditional jump is used, the target will be affected by the second stack item. For example, in Figure 2, the jump target of *JUMPI* (ID 140) is read from previous instruction *PUSH* and will be affected by the second stack item, i.e., $ISZERO(GT(10, num))$ (details see Section 5.3.3). If the second item refers to a true value. The jump target is 148; otherwise, the target is 141. The unconditional jump target is also read from the top of the EVM stack. For example, the jump target of *JUMP* (ID 147) in Figure 2 is also read from the previous instruction *PUSH*. Therefore, we need to symbolically execute the EVM bytecode to construct the control-flow edges.

(3) JVM bytecode has well-defined method invocation and return instructions [95]. In contrast, EVM bytecode uses jumps to perform its intra-contract function calls. In this case, to resolve an intra-contract function call, we need to inspect the top stack element to determine the jump target. For example, there are two functions A and B. Function A contains three blocks, e.g., A1, A2, A3; function B contains two blocks, e.g., B1, B2. The code on block A2 calls function B. In EVM bytecode, there is no defined method invocation and

return instructions. Instead, the code pushes the return address to the stack; the arguments and jump target (block identifier of B1) need to be identified through bytecode. To return, the code pops the caller’s block identifier (A3) and jumps to execute the block. Thus, the execution sequences are A1, A2, B1, B2, A3. The identifiers of B1 and A3 should be obtained from bytecode through symbolic execution.

The Fallback Function. The fallback function is a unique feature of smart contracts compared to traditional programs. An example can be found at Line 13 of Listing 5.4, which is the only unnamed function in smart contracts programming [173]. The fallback function does not have any arguments or return values. It will be executed automatically on a call to the contract if none of the functions match the given function identifier [173]. For example, if a transaction calls function ‘A’ of the contract, and there is no function named ‘A’, then the fallback function will automatically be executed to handle the erroneous function invocation. If the function is marked by *payable* [173], the fallback function will also be executed automatically when receiving Ethers.

The Call Instruction and Ether Transfer. Ether transfer is an important feature on Ethereum. In Solidity programming, there are three methods to transfer Ethers, i.e., *address.call.value()*, *address.transfer()*, and *address.send()*. Among these three methods, only *address.call.value()* allows users to send Ethers to a contract address, as the other two methods are limited to 2300 gas units, which are not enough to send Ethers. *address.send()* returns a boolean value, while *address.transfer()* throws an exception when errors happen and returns nothing. All of these three methods can generate a CALL instruction in contract bytecode. Other behaviors, e.g., function call, can also generate CALL instructions. A CALL instruction reads seven values from the top of EVM stack. They represent the gas limitation, recipient address, transfer amount, input data start position, size of the input data, output data start position, size of the output data, respectively.

4.2.2 Contract Defects in Smart Contracts

Our previous work [31] defined 20 contract defects for smart contracts. We divided these contract defects into five “impact” levels; among these contract defects, 11 belong to impact level one (most serious) to three (low seriousness) that might lead to unwanted behaviors. The definition of these 11 contract defects is given in Table 4.1. In this paper, we propose *DefectChecker*, a symbolic execution tool to detect eight of these impact level one to three contract defects. *DefectChecker* does not detect contract defects belonging to levels 4 and 5, as these contract defects will not affect the normal running of the smart contracts according to the definition. For example, *Unspecified Compiler Version* is one of the level 5 smart contract defects. The removal of the contract defects requires the developer of the contract to use a specific compiler like 0.4.25. This contract defect will not affect the normal running of the contract and will only pose a threat for code reuse in the future. This kind of contract defect is also difficult to detect at the bytecode level as much semantic information is lost after compilation.

However, please note that in this work, we do not consider three of the contract defects that belong to impact level 1 to 3 – *Unmatched Type Assignment*, *Hard Code Address* and *Misleading Data Location*, as they are not easy to detect at bytecode level. Our analysis shows that they appear 22, 84, and 1 times among 587 smart contacts, respectively. EVM will remove or add some information when compiling smart contracts to bytecode, which may cover up these taints on the source contract code. For *Hard Code Address*, the bytecode we obtain from the blockchain does not contain information on the *construct* function, while we found most *Hard Code Address* errors appear in *construct* functions. To detect *Unmatched Type Assignment*, we need to know the maximum loop iterations, which is usually read from storage, and is not easy to obtain the value through static analysis. For example, for a loop “*for(uint8 i = 0; i < num; i++)*”, the data range of uint8 is from 0 to 255. Thus, if num is larger than 255, the loop will overflow. However, num is usually a

Table 4.1: The Definitions of contract defects with Impact level 1-3. The first eight contract defects can be detected by *DefectChecker*.

Contract Defect	Definition	Impact Level
<i>Transaction State Dependency (TSD)</i>	Using tx.origin to check the permission.	IP1
<i>DoS Under External Influence (DuEI)</i>	Throwing exceptions inside a loop which can be influenced by external users	IP2
<i>Strict Balance Equality (SBE)</i>	Using strict balance quality to determine the execute logic.	IP2
<i>Reentrancy (RE)</i>	The re-entrancy bugs.	IP1
<i>Nested Call (NC)</i>	Executing CALL instruction inside an unlimited-length loop.	IP2
<i>Greedy Contract (GC)</i>	A contract can receive Ethers but can not withdraw Ethers.	IP3
<i>Unchecked External Calls (UEC)</i>	Do not check the return value of external call functions.	IP3
<i>Block Info Dependency (BID)</i>	Using block information related functions to determine the execute logic.	IP3
<i>Unmatched Type Assignment</i>	Assigning unmatched type to a value, which can lead to integer overflow	IP2
<i>Misleading Data Location</i>	The reference types of local variables with <i>struct</i> , <i>array</i> or <i>mapping</i> do not clarify	IP2
<i>Hard Code Address</i>	Using hard code address inside smart contracts.	IP3

storage variable which is read from storage or depends on an external input. Thus, it is difficult to detect this through bytecode analysis. ***Misleading Data Location*** is also not easy to detect from bytecode. In Solidity programming, *storage* in Solidity is not dynamically allocated and the type of *struct*, *array* or *mapping* are maintained on the storage. Thus, these three types created inside a function can point to the *storage slot 0* by default, which can lead to potential bugs. However, we cannot know whether the point on slot 0 is correct or a mistake made by EVM.

4.2.2.1 Definition of Impact Levels

Below we give representative concrete examples of each of the eight smart contract defects, and introduce the definition of impact level one to three according to our previous work.

- **Impact 1 (IP1):** Smart contracts containing these contract defects can lead to critical unwanted behaviors. Unwanted behaviors can be triggered by attackers, and they can make profits by utilizing the defects.
- **Impact 2 (IP2):** Smart contracts containing these contract defects can lead to critical unwanted behaviors. Unwanted behaviors can be triggered by attackers, but they cannot make profits by utilizing the defects.
- **Impact 3 (IP3):** There are two types of IP3. **Type A:** Smart contracts containing these contract defects can lead to critical unwanted behaviors, but unwanted behaviors cannot be triggered by attackers. **Type B:** Smart contracts containing these contract defects can lead to major unwanted behaviors. The unwanted behaviors can be triggered by attackers, but they cannot make profits by utilizing the defects.

Critical represents contract defects, which can lead to a crash, being controlled by attackers, or can lose all the Ethers. Major represents the contract defects that can lead to the loss of a part of the Ethers [31].

4.2.2.2 Examples of Smart Contract Defects

```
1 contract Victim { ...
2     address owner = owner_address;
3     function sendMoney(address addr){
4         require(tx.origin == owner);
5         addr.transfer(1 Ether);
6     }
7 }
8 contract Attacker{ ...
9     function attack(address vim_addr, address myAddr){
10         Victim vic = Victim(vim_addr);
11         vic.sendMoney(myAddr);
```

```
12    }
13 }
```

Listing 4.1: Transaction State Dependency

(1). Transaction State Dependency (TSD): Contracts need to check whether the caller has the right permission for some permission sensitive functions. The failure of the permission check can cause serious consequences. *tx.origin* can get the original address of the transaction, but this method is not reliable as the address returned by this method depends on the transaction state. Therefore, *tx.origin* should not be used to check whether the caller has permission to execute functions.

Example: In Listing 5.1, The *Attacker* contract can make a permission check fail by utilizing the *attack* function (Line 9). By utilizing this method, anyone can execute *send-Money* function (Line 3) and withdraw the Ethers in the contract.

Possible Solution: *Solidity* provides *msg.sender* to obtain the sender address, which can be used to check permissions instead of using *tx.origin*.

(2). DoS under External Influence (DuEI): Smart contracts will rollback a transaction if exceptions are detected during their running. If the error that leads to the exception cannot be fixed, the function will give a denial of service (*DoS*) error perpetually.

Example: Listing 5.2 shows such an example. Here, *members* is an array which stores many addresses. However, one of the address is an attacker contract, and the transfer function can trigger an out-of-gas exception due to the 2300 gas limitation [212]. Then, the contract state will rollback. Since the code cannot be modified, the contract can not remove the attack address from *members* list, which means that if the attacker does not stop attacking, the following function cannot work anymore.

Possible Solution: Developers can use a boolean value check instead of throwing exceptions in the loop. For example, using “*if(members[i].send(0.1 ether) == false) break;*” instead of line 3 in listing 5.2.

```
1 for (uint i = 0; i < members.length; i++) {
```

```

2     if (this.balance > 0.1 ether)
3         members[i].transfer(0.1 ether);
4 }
```

Listing 4.2: DoS under External Influence

(3). Strict Balance Equality (SBE): Attackers can send Ethers to any contracts forcibly by utilizing *selfdestruct()* [173]. This method will not trigger the fallback function, which means the victim contract cannot reject the Ethers. Therefore, smart contract logic may fail to work due to the unexpected Ethers sent by attackers.

Example: The *doingSomething()* function in listing 5.3 can only be triggered when the balance strict equal to 1 *ETH*. However, the attacker can send 1 *Wei* (1 ETH = 1e18 Wei) to the contract to make the balance never equal to 1 ETH.

Possible Solution: The contract can use “ \geq ” to replace “ $==$ ” as attackers can only add to the amount of a balance. In this case, it is difficult for the attackers to affect the logic of the program.

```
1 if (this.balance == 1 eth) doingSomething();
```

Listing 4.3: Strict Balance Equality:

(4). Reentrancy (RE): In Ethereum, a function can be executed several times in one execution by using the *Call* method. When a contract calls another, the execution waits for the call to finish [131]. Thus, it can lead to multiple invocations and money transfer in some situations.

Example: Listing 5.4 shows an example of a reentrancy defect. There are two smart contracts, i.e., *Victim* contract and *Attacker* contract. The *Attacker* contract is used to transfer Ethers from *Victim* contract, and the *Victim* contract can be regarded as a bank, which stores the Ethers of users. Users can withdraw their Ethers by invoking *withdraw()* function, which contains Reentrancy defects.

First, the *Attacker* contract uses the *reentrancy()* function (L16) to invoke *Victim* contract’s *withdraw()* function in line 3. The *addr* in line 16 is the address of the *Victim* con-

tract. Normally, the *Victim* contract sends Ethers to the callee in line 6, and resets callee's balance to 0 in line 7. However, the *Victim* contract sends Ethers to the *Attacker* contract before resetting the balance to 0. When the *Victim* contract sends Ethers to the *Attacker* contract (L6), the fallback function (L13) of the *Attacker* contract will be invoked automatically, and then invoking the *withdraw()* function (L14) again. The invoking sequence in this example is: L16-17 → L3-6 → L13-14 → L3-6 → L13-14 …, until Ethers run out.

Possible Solution: There are 3 kinds of *Call* methods that can be used to send Ethers in Ethereum, i.e., *address.send()*, *address.transfer()*, and *address.call.value()*. *address.send()* and *address.transfer()* will change the maximum gas limitation to 2300 gas units if the recipient is a contract account. 2300 gas units are not enough to transfer Ethers, which means *address.send()* and *address.transfer()* cannot lead to *Reentrancy*. Therefore, using *address.send()* and *address.transfer()* instead *address.call.value()* can avoid *Reentrancy*.

```

1 contract Victim { ...
2     mapping(address => uint) public userBalance;
3     function withdraw() {
4         uint amount = userBalance[msg.sender];
5         if(amount > 0){
6             msg.sender.call.value(amount)();
7             userBalance[msg.sender] = 0;
8         }
9     }
10    ...
11 }
12 contract Attacker{ ...
13     function() payable{
14         Victim(msg.sender).withdraw();
15     }
16     function reentrancy(address addr){
17         Victim(addr).withdraw();
18     }
19    ...
20 }
```

Listing 4.4: Reentrancy

```

1 for(uint i = 0; i < member.length; i++){
2     member[i].send(1 wei);
3 }
```

Listing 4.5: Nested Call

(5). Nested Call (NC): Instruction *CALL* is very expensive (9000 gas paid for a non-zero value transfer as part of the *CALL* operation) [212]. If a loop contains the *CALL* instruction but does not limit the loop iterations, the total gas cost may have a high risk to exceed its gas limitation.

Example: In listing 5.5, if we do not limit the loop iterations, attackers can maliciously increase its size to cause an out-of-gas error. Once the out-of-gas error happens, this function cannot work anymore, as there is no way to reduce the loop iterations.

Possible Solution: Developers should estimate the maximum loop iterations and limit the loop iterations.

(6). Greedy Contract (GC): Ethers on smart contracts can only be withdrawn by sending Ethers to other accounts or using *selfdestruct* function. Otherwise, even the creators of the smart contracts cannot withdraw the Ethers and Ethers will be locked forever. We define that a contract is a greedy contract if the contract can receive Ethers (contains payable functions) but there is no way to withdraw the Ethers.

Example: Listing 5.6 is a greedy contract. The contract is able to receive Ethers as it contains a payable fallback function in line 2. However, the contract does not contain any methods to transfer money to others. Therefore, the Ethers on the contract will be locked forever.

Possible Solution: Adding a function to withdraw Ethers if the contract can receive Ethers.

```
1 Contract Greedy{  
2     function () payable{  
3         process(msg.sender);  
4     }  
5     function process(address addr) { ... }  
6 }
```

Listing 4.6: Greedy Contract

(7). Unchecked External Call (UEC): Solidity provides many functions (*address.send()*, *address.call()*) to transfer Ethers or call functions between contracts. However, these call-

related methods can fail, e.g., have a network error or run out of gas. When errors happen, these functions will return a boolean value but never throw an exception. If the callers do not check the return values of the external calls, they cannot ensure whether the logic of the following code snippets is correct.

Example: Listing 5.7 shows such an example. Line 1 does not check the return value of the `address.send()`. As the Ether transfer can sometimes fail, line 1 cannot ensure whether the logic of the following code is correct.

Possible Solution: Always checking the return value of the `address.send()` and `address.call()`.

```
1 address . send( ethers ) ; doingSomething() ; // bad
2 if ( address . send( ethers ) ) doingSomething() ; // good
```

Listing 4.7: Unchecked External Call

(8). Block Info Dependency (BID): Developers can utilize a series of block related functions to obtain block information. For example, `block.blockhash` is used to obtain the hash number of the current block. Many smart contracts rely on these functions to decide a program's execution, e.g., generating random numbers. However, miners can influence block information, e.g., miners can vary the block time stamp by roughly 900 seconds [131]. In this case, the block info dependency operation can be controlled by miners to some extent.

Example: The contract in listing 5.8 is a code snippet of a roulette contract. The contract utilizes block hash number to select a winner, and send winner one Ether as bonus. However, the miner can control the result. So, the miner can always be the winner.

Possible Solution: The precondition of a safe random number is that the random number cannot be controlled by a single person, e.g., a miner. The completely random information we can use in Ethereum includes users' addresses, users' input numbers and so on. Also, it is important to hide the values used by the contract for other players to avoid attacks. Since we cannot hide the address of users and their submitted values on Ethereum,

a possible solution to generate random numbers without using block related functions is using a hash number. The algorithm has three rounds:

Round 1: Users obtain a random number and generate a hash value in their local machine. The hash value can be obtained by *keccak256* function, which is a function provided by Ethereum. After obtaining the random number, users submit the hash number.

Round 2: After all the users submit the hash number, users are required to submit the original random number. The contract checks whether the original number can generate the same hash number by using the same *keccak256* function.

Round 3: If all users submit correct original numbers, the contract can use the original numbers to generate a random number.

```
1 address[] participants;
2 uint winnerID = uint(block.blockhash) % participants.length
3 participants[winnerID].transfer(1 eths);
```

Listing 4.8: Block Info Dependency:

4.3 The *DefectChecker* Approach

4.3.1 Design Overview

Figure 4.1 depicts an overview architecture of the *DefectChecker* approach. There are four components of *DefectChecker*, i.e., *Inputter*, *CFG Builder*, *Feature Detector*, and *Defect Identifier*.

The left part of the figure is the *Inputter*, and users can feed bytecode as *input*. Solidity source code is also allowed, but it needs to be compiled into bytecode. Bytecode is then disassembled into opcodes by utilizing API provides by *Geth* [70]. Then, *DefectChecker* splits opcode into several basic blocks and symbolically executes instructions in each block. After that, *DefectChecker* generates the CFG (control flow graph) of a smart contract and records all stack events. During symbolic execution, *Feature Detector* detects three features (i.e., Money Call, Loop Block and Payable Function), all concepts introduced below.

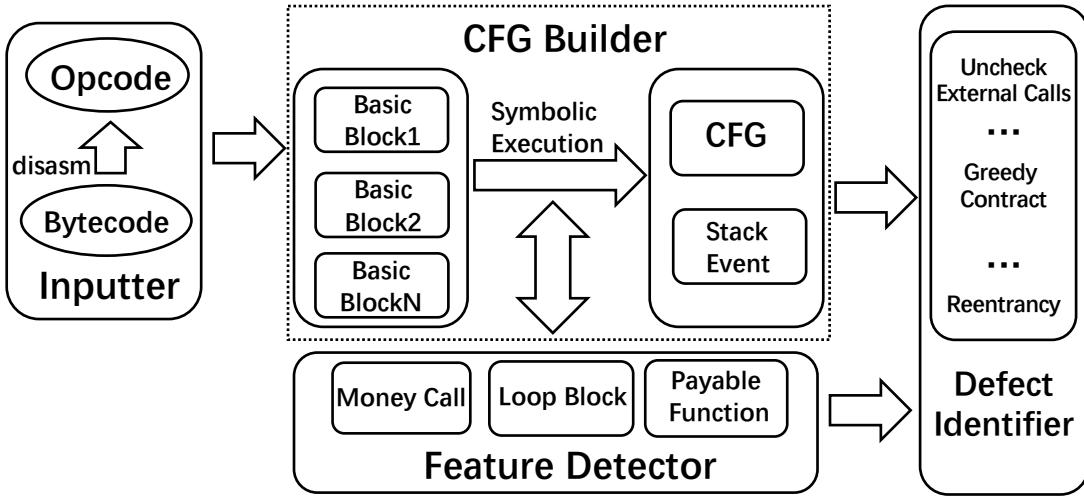


Figure 4.1: Overview architecture of *DefectChecker*

Based on this information, *Defect Identifier* uses eight different rules to identify the contract defects on smart contracts.

Detecting contract defects by bytecode is very important for smart contracts on Ethereum. All the bytecode of smart contracts are stored on the blockchain, but only less than 1% of smart contracts have opened their source code [42]. Smart contracts usually call other contracts, but the callee contracts may not open their source code for inspection. In such a case, the caller smart contracts can only detect whether the callee contract is secure through their bytecode.

4.3.2 Basic Block Builder

A basic block is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit [37]. We first split the opcode into several blocks and give a type of the block according to its exit type. The exit type can be determined by the last instruction on a block. If the last instruction is *JUMP* or *JUMPI*, the block type is *unconditional* or *conditional*, respectively. If the last instruction is a terminal instruction (*STOP*, *REVERT* and *RETURN*), the block type is *terminal*. Some blocks belong to none

of these three types, we call their block type as *fall*. In summary, we consider four types of blocks: *unconditional*, *conditional*, *fall*, and *terminal*.

4.3.3 Symbolic Execution

Unlike other stack-based machines, e.g., JVM where Java bytecode has a clearly-defined set of targets for every jump, the jump position of EVM bytecode needs to be calculated during symbolic execution. Thus, *DefectChecker* needs to symbolically execute each single EVM instruction one at a time to obtain the CFG for smart contracts. EVM is a stack-based machine – when executing an instruction, it reads several symbolic states from the top of the EVM stack and put the symbolic result back to the EVM stack. During the symbolic execution, we can obtain the jump relations between blocks. There are three types of block according to the jump behaviors, i.e., *conditional jump*, *unconditional jump* and *fall execution*. *Stack Event* records all symbolic states on the EVM stack after the execution of each instruction.

```

1 function example(uint num) returns(uint){
2     if (num > 10)
3         return 1;
4     else {
5         return 0;
6     }
7 }
```

Listing 4.9: Code of Figure 2

Figure 4.2 is an example of the symbolic execution of the code in Listing 5.9. There are 4 blocks in this figure, and each block contains several instructions. The instructions in block 1 represent the code *if(num > 10)*. The block 2 and block 3 put the value (0 or 1) to the EVM stack, respectively. The instructions in *block 4* are used to return the value(0 or 1) to the environment. The left-most number in each line indicates instructions' index ID, and the center part is the instruction that needs to be executed. All the instructions will execute sequentially according to their index ID. If the instruction is 'PUSH', the right-most part will have a value that pushes into EVM stack. There is a Program Counter (PC)

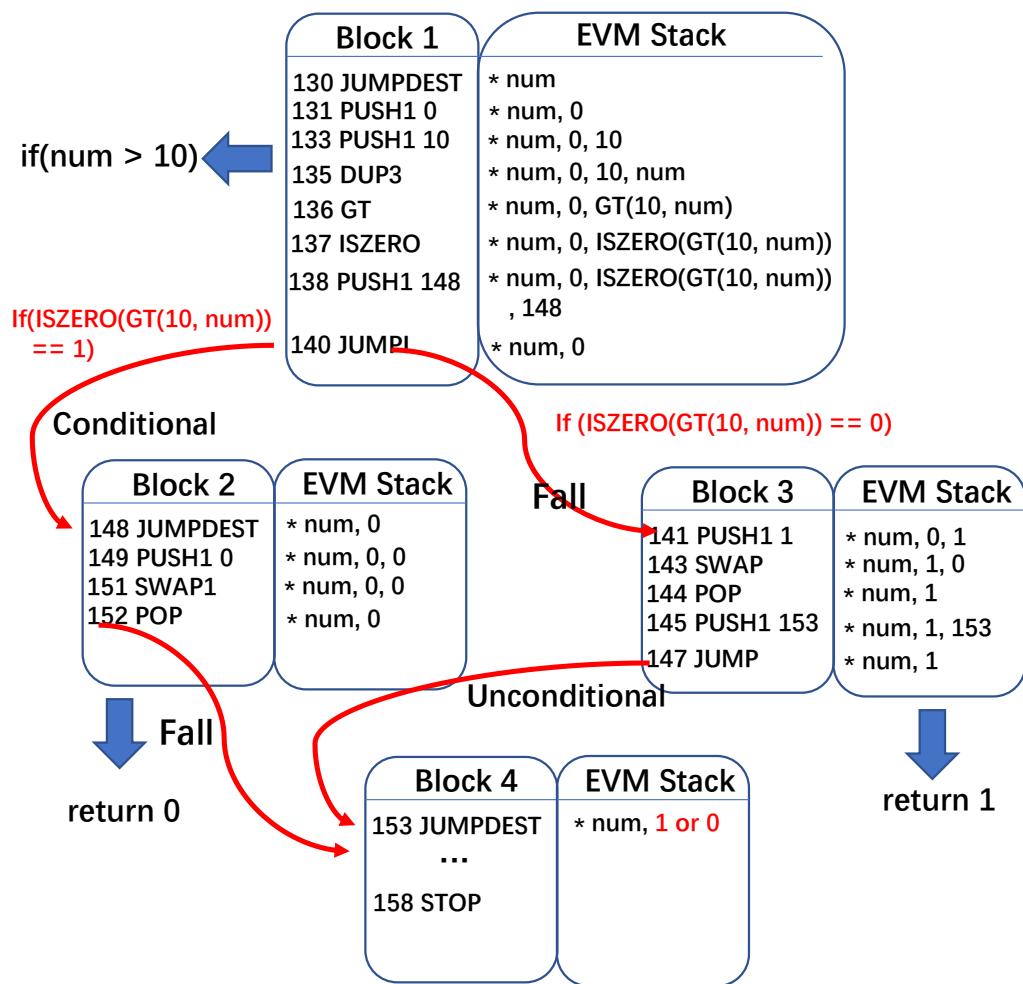


Figure 4.2: Example of Symbolic Execution

that records the ID that being executed at the current time. The PC starts from ID 0 in block 1, and EVM executes this instruction.

The example shown in Figure 4.2 is a part of the code of a contract, so the PC starts from index ID 130 in block 1. Before EVM executes the instruction *JUMPDEST*, there is a symbol *num* in the EVM stack. The symbol *num* represents the input value of the function (L1 of Listing 5.9). *JUMPDEST* marks a valid destination for jumps; it does not read or push any values. So the PC points to ID 131, and EVM pushes a value 0 to EVM stack. Then, ‘10’ is pushed into EVM stack and PC point to 135. *DUP3* duplicates the 3rd stack item. Therefore, the symbol *num* is pushed into EVM stack. *GT* reads two values from the EVM stack. If the first value (at the top of the stack) is greater than the second value, then EVM push 1 into the stack; otherwise, 0 is pushed. We use a symbol *GT(a, num)* to represent the result and push the result into the EVM stack. Then, *ISZERO* reads a value from the top of the EVM stack. *ISZERO* reads one value from EVM. If the value equal to zero, then we push 1 into stack; otherwise, we push 0. We use a symbol *ISZERO(GT(a, num))* to represent the result and push the result into the EVM stack. *JUMPI* (ID 140) reads two values from the stack, the first value represents the jump position ‘148’, and the second value is a conditional expression. If the result of the conditional expression is “1” (true), the the PC jumps to the index ID 148, which indicates the start position of block 2. Otherwise, if the result is “0” (false), the EVM falls to execute the following index ID 141(the start position of Block 3).

Since the result of *ISZERO(GT(a, num))* can be “0” or “1”, this symbolic execution can generate two paths, i.e., Block 1 → Block 2 and Block 1→ Block 3.

We first assume the result of *ISZERO(GT(a, num))* is “1” and the path is Block 1 ->Bock 2. In this case, the PC points to the ID 148. The jump type of this path is *conditional jump*. After executing the instructions on ID 148-152, the EVM falls to execute block 4. The jump type from block 2 to block 4 is *fall*. When executing the first instruction of the block

Table 4.2: The Information Required to Detect Each Contract Defect

Contract Defect	Control Flow Information	Symbolic State
<i>Transaction State Dependency (TSD)</i>		✓
<i>DoS Under External Influence (DuEI)</i>	✓	✓
<i>Strict Balance Equality (SBE)</i>	✓	✓
<i>Reentrancy (RE)</i>	✓	✓
<i>Nested Call (NC)</i>	✓	✓
<i>Greedy Contract (GC)</i>	✓	✓
<i>Unchecked External Calls (UEC)</i>		✓
<i>Block Info Dependency (BID)</i>	✓	

4, the EVM stack holds two values, i.e., *num* and 0. Block 4 then returns the value 0 to the environment and uses instruction *STOP* to finish the execution.

We then assume the result of *ISZERO(GT(a, num))* is “0” and the path is Block 1 ->Block 3. In this case, the PC points to the ID 141. The jump type of this path is *fall execution*. *JUMP* refers to an unconditional jump; it reads one value from the top of the stack. The value reads by *JUMP* in ID 147 is ‘153’. After executing the instructions on ID 141-147, the EVM then jumps to execute block 4. The jump type from block 3 to block 4 is an *unconditional jump*. When executing the first instruction of the block 4, the EVM stack holds two values, i.e., *num* and 1. Block 4 then returns the value 1 to the environment and uses instruction *STOP* to finish the execution.

When executing a conditional jump, we should determine the satisfiability of the conditional expression, which is typically realized by invoking an SMT (satisfiability modulo theories) solver [8], e.g., Z3 [55]. If the SMT solver cannot find a solution, we consider the corresponding program path as infeasible. Therefore, symbolic execution can be used to discover dead code. However, there may be little dead code in EVM bytecode, because the compiler can eliminate dead code during the compilation of smart contracts. To accelerate our analysis, we consider the conditional expression, which is equal to “0” as unsatisfiable and all other conditional expressions as satisfiable, without checking their satisfiability.

4.3.4 Feature Detector

To detect contract defects at the bytecode level, we need to identify some specific behaviors from their opcodes. In this part, we introduce three features that we use when detecting contract defects.

4.3.4.1 Money Call

To detect *Reentrancy*, we need to identify whether a smart contract can transfer Ethers to other contracts. Ethereum provides three methods to transfer Ethers, i.e., `address.send()`, `address.transfer()`, `address.call().value()`. All of these three methods generate a *CALL* instruction. However, only detecting the *CALL* instruction is not enough, as many other behaviors can also generate *CALL* instruction, e.g., calling functions on other contracts or library. In this paper, if a *CALL* instruction is generated by functions which are used to transfer Ethers, we call this *CALL* instruction a *Money-CALL*. Otherwise, the *CALL* instruction is a *No-Money-CALL*. *CALL* reads seven values from EVM stack. The first three values represent the gas limitation, recipient address, transfer amounts, respectively. If the transfer amount is larger than 0, the *CALL* instruction is a *Money-CALL*.

However, only detecting *Money-CALL* is still not enough, as `address.send()` and `address.transfer()` will limit the maximum gas consumption to 2300, which is not enough to send Ethers. Therefore, these two methods also cannot cause *Reentrancy*. If the *CALL* instruction is generated by `address.send()` and `address.transfer()`, a specific number “2300” will be pushed into EVM stack, which represented the maximum gas consumption. So, if *CALL* instruction reads a specific number “2300” from the EVM stack, the *CALL* instruction is generated by `address.send()` and `address.transfer()`. We call this *CALL* instruction a *Gas-Limited-Money-CALL*. Otherwise, if the first value read by *CALL* instruction does not contain a specific value “2300”, we assume that the *CALL* instruction is generated by `address.call().value()`. We call this *CALL* instruction a *Gas-Unlimited-Money-CALL*.

4.3.4.2 Loop Block

After constructing the CFG, we need to detect which block is the start of a loop and which blocks make up the body of the loop. To detect this information, we first traverse the path of the CFG by utilizing *DFS* (Depth-first-search) [180] and then flag all blocks we visit. If there is a block that has been visited, this block is the start of a loop, and other blocks in this cycle are the loop bodies. Since some smart contracts are very complicated, it may contain a large number of paths. To reduce the computational effort, we use the strategy of pruning. For example, block A is the destination of many other blocks, and we find the path of block A does not contain any cycles. We do not need to visit the remaining paths when other paths encounter block A.

4.3.4.3 Payable Function

A smart contract can receive Ethers only if it contains payable functions [212]. To detect whether a function is payable or not, we can inspect the first block of each function. *CALLVALUE* instruction is used to get the received Ether amount. If a smart contract receives Ethers, *CALLVALUE* instruction will get a non-zero value. This value can be checked by the *ISZERO* instruction to know whether a transaction contains Ethers. If the function is not payable, when receiving Ethers, it will throw an exception and terminate the execution.

To find the first block, we first rank all instructions by their index ID. All conditional jumps positioned before the first *JUMPDEST* instruction are the start position of each function. EVM uses a hash value to identify functions; when EVM receives a function call, it first compares the received value to each function's hash value. If a function's hash value is equal to the received hash value, it will jump to the destination, which indicates a function's start position. Otherwise, it will fall to *fallback* function, whose start position is the first *JUMPDEST* instruction.

4.3.5 DefectChecker

Table 4.2 describes the information required to detect each kind of contract defect. To detect TSD and UEC, *DefectChecker* only needs symbolic states computed by symbolic execution, as we only need to check whether *ORIGIN* and *CALL* instructions are read by *EQ* and *ISZERO* instruction, respectively. *DefectChecker* only needs control flow information to detect BID, as we only need to check whether the conditional expression contains block related instructions, e.g., "BLOCKHASH".

To detect the other 5 contract defects, *DefectChecker* needs both control flow information and symbolic states. In the previous subsection, we introduce three features detected by the feature detector, i.e., *Money Call*, *Loop Block*, and *Payable function*. *Money Call* needs symbolic states, so to detect it, *DefectChecker* needs check the values on the EVM stack. *Loop Block* and *Payable function* require control flow information, as they both need CFGs to locate the loop and the start of the function, respectively. *NC*, *DuEI*, *GC*, and *Reentrancy* all need to detect *Money Call*. *DuEI* and *NC* also need to detect *Loop Block*; *GC* needs to detect *Payable function*. To detect *Reentrancy*, *DefectChecker* needs to travel all the paths that contain the *Gas-Unlimited-Money Call*, which needs the help of the CFG. To detect *SBE*, *DefectChecker* needs to check whether the *BALANCE* instruction is read by the *EQ* instruction in the conditional expressions, which needs both control flow information and symbolic state.

Below we describe the detailed patterns that we use to determine whether a smart contract contains one or more of the contract defects.

4.3.5.1 Transaction State Dependency

tx.origin generates an *ORIGIN* instruction. We first locate all *ORIGIN* instructions. We then check whether there is an *ORIGIN* that is read by an *EQ* instruction. The *EQ* instruction reads two values from EVM stack and verifies whether these two values are equal.

If the contract contains this kind of contract defect *ORIGIN* instruction will compare to an address value. Ethereum uses a 40-bit value to indicate an address, and all addresses conform to the EIP55 standard [192].

4.3.5.2 DoS Under External Influence

If a smart contract contains this contract defect, there will be a part of the instructions that check the return value of the *Money CALL*, and then terminate the loop. To detect this contract defect, we first find loop-related blocks. Then, we check whether there is a block that contains *Money CALL*, and the type of the block is *conditional*, as it needs to check the return value. Then, this block jumps to a block, which type is *terminal*.

4.3.5.3 Strict Balance Equality

This kind of contract defect can make a part of the code never be executed. We need to check whether there is a conditional expression that contains the related pattern. *BALANCE* instruction is used to get the balance of a contract. If a *BALANCE* instruction is read by *EQ*, it means there is a strict balance equality check. If this check happens at a conditional jump expression, it means this contract contains this contract defect.

4.3.5.4 Reentrancy

The *SLOAD* instruction is used to get a value from storage [212]. It reads a value (named *Slot ID*) from the EVM stack and puts the result that reads from storage back onto the EVM stack. Using *listing 5.4* as an example, *Victim* contracts do not make the balance of an *Attack* contract to zero (L7) before sending Ethers (L6), which allows an *Attack* contract to withdraw Ethers again. To detect this contract defect, we first need to obtain paths that contain *Gas-Unlimited-Money-Call*, because only this kind of *CALL* can cause *Reentrancy*. We then need to obtain all conditional expressions on these paths. The amount that is sent by the victim contract is usually checked before sending it to attacker contracts, and this

amount is loaded from *storage*. In this case, we need to check if the conditional expression contains *SLOAD* instructions and get its *Slot ID*. If this value still holds and does not be updated when executing *CALL* instruction, it means *CALL* instruction can be executed again and cause *Reentrancy*. To check whether the storage value is updated, we need to detect whether the same *Slot ID* that is read by *SLOAD* is written by *SSTORE* instruction. (*SSTORE* instruction is used to save data to memory. It reads two values from EVM stack, i.e., slot id and value that are written to storage.)

4.3.5.5 Nested Call

Using *listing 5.5* as an example, array *members* is a storage variable, all of its value, including its length, are stored on storage. To get its length, *SLOAD* instruction reads its *Slot ID* δ from EVM stack, and this value is the position that stores the value of *members.length*. To detect this contract defect, the first step is to find the start block of a loop and get the *Slot ID*. Then, we need to check whether this loop limits its size. If the loop limits its size, the same *Slot ID* δ will be read in the loop body again, and this value will be compared with another value. If a smart contract contains a loop that does not limit its size but contains a *Money-Call*, *Nest Call* is detected in this contract.

4.3.5.6 Greedy Contract

A smart contract can transfer money through a *Money CALL* or *selfdestruct* function. *selfdestruct* function generates *SELFDESTRUCT* instruction. If a smart contract contains payable functions but does not have either a *Money CALL* or *SELFDESTRUCT* instruction, the contract is a *Greedy Contract*.

4.3.5.7 Unchecked External Calls

The external call returns a boolean value. If the result is checked by the contract, it will generate an *ISZERO* instruction. To detect this contract defect, we first locate *CALL* in-

structions. Then, we check whether each *CALL* instruction is read by *ISZERO*. If there is a *CALL* that is not checked by *ISZERO*, this contract defect is detected.

4.3.5.8 Block Info Dependency

Detecting this contract defect is similar to *Strict Balance Equality*. This contract defect can allow miners to control the contract, as miners can change the value of some block information, which affects the result of the conditional expression. If the conditional expression contains block related instructions, i.e., “*BLOCKHASH*”, “*COINBASE*”, “*NUMBER*”, “*DIFFICULTY*”, “*GASLIMIT*”, it means the contract contains this contract defect.

4.4 Evaluation

To measure the efficacy of *DefectChecker*, we present results based on applying it to an open-sourced dataset and present our experimental results analysis in this section.

4.4.1 Experimental Setup

All experiments were performed on a PC running Mac OS 10.14.4 and equipped with an Intel i7 6-core CPU and 16 GB of memory. We use Solidity 0.4.25 as the compiler to compile source code into bytecode, and use EVM 1.8.14 to disassemble the bytecode to its opcodes.

4.4.2 Dataset

The dataset we used to evaluate *DefectChecker* was released in our previous work [31]. We first crawled all 17,013 open sourced smart contracts from Etherscan. Then, we randomly selected 600 smart contracts from these contracts. We found 13 smart contracts do not contain any contents. Thus, we removed them from our dataset. Finally, we obtained 587 smart contracts from Etherscan. These contracts have 231,098 lines of the code and more than 4 million Ethers in their balance.

Table 4.3: Some Features of Dataset

Features	Min	Max	Mean	SD
Lines of Code	5	2,239	393.6	356.8
# of Functions	1	174	30.1	621.6
# of Instructions	7	15,355	3,597.3	2,523.7
CC	1	132	30.3	22.4
Ethers	0	1,500,000	7,844.9	1,704,552.7

Table 4.3 shows some key features of the dataset, i.e., lines of code, number of functions in the contracts, number of instructions in the contracts, cyclomatic complexity [137] and Ethers held by the contracts. Cyclomatic complexity is a software metric that indicates the complexity of a program, and it is computed by analyzing the control flow graph. The formulation to compute it is: $E - N + 2P$. E is the number of edges on CFG; N is the number of nodes on CFG and P is the number of connected components on CFG. Since CFG is a connected graph, so P always equal to 1, and the formulation can be simplified as: $E - N + 2$.

The simplest contract in our dataset only contains one constructor function with 7 instructions and a cyclomatic complexity of 1. The contract with the highest cyclomatic complexity has 11,696 instructions and 2,004 lines of code. The richest contract in our dataset holds 1.5 million Ethers, while the poorest contract has no Ethers in its balance.

Two authors of our previous work manually labeled the dataset. They both have three years of experience working on smart-contract-based development and research, and took part in the process of defining contract defects. Thus, they have a very good understanding of the smart contract programming and contract defects introduced in this paper. They first manually labeled the dataset independently. Then, they discussed the disagreements after completing the labeling process and gave the final results. Their overall Kappa value [47] was 0.71, which shows a substantial agreement between them.

In this work, we developed a tool named *DefectChecker* to detect eight contract defects

with severity impact levels 1-3. The numbers of each type of contract defect in our dataset are shown in Table 4.4. This shows that *Block Info Dependency* is the most frequent contract defect in our dataset, while *Transaction State Dependency* and *Strict Balance Equality* are the least popular. Their numbers are 42, 5, and 5, respectively. *DefectChecker* aims at Solidity version 0.4.0+, which is the most widely used version at the time of writing this paper [74]. However, some smart contracts are designed for Solidity version 0.2.0+ and 0.3.0+. Thus, we removed eight smart contracts and used the remaining 579 smart contracts as our ground truth.

Among the six tools we introduced in Table 4.5, only *Zeus* open sourced their dataset. However, *Zeus* still has four kinds of defects which are not included in their dataset. Also, the *Zeus* authors did not provide the detail of how to built their dataset. Their paper only mentioned that “they manually validated each result” without providing any details, e.g., the number of people who labeled the dataset, and whether they are professional smart contract developers or not. Thus, we did not use these datasets.

4.4.3 Evaluation Methods and Metrics

There are seven measurements obtained from our experiments: True Positive (TP), True Negative (TN), False Positive (FP), False Negative (FN), Precision (P), Recall (R) and F-Measure (F). TP indicates the results which correctly predict a contract defect in a smart contract. TN indicates the results which correctly predict a smart contract does not have a defect. FP and FN indicate the results which incorrectly predict that a smart contract contains and does not contain a contract defect. *Precision*, *Recall*, and *F-Measure* can be calculated as:

$$Precision = \frac{\#TP}{\#TP + \#FP} \times 100\% \quad (4.1)$$

Table 4.4: Experimental results for *DefectChecker*.

Defects	#Defects	#TP	#TN	#FP	#FN	P(%)	R(%)	F(%)
<i>TSD</i>	5	5	474	0	0	100.0	100.0	100.0
<i>DuEI</i>	6	6	466	7	0	46.2	100.0	63.2
<i>SBE</i>	5	4	474	0	1	100.0	80.0	88.9
<i>RE</i>	12	10	461	6	2	62.5	83.3	71.4
<i>NC</i>	13	9	464	2	4	81.8	69.2	75.0
<i>GC</i>	6	6	473	0	0	100.0	100.0	100.0
<i>UEC</i>	22	20	454	3	2	87.0	90.9	88.9
<i>BID</i>	42	41	437	0	1	100.0	97.6	98.8

$$Recall = \frac{\#TP}{\#TP + \#FN} \times 100\% \quad (4.2)$$

$$F\text{-Measure} = \frac{2 \times Precision \times Recall}{Precision + Recall} \times 100\% \quad (4.3)$$

4.4.4 Experimental Results and Analysis

Table 4.4 summarizes the results of applying *DefectChecker* to our previous work’s dataset. The first column is the contract defects that need to be detected. The second column is the number of contract defects in our dataset (ground truth). The remaining seven columns are used to measure the performance of *DefectChecker*. Below, we discuss the analysis of each contract defect.

(1). Transaction State Dependency. *DefectChecker* detects 5 smart contracts containing this contract defect among 579 smart contracts with 0 false positives and negatives.

(2). DoS Under External Influence. *DefectChecker* detects 13 smart contracts that have this contract defect among 579 smart contracts with 7 false positives and 0 negatives. The 7 errors are due to the error identification of a loop.

In our detection method, we first split the bytecode into several blocks. Then, symbolic execution is used to find the edge between blocks. We traverse the path of CFG by using *DFS*. If there is a block that has been visited, we regard this block as the start of the loop (See Section 5.3.4.2). Since we regard all the paths are reachable, thus we only flag whether two blocks have an edge. This mechanism leads to false positives in detecting loops.

In Listing 5.10, all the L9, L10, and L11 hold a single block, respectively, and function *sub()* holds several blocks. EVM first executes the block of line 9, then executes the blocks of function *sub()* in line 2. After the execution of blocks of line 10, line 11, respectively, the blocks of function *sub()* will be executed again. Therefore, when traversing the CFG by using DFS, we can find that there is a cycle (fun *sub()*→L10→L11 →fun *sub()*). Since we regard all the paths are reachable, we cannot know that the blocks of function *sub()* cannot jump the block of L10, after executing the block of L11.

This kind of false positive can be addressed if we execute the loop continuously. Using a loop “*for(int i = 0; i < 100; i++)*” as an example; we need to record the state of variable *i*, and check whether the expression (*i < 100*) is satisfied or not. If we prove the loop can execute continuously, we can confirm it is a real loop not the error we show in Listing 5.10. However, we need the assistance of an SMT solver to execute the loop, and executing the loop continuously is also time consuming. Thus, we believe the advantages of removing the use of an SMT solver in our approach outweighs the disadvantages.

```

1 library SafeMath {
2     function sub(uint256 a, uint256 b) internal returns (uint256) {
3         assert(b <= a);
4         return a - b;}}
5 contract Mainsale {
6     using SafeMath for uint256;
7     uint256 public total;
8     function() payable {
9         uint amount = total.sub(100);
10        msg.sender.transfer(amount);
11        uint contri = msg.value.sub(amount);}}
```

Listing 4.10: Error Loop Example

(3). *Strict Balance Equality*. *DefectChecker* detects 4 smart contracts that contain

Strict Balance Equality with 0 false positives and 1 false negative. The cause of the error is that the contract defect related to several functions. For example, the contract in listing 5.11 uses a global variable *balance* to represent the contract's balance. Callers first call function *getBalance* to obtain the balance. The balance will then be checked in Line 5. To detect this contract defect, we need to know that the global variable *balance* represents the contract balance. Therefore, the contract defect can only be detected when we know users will first invoke *getBalance()* and then call *DefectFunction()*. However, it is not easy to detect this contract defect at the bytecode level, as the two operations (i.e., *balance == 1 eth* and *balance = this.balance*) are in two independent functions, and we do not know the calling sequence.

```

1 contract Demo{
2     uint balance = 0;
3     function getBalance(){ balance = this.balance; }
4     function DefectFunction() {
5         if( balance == 1 eth )
6             doSomething(); }
```

Listing 4.11: Strict Balance Equality - False Negative Example

(4). Reentrancy. *DefectChecker* detects 16 smart contracts that contain *Reentrancy*, with 6 false positives and 2 false negatives. The false positives are because of error-money-call detection. A smart contract contains *Reentrancy* must have a *Gas-Unlimited-Money-Call*. To detect it, we first need to check whether the gas limits set are larger than 2,300 gas and the transfer amount is larger than 0. However, in some examples, these two values are represented by complicated symbolic expressions. Some expressions also contain values that read from storage (read by *SLOAD*). Thus their specific values can not be determined by static analysis. Therefore, *DefectChecker* failed to detect them. When *DefectChecker* encounters complicated symbolic expressions, the default value is larger than 2,300 gas and larger than 0, this leads to false positives. When detecting this contract defect, we need to check whether the Slot ID read by *SLOAD* instruction still holds when executing *CALL* instruction. Some Slot IDs are also represented by complicated symbolic expres-

sions. *DefectChecker* failed to detect whether they are equal, which leads to reporting false negatives.

When detecting Money-Call, we use *Gas-Limited-Money-Call* as default, if we cannot figure out the exact value of the gas limit symbolically. We also conduct another experiment, which uses Gas-Limited-Money-Call as the default. However, *DefectChecker* failed to detect any *Reentrancy* default. The reason is that the *Gas-Limited-Money-Call* usually is easy to detect, as *address.transfer()*, *address.send()* will put a specific value “2300” to the EVM stack. Thus, we just need to detect the specific value. However, the gas limit of Gas-Unlimited-Call is not easy to detect, as it usually uses a complicated expression to represent the gas. Since *address.call.value()* will not change the gas cost. In most situations, this method will not lead to an out-of-gas error. This is the reason why we use Gas-Unlimited-Call as our default.

(5). Nested Call. *DefectChecker* detects that 11 smart contracts contain a *Nested Call* defect. Among these 11 smart contracts, we have 2 false positives and 4 false negatives. The cause of the false positives is also the error identification of the loop, which is the same with *DoS Under External Influence*. The false negatives are because of the complicated data structure. When detecting this contract defect, the first step is to know whether the loop iterations are related to the array’s length. We use the *SLOAD* instruction related pattern to obtain the loop iterations, as described in Section 4.3.5.5. However, as shown in Listing 5.12, *self* is a structure and its length is obtained through an external function. Since external functions can be designed in different ways, it is challenging to design a pattern to detect it.

```
1 for (uint i; i<self.keys.length; i++) {  
2     self.data[ self.keys[i] ].transfer(1 Ether);}
```

Listing 4.12: Nest Call - False Negative Example

(6). Greedy Contract. *DefectChecker* detects 6 *Greedy Contracts*, with 0 false positives and negatives.

Table 4.5: Input and Defects Detected of Each Tool

Tools	Input	TSD	DuEI	SBE	RE	NC	GC	UEC	BID	# of Other Defects
<i>DefectChecker</i>	Bytecode	✓	✓	✓	✓	✓	✓	✓	✓	0
<i>Oyente</i> [131]	Bytecode				✓			✓	✓	1
<i>Maian</i> [146]	Bytecode						✓			2
<i>Securify</i> [186]	Bytecode				✓			✓		7
<i>Mythril</i> [51]	Bytecode	✓	✓	✓	✓	✓	✓			28
<i>Contractfuzzer</i> [111]	Bytecode + ABI				✓		✓	✓	✓	3
<i>Zeus</i> [113]	Source Code	✓			✓		✓	✓	✓	3

```

1 function Example(Address addr) returns (bool) {
2     return addr.send();

```

Listing 4.13: Unchecked External Call - False Positive Example

(7). **Unchecked External Call.** *DefectChecker* reports 23 contracts have this kind of contract defect, with 3 false positives and 2 false negatives. We analyzed the false positive examples and find that these contracts use the return value of *send()* as function's return value and check the return value in other functions. For example, *addr.send()* as shown in listing 5.13 is the return value of function *Example*, and the value is checked in the callee programs. The false negatives are because the defect happens in a constructor function, while the bytecode of the constructor function is not contained in runtime bytecode. Therefore, we missed it. However, the contract defects in the constructor function will not harm the deployed contracts, as the constructor function will only be executed once when deploying the contracts to the blockchain.

(8). **Block Info Dependency.** *DefectChecker* detects 41 smart contracts contain this contract defects, with 0 false positives and 1 false negative. The cause of the false negative is similar to the one with *Strict Balance Equality*. The defect contract uses a global variable to represent block information and uses this global variable in other functions, which causes the contract defect to be detected.

Table 4.6: Experiment result of *Oyente*.

Defects	#Defects	#TP	#TN	#FP	#FN	P(%)	R(%)	F(%)
<i>RE</i>	12	2	94	373	10	2.1	16.7	3.7
<i>UEC</i>	22	16	448	9	6	64.0	72.7	68.1
<i>BID</i>	42	11	431	6	31	64.7	26.2	37.3

Table 4.7: Experiment result of *Mythril*.

Defects	#Defects	#TP	#TN	#FP	#FN	P(%)	R(%)	F(%)
<i>TSD</i>	5	0	474	0	5	0	0	0
<i>DueI</i>	6	1	245	228	5	0.4	16.7	0.8
<i>SBE</i>	5	0	474	0	5	0	0	0
<i>RE</i>	12	5	280	187	7	2.6	41.7	4.9
<i>NC</i>	13	2	414	52	11	3.7	15.4	6.0
<i>UEC</i>	22	11	436	21	11	34.4	50.0	40.8

4.4.5 Comparison with state-of-the-art tools

In our previous work, we investigated whether there are existing tools that can detect some of the contract defects we have defined. We first collected all the papers from top Security and SE conferences/journals, i.e., CCS, S&P, USENIX Security, NDSS, ACSAC, ASE, FSE, ICSE, TSE, TIFS, and TOSEM from 2016 to 2019. Then, we only retain the papers whose titles have the key words “smart contract”, “Ethereum” or “blockchain”. After that, we manually read the abstract to verify their relevance. Finally, we found only four papers that are related to smart contract defects, i.e., Oyente [131], Maian [146], Zeus [113], and ContractFuzzer [111].

To enlarge our baseline methods, we use the same method as proposed by Kitchenham et al. [118]. We first read the references of these 4 relevant papers, and tried to find whether there are existing tools that can detect the defined contract defects. If there is a relevant paper, we read its references repeatedly, until no new paper can be found. In this way we also found two other tools, i.e., Securify [186] and Mythril [51].

Table 4.8: Experiment result of *Securify*.

Defects	#Defects	#TP	#TN	#FP	#FN	P(%)	R(%)	F(%)
<i>RE</i>	12	1	439	28	11	3.5	8.3	4.9
<i>UEC</i>	22	10	457	0	12	100.0	45.5	62.5

Table 4.5 shows the input and contract defects that can be detected by these tools. The last column shows the number of the defects can be detected by these tools except the mentioned 8 contract defects. As we know, the bytecode of smart contract on Ethereum are visible to everyone, but only less than 1% of the smart contracts open up their source code [42]. Therefore, detecting contract defects from the bytecode level is very important. To make the comparison fair, we select *Oyente*, *MAIAN*, *Securify* and *Mythril* as our baseline tools, since they can detect contract defects at the bytecode level, the same as *DefectChecker*. However, we found that *Maian* has not been updated to support the latest Ethereum environment and so we could not run *MAIAN* on our dataset. For example, they use methods provided by *web3* [202] to obtain contracts' information on Ethereum. However, the methods they used have been removed and did not support the current version of Ethereum that we used. In addition, *DefectChecker* gets 100% F-Measure when detecting *Greedy Contract*. In this case, we do not compare with *MAIAN*, and choose *Oyente*, *Securify* and *Mythril* as our baseline tools.

Table 4.9: Result Comparison(F-Measure) between Four Tools

Tools	TSD	DuEI	SBE	RE	NC	GC	UEC	BID
<i>DefectChecker</i>	100.0%	63.2%	88.9%	71.4%	75.0%	100.0%	88.9%	98.8%
<i>Oyente</i>	/	/	/	3.7%	/	/	68.1%	37.3%
<i>Securify</i>	/	/	/	4.9%	/	/	62.5%	/
<i>Mythril</i>	0%	0.8%	0%	4.9%	6.0%	/	40.8%	/

Oyente detects three kinds of security-related vulnerabilities for smart contracts. These three kinds of security-related vulnerabilities are the same as our *Unchecked External Calls*, *Block Info Dependency* and *Reentrancy*. *Mythril* [51] is a tool developed by *ConsenSys*, which is a leading global blockchain technology company. They find security problems from online posts or news, which is similar to our previous work [31]. Our previous work analyzed the posts from StackExchange posts and defined 20 contract defects. *Mythril* can detect 6 contract defects as shown in Table 4.7. *Securify* is a smart contract security analyzer that takes EVM bytecode as input. It first decompiles EVM bytecode and analyzes

the semantic facts of the decompiled code. In our study, *Securify* uses several security patterns to detect related vulnerabilities. *Securify* can detect *Reentrancy* and *Unchecked External Call*, which can also be detected by *DefectChecker*.

Table 4.6 shows the results of running *Oyente* on our previous dataset [31]. The F-score of *Oyente* in detecting RE, UEC, and BID are 3.7%, 68.1%, and 37.3%, respectively, while the numbers for *DefectChecker* are 71.4%, 88.9%, and 98.8%, respectively. We found that *Oyente* only considers *BLOCKHASH* instructions when detecting *Block Info Dependency*, while there are many other instructions, e.g. NUMBER (NUMBER instruction is used to get block's number), that can lead to this contract defect. Besides, *Oyente* also has many false positives when detecting *Reentrancy*. The reason is that they do not distinguish between *send()*, *transfer()* and *call()* functions at the bytecode level, while *send()* and *transfer()* will limit gas to 2300 unit, which cannot cause *Reentrancy*. In addition, the most important reason for these errors is *code coverage*. Code coverage means the percentage of instructions executed. The average code coverage for *Oyente* is 18.9%, while the number for *DefectChecker* is 77.1%. Low code coverage means only a small part of the code can be analyzed for contract defect occurrence, which can lead to a large number of false positives and negatives. There are three reasons that lead to the low coverage of *Oyente* compared to *DefectChecker*. First, *Oyente* checks whether a path can be reached, while *DefectChecker* assumes that all the paths are reachable. *Oyente* also only optimizes for Solidity Version 0.4.19, but there is a wide version coverage in our dataset. Finally, the jump positions of some unconditional jump might not be easy to find. To be specific, the jump position might be a result of a complicated expression. Thus both *Oyente* and *DefectChecker* can fail to detect these unconditional jumps, and it is the reason why *DefectChecker* misses some blocks.

Table 4.7 shows the results of *Mythril*. *Mythril* fails to detect *Transaction State Dependency* and *Strict Balance Equality* in our dataset. In addition, its results contain many

false positives, especially in detecting *Reentrancy* and *DoS Under External Influence*. We found that *Mythril* is similar to *Oyente* - it fails to distinguish between *call()* with *transfer()* and *send()*, which will not lead to *Reentrancy*. Besides, *Mythril* failed to distinguish loop related patterns, which lead to errors when detecting loop related defects, e.g., *DoS Under External Influence* or *Nest Call*.

Table 4.8 presents the results of *Securify*. *Securify* can detect two common defects with *DefectChecker*, i.e., *Reentrancy* and *Unchecked External Call*. All the *DefectChecker*, *Oyente*, *Mythril*, and *Securify* can detect these two defects. The performance of *Securify* in testing *Reentrancy* (4.9%) is better than *Oyente* (3.7%), and similar to *Mythril* (4.9%), but much worse than *DefectChecker* (71.4%). In terms of detecting *Unchecked External Call*, the F-score of *Securify* (62.5%) is a little bit worse than *Oyente* (68.1%) and much better than *Mythril*. *DefectChecker* still get the best F-score, which receives 88.9% in detecting *Unchecked External Call*

To compare the results between all four tools, we add a comparison of F-measure in Table 4.9, which shows that *DefectChecker* obtains the best F-measure of all four tools.

Table 4.10: Overall Precision, Recall, and F-Measure of Each Tool

Tools	O. P. (%)	O. R (%)	O. F. (%)
<i>DefectChecker</i>	88.3	90.9	88.8
<i>Oyente</i>	54.6	38.2	40.9
<i>Securify</i>	65.9	32.4	42.2
<i>Mythril</i>	13.3	30.2	16.5

Table 4.11: Time Consumption of Each Tool

Tools	Avg.	Max	Min	S.D.
<i>DefectChecker</i>	0.15s	2.42s	0.04s	5.43
<i>Oyente</i>	18.48s	1,096.32s	0.28s	2,877.64
<i>Securify</i>	21.55s	1,203.99s	0.37s	3,384.39
<i>Mythril</i>	103.55s	2,480.26s	1.58s	13,063.80

We also calculate the overall precision, recall, and F-measure of all four tools on the

whole experimental dataset. Using overall-precision as the example, the overall result is calculated by $\frac{\sum_{i=1}^n p_{c_i} \times |c_i|}{\sum_{i=1}^n |c_i|}$, in which p_{c_i} is the precision of the contract defect i , $|c_i|$ is the number of contract defect i in the whole dataset. The results are given in Table 4.10, which clearly shows that *DefectChecker* obtains the best results in detecting contract defects.

Time Consumption. We calculate the time to analyze one smart contract to evaluate each tool. To make the evaluation accurate, we kill all the background processes in our machine when testing the tool to ensure the environment is clean. For each tool, we run it for 10 times and record the average time to test one smart contract in our dataset.

Table 4.11 shows the time consumption results of each tool. The second column of the table gives the average time consumption to test a smart contract for each tool. The speed of *DefectChecker* is the fastest in these four tools. It only needs 0.15s to analyze one smart contract. *Oyente* and *Securify* have similar running times. *Oyente* needs 18.48s to analyze one smart contract, and the time for *Securify* is 21.55s. *Mythril* is the slowest tool; it needs 103.55s to analyze one smart contract. The maximum time to analyze a smart contract of *DefectChecker* is 2.42s, while the time for *Oyente*, *Securify*, and *Mythril* are 1096.32s, 1203.99s and 2480.26s, respectively. The simplest smart contract in our dataset only contains 7 lines with a single constructor function. *DefectChecker* needs 0.04s to analyze it, while the time for *Oyente*, *Securify*, and *Mythril* are 0.28s, 0.37s and 1.58s, respectively. *DefectChecker* also has the smallest Standard Deviation value among these four tools, which shows that *DefectChecker* has the most stable speed in analyzing a smart contract.

In conclusion, the efficiency of these four tools is in order: *DefectChecker* > *Oyente* > *Securify* > *Mythril*.

4.4.6 Threats to Validity

Internal Validity. We used a dataset released in our previous work [31] as the ground truth to evaluate *DefectChecker*. Since the people who developed *DefectChecker* are the same as the people who labeled the dataset, it is likely that their familiarity with the dataset might lead to potential optimization or omissions when developing *DefectChecker*. We tried to use the datasets of the baseline tools to evaluate *DefectChecker*. However, we failed to find the dataset. Luu et al. run *Oyente* on 19,366 contracts. They only manually check the correctness of some examples, instead of using a complete dataset to evaluate *Oyente*. We can only find some false positive and true positive values on their paper. *Securify* uses a complete dataset which consists of 100 smart contracts. However, they do not open their dataset to the public. *Mythril* is a tool from industry. They even do not have an evaluation section in their technical papers. Thus, we had to build our own dataset. To reduce the influence of our dataset, we first wrote a few demo smart contracts when developing *DefectChecker* and used these to conduct small-scale testing of our proposed tool. Then, we conducted large-scale testing by using real world bytecode we crawled from the Ethereum blockchain. The dataset is the same as that we introduced in Section 5.5. During this large-scale testing, we randomly choose a set of smart contracts that can find their source code. We use these smart contracts to improve the performance and patterns that are used to detect contract defects. We admit that the familiarity with the ground truth dataset might lead to a bias, but the methods we used to develop *DefectChecker* can reduce this influence.

External Validity. The dataset we used to evaluate *DefectChecker* is based on manual analysis, which may contain false positives and negatives. To address this problem, we double-checked the results and used them to update the dataset when we found some mistakes. Another threat is that Solidity is a fast-growing programming language. There are nine versions released in 2018, which may add or modify any features of the previous version. *DefectChecker* is designed based on Solidity version 0.4.0+, which is the most

popular version in the time of writing the paper [74]. In the future, more smart contracts may use higher versions, which may make our tool unable to work.

4.5 A Large Scale Evaluation

In the previous section, we showed that *DefectChecker* has an excellent performance when applied to a small scale dataset. In this section, to validate *DefectChecker* is still usable to find contract defects in real-world smart contracts, we ran *DefectChecker* on a large scale dataset that we crawled from Ethereum blockchain, and show the contract defects as found by *DefectChecker*. We give two real-world attacks as case studies to show how harmful these contract defects are.

4.5.1 Dataset

To identify whether contract defects are actually prevalent in a large-scale, real-world dataset, we crawled bytecode from Ethereum blockchain by 2019.01 and obtained 183,706 distinct bytecode. Since some smart contract versions are not supported by *DefectChecker*, and so we removed them from our experimental dataset. Finally, we ran *DefectChecker* on 165,621 distinct smart contract bytecode. All these bytecode are runtime bytecode. Runtime bytecode does not contain information on their constructor function. It is the default bytecode stored on the Ethereum.

4.5.2 Contract Defects on Ethereum

We ran *DefectChecker* on 165,621 smart contract bytecode. The detailed results are given in Table 4.12, which aims to show the frequency of each defect on Ethereum. Since *DefectChecker* only identifies whether a contract contains a defect or not, if the same kind of defects appears multiple times in a smart contract, we only count it once in Table 4.12. The second column of the table shows how many contracts contain related defects, and the

Table 4.12: Contract Defects in Ethereum

Contract Defects	# Defects	# Percentage
<i>Transaction State Dependency</i>	1,669	1.0%
<i>DoS Under External Influence</i>	2,116	1.3%
<i>Strict Balance Equality</i>	390	0.2%
<i>Reentrancy</i>	3,892	2.4%
<i>Nested Call</i>	1,043	0.6%
<i>Greedy Contract</i>	3,139	1.9%
<i>Unchecked External Calls</i>	12,439	7.5%
<i>Block Info Dependency</i>	5,201	3.1%

last column gives the percentage of how many contracts contain the defect. If a contract contains multiple defects, all of the defects are counted.

Unchecked External Calls is the most frequent contract defect in the Ethereum, and about 7.5% of real world smart contracts contain this defect. There are about 3.1% of smart contracts that contain *Block Info Dependency*, which is the second most popular contract defect on the blockchain. *Strict Balance Equality* is the rarest of our contract defects. *DefectChecker* only detects 390 smart contracts that have this contract defect. The percentage of *Nested Call* is also less than 1%, with 1,043 (0.6%) smart contracts having this kind of contract defect. The percentage of *Transaction State Dependency* and *DoS Under External Influence* are similar on Ethereum, at about 1.0% and 1.3%, respectively. There are 3,139 greedy contracts on the Ethereum, and 3,892 smart contracts containing the *Reentrancy* problem, which can lead to serious security problems.

We found that there are 16 smart contracts that contain 4 kinds of contract defects, which are thus the most defective contracts. The number of smart contracts that contain 3 kinds of contract defects is 539, and 3,520 smart contracts contain 2 kinds of contract defects. About 25,815 smart contracts contain at least one kind of defect, which means that about 15.9% smart contracts on Ethereum contain some kinds of defects, as reported by our *DefectChecker*.

We utilized cyclomatic complexity [137] and the number of instructions to conduct a

further analysis. We computed the cyclomatic complexity and number of instructions for contracts in our dataset. We found that the average cyclomatic complexity of smart contracts in Ethereum is 21.3, and the average number of instructions are 2,342.6. Figure 4.3 shows the relationship between the number of the contract defects that contained in smart contracts and the number of instructions & cyclomatic complexity. The x-axis means the number of contract defects in a smart contract. The left y-axis is the number of x, and the right y-axis is the number of cyclomatic complexity. The two lines have a similar trend.

The number of instructions is proportional to the length of a contracts' code, which can show the contracts' complexity at the code level. The number of cyclomatic complexity indicated the complexity of a program. We performed a generalized linear regression with the Poisson error distribution model provided by R [93] to analyze the relationship between the number of defects with instructions, and the number of defects with cyclomatic complexity. In our model, we use the number of instructions and cyclomatic complexity to predict the number of defects, respectively. Since both the correlation coefficients are positive (0.001 with std. error = 0.0009 and 0.023 with std. error = 0.0179, respectively), it shows that the more complex a contract is, the higher is its probability to contain defects. We calculated the correlation level between these two complexity measures using the Pearson correlation method [13] at a 5% significance level. The statistical test shows that the correlation coefficient is 0.702 with $p - value < 0.05$. These correlation results imply that the number of instructions and cyclomatic complexity is correlated, and we can use only one of them as a predictor.

4.5.3 Case Study

DefectChecker found some real-world attacks / financial loss from our large-scale testing on the full Ethereum dataset. In this subsection, we give two examples to show the importance of detecting such contract defects.

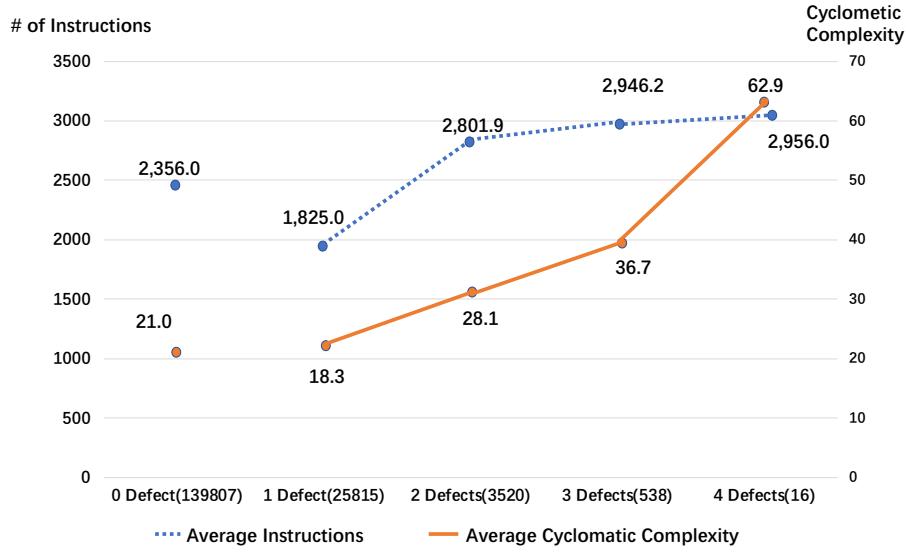


Figure 4.3: The relationship between the number of contract defects and number of Instructions & Cyclomatic Complexity

Case Study 1: The first example is shown in Listing 5.14. There are 2,335.8 Ethers in the contract balance, and it is worth \$552,720 by Mar. 2020. Unfortunately, all the Ethers are locked because of the contract defect, i.e., *Nested Call*. The buggy function in Listing 5.14 is named *sendReward()*. We highlight two lines of the code (Line 2 and Line 14), which are related to two contract defects, i.e., *Nested Call* and *DoS Under External Influence*.

There is a loop in the function *sendReward()*, and the loop iterations are increased with the length of *investors[]*. However, the contract does not limit its loop iterations. As we know, sending Ethers is expensive as it needs a large amount of gas consumption, and the contract sends Ethers to the contract users in Line 14. So, the gas consumption of executing *sendReward()* will increase in the length of *investors[]*. When we check the transaction of the contract, we can find that the contract can work normally at first, as the total gas consumption of *sendReward()* does not exceed its maximum gas limitation at that time. However, with the increase of the length of *investors[]*, the total gas cost increases rapidly. The gas cost then eventually exceeds the gas limitation, and leads to an out of gas error. Even worse, since the length of *investors[]* cannot be reduced, once the error

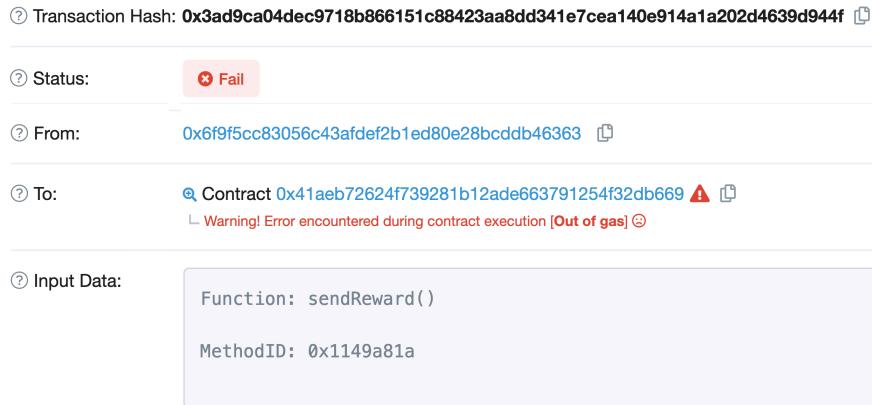


Figure 4.4: Transaction Detail of Case Study 1

happens, the *sendReward()* cannot be called anymore, which means all the Ethers in the balance are locked forever. Figure 4.4 shows the detail of a failed transaction. It is clear that when a user calls *sendReward()*, the out-of-gas error happens.

```

1 function sendReward() public isOwner{
2     for (uint i = 0; i < investors.length; i ++){
3         address _add = investors[i];
4         User memory _user = addressToUser[_add];
5         if (_user.gameOver){
6             autoReInvest(_add);
7             _user.rebirth = now - (oneLoop / 2);
8             addressToUser[_add] = _user;
9         } else {
10             if (SafeMath.sub(now, _user.rebirth) >= oneLoop){
11                 address payable needPay = address(uint160(_add));
12                 uint staticAmount = getStatic(_add);
13                 if (staticAmount > 0){
14                     needPay.transfer(staticAmount);
15                 }
16             ...
17         }
18     }
}

```

Listing 4.14: Case Study 1 - Contract with Nested Call. Code from Contract: 0x41AeB72624f739281b12aDE663791254F32DB669.

It should be noticed that although the financial loss in the real world example is caused by *Nested Call*, the contract shown in Listing 5.14 also has another contract defect, namely *DoS Under External Influence*. This contract defect can also lead to the lock of Ethers. Specifically, if the *needPay* (Line 14) is a contract address, the maximum *Gas Limit* will

be restricted to 2300 gas units, which is not enough to transfer Ethers. Thus, an *out-of-gas error* will happen in Line 14, and the Ether transfer cannot succeed.

Case Study 2: A second example is a bank contract, which is shown in Listing 5.15. Users can send Ethers to the *Deposit()* function, and withdraw its Ethers by calling the *CashOut()* function. First, the contract sends Ethers on Line 11 and then reduce the caller's balance on Line 12. However, it can lead to the *Reentrancy* if the caller is an attacking contract. When the victim contract sends Ethers to the attack contract. The fallback function of the attack contract can recall the *CashOut()* function, and steal Ethers of the victim contract. Then, all of the balance in the contract was stolen by the attackers.

Figure 4.5 shows an attacking transaction which was launched by an attacking contract. The address of the attacking contract starts with 0xdefbe, and the address of the victim contract starts with 0xbabfe. The attack happens three times on block 4919015, 4919567, and 4919662, respectively. First, the attacking contract sent 1 Ether to the victim contract. Then, the victim contract returned back Ethers to the attack contract. From these 3 attacks, the attacking contract stole about 5 Ethers from the victim contract, which were worth about \$1,200 at the time of writing the paper. We only show one example in Figure 4.5. Actually, the victim contract was attacked by multiple attacking contracts, so the financial loss was far more than 5 Ethers.

```

1 function Deposit() public payable{
2     if (msg.value >= MinDeposit){
3         balances[msg.sender] += msg.value;
4         TransferLog.AddMessage(msg.sender, msg.value, "Deposit");
5     }
6 }
7
8 function CashOut(uint _am)
9 {
10    if (_am <= balances[msg.sender]){
11        if (msg.sender.call.value(_am)()){
12            balances[msg.sender] -= _am;
13            TransferLog.AddMessage(msg.sender, _am, "CashOut");
14        }
15    }
16 }
```

Block	From	To	Value
4919850	0xbabfe0ae175b84...	0xdefbe144c325d3...	2.000000000002421899 Ether
4919662	0xdefbe144c325d3...	0xbabfe0ae175b84...	1 Ether
4919567	0xbabfe0ae175b84...	0xdefbe144c325d3...	2.000000000002421899 Ether
4919567	0xbabfe0ae175b84...	0xdefbe144c325d3...	1 wei
4919567	0xbabfe0ae175b84...	0xdefbe144c325d3...	1 Ether
4919567	0xdefbe144c325d3...	0xbabfe0ae175b84...	1 Ether
4919015	0xbabfe0ae175b84...	0xdefbe144c325d3...	2.000000000004843799 Ether
4919015	0xbabfe0ae175b84...	0xdefbe144c325d3...	1 wei
4919015	0xbabfe0ae175b84...	0xdefbe144c325d3...	1 Ether
4919015	0xdefbe144c325d3...	0xbabfe0ae175b84...	1 Ether

Figure 4.5: Transaction Lists of Case Study 2

Listing 4.15: Case Study 2 - Contract with Reentrancy. Code from Contract: 0xbABfE0AE175b847543724c386700065137d30e3B.

4.5.4 Threats to Validity

Internal Validity. The dataset we used was crawled from Ethereum, which contains different Solidity versions. *DefectChecker* only supports versions higher than 0.4.0+, and about 20,000 contracts had to be removed from our dataset, which may influence the overall results. However, the bytecode we removed is from many years ago, since the first version of 0.4.0+ was released on Sept. 2016. Even though there are many contract defects in the removed bytecode, these do not represent current smart contract usage.

Another key threat is that we used our *DefectChecker* to get the results, but *DefectChecker* also reports false positives and negatives, as shown in the previous section. However, *DefectChecker* is the most accurate and efficient tool that detects contract defects in the bytecode level, as we also demonstrated in the previous section. Therefore, we believe the results and our conclusions from it are reasonable.

External Validity. There are more than 1,000 smart contracts being deployed to Ethereum every day [38]. Many guidance and security detection tools [132, 146] are released to the public, which can help to improve the quality of smart contracts. In this case, the contract defects in smart contracts may decrease, which may lead to different results to what we found and reported in this section.

4.6 Related Work

Contract Defects on Smart Contracts. Our previous work [31] is the first work that defines 20 smart contract defects on Ethereum by analyzing the post on StackExchange [175]. We first crawl all 17,128 Stack Exchange posts by the time of writing the paper and use key words to filter solidity related posts. After getting Solidity related posts, two authors of the paper use *Open Card Sorting* to find 20 contract defects and divide them into five categories, i.e., *security*, *availability*, *performance*, *maintainability*, and *reusability defects*. According to their paper, although previous works define several security defects, they did not consider the practitioners' perspective. Therefore, we first designed an online survey to collect feedback from developers to validate whether the developers regard the contract defects are harmful. This feedback showed that all the defined contract defects are harmful to smart contracts. We assigned five impact levels to the defined 20 contract defects according to our survey results and the symptoms of the defects. According to our definition, contract defects with impact level 1-3 can lead to unwanted behaviors of contract, e.g., a contract being controlled by attackers.

Smart Contract Security Problems and Detection Tools. Luu et al. [131] introduced four security issues in their work, i.e., mishandled exception, transaction-ordering dependence, timestamp dependence, and reentrancy attack. They proposed a tool named *Oyente*, which is the first symbolic execution based bug detection tool for smart contracts. They first split the bytecode into several blocks, and built a skeletal control flow graph for the

detected contract. Then, they utilized Z3 [55] as their SMT solver and symbolically executed each instruction to obtain the full control flow graph. Finally, they designed different patterns to detect whether the input contracts contain the defined security problems. Oyente measured 19,366 existing Ethereum contracts and found 8,519 of them contain the defined security problems.

Kalra et al. [113] developed a tool named Zeus. The tool feeds source code as input and translates them to LLVM bytecode. Zeus can detect seven kinds of security problems (four of them are the same with Oyente), and the other three problems are *unchecked send*, *Failed send*, *Integer overflow/underflow*. They also compared their result to Oyente and found Oyente contains many false positives and false negatives. Zeus crawled 1,524 distinct smart contracts from Etherscan [74], Etherchain [67] and EtherCamp [66] explorers to evaluate their tool. The result illustrates that about 94.6% of contracts contain at least one security problem. However, the needs of source code limited their usage.

Jiang et al. [111] proposed a tool named *ContractFuzzer* to test seven security issues. *ContractFuzzer* is the first tool that utilizes fuzzing technology to detect security problems on smart contracts. They tested 6,991 smart contracts and found that 459 of them have issues. However, only less than 0.5% of smart contracts open their ABI to investigate on Ethereum [74], while their tool needs smart contract ABI or source code to generate test case, which limited their usage. In addition, our dataset consisted of 579 bytecode smart contracts, which are not supported by *ContractFuzzer*.

Nikolic et al. [146] developed a tool named MAIAN, which contains two major parts: symbolic analysis and concrete validation. Similar to Oyente, MAIAN utilizes symbolic execution and defines several execution rules to detect these security issues. Their tool takes input data as either bytecode or source code. MAIAN has a different concern compared to our tool. They focus on security issues that can lead to a contract not able to release Ethers, can transfer Ethers to arbitrary addresses, or can be killed by anybody. Their results were

deduced from 970,898 smart contracts and they found that a total of 34,200 (2,365 distinct) contracts contain at least one of these three security issues.

ConsenSys is a leading blockchain technology company. They built a website named SWC Registry [171] (Smart Contract Weakness Classification and Test Cases) to collect smart contract security problems from both online posts and news through crowdsourcing. *Mythril* [51] is a tool to detect security problems on this SWC Registry, and their first version was released in May 2018. The method used by *Mythril* is similar to *Oyente*. It first builds a CFG and utilizes Z3 [55] as an SMT solver. Then, it designs several rules to detect related problems. *Mythril* is a tool developed by industry; their instruction manual does not contain any evaluation section on the tool.

Securify [186] is a tool released by Tsankov et al. *Securify* is the first tool that utilizes semantic information to detect security problems on smart contracts. It first decompiles EVM bytecode to and analyzes the semantic facts, including data flow and control flow dependencies. Finally, it checks several security patterns that are written in a specialized domain-specific language to detect related security problems. *Securify* focuses on two kinds of security problems, i.e., *Stealing Ether* and *Frozen Funds*. There are 9 security issues can that be detected by *Securify*. Tsankov et al. evaluate their tool based on two datasets. First, a large-scale evaluation based on 24,594 smart contracts. Their results show that more than 70% of smart contracts contain at least one of the security problems. Then, they use a small-scale evaluation based on 100 smart contracts to evaluate their proposed tool's effectiveness. To simplify manual inspection, all of these 100 smart contracts are up to 200 lines of code. According to their paper, *Securify* can find more security violations compared to *Oyente* and *Mythril*.

In this paper, we propose a tool named *DefectChecker*, which is the most accurate and the fastest symbolic execution model of smart contract defect detection tool. *DefectChecker* can detect contract defects by analyzing bytecode, while *Zeus* and *ContractFuzzer* need

source code and contract ABI, respectively. The bytecode of smart contracts are visible to everyone, while only 1% of smart contracts open up their source code and ABI for the public [42], which restricts their usage. *MAIAN* uses a dynamic analysis method to detect security problems, which is different from our static analysis method. However, we find their tool can not support the current version of Ethereum that we used. *Oyente*, *Mythril*, and *Securify* use symbolic execution to detect security problems, which are similar to *DefectChecker*, but *DefectChecker* uses *Stack Event* and *Feature Detector* to instead the usage of SMT solver, which makes *DefectChecker* requires less runtime and yet is more accurate than these tools.

Oyente, *Mythril*, and *Securify* can detect other contract defects that are not supported by *DefectChecker*. Especially for *Mythril*, which can detect 34 kinds of contract defects. We admit that some tools can detect more contract defects than *DefectChecker*, but it is not the main motivation of this paper. Previous works, e.g., *Oyente*, *Securify*, only proposed several security defects of smart contracts without validating they are really harmful. This is not beneficial for the development of the smart contract ecosystem. In our previous work, we validated whether smart contract developers consider the contract defects we found from StackExchange posts are harmful by using an online survey. In this paper, we proposed *DefectChecker*, which aims to automatically detect the validated contract defects. We use *Oyente*, *Mythril*, and *Securify* as baseline methods with the aim to show the method we use is more accurate and efficient than these state-of-the-art tools.

Our *DefectChecker* is extensible. As shown in Figure 4.1, there are three components of *DefectChecker*, i.e., *CFG Builder*, *Feature Detector*, and *Defect Identifier*. *Defect Identifier* uses eight different rules to identify the contract defects, while the other two components can also be used to detect other defects. When detecting other defects, we can define new rules that use the data provided by our Feature Detector, CFG, and Stack Event components. There are many tools built based on the top of Oyente. For example, our previous work

GasChecker [36] is a tool to detect gas-inefficient Smart Contracts. The tool uses the CFG generated by *Oyente* to detect related gas-inefficient issues. *DefectChecker* has higher efficiency in generating CFG compared to *Oyente*. *GasChecker* can also use the CFG generated by *DefectChecker*. Thus, *DefectChecker* is also extensible to detect other kinds of issues.

4.7 Conclusion and Future Work

In this paper, we proposed *DefectChecker*, which utilizes symbolic execution to detect smart contract defects by analyzing the contracts’ bytecode. *DefectChecker* uses different rules to detect 8 contract defects and achieves a very good result when running on our previous work’s dataset. The scores for our tool are much higher than those of the state of the art work e.g. (*Oyente*, *Mythril*, and *Securify*). We also crawled 165,621 distinct bytecode smart contracts from Ethereum and ran *DefectChecker* on these. Our results show that about 15.89% of smart contracts on Ethereum contain at least one instance of our 8 identified kinds of contract defects.

Two groups can benefit from this work. For smart contract developers, they can utilize *DefectChecker* to check their smart contracts and make them more robust. As *DefectChecker* can detect contract defects from bytecode without the need for source code, developers can utilize *DefectChecker* to check whether the smart contracts they call are secure or not, even if the callee contracts are not open sourced. This can also make their contracts safer. For software engineering researchers, *DefectChecker* provides a good framework to help them solve other smart-contract-related research problems as the CFG generated by *DefectChecker* can be used for other purposes.

DefectChecker has some false positives / negatives when detecting defects, e.g., *NC*, *DuEI*. As we described in Section 5.4.4, adding a SMT Solver can reduce some error cases, while it will also increase the time consumption for analyzing a contract. Future work could

explore how to combine the method used by *DefectChecker* and a SMT solver, to balance both efficiency and accuracy. Specifically, researchers could identify which kinds of code patterns can lead to the errors made by *DefectChecker*. For example, *DefectChecker* regards all paths to be reachable, while some conditional expressions are always evaluated to false, which can lead to the false positives in detecting loops. Developers can use a SMT solver to check the conditional expression in the loop related blocks. This method can increase the accuracy in detecting loop related blocks.

Chapter 5

Why Do Smart Contracts Self-Destruct? Investigating the Selfdestruct Function on Ethereum

Chen, J., Xia, X., Lo, D., Grundy, J.C. Why Do Smart Contracts Self-Destruct? Investigating the Selfdestruct Function on Ethereum, to appear in *ACM Transactions on Software Engineering and Methodology*.

Abstract: The *selfdestruct* function is provided by Ethereum smart contracts to destroy a contract on the blockchain system. However, it is a double-edged sword for developers. On the one hand, using *selfdestruct* function enables developers to remove smart contracts (SC) from Ethereum and transfers Ethers when emergency situations happen, e.g. being attacked. On the other hand, this function can increase the complexity for the development and open an attack vector for attackers. To better understand the reasons why SC developers include or exclude the *selfdestruct* function in their contracts, we conducted an online survey to collect feedback from them and summarize the key reasons. Their feedback shows that 66.67% of the developers will deploy an updated contract to the Ethereum after destructing the old contract. According to this information, we propose a method to find the self-destructed contracts (also called predecessor contracts) and their updated version (successor contracts) by computing the code similarity. By analyzing the difference between the predecessor contracts and their successor contracts, we found five reasons that led to the death of the contracts; two of them (i.e., *Unmatched ERC20 Token and Limits of*

Permission) might affect the life span of contracts. We developed a tool named LIFESCOPE to detect these problems. LIFESCOPE reports 0 false positives or negatives in detecting *Unmatched ERC20 Token*. In terms of *Limits of Permission*, LIFESCOPE achieves 77.89% of F-measure and 0.8673 of AUC in average. According to the feedback of developers who exclude *selfdestruct* functions, we propose suggestions to help developers use *selfdestruct* functions in Ethereum smart contracts better.

5.1 Introduction

The great success of Bitcoin [141] shows the enormous potential of blockchain technology [18]. People usually regard Bitcoin as a representative of blockchain 1.0 [124], the first generation of blockchain technology. In blockchain 1.0, the blockchain technology is usually used to make cryptocurrency [207], e.g., Bitcoin, Ripple Coin [164]. The usage scenario of cryptocurrencies in blockchain 1.0 is limited, as the main application for them is storing and transferring values.

The birth of Ethereum [68] changed this situation at the end of 2015. Ethereum leverages a technology named *smart contracts*, which can be regarded as a program that runs on the blockchain. Smart contracts are usually developed in a high-level programming language, e.g., Solidity [173]. The blockchain technology provides an immutability feature for smart contracts, which means all of the smart contracts are self-executed and can't be modified. Even the creator of the contract cannot modify the code after deploying the contract to the Ethereum. By utilizing a smart contract, developers can easily design their DApps (decentralized applications) [205]. The appearance of Ethereum marked the point that blockchain technology upgraded from blockchain 1.0 to blockchain 2.0. By Sept. 2019, millions of smart contracts on the Ethereum [74] have been applied to different fields, such as gaming [53] and monetization [78], with many other application domains under exploration.

However, the features of Ethereum also make it easy to be attacked. First, Ethereum is a permission-less network; smart contracts on Ethereum can be executed by everyone, including attackers. Second, all the data stored on the blockchain, transactions, and bytecode of smart contracts are visible to the public, which makes smart contracts become attractive targets for attackers. In 2016, attackers utilized a vulnerability (reentrancy [131]) to attack a smart contract owned by an organization named DAO (Decentralized Autonomous Organization). This attack made the organization lose 3.6 million Ethers¹. People usually call this attack a DAO attack [54]. Actually, the attack continued for several days and the organization even noticed that their contract had been attacked at that time. However, they could not stop the attack or transfer the Ethers because of the immutability feature of smart contracts.

This DAO attack attracted great attention from both academia and industry. Some previous works [50, 85] advice contracts add some mechanisms to stop the contracts or transfer the Ethers when emergency situations happen, e.g., a contract being attacked. In this case, the owners can reduce the impact of financial loss. Solidity provides a novel *selfdestruct* function [173]. By calling this *selfdestruct* function, a smart contract can be removed from the blockchain and all the Ethers on the contract will be transferred to a specified address, which is an unique identification for the account² on Ethereum.

This *selfdestruct* function is however a double-edged sword for developers. On the one hand, the function enables contract owners have the ability to reduce financial loss when emergency situations happen. On the other hand, this function is also harmful. The function might open an attack vector for attackers. It may also lead to a trust concern from the contract users, as the contract owners can transfer user's Ethers that are stored on the contract. These conflicting features make the *selfdestruct* function valuable to be

¹Ether is the cryptocurrency generated by the Ethereum platform. An Ether worth \$1270 on Jan. 2021.

²There are two types of accounts on Ethereum, i.e., External Owned Account (EOA) and contract account. EOA is controlled by users. Contract account is controlled by its code.

investigated. **In this paper, we call a smart contract that has executed the *selfdestruct* function as a ‘self-destructed contract’.** To better understand the developers’ perspective about this unique Ethereum smart contract *selfdestruct* function, we designed an online survey to collect their opinions, and to help us to answer the following research question:

RQ1: Why do smart contract developers include or exclude selfdestruct functions in their contracts?

We sent our survey to 996 smart contract developers and received 88 responses. Their feedback shows that there are six reasons why developers exclude the *selfdestruct* function. The top two most popular reasons are *security concerns* and *trust concerns*. Developers are worried that the *selfdestruct* function in their contract will open an attack vector for attackers. Besides, this function can also reduce users’ confidence of the contract as the contract owner have ability to transfer users’ Ethers that stored on the contract balance. To address these concerns of developers, we provide six suggestions in Section 5.6, which can help developers to better use the *selfdestruct* function. In terms of why developers include a *selfdestruct* function, our survey feedback shows that two thirds of developers will kill their contracts when security vulnerabilities are found, or if they want to upgrade their smart contract’s functionalities. After fixing bugs or upgrading the contracts, they will deploy a new version of the contract. This finding inspired our second research question:

RQ2: Why do smart contracts on Ethereum self-destruct?

We called the *self-destructed* contract as a ‘*Predecessor*’ contract, and its upgraded version as a ‘*Successor*’ contract. By comparing the difference between *Predecessor* contract and *Successor* contract, we can identify the reasons why contract destructed, e.g., security reasons. We propose a method that leverages a clone detection tool (SMARTMBED [89, 90]) to find *Predecessor* contracts and their *Successor* contracts. Then, we summarize 5 common reasons why contracts destructed by conducting *open card sorting* [174].

As a result, we summarize 5 common self-destruct reasons, detailed in Table 5.2. Two

of them – *Unmatched ERC20 Token* and *Limits of Permission* – might affect the life span of contracts. Therefore, an automatic tool to detect these problems would be helpful to extend the life span of smart contracts. This motivated us to investigate our third research question:

RQ3: How can we detect lifespan-based smart contract problems automatically?

We designed a tool named LIFESCOPE, which can be used to detect *Unmatched ERC20 Token* and *Limits of Permission* problems. For *Unmatched ERC20 Token*, LIFESCOPE uses ASTs (Abstract Syntax Trees) to parse the source code and extract related information. LIFESCOPE obtains 100% of F-measure for detecting this problem. For *Limits of Permission*, LIFESCOPE first transfers code to a TF-IDF representation and utilizes a machine learning method to predict the permission. LIFESCOPE achieves an F-measure and AUC of 77.89% and 0.8673 for this task.

The main contributions of this paper are:

- To the best of our knowledge, this is the most comprehensive empirical work that investigates the *selfdestruct* function of smart contracts in Ethereum. We conduct an online survey to collect feedback from developers. According to this survey feedback, we summarize 6 reasons why developers add *selfdestruct* functions and 6 reasons why they do not add them to their smart contracts.
- We design an approach to find 5 reasons why smart contracts self-destructed. These self-destruct reasons can be used as a guidance when practitioners develop their contracts. Also, our approach gives inspiration for researchers. They can use the same approach to find more self-destruct reasons and apply the method to other smart contract platforms, e.g., Ethereum Classic³ [206].
- We propose a tool named LIFESCOPE to detect two problems that might shorten the

³Ethereum Classic is another popular blockchain platform which support the running of smart contracts.

life span of smart contracts. LIFESCOPE obtains 100% of F-measure in detecting *Unmatched ERC20 Token*. And it achieves an F-measure and AUC of 77.89% and 0.8673, respectively in detecting *Limits of Permission*.

- According to the feedback from our survey, there are six common reasons why some developers do not use *selfdestruct* function. We give five suggestions for developers to address these issues and to help them better use the *selfdestruct* function in their smart contacts.

The organization of the rest of this paper is as follows. In Section 6.2, we present the background knowledge of smart contracts. Then, we show the answer to the three research questions in Section 6.3-6.5, respectively. We discuss the implication, how to better utilize *selfdestruct* function and threats to validity in Section 6.6. After that, we introduce related works in Section 6.7. In Section 6.8, we conclude the whole work and present our future work.

5.2 Background

In this section, we briefly introduce the background information about smart contracts, the Ethereum system, and some features and knowledge about smart contract programming.

5.2.1 Smart Contracts

Bitcoin was the first cryptocurrency that utilized blockchain as its underlying technology. It allows users to encode scripts to process transactions. However, the scripts on Bitcoin are not Turing-complete, which restricts the usage of Bitcoin [141]. In contrast, Ethereum leverages a technology named *smart contracts*. These can be regarded as self-executed programs that run on the blockchain. When developers deploy smart contracts to Ethereum, the source code of the contracts will be compiled into bytecode and reside on the blockchain forever. The storage of Ethereum is very expensive, as all the data stored on the blockchain

will be copied on each node, a so-called distributed ledger. To minimize the data space, the source code of the smart contracts will not be stored on the blockchain. Once a contract is deployed to the blockchain, the contract is identified by a 20-byte hexadecimal address. Arbitrary users can call the functions of a smart contract by sending transactions to the contract address.

```

1 pragma solidity ^0.4.25;
2 contract Example{
3     address owner_addr;
4     address [] participants;
5     uint participantID = 0;
6     function constructor(){
7         owner_addr = msg.sender;
8     }
9     function() payable{
10        if(msg.value != 1 Ether)
11            revert();
12        participants[participantID] = msg.sender;
13        participantID++;
14        if(this.balance == 10 Ether)
15            getWinner();
16    }
17    function getWinner(){
18        uint random = uint(block.blockhash(block.number)) % participants
19                    .length;
20        participants[random].transfer(9 Ether);
21        participantID = 0;
22    }
23    modifier onlyOwner{
24        if(msg.sender != owner_addr)
25            -;
26    }
27    function Selfdestructs(address addr) onlyOwner(){
28        selfdestruct(addr);
29    }
}

```

Listing 5.1: A simple contract

Listing 5.1 is an example of a smart contract that implements a simple gambling game by using Solidity [173]. Solidity is the most popular smart contract programming language on the Ethereum platform. Users can send 1 Ether to the contract. Once the contract receives 10 Ethers, the contract will choose 1 user as the winner randomly and send 9 Ethers to him/her.

The first line is called the version pragma, which is used to identify the compiler version of the contract. Lines 3-5 are the global parameters, and the function on line 6 is the constructor function of the smart contract. The constructor function can only be executed once when deploying the contract to the blockchain. Therefore, this function is usually used to store the owner's information. Specifically, line 7 stores the owner's address by using *msg.sender*. (*msg.sender* is used to obtain the address of the transaction sender.) Function on line 9 named fallback function, which is the only unnamed function of the smart contract. This function will be executed automatically when an error function call happens. For example, a user calls function “ δ ”, but there is no function named “ δ ” in the contract. In this situation, a fallback function will be executed to handle the error call. If the fallback function is marked by a keyword named *payable*, the fallback function will also be executed automatically when the contract receives Ethers. Lines 10 and 11 guarantee that each user sends 1 Ether to the contract. If the user sends other amounts of Ethers, the transaction will be rolled backed by executing *revert()*, which is a function provided by *Solidity*. When the contract receives 10 Ethers (line 14), the contract will choose 1 user to send 9 Ethers by using function *getWinner* (line 17). The contract generates a random number by using the block info related functions⁴ in line 18. Then, the contract sends 9 Ether to the winner in line 19.

5.2.2 Function Modifier

Ethereum is a permission-less network – everyone can call methods to execute smart contracts. Developers usually add permission checks for permission-sensitive functions. For example, the contract in Listing 5.1 records the owner's address in its constructor function (line 7). In this case, the contract can compare whether the caller's address is the same as the owner's address. Solidity provides *Function Modifiers* which are used to add prerequi-

⁴(*block.blockhash* and *block.number* are the functions provide by Solidity to obtain block related information. Since block hash number is random; so it can be used to generate random numbers sometimes.)

sites checks to a function call. A function with function modifier can be executed only if it passes the check of the modifier.

Listing 5.1 line 22 shows a modifier named *onlyOwner*. This modifier requires the transaction creator (*msg.sender*) should be the owner of the contracts (*owner_addr*). Function *Selfdestructs* on line 26 contains this modifier. Therefore, only the owner of the contract can call *selfdestruct* function in line 27.

5.2.3 Selfdestruct Function

The *selfdestruct* function in Listing 6.1 line 27 is the only way to remove the contract from Ethereum. When executing this method, the caller can transfer all Ethers on balance to a specific address (*addr*) (line 27). Then, the contract will be discarded. If others transfer Ethers to the self-destructed contract address, the Ethers will be locked forever. Calling *selfdestruct* function when a contract is no longer needed can help clean up the Ethereum environment. To motivate this, Ethereum refunds up to half of the gas used by a contract transaction calling the *selfdestruct* function to the transaction sender. This mechanism is also utilized by GasToken [76], which allows users to store gas when the gas price is low and use the gas when it is expensive. Specifically, a user can create a simple contract (GasToken) that contains *selfdestruct* function when the gas price is low. Then, the user can destruct the GasToken to save the gas when the gas price is high. However, GasTokens also have the downside to the Ethereum network, as it leads to the creation of millions of “useless” contracts, which is against the original motivation of the gas refund. EIP-3529 [194] and EIP-3298 [193] are two Ethereum improvement proposals that suggest reducing the gas refunds. EIP-3529 recommends reducing the gas refund from up to 1/2 gas used by a transaction to 1/5, and EIP-3298 even recommends removing the gas refund directly. These two EIPs imply that the GasToken might be nullified in the future.

The *selfdestruct* function is sometimes harmful as the immutability feature can be bro-

ken. Immutability is a special and important feature of smart contracts compared to traditional programs. Once a contract is deployed to the blockchain, none can modify the contract, even the owner. However, this function can allow the owner to kill the contract and make the contract disappear from the blockchain. This might reduce the confidence of the users, as the owner can transfer all the Ethers of the contract. For example, the owner can transfer all the Ethers by calling the *selfdestruct* function on contract in Listing 5.1 when the contract receives 9 Ethers. In this case, all the users are losers.

```

1 contract Victim {
2     mapping(address => uint) public userBalannce;
3     function withDraw() {
4         uint amount = userBalannce[msg.sender];
5         if(amount > 0){
6             msg.sender.call.value(amount)();
7             userBalannce[msg.sender] = 0;
8         }
9     }
10    ...
11 }
12 contract Attacker{
13     function() payable{
14         Victim(msg.sender).withDraw();
15     }
16     function reentrancy(address addr){
17         Victim(addr).withDraw();
18     }
19    ...
20 }
```

Listing 5.2: The Demo of the DAO Attack

5.2.4 The DAO Attack - A Motivation Example of the *selfdestruct* Function

In 2016, attackers found a vulnerability named Reentrancy [131, 85] in a smart contract of the Decentralized Autonomous Organization (DAO organization), and this vulnerability made the DAO organization lost 3.6 million Ethers (\$270/Ether on Feb. 2020). People usually call this infamous attack a DAO attack.

List 5.2 is a demo of the DAO attack. There are two smart contracts, i.e., *Victim* contract and *Attacker* contract. The *Attacker* contract is used to transfer Ethers from *Victim* contract,

and the *Victim* contract can be regarded as a bank, which stores the Ethers of users. Users can withdraw their Ethers by invoking *withDraw()* function. However, *withDraw()* function contains the *Reentrancy* vulnerability in line 6-7.

First, the *Attacker* contract uses *reentrancy()* function (line 16) to invokes *Victim* contract's *withDraw()* function in line 3. The *addr* in line 17 is the address of the *Victim* contract. Normally, the *Victim* contract sends Ethers to the callee in line 6, and resets callee's balance to 0 in line 7. However, Ethereum does not support concurrency, which means *Victim* contract sends Ethers to *Attacker* contract before resetting the balance to 0. When the *Victim* contract sends Ethers to the *Attacker* contract, the fallback function (line 13) of the *Attacker* contract will be invoked automatically, and line 7 is not executed at that time. So, the *Attacker* contract can invoke *withDraw()* function repeatably.

Actually, the DAO attack continued for several days and the organization even noticed that their contract had been attacked at that time. However, they could not stop the attack or transfer the Ethers because of the immutability feature of smart contracts. If the contract contains a *selfdestruct* function, the DAO organization can transfer all the Ethers easily, and reduce the financial loss.

5.2.5 ERC20 Standard

Motivated by the great success of Bitcoin, thousands of cryptocurrencies have been created in recent years. However, most of them do not have their own blockchain system. Instead, they are usually implemented by smart contracts that run on the Ethereum, also called *tokens*. To ensure different tokens can interact accurately and be reused by other applications (e.g., wallets and exchange markets), Ethereum provides ways to standardize their behaviors. ERC20 [78] is the most popular token standard on Ethereum. It defines 9 standard interfaces (3 are optional) and 2 standard events. To design ERC20 tokens, developers should strictly follow the standard. For example, the standard method *transfer* is declared

as “*function transfer(address _to, uint256 _value) public returns (bool success)*”. This function is used for transferring tokens to a specific address (*_to*). The ERC20 standard requires this function to throw an exception if the caller’s account balance does not have enough tokens to spend. Besides, the function should fire an event named “*TRANSFER*” to inform the caller whether the tokens are transferred successfully.

5.2.6 Card Sorting

Card sorting is a research method to organize data into logical groups [174]. Due to the low-tech and inexpensive nature of card sorting, it is widely used to help users understand how users would organize and structure the data that makes sense to them. The users who conduct a card sorting process first need to identify the key concepts and write them into labeled cards, which can be actual cards or a piece of paper. Then, they are asked to classify them into groups that they think are appropriate. By utilizing card sorting, users can 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 commonly used for organizing data with no predefined groups. 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 cluster into one of the groups. Hybrid card sorting combines open card sorting and closed card sorting. Hybrid card sorting has predefined groups but allows the creation of new groups during the process.

5.3 RQ1: Developer’s Perspective about *selfdestruct* Function

5.3.1 Motivation

Usage of the *selfdestruct* function can enable developers to destruct their contracts and transfer Ethers when emergency situations happen, e.g. a contract is being attacked or is found to be buggy. However, this function is also harmful for both contract users and contract owners. In our analysis, we crawled all of the 54,739 verified smart contracts from Etherscan [74] by the time of writing (for details see Section 5.4.2.1), and found 2,786 (5.1%) smart contracts contain a *selfdestruct* function in their source code. In this RQ, we aim to investigate the developers’ perspective about using the *selfdestruct* function in Ethereum smart contracts. By understanding the reason why they include or exclude *selfdestruct* functions in their contracts, we can better understand the advantages and disadvantages of this function. We then want to design some guidance about using the *selfdestruct* function (can be found at Section 5.6.2), which enables developers to design a more robust smart contract.

5.3.2 Approach

5.3.2.1 Validation Survey

In this paper, we utilize the methods proposed by Kitchenham et al. [120] to design a survey for collecting the opinions from smart contract developers. To increase the response rate, we make the survey anonymous [187] and provided a raffle for developers who take part in our survey. Participation in the raffle is voluntary; we chose two respondents who provided their email addresses as the winner, and gave them \$50 Amazon gift cards as the reward. We first use a small scale survey to collect feedback about our survey. The feedback includes: (1). Whether the expression about our question is easy to understand. (2). Whether the time to finish the survey is reasonable. After the small scale survey, we

refine our questionnaire based on the feedback we collected. Finally, conduct a large scale investigation to collect our data.⁵

5.3.2.2 Survey Design

To understand the background of the respondents better, we first collect their demographic information. These five questions can help us have an overall understanding of the respondents.

a. Demographics:

- 1. Professional smart contract developer? : Yes / No
- 2. Involved in open source software development? : Yes / No
- 3. Main role in developing smart contract: Testing / Development / Management / Other
- 4. Experience in years (decimals ok)?
- 5. Current country of residence ?

After that, the respondents are required to choose yes / no in the question 6. If they choose yes, they are required to answer question 7; otherwise, they should answer question 8. Both question 7 and 8 contain a textbox, which enables respondents to input their answer. (There is no length requirement / restriction of their inputs.)

b. Questions about selfdestruct Functions:

- 6. Will you add *selfdestruct* functions in your future smart contracts? : Yes (Go to Q7) / No (Go to Q8)
- 7. Why do you add *selfdestruct* functions?
- 8. Why do you not add *selfdestruct* functions?

⁵ETHICS COMMITTEE APPROVAL

To increase the response rate, we prepare two kinds of survey⁶, i.e., English Version and Chinese Version, as Chinese is the most spoken language and English is an international language in the world. The Chinese version survey is carefully translated to ensure the contents between the two versions are the same.

5.3.2.3 Recruitment of Respondents

To receive sufficient response from different backgrounds, we first sent our questionnaire to our contacts who are working in world-famous blockchain companies or doing related research in academic institutions, e.g., *Ant Financial, The Hong Kong Polytechnic University, NUS, The University of Manchester*. Then, we also collect developers' email addresses on their Github homepage who are contributing to open-sourced blockchain projects. We collected 1,238 email addresses from Github. Due to the scale of the smart contract projects, 1,238 are the numbers of contributors of the top 100 most popular (ranked by stars) smart contract related projects, which is a good number compared to previous smart contract related surveys [32, 22, 28].

5.3.3 Result

Since some email addresses we collected are illegal or abandoned, we successfully sent our survey to 996 developers, and receive 88 responses from 32 countries (The response rate is 8.84%). The top three countries in which respondents reside are China (29.89%), the USA (8.05%) and the UK (5.57%). Three of the respondents claim that they are not professional smart contract developers and have no experience in developing smart contracts. Therefore, we exclude their responses and use the remaining 85 responses for analysis. The average years of experience in developing Ethereum smart contracts are 1.96 years (standard deviation is 1.05) for all of our respondents. As the survey was undertaken in Sept. 2019 (about

⁶The two surveys and related feedback can be found at: <https://zenodo.org/record/5518527#.YUl-bWYZYUE>

4 years since Ethereum was first published), 1.96 average years of experience shows that they have good experience in developing smart contracts. Among these respondents, 62 (72.94%), 10 (11.76%), 5 (5.88%) described their job roles as development, testing, and management, respectively. The other 8 respondents said they have multiple roles, such as security auditor and research.

Guided by previous works [224, 32], we performed open card sorting [174] to analyze the survey feedback to summarize the reasons why developers include or exclude the *selfdestruct* function in their smart contracts. Feedback that we received in Chinese was first translated into English. After that, we manually converted each feedback into several separate units with coherent meaning, as some feedbacks contain several reasons. Then, a card was created for each separate unit with a title (ID) and description (feedback content). Two experienced smart contract researchers were involved in the card sorting. The detailed steps are:

Iteration 1: Two researchers randomly chose 20% of the cards. Each card was analyzed by the two researchers together. They were required to summarize a detailed reason, e.g, Security concern, Trust concern. If the root concern is unclear, they omit the card from the card sort.

Iteration 2: The same two researchers analyzed the remaining 80% of the cards independently by following a similar method as iteration 1. Some new reasons are found in this step. After they have gone through the cards independently, they compared their results and discussed any differences. Finally, 6 reasons for including and 6 reasons for excluding *selfdestruct* function were summarized.

There is a threat that what developers told us, i.e., their reasons for adding a self-destruct function to their smart contracts, may be different from what they do in reality. We do not claim that our user study is final and complete; rather, we view it as a first step to better understand the usage of *selfdestruct* function. We invite others to replicate our study with

additional surveys and interviews.

5.3.3.1 Reasons for including the *selfdestruct* function

33 (38.82%) of the respondents claim that they will add the *selfdestruct* function in their smart contracts. We analyzed the feedback of these respondents and summarized five key reasons. **As some respondents give more than one reason, the sum of these is higher than 33.**

Reason 1: Security Concerns. **18 (54.55%) respondents** claim that they use the *selfdestruct* function to stop the contracts when security vulnerabilities are detected in their contracts. After fixing the vulnerabilities, they can deploy a new contract.

Reason 2: Clean Up Environment. Blockchain is a distributed ledger where each node stores all the data. After destructing the contracts, the functions of the contracts cannot be called anymore. **11 (33.33%) respondents** mention that when the duty of the contract is finished, they will call the *selfdestruct* function to remove the contracts from the blockchain, which can clean up the blockchain environment.

Reason 3: Quickly Withdraw Ethers. By using the *selfdestruct* function, the owner of the contract can remove all the Ethers to a specific address. **9 (27.27%) respondents** claim this function can help them transfer assets quickly.

Reason 4: Upgrade Contracts. **4 (12.12%) respondents** said they may need to upgrade their contracts in the future. Adding a *selfdestruct* function is the easiest method to upgrade their contract. This function allows them to remove the old version of the contract and deploy a new version.

Reason 5: Business Requirement. The business requirement is also a reason why developers add *selfdestruct* function. **2 (6.06%) respondents** said their business partners require them to add the *selfdestruct* function.

Reason 6: Gas Refund. As we introduced in Section 5.2.3, the gas refund feature

of the *selfdestruct* function allows the transaction sender to get up to half of the gas back.

1 (3.03%) respondent mentioned that he/she adds the *selfdestruct* function to get the gas back.

According to our survey, 22 / 33 respondents claim that they add *selfdestruct* function for security concerns or to upgrade contracts. These two motivations can lead to redeployment of smart contracts after developers destruct the contracts. Besides, the survey feedback also show that *selfdestruct* function is useful for contract developers to handle emergency situations, e.g., when serious security issues are found in the contracts.

5.3.3.2 Reasons for excluding the *selfdestruct* function

52 (61.18%) of the respondents claim that they will not add *selfdestruct* functions in their smart contracts. **As some respondents give more than one reason, the sum of these is higher than 52.**

Reason 1: Security Concerns. The *selfdestruct* function can also lead to serious security problems if the contract does not handle access permissions correctly or the private keys of owners are leaked. **19 (36.54%) respondents** worried that the *selfdestruct* function might open an attack vector for adversaries to exploit. Limiting the permission of calling *selfdestruct* function is not difficult. For example, a contract can only allow specific addresses to execute this function. However, it is also possible that the private keys⁷ of these addresses might be stolen. Once the private keys are stolen by attackers, the smart contract can then be destructed by attackers and all the Ethers will be lost.

Reason 2: Trust Concerns. **16 (30.77%) respondents** who give this reason believe that including a *selfdestruct* function might lead to trust concerns from the contract users. To be specific, a *selfdestruct* function allows the owner to kill the contract and make the contract disappear from the blockchain. Also, the owner can transfer all the Ethers, which

⁷There are two kinds of accounts in Ethereum, i.e., externally owned account (EOA) and contract account. A contract account is controlled by its code, and an EOA is controlled by the private key.

raises a trust concern for the user.

Reason 3: Requirement Concerns. **16 (30.77%) respondents** mention that their contracts do not use *selfdestruct* function as their contracts do not have Ethers. Therefore, they do not need to transfer Ethers. Besides, when they want to add some new functionalities, they said they could deploy a new smart contract and ignore the old one.

Reason 4: Unfamiliarity. **7 (13.46%) respondents** claim that they are unfamiliar with *selfdestruct* function. They are worried that they might misuse the *selfdestruct* function, and lead to the bugs.

Reason 5: Additional Complexity. **4 (7.69%) respondents** told us that they need to add more tests if they add *selfdestruct* functions in the contracts, which can introduce additional complexity to their contracts.

Reason 6: Additional Financial Risk. Risk of losing Ether after destroying the contract is also a concern for the developers. **2 (3.85%) respondents** worried that people may send Ethers to the self-destructed contract, and these Ethers will be locked forever.

According to our survey feedback, *selfdestruct* function might be risky for both contract developers and contract users. We find six reasons why developers destruct their contracts, and the reasons “Security concern” and “Upgrade contracts” can lead to the redeployment of smart contracts. These two reasons are also the most common reasons (66.67%) why developers destruct their contracts. This finding gives us the motivation of RQ2 that we can find some security issues by comparing two versions of contracts.

5.4 RQ2: Reasons for Self-destruct

5.4.1 Motivation

According to our survey, 22 out of 33 respondents claim that they add a *selfdestruct* function for security concerns or to upgrade contracts. These two motivations can lead to redeployment of smart contracts after developers destruct the contracts. Therefore, by compar-

Table 5.1: Information for the 756 self-destructed contracts

	Max	Min	Avg.	Median
Life Span	879.6 days	\leq 1 hour	40.8 days	4.3 days
No. Trans	100582	2	538.8	7
No. Eths	208.6	0	0.44	0

ing the difference between the two versions of the contract, we can find the reasons why contracts self-destructed. Consider the following scenario.

Bob is a smart contract developer. He developed a smart contract several weeks ago, and his company uses this contract to receive money from other companies. However, they find that a function in the smart contract does not limit the caller's permission, which can lead to serious security problems. Therefore, Bob has to destroy the contract by involving *selfdestruct* function and deploy a new contract to the blockchain. The new contract adds a permission check to avoid this vulnerability. Bob and his colleagues try their best to inform other companies not to transfer Ethers to the self-destructed contract anymore. However, it requires a long time to inform all companies. Many users still transfer Ethers to the self-destructed contract, and all the Ethers sent to the contract are lost forever. It causes a great financial loss to Bob's company.

From this scenario, we see that calling the *selfdestruct* function may lead to great financial loss. Therefore, we should try to make contracts robust. If we tell Bob that many previous contracts are destructed because a function in the smart contract does not limit the caller's permission, he might check whether his contract contains the same problem and can avoid this problem.

In this section, we compare the self-destructed contracts and their successor contracts to summarize reasons why contracts self-destructed. The reasons we identify can guide smart contract developers and help them refine their contracts.

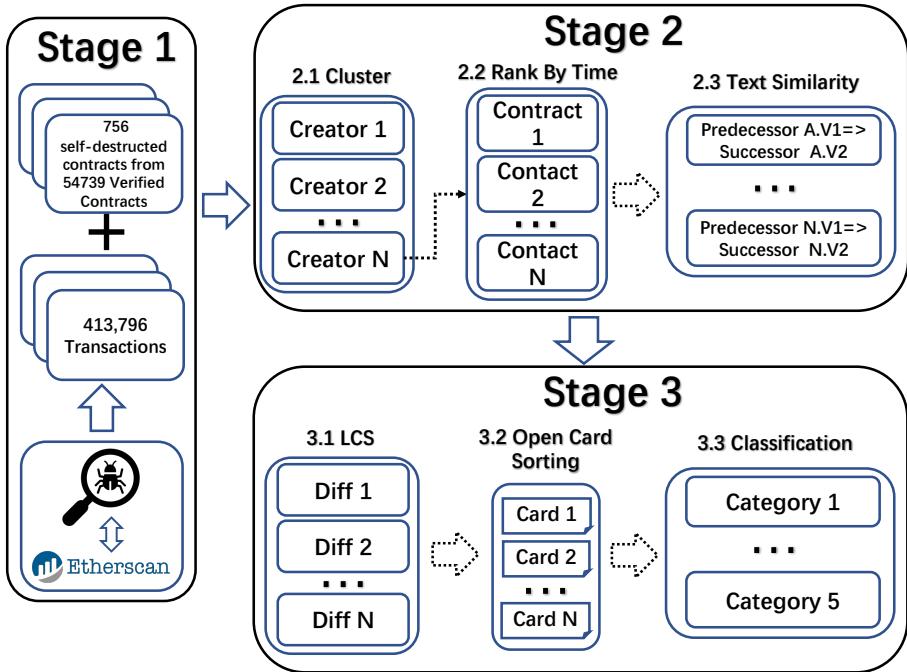


Figure 5.1: Overview architecture of finding the self-destructed reasons

5.4.2 Approach

Figure 5.1 depicts the detailed steps to identify the reasons why some smart contracts have been destructed. Our method consists of three stages. In the first stage, we crawl all verified contracts and their transactions from Etherscan. We crawled 54,739 smart contracts altogether. We found that 2,786 (5.1%) of these smart contracts among 54,739 contracts contain a *selfdestruct* function, and 756 (27.14%) contracts have been destructed. In the second stage, we first divide crawled contracts into several groups by their creators' addresses. In this case, we can find smart contracts that are created by the same authors. We only use groups that contain self-destructed contracts and rank all the contracts in the same group by contracts' creation time. Then, we compute the code similarity of contracts in each group to find self-destructed contracts (also called predecessor contracts) and their successor contracts. In the last stage, we compare the difference of predecessor contracts and their successor contracts by using *open card sorting*. Finally, we summarized 5 reasons

why contracts self-destructed.

5.4.2.1 Stage 1. Data Collection:

Stage 1 is used to collect data for the following two stages. Our data contains three parts, i.e., *verified contracts*, *self-destructed contracts*, and *contract transactions*.

Verified Contracts: Verified contracts are crawled from Etherscan. To crawl the source code of verified smart contracts from Etherscan, we first need to know the contract addresses of verified contracts. Figure 5.2 is a smart contract on Etherscan. By obtaining the contract address, we can easily download the source code, transactions, and other information of the contract. The contract address list is provided by Etherscan (<https://etherscan.io/contractsVerified>). However, Etherscan only shows the last 500 verified contract addresses since Jan. 2019. The data used in this paper are crawled before Jan. 2019 and the last 500 verified contracts by the time of the writing. We finally obtained 54,739 verified contract addresses. After obtaining the contract addresses, we can crawl the source code from Etherscan directly.

Self-destructed Contracts: Finding whether a verified smart contract has been self-destructed is straightforward. If a contract has self-destructed, there will be a label (*Self Destruct*) given on Etherscan (see Figure 5.2). We found 756 self-destructed contracts from 54,739 verified smart contracts. The detailed information (creation time, destructed time, number of transactions / balances) of each contract can be found at https://zenodo.org/record/5518527#.YUl-bWYzYUE/Contract_Info_Suicide.csv. Table 5.1 shows a brief summary of these 756 contracts. Life span is the time interval between the transaction of creation and destruction. No. Trans / Eths records the transactions and balance of a contract before its destruction, respectively.

Transactions: Transactions on Ethereum record the information of the external world interacting with the Ethereum network. All the transactions can be found on Etherscan.

The screenshot shows the Etherscan interface for a specific smart contract. At the top, there's a navigation bar with tabs for 'Buy', 'Earn Interest', and 'Crypto Credit'. Below the header, a message from Etherscan says 'Sponsored slots available. Book your slot here!'. The main content area has two sections: 'Overview' and 'More Info'. In the 'Overview' section, it shows 'Balance: 0 Ether' and 'Value: \$0.00'. In the 'More Info' section, it shows 'Creator: 0x4fa1728b8e015c1...'. Below these sections is a tab bar with 'Transactions', 'Internal Txns', 'Contract Self Destruct' (which is currently selected), 'Analytics', and 'Comments'. Under the 'Contract Self Destruct' tab, it says 'Contract Self Destruct called at Txn Hash 0xa51fd68f6fdb810e4712da8a764107ab53c5632012e3badf809ca147a9b88fd2'. A note below states 'Contract Source Code Verified (Exact Match)'. The 'Contract Source Code (Solidity)' section shows the following code:

```

1  /**

```

Figure 5.2: A smart contract on Etherscan

The screenshot shows the transaction history for the same smart contract. At the top, there's a navigation bar with tabs for 'Transactions', 'Internal Txns', 'Contract Self Destruct', 'Analytics', and 'Comments'. Below the header, it says 'Latest 6 txns'. The table lists six transactions:

Txn Hash	Block	Age	From	To	Value	[Txn Fee]
0xa51fd68f6fdb810...	5426322	683 days 5 hrs ago	0x4fa1728b8e015c1...	IN 0xfa1d63b87f40c92...	0 Ether	0.000053492
0xd7fc5479177be3...	5419608	684 days 8 hrs ago	0x58d736ac195d1e...	IN 0xfa1d63b87f40c92...	0 Ether	0.000053924
0x8fb997e1bb2f5a1...	5419588	684 days 8 hrs ago	0x58d736ac195d1e...	IN 0xfa1d63b87f40c92...	0 Ether	0.0000583924
0x60b9b574b4e0dc...	5415974	684 days 22 hrs ago	0x58d736ac195d1e...	IN 0xfa1d63b87f40c92...	0 Ether	0.000212336
0x9c4c37b093458c...	5414930	685 days 2 hrs ago	0x4fa1728b8e015c1...	IN 0xfa1d63b87f40c92...	0 Ether	0.000106168
0x923edc9ba70e9d...	5414589	685 days 4 hrs ago	0x4fa1728b8e015c1...	IN Contract Creation	0 Ether	0.007489624

Red arrows point to the 'From' address in the first and last rows, labeled 'Destructor' and 'Creator' respectively.

Figure 5.3: Transactions of a Self-destructed Contract

We collect all 413,796 transactions of 756 self-destructed smart contracts. Figure 5.3 is the transactions of a self-destructed contract. In the first transaction, we can find who deployed the contract (creator) and we can find who destructed the contract (destructor) in the last transaction.

5.4.2.2 Stage 2. PS Pairs Generation:

A *PS* (predecessor and successor) pair is denoted as: $PS = \langle \mathbb{P}, \mathbb{S} \rangle$, where \mathbb{P} is a smart contract that has executed the *selfdestruct* function, named as the predecessor contract. \mathbb{S} represents \mathbb{P} 's upgradeable version, that we call a successor contract. \mathbb{P} and \mathbb{S} are deployed by the same address and have similar functionalities.

The aim of stage 2 is to find all the PS pairs in our dataset, which helps to reduce the manual effort when summarizing the reasons for self-destruct in stage 3. For some self-destructed contracts, it is not easy to find their successor contracts. For example, in our survey feedback, some developers mentioned that they added the *selfdestruct* function is only to quickly transfer Ethers and will not redeploy the contract to Ethereum. For this kind of contracts, we cannot find their successor ones, and thus we removed them from our analysis list. Some developers said they would destruct the contracts when bugs are found. Then, they will redeploy a smart contract after fixing the bug. Thus, we can compare the difference of two versions of contracts to find the bugs, and the two versions of the contracts are likely to have high similarity. In this stage, we calculate the similarity to find the successor contracts of the self-destructed contracts. We found 436 contracts among 756 self-destructed contracts that have successor contracts.

Step 2.1 Cluster: We first find the creator addresses of all the 54,739 verified smart contracts through their transactions. In this step, we inspect the first transaction of each smart contract as the first transaction contains the creator address and creation time. Then, we classify the contracts into several groups according to their creator addresses. If two

contracts have the same creator address, they will be classified into the same group. We only choose groups that contain self-destructed contracts.

Step 2.2 Rank by Time: A PL (predecessor and alive) pair is denoted as: $PL = \langle \mathbb{P}, \mathbb{L} \rangle$, where \mathbb{P} is a smart contract that has executed the selfdestruct function. \mathbb{L} represents a smart contract that has been deployed later than \mathbb{P} . \mathbb{P} and \mathbb{L} are deployed by the same address.

In this step, we first rank contracts in each group by their creation time, which can be obtained from the first transaction. Then, we can obtain several PL pairs. For example, one group contains five contracts, they are contract a, b, c, d, e and these five contracts are ranked by creation time. Contract b and d are the self-destructed contacts in these five contracts. Finally, we output four PL pairs, i.e., (b, c) , (b, d) , (b, e) and (d, e) .

Step 2.3 Text Similarity: We compute the code similarity between two contracts to identify whether the later created contract is the successor contract of the self-destructed contract. SMARTMBED [89, 90] is the only tool that is specialized for calculating the similarity between smart contracts developed by Solidity at the time of writing this paper. According to their paper, SMARTMBED obtains excellent performance in calculating the similarity of Ethereum smart contracts and outperforms the traditional similarity-checking / clone-detection tools, e.g., Deckard [112]. Thus, we use SMARTMBED to calculate the code similarity instead of using other similarity checking techniques, e.g., vanilla [216].

SMARTMBED first converts a smart contract into a code embedding by parsing the AST of a smart contract (details see Section 5.7) and then calculate the similarity between two contracts. The similarity metric is calculated as: $Similarity(c_1, c_2) = 1 - \frac{Euclidean(e_1, e_2)}{\|e_1\| + \|e_2\|}$, where c_1 and c_2 are two smart contracts; e_1 and e_2 are their corresponding code embeddings, which can be presented as $e_i = \{w_{i_1}, w_{i_2}, \dots, w_{i_n}\}$. Euclidean function is used to point the distance between e_1 and e_2 , which is calculated by $Euclidean(e_1, e_2) = \sqrt{(w_{1_1} - w_{2_1})^2 + (w_{1_2} - w_{2_2})^2 + \dots + (w_{1_n} - w_{2_n})^2}$. Although the tool is aimed at finding bugs, their first step computes code similarity between the given smart contract and

history contracts. We modified the source code of SMARTEMBED to compute the similarity between two contracts. If their similarity is larger than 0.6, they might be relevant and we assume the later created contract is the successor of the self-destructed contract. We also called this self-destructed contract as the predecessor contract of the successor contract. We found 436 self-destructed contracts have their successor contracts with 1513 *<predecessor contract, successor contract>* pairs. We note that 0.6 is a conservative threshold (original paper assumes similarity 0.95 are cloned); we might include many irrelevant pairs in our dataset, but it will not influence our result as we conduct a manual analysis in the subsequent step. Increasing the threshold can remove some irrelevant pairs to reduce the manual effort, but it might make us miss some true matching pairs. Besides, some token contracts might have duplicated code with high similarity, but it still will not affect the results. Because we will analyze the difference of the similar token contracts manually to identify whether the later created contracts are the successor contracts of the prior contracts.

5.4.2.3 Stage 3. Reason Generation:

In stage 2, we found 436 contracts among 756 self-destructed contracts that have successor contracts. Note that the PS pairs we found in stage 2 might contain many false positives. For example, two contracts can obtain very high code similarity if they use many common open-source libraries, but they are not the valid PS pairs. Thus, we need manual analysis to remove these false positives.

It is a time-consuming and error-prone process to analyze the predecessor contract and successor contract directly. To deal with these issues, we perform two steps. First, to reduce the manual effort, we use a tool named *DiffChecker* [182] to help us find the difference between the predecessor and successor contract. The basic idea of *DiffChecker* is to use the Longest Common Substring (LCS) [82] algorithm to find the longest string (or strings) that is a substring (or are substrings) of two or more strings. We use *DiffChecker* to highlight

the difference to reduce manual efforts. Second, to increase the reliability of our results, we conduct open card sorting to summarize the reason why a smart contract self-destructed. Guided by previous works [32, 174], we create one card for each *PS* pair. Each card highlights the difference between the two contracts.

The detailed steps of the open card sorting we used are:

Iteration 1: We randomly chose 20% of the cards, and two developers with 3 years of smart contract development analyzed the difference of the code and discussed the reason why contracts self-destructed. They first quickly read the two contracts to identify whether they are relevant (The two contracts have similar functionalities). If they are irrelevant, the card will be discarded. Then, they carefully read the difference between the contracts and discuss the reason for this difference. For example, Figure 5.4 is a real example of a predecessor contract (left) and its successor contract (right) in our dataset. The three differences between the two contracts are highlighted. First, the developer added a *Transfer* event in Line 356 of the successor contract. Second, in the predecessor contract, Ethers can only be sent to an address whose balance is zero. This restriction was removed in the successor contract. Finally, the *selfdestruct* function was also removed. According to our definition, all of these three modifications change the code representation of the contract. Thus, the reason for the Self-destruct is regarded as *Functionality Changes*. For some contracts, it is not easy to find the reason for the self-destruct usage and they were omitted from our card list. All the reasons are generated during the sorting.

Iteration 2: The same two smart contract developers independently categorized the remaining 80% of the cards into the initial classification scheme. Then, they compared their results and discussed disagreements. We used Cohen's Kappa [47] to measure the agreement between the two developers. Their overall Kappa value is 0.84, indicating strong agreement.

We finally identified 5 reasons why contracts have been self-destructed. This informa-

```

353     function ZipToken() public {
354         totalSupply_ = INITIAL_SUPPLY;
355         balances[msg.sender] = INITIAL_SUP
356         PLY;
357     }
358     function distributeTokens(address[] ad
359         dresses, uint[] values) public onlyOwner {
360             require(addresses.length == value
361             s.length);
362             for (uint i = 0; i < addresses.len
363                 gth; i++) {
364                 address a = addresses[i];
365                 uint v = values[i];
366                 if (balanceOf(a) == 0) {
367                     transfer(a, v);
368                 }
369             function die() public onlyOwner {
370                 selfdestruct(msg.sender);
371             }
372         }
353     function ZipToken() public {
354         totalSupply_ = INITIAL_SUPPLY;
355         balances[msg.sender] = INITIAL_SUP
356         PLY;
357         Transfer(0x0, msg.sender, INITIAL_
358         SUPPLY);
359     }
360     function distributeTokens(address[] ad
361         dresses, uint[] values) public onlyOwner {
362         require(addresses.length == value
363             s.length);
364         for (uint i = 0; i < addresses.len
365                 gth; i++) {
366             address a = addresses[i];
367             uint v = values[i];
368             transfer(a, v);
369         }
370     }
371 }
```

Figure 5.4: An example considered in our card sort that contain a predecessor contract (0x3b96990a8ef293cdd37c8e1ad3d210a0166f40e1) and successor contract (0xedd7c94fd7b4971b916d15067bc454b9e1bad980).

tion is shown in Table 1 and the detailed information is shown in the following subsection.

5.4.3 Reasons for Self-destruct

In this subsection, we give detail explanations of the 5 self-destruct reasons and their distribution in our dataset.

5.4.3.1 Definitions

The short descriptions of 5 self-destruct reasons are given in the first two columns of Table 5.2. Below we give a detailed description each reason.

(1) Functionality Changes: Due to the immutability of smart contracts, it is not easy to upgrade smart contracts. However, during the entire smart contract life cycle, it is necessary for developers to add, remove or change some functionalities to respond to the new requirements. Functionality changes will change the code representation of a contract, e.g., Abstract Syntax Tree (AST), Control Flow Graph (CFG). According to our analysis, we find that functionality changes are the most common reason why smart contracts are

Table 5.2: Reasons of Self-destruct and their distributions among 340 self-destructed smart contracts.

Category	Description	Distribution
Functionality Changes	Adding, removing or changing Functionalities for upgrading contracts to respond new requirements. Functionality changes will change the code representation of a contract, e.g., Abstract Syntax Tree (AST), Control Flow Graph (CFG).	156 (45.88%)
Limits of Permission	Adding permission checks for the sensitive functions.	25 (7.35%)
Unsafe Contracts	Removing the security problems of the contracts.	95 (27.94%)
Unmatched ERC20 Token	Modifying the contract to make it follows the ERC 20 standard	19 (5.59%)
Setting Changes	Changing the variable or function states of the contracts, such as renaming a contract, and changing the amount of ERC20 token supplement, changing a public function to private function. Setting Changes will not remove or add new code from a contract.	56 (16.47%)

self-destructed. When new requirements appear or some requirements are changed, some developers choose to deploy a new smart contract and the old version of the contract will be destructed.

(2) Limits of Permission: Ethereum is a permission-less network [33] and anyone can call the functions of the contracts by sending a transaction. Thus, it is important to limit the access permissions for some sensitive functions. According to our analysis, we find some predecessor contracts do not check permissions of the callers in some sensitive functions, e.g., Ether transfer. Thus, everyone can execute the sensitive functions. In the successor contract, they add permission checks to limit access permissions.

Example: Figure 5.5 is a real example of Limits of Permission. The predecessor contract does not limit the permission of calling the selfdestruct function. In the successor contract, the function adds a modifier *onlyOwner* to check the permission. Thus, only the contract owner can call this function.

(3) Unsafe Contracts: Previous works [131, 113, 146, 186, 32] highlighted several security problems of smart contracts. For example, *Oyente* highlighted four security issues,

```

227     function selfdestructs() payable public {
228         selfdestruct(owner);
229     }
190     function selfdestructs() onlyOwner payable public {
191         selfdestruct(owner);
192     }

```

Figure 5.5: A real diff example of Limits of permission in our dataset. Predecessor Address: 0xfa1d63b87f40c92d27bfb255419c1ea8c49086de; Successor Address: 0x64b09d1a4b01db659fc36b72de0361f2c6c521b1

```

27     msg.sender.send(withdraw_amt); // ok send it back to me
28
29
44     if (!msg.sender.send(withdraw_amt)) throw; // everything
45     ok, send it back to me
46

```

Figure 5.6: A real diff example of Unsafe Contract in our dataset. Predecessor Address: 0x8b099bdcfea93faecfac13d0dbc1d08c4e1ec595; Successor Address: 0x17683235257f2089e3e4acc9497f25386a529507

Zeus described four security problems of smart contracts (see Section 5.7). We find many predecessor contracts contain security problems like *reentrancy*, which can lead to Ether loss. Developers usually fix these security issues in the successor contracts.

Example: Figure 5.6 is a real example of Unsafe Contracts. The predecessor contract does not check the return value of the `msg.sender.send()`, which might lead to security issues of the contract. In the successor contract, the function checks whether the Ether send is successful. If not, the transaction will be thrown.

(4) Unmatched ERC20 token: ERC20 [78] is the most popular standard interface for tokens in Ethereum. If the implementation of token contracts does not follow the ERC20 standard strictly, the transfer between tokens may lead to errors. We find many predecessor contracts are token contracts but do not strictly follow the ERC20 standard, while their successor contracts do follow the standard.

Example: ERC20 requires a transfer function to return a boolean value to identify

```

55 contract ERC20Basic {
56     uint public totalSupply;
57     function balanceOf(address who) constant returns (uint);
58     function transfer(address to, uint value);
33 contract ERC20Basic {
34     uint256 public totalSupply;
35     function balanceOf(address who) constant returns (uint256);
36     function transfer(address to, uint256 value) returns (bool);

```

Figure 5.7: A real diff example of Unmatched ERC20 token in our dataset. Predecessor Address: 0x848217a9569ca64ffba9d000cda05f9d2fa97f5; Successor Address: 0xf42230a7e21375c29648ae9544f7da394e20ead3

```

447 uint256 public constant ALLOC_TEAM = 330 * TOKEN_MULTIPLIER;
448 // 7%
449 uint256 public constant ALLOC_ADMISORS = 70 * TOKEN_MULTIPLIER;
450 // 10%
451 uint256 public constant ALLOC_FOUNDER = 100 * TOKEN_MULTIPLIER;
452 // 50%
453 uint256 public constant ALLOC_AIRDROP = 500 * TOKEN_MULTIPLIER;
454 ...
455 uint256 public constant AIRDROP_CLAIM_AMMOUNT = 500;
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
759

```

Figure 5.8: A real diff example of Setting Changes in our dataset. Predecessor Address: 0xa41aa09607ca80ee60d2ce166d4c02a71860e5c5 ; Successor Address: 0x41c6af7b388e80030e63f2686dc2ff9bfd1267c9

whether the transfer is successful. However, the transfer function in the predecessor contract in Figure 5.7 does not return anything. Users usually use third-party tools to manipulate their tokens and these tools capture token transfer behaviors by monitoring standard ERC20 method [42]. If the contract does not match the ERC20 standard, the token may fail to be transferred by third-party tools. In the successor contract, the return value of the *transfer()* function is added.

(5) Setting Changes: Similar to *Functionality Changes*, it is likely that developers will change some settings of smart contracts in response to new requirements. For example, the token-related contracts usually have some default values, e.g., total token supply, number of decimals, and token name. Due to immutability, if developers want to change the total token supply, they have to destruct the contract and deploy a new one. The main difference between *Setting Changes* and *Functionality Changes* is that *Setting Changes* will not add or remove code from contracts. Thus, the structure of AST and CFG should be the same between two contracts, if the reason for their selfdestruct is *Setting Changes*.

Example: Figure 5.8 is a real example of Setting Changes. The token supplement in the predecessor contract is too small. Thus, the successor contract changes the token supplement by multiple a value DECIMALS (DECIMALS equals to 10^{18} in the successor contract.)

5.4.3.2 Distribution

We use SMARTMBED to find the pair $\langle predecessor\ contract, successor\ contract \rangle$, and set the threshold value to 0.6, which might obtain some irrelevant pairs. After manually removing irrelevant pairs, we found 340 contracts (**some contracts have multiple self-destruct reasons. If multiple changes are found in the successor contract, we regard all of the changes as contributing to the self-destruction.**) for which we can identify the reason(s) why they are self-destructed and give the distribution of the five self-destruct reasons in the last column of table 5.2. 96 contracts cannot find the reason why they are self-destructed, and thus they are omitted from our dataset.

It is clear that *Functionality Changes*, *Unsafe Contract*, and *Setting Changes* are the top three most popular reasons that lead to contracts destructed; the number are 156, 95 and 56, respectively. The number of the other two reasons are similar, there are 25 contracts destructed for *Limits of Permission* and 19 for *Unmatched ERC20 Token*.

It should be noted that it is not easy to find all the security issues in our dataset. One the one hand, we only checked the security issues reported by *Oyente*, *Zeus*, *Mythril*, *Securify*, *Maian* [131, 113, 146, 186, 51]. On the other hand, manually checking for security issues is very error-prone and time-consuming. To reduce the errors, we utilized tools by *Oyente*, *Zeus*, *Mythril*, *Securify*, *Maian* and manually checked each contract it found. We first use the tools to check smart contracts. Then, two developers with 3 years of smart contract development experience manually identified whether the results are correct. If the reported results were different, they discussed to obtain the final result. Note that the code of *Oyente*, *Mythril*, *Securify*, *Maian* can be found on Github, and we rerun the tool to get the result. We did not find the code of *Zeus*, but *Zeus* provides their evaluation results which inform whether a contract address contains vulnerabilities or not. Thus, we use their evaluation results directly. If the detected contract does not appear in their evaluation results, we regard *Zeus* as not finding any vulnerabilities in the contract.

5.5 RQ3: LifeScope: A Self-destruct Issues Detection Tool for Smart Contracts

5.5.1 Motivation

In the previous section, we introduced five smart contract self-destructed reasons by comparing the difference between predecessor contracts and their successor contracts. Among these five reasons, *Functionality Changes* and *Change Setting* depend subjectively on the contract owner’s requirements. Specifically, different developers might make different decisions of whether a smart contract should be self-destructed according to their requirements, even if the smart contracts are the same. It is thus hard to say these two reasons can affect the life cycle of smart contracts. However, smart contracts that contain the other three self-destruct reasons might have a short life span, as they can lead to unwanted behaviors of the smart contracts. Detecting whether a smart contract contains these self-destruct reasons might increase the life span of the contract. Manual analysis is time-consuming and error-prone. Therefore, designing a tool to detect whether a contract contains these self-destructed reasons before deploying them to the Ethereum is important. Security issues is a big concept. In the last section, we use the security vulnerabilities defined in previous works, e.g., *Oyente*, *Zeus*, *Mythril*, *Security*, *Maian* [131, 113, 146, 186, 51], to find unsafe contracts. These have already proposed several tools to detect security issues with high accuracy. The accuracy of *Zeus* is almost 100% according to their paper, and designing a more accurate and comprehensive security detecting tool is not the main target of this paper. In this case, we do not redevelop a tool to detect security issues introduced in these previous works.

5.5.2 Approach

```
1 function totalSupply() public returns (uint256)  
2 function balanceOf(address _owner) public view returns (uint256 balance)  
3 function transfer(address _to, uint256 _value) public returns (bool  
success)
```

```

4 function transferFrom(address _from, address _to, uint256 _value) public
    returns (bool success)
5 function approve(address _spender, uint256 _value) public returns (bool
    success)
6 function allowance(address _owner, address _spender) public view returns
    (uint256 remaining)
7 event Transfer(address _from, address _to, uint256 _value)
8 event Approval(address _owner, address _spender, uint256 _value)

```

Listing 5.3: ERC20 Functions and Events

We propose a tool named *LifeScope* to detect the remaining two issues, i.e., *Limits of Permission* and *Unmatched ERC20 Standard* that can lead to contracts being destructed. Since the aim of *LifeScope* is extending the life span of a smart contract by finding the self-destruct reasons, and smart contracts are immutable to be modified after deploying to the blockchain. Therefore, it is meaningless to detect the two self-destruct reasons through bytecode, although smart contracts are stored in the form of bytecode.

LifeScope detects the self-destruct issues at source code level, which utilizes AST (abstract syntax tree) to parse the smart contracts and extract related information to detect *Unmatched ERC20 Standard*. For *Limits of Permission*, *LifeScope* first transfers the contract to a TF-IDF representation and then utilizes machine learning algorithms to predict this problem. These two problems are not only limited to contracts that contain the *self-destruct* function. Any smart contracts can be analyzed with *LifeScope* to detect these two problems before deploying them to the Ethereum.

5.5.2.1 Unmatched ERC20 token

The method to detect the *Unmatched ERC20 Token* is shown in Algorithm 1. The input is an AST (Abstract Syntax Tree) of a smart contract, which is generated by the *Solidity* compiler [214]. AST is a tree structure that contains the syntactic information of source code. By analyzing the AST, we can generate a list of pairs $\langle fun_n, funInfo \rangle$. fun_n is the name of a function; $funInfo$ contains information about the function fun_n , i.e., parameter types and return types. ERC20 standard defines nine functions and two events.

Algorithm 1: Algorithm to Detect Unmatched ERC20 Token

Input: AST of a smart contract
Output: Is an Unmatched ERC20 Token

```
1 Extract  $\langle fun_n, funInfo \rangle$  pair list from AST;
2 Extract  $\langle event_n, eventInfo \rangle$  pair list from AST;
3 appearedFunc = 0;
4 legalFunc = 0;
5 legalEvent = 0;
6 isMachedERC20 = false;
7 for  $fun_i, funInfo_i \in \langle fun, funInfo \rangle$  pair list do
8   if  $fun_i$  is one of ERC20 Standard Function then
9     |   appearedFunc++;
10    |   if  $funInfo_i$  is same to ERC20 Standard Function then
11      |     |   legalFunc++;
12    |   end
13  end
14 end
15 for  $event_i, eventInfo_i \in \langle event, eventInfo \rangle$  pair list do
16   if  $event_i$  is one of ERC20 Standard Event then
17     |   if  $eventInfo_i$  is same to ERC20 Standard Event then
18       |     |   legalEvent++;
19     |   end
20   end
21 end
22 if  $appearedFunc < 5$  then
23   |   return not_ERC20_Contract;
24 end
25 if  $legalFunc == 6$  and  $legalEvent == 2$  then
26   |   isMachedERC20 = true;
27 end
28 return isMachedERC20;
```

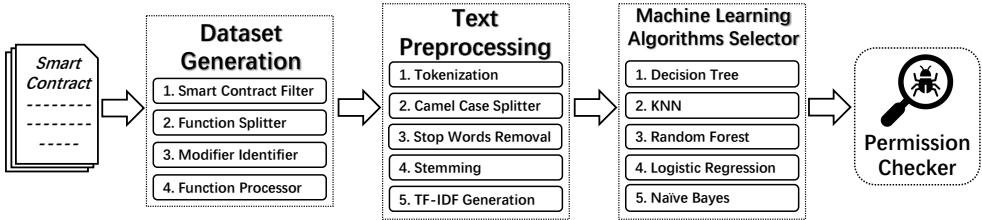


Figure 5.9: Approach to Check Permissions

Among the nine functions, three are optional, and six are compulsory. The six compulsory functions and two events are shown in Listing 5.3. We traverse all the functions in the smart contract. If the function name fun_n is one of the six compulsory functions, we then compare whether the input parameter types and return types are the same with the ERC20 standard. For example, if a contract contains a function named *transfer*, we then check whether this function contains two parameters and their types are *address* and *uint256*, respectively. Besides, the function should have a return value, and the type of the return value is *bool*. We use the same method to check contract events. Finally, if all the six compulsory functions and two events appear in the contract, this is a matched ERC20 smart contract. We follow the previous work [87], if less than five ERC20 functions appear in a smart contract, we regard it as not an ERC20 smart contract. Otherwise, it is an unmatched ERC20 token. Note that Solidity allows abstract functions or interfaces which do not have code implementation. We only consider a function as an ERC20 function if it has complete implementation. Besides, the bytecode of a public storage variable is the same as a view function without any input parameters and only has one return value. Thus, the function *totalSupply()* can also be represented as a global variable, i.e., *uint totalSupply = amount;* If a smart contract does not contain a function named *totalSupply()*, but contains a public variable named *totalSupply*, it will also be regarded as a matched ERC20 function.

5.5.2.2 Limits of Permission

It is hard to prescribe functions need to check their permissions. Therefore, It is not easy to detect this issue by using programming analysis methods. We utilize a machine learning method to predict whether a function needs to check for its caller permission. Figure 5.9 describes the overall architecture that we used. The method contains three parts, i.e., *Dataset Generation*, *Text Preprocessing* and *Machine Learning Algorithm Selector*.

(a). **Dataset Generation:** The aim of this step is to extract pairs $\langle func, permission \rangle$ from smart contracts. In each pair, $func$ is the source code of a function in the smart contract, and $permission$ means whether the function needs to check the caller's permission. Since security vulnerabilities are ubiquitous in smart contracts on Ethereum [131, 43, 113, 146], some alive contracts might also miss checking the permissions of the functions. This situation gets even worse in self-destructed contracts, as the reason for the destruct might be missing permissions. Therefore, it is not reliable to use these contracts as our ground truth. To ensure the correctness of our dataset, we should use contracts that correctly check their permissions for the contracts. However, it is not easy to ensure the correctness of the contract, and manually check whether a function needs to check its permission is also error-prone and subjective.

To obtain the dataset, we first rank all the alive verified contracts by their transaction numbers. We then choose all of the contracts whose transaction numbers are larger than 500, as transactions can be regarded as test cases for the contract. Previous works check the transaction input to detect malicious attacks [81, 199, 35]. The more the number of normal running transactions a contract has, the less likely the contract has permission problems. After this step, 5,986 contracts remained. Financial gain is an important motivation behind the attacks on Ethereum smart contracts [33]. We finally find 1 Ether can get an appropriate number of the dataset (875 smart contracts with 29,313 functions). Thus, we choose contracts whose balance have more than 1 Ether (1 Ether worths about \$1400 at Feb. 2021).

The sensitive information of smart contracts, e.g., contract bytecode, balance, are visible to the public. Finally, 875 contracts whose balance has larger than 1 Ether and transaction numbers larger than 500 transactions remain.

After getting these contracts, we first remove the comments on the contracts and then split the contracts into functions. We obtain 29,313 functions from these 875 smart contracts. We then need to identify whether these functions contain modifiers, as the permission is usually checked by the modifier. If a function contains modifier, we remove the modifier from the function. For example, the original function is *function transferMoney(address addr) onlyOwner{}*. We remove the modifier *onlyOwner* from the function and only use *function transferMoney(address addr){}* to training the machine learning model. In our dataset, we obtain 29,313 functions, with 4,393 of them needing to check permissions.

(b). Text Preprocessing: Before training the machine learning module, we need to transfer each processed function into a set of bag-of-words (BoW). The dataset is split into a training set and a test set. Both of them are first processed by the following steps:
(i) Tokenization: Each processed function is divided into a list of words by punctuation and space that usually do not contain any information. For example, *function transferMoney(address addr){}* will be transferred into "function", "transferMoney", "address" and "addr"
(ii) Camel Case Splitter: We separate function names, variable names and identifiers according the rules of Camel Case [210]. For example, "transferMoney" will be separated into "transfer" and "Money". Then, we transfer all the words to their lower case.
(iii) Stop Words Removal: Stop words means meaningless words (e.g., "to", "as", "is"). We adopt NLTK (a python library) stop words list in this step. We also remove tokens of less than 3 characters.
(iv) Stemming: this step is used to transfer words into their stem form. For example, "running" is replaced by "run". In this paper, we use Porter's stemmer [161] to transfer the words.

After these four steps, we use TF-IDF (term frequency - inverse document frequency) [7] to represent each processed word in the function. (Noticed that to make the result more reliable, the BoW model is only built by the training set.) This is described as:

$$w_{i,j} = tf_{i,j} * \log\left(\frac{\#\text{of functions}}{df_i}\right) \quad (5.1)$$

Here, $w_{i,j}$ is the weight of the word i in the function j . $tf_{i,j}$ is the term frequency of word i in the function j . df_i is the number of functions that contain word i . Finally, each function is represented as $fun_j = (w_{1,j}, \dots, w_{i,j}, \dots, w_{n,j})$.

(c). Machine Learning Algorithms Selection: Checking the permission of functions is a binary classification problem. We tried five popular machine learning algorithms to find an appropriate algorithm for predicting the permission and use the algorithm that obtains the best F-Score for this task. The five algorithms are *Decision Tree*, *KNN*, *Random Forest*, *Logistic Regression*, and *Naive Bayes*. We use a python library named `sklearn` [170] with its default configuration to implement these five algorithms.

5.5.3 Evaluation For Unmatched ERC20 Token

5.5.3.1 Dataset

The dataset for *Unmatched ERC20 token* consists of two parts, i.e., smart contracts with and without *selfdestruct* function. The dataset with *selfdestruct* function has 756 self-destructed contracts, which is the same dataset introduced in RQ2. Among these contracts, 127 of them are discarded due to the unsupported compiler version. (LifeScope supports Solidity compiler versions that are equal to or higher than 0.4.25). So, there are finally 629 self-destructed contracts in our dataset. The two researchers manually labeled the dataset and found 164 of them are ERC 20 token contracts. In these 164 ERC20 token contracts, 70 (11.13%) of them are *Unmatched ERC20 tokens*.

For the dataset without *selfdestruct* function, we use an open source dataset proposed by our previous work [32]. The dataset contains 587 smart contracts, which has the ground truth of *Unmatched ERC20 tokens*. The ground truth is labeled by two experienced researchers. They first analyzed the contracts independently, and then discussed any differences, which ensured the correctness of the dataset. Since the dataset is randomly selected, some contracts contain the *selfdestruct* function and a low compiler version. After removing the incompatible contracts, 358 contracts remained. Among these 358 smart contracts, 141 are ERC20 token contracts, and 36 (25.53%) of them have an Unmatched ERC20 token.

5.5.3.2 Result

For the dataset with *selfdestruct* function, *LifeScope* finds 70 *Unmatched ERC20 token*, with 0 false positive and negative. For the dataset without *selfdestruct* function, *LifeScope* finds all 36 Unmatched ERC20 token, with 0 false positives and negatives. The results show that *LIFESCOPE* can also detect the *Unmatched ERC20 token* problem in both smart contracts with and without *selfdestruct* functions.

5.5.3.3 Comparison:

TokenScope [42] is a novel transaction based tool to identify the inconsistency of ERC20 tokens. It can identify whether a contract is a legal ERC20 token by investigating its transactions. In this paper, we re-execute *TokenScope* and use the same dataset to compare with *LifeScope*.

The dataset with a *selfdestruct* function contains 164 ERC20 contracts. Among these contracts, 94 are legal ERC20 contracts, and 70 have an unmatched ERC20 Token. *TokenScope* correctly predicts 88 contracts are legal ERC20 tokens, and 49 are unmatched ERC20 tokens. However, it mistakenly predicts 6 contracts are not legal ERC20 tokens, and 21 are ERC20 tokens.

The dataset without *selfdestruct* function contains 141 ERC20 contracts. Among these contracts, 105 are legal ERC20 contracts, and 36 are unmatched ERC20 Tokens. *TokenScope* correctly predicts 96 contracts are legal ERC20 tokens, and 17 are unmatched ERC20 tokens. However, it mistakenly predicts 9 contracts are not legal ERC20 tokens, and 19 are ERC20 tokens.

TokenScope needs transactions to identify whether a contract has legal ERC20 tokens. However, since some contracts only have a signal transaction, this leads to the false positives of *TokenScope* (mistakenly predicting a contract is not an ERC20 token). Besides, we find *TokenScope* cannot check the return value of a function. Specifically, ERC20 standard requires the transfer function to return a boolean value, while *TokenScope* cannot identify whether a function has a return value, which leads to false negatives (mistakenly predicting a contract is an ERC20 token).

In conclusion, *LifeScope* performs better than *TokenScope*.

Etherscan also can identify whether a contract is an ERC20 contract. We did not compare the result with Etherscan because we have different standards to define whether a contract is an ERC20 token. ERC20 standard defines nine functions and two events. Among the nine functions, three are optional, and six are compulsory. For our paper, we use a definition in previous work [87]: if less than five compulsory ERC20 functions appear in a smart contract, we regard it as not an ERC20 smart contract. A smart contract is defined as an unmatched ERC20 contracts only if it has more than or equal to five compulsory ERC20 functions, and some of these functions do not follow ERC20 standards. However, Etherscan has a different definition for ERC20 contracts. Etherscan regards a contract that has ERC20 related transactions as an ERC20 contract even though it only contains one ERC20 function, e.g., *transfer()* and has related transactions. Thus, the result of Etherscan will be very different from our method, and it is the reason why we did not choose it as our comparison method.

5.5.4 Evaluation For Limits of Permission

5.5.4.1 Dataset

The method that was used to generate the dataset of *Limits of Permission* is introduced in section 5.5.2. To better evaluate the results, we use a cross-validation method of training-testing sets. First, we divided our dataset into 10 parts of equal sizes. Then, we conduct the training using 7 parts of the dataset and 3 parts for testing. Specifically, we give an ID (0 to 9) to each part. In the first round, the parts with ID 0 to 2 are the testing sets, and 3-9 are the training sets. In the last round, the parts with ID 9, 0, 1 are the testing parts, and the remaining parts are the training set. We continue this process 10 times. Finally, we report the average results. There are around 20,328 case in our training sets and 8985 cases in our testing test.

5.5.4.2 Evaluation Methods and Metrics

We use five measurements to evaluate the results, i.e., precision, recall, F1-Measure, accuracy, and AUC. Precision, Recall, F-measure, and Accuracy can be calculated as:

$\frac{\#TP}{\#TP+\#FP}$, $\frac{\#TP}{\#TP+\#FN}$, $\frac{2 \times P \times R}{P+R}$, $\frac{\#TP+\#TN}{\#TP+\#TN+\#FN+\#FP}$, respectively. TP (true positive) indicates the number which correctly predicts a function needs to add a permission check. TN (true negative) indicates the number which correctly predicts a function does not need to add a permission check. FP (false positive) and FN (false negative) indicate the number which incorrectly predicts that a function needs or does not need to add a permission check. AUC (area under the curve) is calculated by plotting the ROC curve (receiver operator characteristic).

5.5.4.3 Result

Table 5.3 shows the result of predicting *Limits of Permission* by using five machine learning algorithms. We found that the *Decision Tree* algorithm obtains the best F-Score for this task. Therefore, we finally use the *Decision Tree* to predict whether a function needs to

Table 5.3: Results of Predicting *Limits of Permission* by using Five Machine Learning Algorithms

	Precision	Recall	F-Measure	Accuracy	AUC
Decision Tree	78.91%	77.09%	77.89%	93.45%	0.8673
KNN	72.75%	50.40%	59.50%	89.71%	0.7352
Random Forest	83.73%	70.82%	76.05%	94.88%	0.8499
Logistic Regression	84.11%	52.78%	64.82%	91.40%	0.7551
Naive Bayes	23.33%	88.43%	35.66%	52.15%	0.6708

Table 5.4: Results of Predicting *Limits of Permission* for 10-fold cross-validation by using Decision Tree

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	AVG
Precision	79.92%	79.08%	77.45%	76.76%	77.14%	77.04%	79.20%	77.32%	82.96%	82.19%	78.91%
Recall	69.08%	73.14%	77.28%	88.19%	82.50%	78.40%	81.34%	76.93%	76.84%	73.39%	77.09%
F-Measure	74.10%	76.00%	77.37%	79.29%	79.73%	77.72%	80.26%	77.13%	79.78%	77.54%	77.89%
Accuracy	92.35%	93.16%	93.72%	93.97%	94.01%	93.45%	93.88%	93.33%	93.72%	92.95%	93.45%
AUC	0.8291	0.8489	0.8682	0.8896	0.8921	0.8721	0.8874	0.8653	0.8690	0.8511	0.8673

check for its callers' permission.

The detailed results of predicting *Limits of Permission* for 10-fold cross-validation by using Decision Tree are shown in Table 5.4. Our method obtains 78.91% of precision, 77.09% of recall, 77.89% of F-measure, 93.45% of Accuracy, and 0.8673 of AUC. Prior works [80, 79] suggest that a classifier performs reasonably well if its AUC is larger than 0.7.

5.5.4.4 Comparison

To evaluate the performance of *LifeScope* in detecting *Limits of Permission*, we design a keywords-based method to identify whether a function needs to check its permission. Following we describe the details of the comparison method.

Feature Selector: We use decision tree to predict whether a function needs to add a permission check, which enable us to know which words lead to a function being classified as needing permission check. Guided by previous works [102, 168, 218], we employ a widely used feature selection technique named Information Gain, to select useful features in our prediction.

Our dataset can be denoted as $F = (F_1, L_1), (F_2, L_2), \dots, (F_n, L_n)$, where F_i represents

the i^{th} function, and L^i is the label, which means whether the F_i needs to check its permission (t) or not (\bar{t}). The word vector of F_i is represented as $F_i = \{w_1, w_2, \dots, w_n\}$, where n represents the number of different words appeared in F_i , and w_i represents the i^{th} words. There are four relationships between the word w and a function F_i .

1. (w, t) : function F_i contains the word w , and the function needs to check its permission.
2. (w, \bar{t}) : function F_i contains the word w , but the function does not need to check its permission.
3. (\bar{w}, t) : function F_i does not contain the word w , but the function needs to check its permission.
4. (\bar{w}, \bar{t}) : function F_i does not contain the word w , and the function does not need to check its permission.

Based on these relationships, the information gain (IG) of word w' and label t' is defined as:

$$IG(w, t) = \sum_{t' \in \{t, \bar{t}\}} \sum_{w' \in \{w, \bar{w}\}} p(w', t') \times \log \frac{p(w', t')}{p(w') \times p(t')} \quad (5.2)$$

In the equation, $p(w', t')$ is the probability of the word w' in a function with label t' . $p(w')$ means the probability of word w' in a function and $p(t')$ represents the probability of a function with label t' .

Behaviors of permission check: Information gain reflects the amount of information required for predicting a label (needs or does not need to check the permission). The higher IG score a word has, the more important the word to distinguish the label. We rank the IG score of each word and list the top 50 words with highest information gain score in Table 5.5. The first line has the highest IG score, and the score decreases from left to right in each line. Note that a word with a high IG score means it has a high contribution

to predicting the label, not necessarily mean that it indicates whether a function needs permission check. Thus, we manually check the word in Table 5.5 to find useful behaviors.

From the words, we then summarize four kinds of behaviors that needs to check functions' permission:

(1). **Ether Transfer.** Specifically, the words "msg", "sender", "transfer", and "eth" are related to Ether transfer methods on Ethereum, i.e., *msg.sender.transfer(eth)*; the word "withdraw" reflects a behavior related to withdraw balance.

(2). **Sensitive states change**, e.g., changing permission owners, increasing or decreasing total supply of tokens, stopping or starting the functionalities. Specifically, the words "ownership", "administr", "mint" and "renounce" are all related to sensitive states of a contract.

(3). **Inline assemble and selfdestruct function.** Ethereum provides some functionalities which need to limit permission. Specifically, the words "mload" and "assemble" are related to the inline assemble. The word "selfdestruct" is related to the *selfdestruct* function.

(4). **Emergency Management.** Smart contracts are difficult to be modified once deployed. Thus, there are some functions used to handle the emergency situations. Specifically, the words "emerge", "pause", "stop", "unpause", "selfdestruct" are related to emergency management.

We use the top 50 words with the highest IG score to find four behaviors that need to check functions' permission. To prove the completeness of the four behaviors, we also manually check the top 51-100 words with the highest IG score. We find there are no additional behaviors that can be found. All the words can be classified into these four categories.

Result of the Comparison Method: In the last part, we summarize four kinds of behaviors that need to check functions' permission according to the top 50 words with the

Table 5.5: Top 50 words with highest information gain score.

ownership	transfer	name	oracle	pidx	gen	address	pause	human	round
stop	unpause	assert	online	data	addr	core	log	compress	assemble
withdraw	gap	event	mint	mask	eth	within	msg	transaction	mload
earn	emerge	spender	code	sender	renounce	determine	calc	sqrt	team
propose	math	divest	div	selfdestruct	filter	administr	confirm	investor	sale

highest IG score. According to these words, we design a simple keywords based method to identify whether a function needs to check its permission. Specifically, if a function contains one of the key word "msg.sender.transfer", "ownership", "administr", "mint", "renounce", "mload", "assemble" "emerge", "pause", "stop" and "unpause", we assume the function needs to check its permission. Notice that we merge some keywords, e.g, "msg", "sender", "transfer" to "msg.sender.transfer" as "msg.sender" are widely used in Ethereum. We finally obtain 31.53% precision, 11.41% recall, 16.75% F1-measure, and 83.01% accuracy, which are much worse as compared to our machine learning based method (78.91% precision, 77.09% recall, 77.89% F1-Measure, and 93.45% accuracy).

5.6 Discussion

We first summarize the key implications of our work for researchers, practitioners, and educators. Then, we give 6 suggestions on how to better use *selfdestruct* function according to the feedback of the survey. Finally, we summarize the main threats of validity.

5.6.1 Implications

5.6.1.1 For Researchers

Research Guidance. In this paper, we found 5 reasons why smart contracts destructed by comparing the difference between self-destructed contracts and their successor contracts. With the increasing number of smart contracts, researchers can apply our methods to find more problems that can affect the life span of smart contracts. Our study focuses on Ethereum smart contracts, but many other blockchain platforms also support the running

of smart contracts, e.g., Ethereum Classic [206], Expanse [77]. Both Ethereum Classic and Expanse were created by the hard fork of Ethereum, but currently they are independent blockchain systems with many years of development. Both of them support the running of smart contracts based on EVM and support the *selfdestruct* function. There might be some different reasons for the self-destruct; researchers can use our methods to identify the reasons on these platforms.

5.6.1.2 For Practitioners

Our work is the first that uses an online survey to collect feedback from smart contract developers on why they include or exclude *selfdestruct* function. Their feedback shows that adding a *selfdestruct* function can help developers transfer Ethers when emergency situations happen. However, using this function can also lead to several problems. To address the drawbacks of adding a *selfdestruct* function, we give 6 suggestions in the next section. These can help developers better use the *selfdestruct* function in their contracts. Smart contract developers can develop a smart contract according to our suggestions and open source the code for other developers to use. We also summarized 5 common reasons why contracts self-destructed and developed the LIFESCOPE tool to detect 2 self-destruct reasons. Removing these problems might extend the life span of smart contracts.

5.6.1.3 For Educators

selfdestruct is an important feature of smart contracts. However, most blockchain tutorials focus on teaching how to develop smart contracts and knowledge about blockchain. Educators should pay more attention to these unique functions of smart contacts. For example, educators should mention the importance and drawbacks of adding a *selfdestruct* function when they introduce this function. The feedbacks of our survey in RQ1 can provide good materials for them.

5.6.2 Towards More Secure `selfdestruct` Functions

In Section 5.3, we summarized six reasons why smart contract developers exclude the `selfdestruct` function from their contracts. In this part, we give six suggestions about how to better use `Selfdestruct` function according to the summarized worries.

Suggestion 1. Limit Usage Scenario: Adding `selfdestruct` function can increase the complexity of the development and risk of attacks. Some smart contract developers claim that the `selfdestruct` function is mainly used to remove the code and transfer Ethers. However, they do not need this function if there is no Ether in their contracts. Removing the contracts from the blockchain is also unattractive to them. Even if their contracts are attacked and controlled by attackers, they can discard the old contracts and deploy a new version. To reduce the risk and the workload of development, a `selfdestruct` function is better to add in contracts that contain Ethers.

Suggestion 2. Permission Check: Calling a `selfdestruct` function can lead to irreversible consequences. The contract has to check the permission of the caller in each transaction. A common method to check the permission is recording the owner's address in the constructor function. Then, checking whether the caller is the owner in each transaction.

Suggestion 3. Distribute the Rights and Modularization: The trust concern is an important reason why developers exclude a `selfdestruct` function. The trust concern contains two parts according to the feedback, i.e., human related and code related concern.

For human related concern, users might worry that the owner of the contract can destructure the contract and transfer all the balance if the contract has a `selfdestruct` function. For example, the `gambling` contract shown in Listing 6.1 claims that users can transfer 1 Ether to the contract. When the contract receives 10 Ethers, the contract will choose one user as the winner and transfer 9 Ethers to the winner as a bonus. However, if the contract has a `selfdestruct` function, users might worry that the owner might transfer the money out at any time.

To reduce this kind of concern, the owner could build a DAO (Decentralized autonomous organization) for the contracts. In a DAO system, the DAO controls the contracts, and the users of the contracts control the DAO by using digital tokens which give them voting rights. The users who hold tokens (voting rights) can submit a proposal, e.g., executing the *selfdestruct* function. Then, the proposal will be checked by a group of volunteers called “curators” to check the legality of the submitted proposal and the identity of the submitter. Finally, the users who owned the DAO tokens vote to accept or reject the proposal.

However, adding a DAO pattern to the contract can increase code complexity, which is also a big concern according to our developer survey feedback. With the increase of code complexity, the probability of containing security vulnerabilities is also increasing, which leads to code related trust concerns for developers. To address these two concerns, we suggest that smart contract developers can open source and modularize this part of their code (DAO patterns) to a library. Other developers can then help polish the code together, and can make the code easier to use in the future.

Suggestion 4. Delay Self-destruct Action: The Ethers sent to a self-destructed contract will be locked forever, which increases the risk of using *selfdestruct* function. As we described in Section 5.4.1, the contract owner might find it difficult to inform all the users in a short time after the contract self-destructed. In this case, some users might send Ethers to the self-destructed contract and this may lead to financial loss. To address this problem, we suggest that the contract can delay the self-destruct action and throw an event to inform the users that the contract will self-destructed in the near future. On the one hand, delaying self-destruct action can give time for voting (Suggestion 3). On the other hand, it can provide time to inform users that the contract will be destructed.

Suggestion 5. Pause Functionality: The options in Suggestion 3 and 4 require time to implement. However, when a contract is being attacked, any delay might lead to enormous

financial loss. In this case, pausing the functionality when performing the methods described in Suggestion 3 and 4 are important. OpenZeppelin provides a *Pausable* contract template [151, 152], which can be easily used through inheritance. The three functions and two modifiers are shown in Listing 5.4. The two modifiers, i.e., *whenNotPaused* and *whenPaused*, can be added to control its states. Specifically, *whenNotPaused* makes functions callable only when the contract is not paused, and *whenPaused* makes a function callable only when the contract is paused. The state of the contract is obtained by a boolean value named *_paused*, which can change its state by the function *_pause()* and *_unpause()*.

```

1 modifier whenNotPaused() { require(!paused(), "Pausable: paused"); _; }
2 modifier whenPaused() { require(paused(), "Pausable: not paused"); _; }
3 function paused() public view virtual returns (bool) { return _paused; }
4 function _pause() internal virtual whenNotPaused { _paused = true; }
5 function _unpause() internal virtual whenPaused { _paused = false; }
```

Listing 5.4: OpenZeppelin Pausable contract

Suggestion 6. Refund Values: Some smart contracts might store the values of users. For example, users might hold tokens in ERC20 contracts. Thus, destructing the smart contracts will lead to financial loss of users. Before executing the *selfdestruct* function, the contract owners should refund users' assets.

5.6.3 Tokens of Destructed Unmatched ERC20 Contracts

Unlike other kinds of contracts, ERC20 contracts usually store values (tokens) of users. Destructing the contract and transferring the balance is not enough for ERC20 contracts, as the tokens will be locked with the execution of the *selfdestruct* function. We investigate the transaction of 70 destructed unmatched ERC20 tokens introduced in Section 6.5.3 and find that 53 of them contain transactions that call *transfer()*, which means there are token transfers in these contracts. However, only 3 contracts have refund-related transactions. These three contracts pay back a certain amount of Ethers according to the tokens that users have. For the other 50 smart contracts, the contract owners destruct the contracts directly without considering the benefits of users. This situation might suggest that ERC20

tokens with *selfdestruct* function might have a high risk for users.

5.6.4 Inconsistency

5.6.4.1 *selfdestruct* functions on Etherscan vs. Survey Results.

In this paper, we collected 54,739 open-source smart contracts but only found 5.1% of them contain the *selfdestruct* function. However, in our survey, 38.82% of the developers claim that they will add *selfdestruct* functions to their contracts. There is an inconsistency between the practitioner's perception and their behavior (38.82% vs. 5.1%). The inconsistency might indicate that many developers admit the importance of *selfdestruct* function but they give up on adding it during the actual development process. The reasons why developers do not add the *selfdestruct* functions in actual developing process might have already been included in Section 6.3.3.2. Therefore, designing guidelines for using the *selfdestruct* function might be helpful. Future work can aim to design guidelines, development models or tools to address these problems. We also give five suggestions in the Section 5.6.2.

5.6.4.2 Self-destructed Contracts on Blockchain vs. Etherscan.

We summarized 5 reasons why smart contracts were destructed based on 756 self-destructed contracts collected from Etherscan. However, there are millions of destructed contracts on Ethereum blockchain. The inconsistency between the number of self-destructed Contracts on Ethereum blockchain and Etherscan might make our finding not so reliable. To investigate this inconsistency, we collected all self-destructed-related traces by Jan. 2019 (The date is the same as the verified contract we collected from Etherscan.) Table 5.6 shows the self-destructed-related information we collected from Ethereum blockchain. There are 30,486,241 self-destructed traces which were generated by 2,084,841 self-destructed contracts. When a contract is destructed, the contract will transfer its balance to another account; we call these contracts used to receive the balance as "dest" accounts. We found 19,131,801 dest accounts. Noticed that one transaction can

have several self-destructed-related traces and one self-destructed-related trace related to one self-destructed operation. There is a difference between the number of self-destruct traces and the number of self-destructed contracts because a contract that has been marked as *destructed* still exists until the end of the transaction. The contracts can still be called and may execute further self-destructs. This mechanism was used by attackers to launch the DDoS attack in block No. 2.3M to 2.7M [97]. 27,560,501 (90.4%) self-destruct traces and 19,127,397 (99.98%) dest accounts were generated in this DDoS attack. Thus, to make the analysis more accurate, we removed the self-destructed-related information. generated on this DDoS attack, and there are 2,925,740 self-destruct traces generated by 2,080,319 self-destructed contracts. An interesting finding is that all the 2,080,319 self-destructed contracts are related to only 4,404 dest accounts. Among these 4,404 dest accounts, 2,716 are EOAs, 1,263 are self-destructed contracts and 425 are alived contracts (contracts that can be found on blockchain), while the number of dest accounts of 756 destructed contracts collected from Etherscan is 472 (415 EOA, 16 self-destructed contracts and 41 alived contracts). It shows that although there are large number of self-destructed contracts on Ethereum blockchain, most of them were generated by a limited number of accounts.

Previous work [59] also investigated the self-destructed contracts on Ethereum. They analyzed 7.3 M self-destructed contracts, and 7.2M (98.6%) were short-lived contracts (4.2M, 57.53%)⁸, GasTokens (2.8M, 38.35%), and ENS (Ethereum Name Service) deeds (0.2M, 2.74%). All these 98.6% self-destructed contracts will not be included in the analysis of these papers because these contracts usually do not contain much information. For example, the recommended template of recommended GasToken [76] only contains seven instructions with two lines of code. Short-lived contracts are usually contract-created simple contracts that are created and destructed in a single transaction. Thus, there is no need to open source these contracts on Etherscan and will not be included in our analysis.

⁸A contract called short-lived when the creation and subsequent selfdestruct is executed in the same transaction.

Table 5.6: self-destructed-related Info. on Ethereum blockchain

	Trace	self-destructed Contracts	Dest Accounts	Trans.
Total	30,486,241	2,084,841	19,131,801	374,735
Block No. 2.3M - 2.7M	27,560,501	4,522	19,127,397	53,374
Others	2,925,740	2,080,319	4,404	321,361

Table 5.7: Open Interview Questions (Excerpt)

ID	Question
1	Do you used selfdestruct function?
2	Do you know the gas refund feature?
3	Do you know selfdestruct function can refund gas?
4	Do you add selfdestruct function for refunding gas? why?
5	Do you know GasToken and how do you think about it?

Our analysis and Di Angelo and Salzer’s [59] finding can show that the self-destructed contracts we collected from Etherscan have a good coverage of selfdestruct function usages apart from DDoS attack, GasToken, and ENS deeds. It is also likely that we might miss to report some usages as the limitation of manual analysis used in this paper. However, this paper also highlights a new direction and might trigger more studies in the future.

5.6.4.3 *GasToken* Contracts on Blockchain vs. Survey Results.

Previous work [59] reported that about 38.35% self-destructed contracts on Ethereum blockchain are GasToken. In our survey, only one respondent mentioned that the motivation of including the *selfdestruct* function is refunding gas. To investigate this inconsistency, we performed an interview to collect developers’ perspectives. From our survey, 33 respondents claim that they will add the selfdestruct function in their smart contracts. 18 out of 33 left their emails, and we sent emails to ask them whether they agreed to have a further interview about the gas refund feature. Finally, 8 respondents accepted the interview invitations, and all interviews were performed remotely via Skype or WeChat. We conducted semi-structured interviews followed by Zhou et al.’s method [224]. Specifically, we first introduced our work and asked the demographic questions shown in Section 6.3.2.2. The interviewees had an average experience of 3.9 years in smart contract devel-

opments with various roles, including 3 developers, 1 manager, 1 tester and 2 researchers. Then, we used some open questions listed in Table 5.7 to guide the discussion. The first question is used to confirm their qualification and all interviewees answered “Yes”. For the second question, 6 interviewees said they did not know it, and the interviews were finished. For the other 2 interviewees, both said they had a deep understanding of the gas refund feature and knew the *selfdestruct* function could refund gas. Thus, we continued to the fourth and fifth questions to collect their feedback. As our interviews were semi-structured, we also asked follow-up questions to dig deeper according to their answers. For example, one interviewee said they have no need to use *selfdestruct* to return gas back, and we then asked them why this was the case.

We found the following reasons why they will not include *selfdestruct* function just for refunding gas. First, the most important reason is the lack of knowledge about this feature. Although the interviewees had an average experience of 3.9 years in smart contract developments, only two of them know this feature. Second, one interviewee said they focus more on the functionalities and the security of smart contracts. They do not pay for the gas and thus they do not care how to save gas for the contracts. Besides, both two interviewees mentioned that adding *selfdestruct* function will increase development cost. However, only up to half of the gas used by a transaction that calls the *selfdestruct* function will be returned. They believe the loss outweighs the gain.

In terms of the GasToken, the two interviewees gave various views, but all the views were negative. The first interviewee mentioned that he did not believe GasToken can save gas or used it to make profits. Developers can only save a little gas fee when the gas price on free time is several times higher than mint time. If the gas price on free time is similar or lower than mint time, using GasTokens will even cost more, as the creation of the GasTokens will also cost gas. Besides, the miner can only receive half of the gas fee because of the GasToken, and they might refuse to process the transaction. Thus, the usage

of GasTokens seems narrow. Another interviewee said that GasToken is harmful for the Ethereum ecosystem, as it will create a large number of useless contracts on Ethereum. All of these useless contracts will be stored in nodes because of the distributed ledger nature of Ethereum, which wastes many storage and network resources.

Except from the views we collected from the interview, in the previous subsection (Section 5.6.4.2), we found that all the 2,080,319 self-destructed contracts only related to 4,404 dest accounts. This also shows that although there are a large number of GasTokens on Ethereum, they are only used by a small number of users. It is an interesting topic to conduct a comprehensive investigation of the GasTokens, e.g., their main usage scenarios, but they are out-of-scope to this paper. Thus, we will conduct the investigation in our future work.

5.6.5 Threats to Validity

Internal Validity. In RQ1, we sent our survey to 996 developers and received 88 responses. The response rate is 8.84%. We used the feedback of these 88 responses to summarize key reasons why developers include or exclude *selfdestruct* function in their contracts. Due to the limited number of feedback. There might however still be other reasons we did not cover in our survey. We collected all contributors emails from the top 100 most popular smart contract related projects. We also tried to make our survey as simple as possible and give 2 respondents \$50 Amazon gift card to increase the response rate. We finally obtained a 8.84% response rate, which is also acceptable [215].

In RQ2, we use SMARTEMBED to compute the similarity between smart contracts. If the similarity of two contracts larger than 0.6, we think they are a predecessor contract and its successor contract. The similarity threshold can influence the manual effort we need to pay. If the similarity is too large, we might miss some predecessor contracts and their successor contract. Otherwise, if the similarity is too small, we need to pay more effort

to distinguishing whether the two contracts are relevant or not. The similarity threshold used in the paper of SMARTMBED is 0.95, and it found few contracts are relevant if the similarity is lower than 0.7. To reduce the number of unidentified relevant contracts, we conservatively reduce the threshold to 0.6. We used *Open Card Sorting* to find 5 common self-destruct reasons. Due to the limitation of our understanding of the smart contracts, we might miss some self-destruct reasons. To reduce the threat of human factors, we followed the process of card sorting strictly, and the developers all have rich experience (>3 years) in smart contract related research. Researchers can also use the same method we proposed in RQ2 to find other self-destruct reasons in the future.

In RQ3, we use contracts whose number of transactions are larger than 500, and balance is larger than 1 Ether as the ground truth. We regard these contracts having a low probability of having permission problems. However, it is still possible we might find some functions in this group that have permission problems, but we believe the number of these functions is small.

We performed an interview to collect why developers do not include selfdestruct function for refunding gas. However, only 8 respondents accepted our interview, and 6 of them did not provide much information as they do not know the gas refund feature of Ethereum. Thus, it is likely that some points might be missing. Fortunately, we still obtain many reasonable feedbacks which can answer the inconsistency. We acknowledge that a comprehensive and convincing investigation about the GasTokens is an interesting topic, but it is not the core focus of this paper and needs further research efforts that we reserve for future work.

External Validity. The smart contracts used in this paper were up to Jan. 2019. *Solidity*, the most popular programming language for the smart contract, is fast-growing. From Jan. 2019 to the time of writing, there are 11 versions updated and released. Many new features have been removed and added in these versions. Ethereum also might be updated

in the future through a hard fork [109]. In this case, the self-destruct reasons might be changed because of a major update to Ethereum and Solidity. Addressing this threat needs more research effort, but the method we proposed to find the self-destruct reasons is still working.

In this paper, we first cluster contracts with their creator addresses. However, it is likely that some developers may use multiple addresses to deploy smart contracts. In this case, our method may fail in finding successor contracts of some self-destructed contracts, and our analysis may miss uncovering some reasons for the use of self-destruct. Because of the anonymity of Ethereum, it is difficult to cluster contracts with multiple creator addresses, even if they are owned by the same developer. Fortunately, from the collection of 756 self-destructed contracts that we analyzed, we find that for 436 of them we could find their successor contracts. In total we found 1513 *<predecessor contract, successor contract>* pairs. We use open card sorting to summarize the reasons why contracts self-destructed (details see Section 6.4.2.3). There are two iterations in the open card sorting. In the first iteration, we use 20% of the cards (one card is a *<predecessor contract, successor contract>* pairs) and get all the reasons. No new reason was found by using the remaining 80% of cards in the second iteration, which means 1513 pairs were enough to support the analysis work reported in this paper.

5.7 Related Work

SMARTEMBED [89, 90] is the first tool that uses a clone detection method to detect bugs in smart contracts. The tool contains a training phase and a prediction phase. In the training phase, their dataset contains two parts, i.e., source code database and bug database. The source code database consists of the source code of all the open source smart contracts in the Ethereum. The bug database records the bugs of each smart contract in a source code database. SMARTEMBED first converts each smart contract to an AST(Abstract syntax tree).

After normalizing the parameters and irrelevant information on the AST, SMARTEMBED transfers the tree structure to a sequence representation. Then, they use *Fasttext* [21] to transfer code to embedding matrices. Finally, they compute the similarity between the given smart contracts with contracts in their database to find the clone contracts and clone related bugs. Although the tool is aimed at finding bugs, their first step is computing the code similarity between the given smart contract and history contracts in their database. Therefore, we can modify their code to compute the similarity between two given smart contracts (used in RQ2).

Bartoletti et al. [10] found that the infamous Ponzi schemes migrated to the digital world. Many frauds use Ethereum to design Ponzi schemes contracts for earning money. They manually analyzed 1,382 verified smart contracts on Etherscan and find 137 of them are Ponzi scheme contracts. Then, they divided these Ponzi scheme contracts into four categories, i.e., array-based pyramid schemes, tree-based pyramid schemes, handover schemes, and waterfall schemes. Bartoletti et al. opened their dataset to the public but do not provide a tool to detect whether a contract is a Ponzi scheme contract. To address this limitation, Chen et al. [204] proposed a method that uses a machine learning algorithm (XGBoost [34]) to distinguish Ponzi scheme contracts. They use account features and code features to train the module. The account features are extracted from the transactions, e.g., the number of payment transactions, the balance in the contracts. The code features can be obtained from contract bytecode. They count the frequency of each opcode in the contract bytecode. Both account features and code features do not need the source code of contracts. Therefore, their method can predict arbitrary contracts on the Ethereum.

Oyente [131] is the first tool for security examination for smart contracts based on symbolic execution. Their work introduces four security issues on smart contracts, i.e., mishandled exception, transaction-ordering dependence, timestamps dependence, and reentrancy attack. To detect these security issues, Oyente first constructs a CFG (control flow

graph) based on symbolic execution. After that, they design different rules to detect these four security issues. Kalra et al. [113] proposed a tool named Zeus, which can detect seven kinds of security problems; four of them are the same with Oyente; the other three issues are *failed send*, *interger overflow/underflow* and *transaction state dependence*. Zeus can detect security issues at the source code level. They use LLVM bytecode to represent the Solidity source and detect related patterns through LLVM bytecode. ContractFuzzer [111] is the first fuzzer to detect seven security issues in smart contracts. Four security issues are the same as Oyente; the other three issues are *gasless send*, *dangerous delegatecall* and *freezing ether*. ContractFuzzer utilizes ABI (abstract binary interface) of smart contracts to generate fuzzing inputs and defines test oracles to detect security issues.

Chen et al. [32] define 20 smart contract defects on Ethereum. They first crawl 17,128 Stack Exchange posts and use key words to filter solidity related posts. After getting Solidity related posts, they use *Open Card Sorting* to find 20 contract defects and divide them into five categories, i.e., *security*, *availability*, *performance*, *maintainability*, and *reusability defects*. According to their paper, although previous works define several security defects, they did not consider the practitioners' perspective. Therefore, they design an online survey to collect feedback from developers. The feedback shows that all the defined contract defects are harmful to smart contracts. They assign five impact levels the defined 20 contract defects. Defects with impact level 1-3 can lead to unwanted behaviors of contract, e.g., crashing or a contract being attacked.

Li et al. [123] proposed a symbolic execution analysis tool named SOLAR to detect violations of two standards, i.e., ERC20 and ERC721. SOLAR is built on top of Manticore [139] which is a well-known symbolic execution framework for smart contracts, and uses boolector [145] as the SMT solver to check the symbolic constraints. Since Manticore does not fully support EVM instructions, SOLAR is extended to fully support EVM instructions. Their experimental results show that SOLAR is significantly more effective

than previous tools, e.g., Mythril [51], and can find more errors with fewer false positives.

Di Angelo and Salzer. [59] introduced the usage of smart contracts on Ethereum by analyzing about 20 million deployed smart contracts. They defined ten kinds of usages of smart contracts on Ethereum and found that most of the deployed smart contracts remain unused and tokens are the most popular applications of Ethereum. self-destructed contracts is one important research dimension in their work. They found 7.3M self-destructed contracts on their dataset. According to their analysis, 4.2 M of them were created and then self-destructed in the same transaction; 2.8 M were GasTokens, and 0.2 M were ENS (Ethereum Name Service) deeds. Eight thousand of the remaining nine thousand contracts were self-destructed for unknown reasons and 778 were wallets. Their other work [58] also investigated the usage of self-destructed contracts and found 48,506 contracts self-destructed multiple times (some of them up to 10 920 times). They called a contract a mayfly if the contract was created and self-destructed in the same transaction. They found 1,856,655 mayflies that were created by just 8,992 distinct addresses, and most of the mayflies appeared during the DDos period of 2016. All of their work and this work investigated the usage of `selfdestruct` function on Ethereum, but our work is more comprehensive and has a different focus. Specifically, we first conducted an online survey to collect developers' feedback about why they add or do not add `selfdestruct` function. Then, we proposed a method to find the reasons why smart contracts self-destructed. Finally, we propose a tool to detect two problems that might shorten the lifespan of smart contracts.

5.8 Conclusions and Future Work

In this paper, we conducted a comprehensive empirical study on the use of the `selfdestruct` function on Ethereum. To understand the smart contract developers' perspective, in RQ1, we designed an online survey to collect reasons from developers why they include and exclude the `selfdestruct` function in their contracts. We summarized 6 reasons for includ-

ing and 6 reasons for excluding *selfdestruct* function in their contracts, respectively. The feedback also shows that 22 / 33 respondents claim that they add *selfdestruct* function for security concerns or to upgrade contracts. These two motivations can lead to redeployment of smart contracts after developers destruct the contracts. According to this information, we propose an approach that can find the upgrade version of the self-destructed contracts in RQ2. After that, we used the *open card sorting* method and summarized 5 reasons why contracts might destruct. Two of them – *Unmatched ERC20 Token* and *Limits of Permission* – can affect the life span of smart contracts. To detect these problems, we developed a new tool named LIFESCOPE in RQ3, which reports 0 false positive / negative in detecting *Unmatched ERC20 Token*, and achieves an F-measure and AUC of 77.89% and 0.8673 for detecting the *Limits of Permission* issue. Finally, to help developers use the *selfdestruct* function better, we give 6 suggestions based on the feedback of our survey and our smart contract analysis.

Apart from the *selfdestruct* function, using *Delegatecall* or *Callcode* can also be used to design an upgradeable smart contract [143]. Specifically, we need a proxy contract and a logic contract to design an upgradeable contract. The proxy contract stores the storage variables and Ethers, and the logic contract contains the logical code. The proxy contract uses *Delegatecall* to call the code of the logic contract. If there are some security issues found on the logic contract, developers can deploy a new logic contract and discard the old logic contract. By comparing the difference between the old logic contract and new logic contract, we can find some reasons why contracts need upgrade. In Section 6.6.3, we only investigate the transaction of destructed unmatched ERC20 tokens. However, it is likely that the matched ERC20 tokens also do not transfer values back to the users. Also, the upgradeable contracts will also lock the tokens of users. In the future, we will conduct more comprehensive work to investigate the value lock on ERC20 tokens.

We found 2,789 (5.1%) smart contracts in total contain the *selfdestruct* function, while

only 199 (0.36%) contracts contain the Delegatecall or Callcode functions. The data size is one of the main reasons why we choose to investigate *selfdestruct* function first. In the future, we plan to conduct an empirical study to investigate the upgradeable smart contracts on Ethereum when there are more smart contracts that contain Delegatecall or Callcode function. Specifically, there are many new issues of designing an upgradeable smart contract. For example, although the old logic contract is discarded, the contract can still be called by other contracts, which might lead to new security issues; designing an upgradeable smart contract can also increase the development cost [33]. Thus, we first plan to investigate developers' motivation about why they design their contracts as upgradeable. Then, we plan to design a method to find the *<old logic contract, new logic contract>* pairs. Unlike *selfdestruct* function where we need to compare similarity to find the pair, for the upgradeable contract, we can check the transaction details of the proxy contract. From the transactions, we can find the discarded logic contract and new logic contract. After that, we can compare their differences to find the reasons why developers upgrade a smart contract. Finally, we will update LIFESCOPE to detect the additional problems.

We analyzed self-destructed contracts with source code and conducted a preliminary analysis for the self-destructed contracts on Ethereum blockchain. In the future, we will perform a comprehensive empirical study to analyze the self-destructed contract on Ethereum. Specifically, we first investigate how many GasTokens on Ethereum; how many users of the GasTokens, and the usage scenarios of the GasTokens. Besides, it is interesting to analyze remaining self-destructed contracts. For example, are there any other DDoS attacks happening; why millions of self-destructed contracts are only related to 4,404 dest accounts, and why some developers created a large number of self-destructed contracts.

We observed the difference between a PS pair to summarize the reasons why smart contracts were destructed. However, we only stand at a high level to present our observations instead of digging out a more detailed reason. Specifically, *Setting Changes* and *Function-*

ability Changes are two broad definitions. Developers might change the functionalities for several reasons, e.g., adding business requirements, fixing bugs, increasing readabilities. Besides, when multiple changes happen, we regard all of them as the reasons that contributed to the self-destruct. Actually, their importance might be different, and maybe only one of them was the real reason why smart contracts were destructed. In the future, we will conduct a more comprehensive empirical study to find more specific reasons that lead a contract self-destruct.

Chapter 6

Conclusion and Future Work

Due to the revolutionary features of smart contracts, e.g., immutability, smart contracts are much harder to be maintained compared to traditional programs. This thesis aims to help developers maintain their smart contracts by finding and detecting smart contract defects.

6.1 Key Findings and Contributions

To highlight the further studies on smart contract maintenance, we conducted an empirical study in Chapter 2 to investigate (1). what kinds of issues will developers encounter in smart contract maintenance? (2). what are the current maintenance methods used for smart contracts? We found that most developers choose to discard the old contract and redeploy a new contract when they need to patch or add new features on contracts. Before redeploying the new smart contract to Ethereum, an important step is checking its robustness and security. This finding highlights our further researches in finding and detecting smart contract defects.

In Chapter 3, we conducted an empirical study to find smart contract defects from online Q&A posts. Our approach contains four steps: First, we crawled all 17,128 posts from Ethereum StackExchange, the most popular question and answer (Q&A) sites for Ethereum smart contracts. Then, we used key words filtering to select Solidity defects-related posts. After that, we followed the card sorting approach to analyze and categorize

the filtered contract defects-related posts. We totally summarized 20 contract defects from five aspects: security, availability, performance, maintainability, and reusability. To validate the acceptance of our newly defined smart contract defects, we conducted an online survey and received 138 responses and 84 comments from developers in 32 countries. The feedback and comments show that developers believe removing the defined contract defects can improve the quality and robustness of smart contracts.

In Chapter 4, we proposed *DefectChecker*, a symbolic execution-based tool to detect contract defects defined in Chapter 3. *DefectChecker* can detect contract defects from smart contracts bytecode without the need for source code. During the symbolic execution, *DefectChecker* generates the control flow graph (CFG) of smart contracts, as well as the “stack event”, and identifies three features, i.e., “Money Call”, “Loop Block”, and “Payable Function”. By using the CFG, stack event, and the three features, *DefectChecker* uses different rules to detect contract defects and achieves very good results (88.8% of F-score in the whole dataset and only requires 0.15s on average to analyze one smart contract).

In Chapter 5, we presented an approach to find smart contract defects from historical self-destructed contracts. In this approach, we first collected all the verified (open-sourced) smart contracts from Etherscan, and 756 of them are self-destructed. Then, we propose a method to find the self-destructed contracts (also called predecessor contracts) and their updated version (successor contracts) by computing the code similarity. By analyzing the difference between the predecessor contracts and their successor contracts, we found five reasons that led to the death of the contracts; two of them (i.e., Unmatched ERC20 Token and Limits of Permission) are contract defects that might affect the life span of contracts.

We presented two methods to find defects from different perspectives in Chapter 3 and 5. Both of them have related advantages and limitations. Chapter 3 introduced a method to find defects from online posts, a breadth-first method and that us to cover a high range of

defects from a different aspect of smart contracts, e.g., security, performance. However, the limitation is that we cannot dig too much depth into a specific issue. The method proposed in Chapter 5 is totally different, which is a depth-first method that enables us to find many important issues of a specific feature, i.e., the Selfdestruct function. Specifically, a large number of features of the Selfdestruct function were introduced, which could highlight further research, and I will give the details in the next section.

6.2 Limitations and Future Works

6.2.1 Limitations and Future Works for Chapter 2

Limitation 1: Errors might be included on the Empirical Study

In this Chapter, 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 [172]. Thus, it is likely that some findings and results in the publications are out-of-date. Also, some non-peer-reviewed papers were included when we conducted the empirical studies, which might provided some incorrect answers.

Besides, it is possible that the respondents to our survey may provide some dishonest or unprofessional answers. One evidence is that not everybody answered all questions, which might show some respondents only have limited knowledge about the Ethereum maintenance.

Future works. In the future, we will update our findings if some non-peer-reviewed papers update their results. Also, a literature review for smart contract papers published after 2019 will be conducted to update our findings and help us figure out the changes of maintenance issues/methods in recent years.

6.2.2 Limitations and Future Works for Chapter 3

Limitation 1: Results may need update In Chapter 3, we conducted an empirical study, which defined 20 contract defects from online Q&A posts and used a survey to collect developers' perspectives. However, this is an early smart contract work conducted in 2018. However, Solidity is a fast-growing programming language. Some defects we found have been out-of-dated. Also, developers' perspectives might be changed when some new features of Ethereum are added, or some security attacks happen. For example, the survey in Chapter 3 was conducted at the end of 2018. At that time, most developers said adding self-destruct function could increase the security as they could destruct contracts and transfer all the balance when attacks happens. However, according to another survey we conducted in mid 2019, half of the respondents said adding a selfdestruct function could open an attack vector for attackers. Thus, the results we reported in Chapter 3, might be out-of-date today.

Future works. In the future, we will collect more Q&A posts to analyze whether there are new defects on the latest Solidity versions. Also, we will interview more developers to collect their perspective about the newly defined defects.

6.2.3 Limitations and Future Works for Chapter 4

Limitation 1: Results might be out-of-date. In Chapter 4, we developed a tool named *DefectChecker*, which was designed for Solidity v0.4.25 and this is the newest Solidity version at the time of developing *DefectChecker*. However, the newest Solidity version at the time of writing this thesis is v0.8.12, which has many new advanced features, and EVM also added some new instructions in recent years.

Besides, smart contracts were much simpler at the time that work was carried out. The contracts we analyzed were deployed before 2019.01. There were no concepts such as DeFi, NFT at that time. Besides, the smart contracts at that time had high similarities, and copy-paste errors were very popular, which also increased the accuracy. Specifically, most

of the vulnerable contracts have similar code. Thus, optimizing a few number of defect patterns could result to a high detect accuracy.

Also, the defects at that time were relatively simple. For example, most of the Reentrancy at that time is led by `.call.value()`, which is easy to detect. While today's Reentrancy attack is much difficult to analyze, some of them could even be used to launch Flash Loan attacks. It is high probability that *DefectChecker* will fail to detect this kind of Reentrancy.

Finally, the Ethereum ecosystem was immature at that time. Many defects, e.g., unchecking the return value of send, were very popular at that time. However, they were rare today. One reason is that with the development of the Ethereum ecosystem, many tools could warn of these simple defects, e.g., Remix.

Thus, *DefectChecker* as well as many smart contract analysis tools developed at that time, e.g., Oyente, Securify, may have many false positives/negatives today, and the reported results seems not so applicable today.

Future works. In the future, we will update *DefectChecker* to make it able to detect various Solidity versions and more complicated defect patterns. Specifically, we will first investigate the difference between the different Solidity Versions, e.g, which new features are added, any difference in the bytecode. Thus, we are able to analyze different versions of the contracts. Besides, patterns to detect defects need to be updated, and new technologies, e.g., data flow analyze, taint analyze, could be included to increase the accuracy.

Limitation 2: Errors may exist on the large scale dataset.

In Chapter 4, we conducted a large-scale evaluation based on bytecode crawled from Ethereum blockchain by 2019.01 to investigate whether the defects were prevalent on Ethereum. We randomly selected 500 smart contracts with source code to validate the accuracy of *DefectChecker*. These 500 contracts were also included in the large-scale dataset. Thus, we think it is a repeat work to double-check the analysis results on the large-scale evaluation again. However, *DefectChecker* has false positives/negatives in detecting de-

fектs. Thus, the analysis results of the large-scale dataset may still contain errors.

Future works. In the future, we could use other smart contract analysis tools to validate the correctness of the large-scale dataset. Specifically, a large amount of smart contract analysis tools have been presented in recent years. We could run some of them to compare the results. Also, manual work could be involved to increase the accuracy. By using this way, we might be able to obtain a more accurate dataset.

Limitation 3: Missing reporting the impact of contracts. We totally found 25,815 smart contracts have at least one defect during the large-scale evaluation in Chapter 4. However, we miss reporting the impact of the smart contracts, which might mislead the readers. For example, 3,892 contracts were detected containing Reentrancy vulnerabilities in their contracts. However, only 170 contracts whose balance is larger than 0.1 Ethers, and 212 contracts whose number of transactions is larger than 10, and only 7 contracts match these two requirements simultaneously. The results might show that most of the vulnerable contracts we detected might be toy contracts, and the security of Ethereum might not be such serious as we reported in Chapter 4.

Future works. In the future, it is worth investigating whether the vulnerable contracts detected by current smart contract analysis tools are really used or just toy contracts in the future. Specifically, we will collect all the smart contracts on Ethereum to the latest block, and use different smart contract analysis tools to detect vulnerabilities. By using different tools, we could compare the results to ensure correctness. Then, we will analyze the information of the detected vulnerable contracts, e.g., balance, number of transactions, to identify whether a contract is a toy contract. We plan to classify smart contracts into several groups based on their creation times. Thus, we can analyze the security of Ethereum on different time period.

6.2.4 Limitations and Future Works for Chapter 5

Limitation 1: Limited knowledge about survey respondents. In Chapter 5, survey played an important role in our findings. However, although all the developers we recruit claimed they have experience in contributing to open-sourced blockchain projects, many survey respondents have limited knowledge about Ethereum, and thus we might miss some important features. For example, almost all the 88 responses for the survey in Chapter 5 did not mention some advanced features of selfdestruct function. Specifically, no developers know selfdestruct function can be used to design the metamorphic contract (<https://github.com/0age/metamorphic>), which allows a contract to change its bytecode in the same address. Besides, only one developer knows selfdestruct could return gas and be used to design gastoken. The reason for this phenomenon was discussed in Section 5.6.4.3, as they lack knowledge of these advanced features.

Future works. It might be worth reperforming the survey and interviewing some professional developers/researchers. Based on the new results, we might obtain new results / feedbacks, which could lead to new findings. Also, we could present a work to investigate the evolution of the professional skills of Ethereum developers.

Limitation 2: Manual works to find the defects. In Chapter 5, we first used *Diffchecker* to find the difference between two contracts, then we manually analyze the difference to find the defects. As we admitted in the Threads to Validity section, the manual analysis might involve some errors and lead us to miss some cases.

Future works. In the future, we plan to design an automatic method to improve this process. For example, AST and CFG might able to be utilized to analyze the difference between two contracts. Then, we could design patterns to select some interested contracts. In this case, we can perform a more large-scale analysis which is not only limited to the destructed contracts.

Limitation 3: Some advanced Selfdestruct-Related Features need more investigation.

Considering the length and content consistency of the paper, some advanced Selfdestruct-related features were not discussed so much in Chapter 5. For example, designing upgradeable contracts by using *Delegatecall* or *Callcode* is also a good method to maintain smart contracts. According to our investigation in Chapter 5, we found 2,789 (5.1%) smart contracts in total contain the selfdestruct function, while only 199 (0.36%) contracts contain the *Delegatecall* or *Callcode* functions. The data size is one of the main reasons why we choose to investigate selfdestruct function first.

Besides, some Selfdestruct-based attacks/scams were found in Chapter 5. For example, in section 5.6.3, we introduced that most of the selfdestructed-ERC20 token contracts will not return tokens back, which leads to the financial loss of the investors. In section 5.6.4, we found that Selfdestruct could be used to design Gastoken, and it was being utilized to launch a DDoS attack and had many harmful impacts on Ethereum, e.g., waste space. Also, a new opcode named CREATE2 was introduced to the Ethereum virtual machine in February 2019, which allows a contract to change its bytecode in the same address.

Future works. In the future, we can find contract defects by investigating the upgradeable smart contracts on Ethereum when there are more smart contracts that contain *Delegatecall* or *Callcode* function. Besides, all of the above Selfdestructed-based attacks/scams were worth investigating to enhance the security of Ethereum.

References

- [1] Accounting. College cryptocurrency blockchain courses, Oct., 2018. URL: <https://www.accounting-degree.org/college-cryptocurrency-blockchain-courses/>.
- [2] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 66–77, 2018.
- [3] AST. Abstract syntax tree, Mar., 2020. URL: https://en.wikipedia.org/wiki/Abstract_syntax_tree.
- [4] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts (sok). In *International conference on principles of security and trust*, pages 164–186. Springer, 2017.
- [5] Gbadebo Ayoade, Erick Bauman, Latifur Khan, and Kevin Hamlen. Smart Contract Defense through Bytecode Rewriting. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 384–389. IEEE, 2019.
- [6] Shaun Azzopardi, Joshua Ellul, and Gordon J Pace. Monitoring smart contracts: Contractlarva and open challenges beyond. In *International Conference on Runtime Verification*, pages 113–137. Springer, 2018.

- [7] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.
- [8] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.
- [9] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. Dissecting Ponzi schemes on Ethereum: identification, analysis, and impact. *Future Generation Computer Systems*, 102:259–277, 2020.
- [10] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. Dissecting Ponzi schemes on Ethereum: identification, analysis, and impact. *Future Generation Computer Systems*, 102:259–277, 2020.
- [11] Massimo Bartoletti and Livio Pompianu. An empirical analysis of smart contracts: platforms, applications, and design patterns. In *International conference on financial cryptography and data security*, pages 494–509. Springer, 2017.
- [12] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. 2001.
- [13] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer, 2009.
- [14] Mirko Bez, Giacomo Fornari, and Tullio Vardanega. The scalability challenge of Ethereum: An initial quantitative analysis. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 167–176. IEEE, 2019.
- [15] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi,

Thomas Sibut-Pinote, Nikhil Swamy, et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 91–96, 2016.

- [16] Giancarlo Bigi, Andrea Bracciali, Giovanni Meacci, and Emilio Tuosto. Validation of decentralised smart contracts through game theory and formal methods. In *Programming Languages with Applications to Biology and Security*, pages 142–161. Springer, 2015.
- [17] Stefano Bistarelli, Gianmarco Mazzante, Matteo Micheletti, Leonardo Mostarda, and Francesco Tiezzi. Analysis of Ethereum smart contracts and opcodes. In *International Conference on Advanced Information Networking and Applications*, pages 546–558. Springer, 2019.
- [18] Blockchain. What is blockchain, Jan., 2019. URL: <https://en.wikipedia.org/wiki/Blockchain>.
- [19] Barry Boehm and Victor R Basili. Software defect reduction top 10 list. *Foundations of empirical software engineering: the legacy of Victor R. Basili*, 426(37):426–431, 2005.
- [20] Barry W Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.
- [21] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.
- [22] Amiangshu Bosu, Anindya Iqbal, Rifat Shahriyar, and Partha Chakraborty. Understanding the motivations, challenges and needs of Blockchain software developers: a survey. *Empirical Software Engineering*, 24(4):2636–2673, 2019.

- [23] Pierre Bourque, Richard E Fairley, et al. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014.
- [24] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981*, 2018.
- [25] JD Bruce. The mini-blockchain scheme. *White paper*, 2014.
- [26] CCN, Oct., 2021. URL: <https://www.ccn.com/marketcap/>.
- [27] Chainlink. Chainlink random number, Nov., 2021. URL: <https://docs.chain.link/docs/get-a-random-number/>.
- [28] Partha Chakraborty, Rifat Shahriyar, Anindya Iqbal, and Amiangshu Bosu. 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*, pages 1–10, 2018.
- [29] Jialiang Chang, Bo Gao, Hao Xiao, Jun Sun, Yan Cai, and Zijiang Yang. sCompile: Critical path identification and analysis for smart contracts. In *International Conference on Formal Engineering Methods*, pages 286–304. Springer, 2019.
- [30] Jiachi Chen, Xin Xia, Lo David, and Grundy John. Why Do Smart Contracts Self-Destruct? Investigating the Selfdestruct Function on Ethereum. *arXiv preprint arXiv:2005.07908*, 2020.
- [31] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. Domain specific code smells in smart contracts. *arXiv:1905.01467*, 2019.

- [32] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. Defining smart contract defects on ethereum. *IEEE Transactions on Software Engineering*, 2020.
- [33] Jiachi Chen, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. Maintenance-related concerns for post-deployed ethereum smart contract development: issues, techniques, and future challenges. *Empirical Software Engineering*, 26(6):1–44, 2021.
- [34] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.
- [35] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, et al. SODA: A Generic Online Detection Framework for Smart Contracts. In *Proceedings of the 27th Network and Distributed System Security Symposium*, 2020.
- [36] Ting Chen, Youzheng Feng, Zihao Li, Hao Zhou, Xiapu Luo, Xiaoqi Li, Xiuzhuo Xiao, Jiachi Chen, and Xiaosong Zhang. GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts. *IEEE Transactions on Emerging Topics in Computing*, 2020.
- [37] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. In *Proc. SANER*, pages 442–446. IEEE, 2017.
- [38] Ting Chen, Zihao Li, Yufei Zhang, Xiapu Luo, Ang Chen, Kun Yang, Bin Hu, Tong Zhu, Shifang Deng, Teng Hu, et al. Dataether: Data exploration framework for Ethereum. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1369–1380. IEEE, 2019.

- [39] Ting Chen, Zihao Li, Yufei Zhang, Xiapu Luo, Ting Wang, Teng Hu, Xiuzhuo Xiao, Dong Wang, Jin Huang, and Xiaosong Zhang. A large-scale empirical study on control flow identification of smart contracts. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE, 2019.
- [40] Ting Chen, Zihao Li, Hao Zhou, Jiachi Chen, Xiapu Luo, Xiaoqi Li, and Xiaosong Zhang. 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)*, pages 81–84. IEEE, 2018.
- [41] Ting Chen, Zihao Li, Yuxiao Zhu, Jiachi Chen, Xiapu Luo, John Chi-Shing Lui, Xiaodong Lin, and Xiaosong Zhang. Understanding Ethereum via Graph Analysis. *ACM Transactions on Internet Technology (TOIT)*, 20(2):1–32, 2020.
- [42] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. TokenScope: Automatically Detecting Inconsistent Behaviors of Cryptocurrency Tokens in Ethereum. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1503–1520, 2019.
- [43] Ting Chen, Yuxiao Zhu, Zihao Li, Jiachi Chen, Xiaoqi Li, Xiapu Luo, Xiaodong Lin, and Xiaosong Zhang. Understanding Ethereum via graph analysis. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 1484–1492. IEEE, 2018.
- [44] Weili Chen, Mingjie Ma, Yongjian Ye, Zibin Zheng, and Yuren Zhou. IoT service based on jointcloud blockchain: The case study of smart traveling. In *2018 IEEE*

Symposium on service-oriented system engineering (SOSE), pages 216–221. IEEE, 2018.

- [45] Weili Chen, Zibin Zheng, Edith C-H Ngai, Peilin Zheng, and Yuren Zhou. Exploiting blockchain data to detect smart Ponzi schemes on Ethereum. *IEEE Access*, 7:37575–37586, 2019.
- [46] Ram Chillarege et al. Orthogonal defect classification. *Handbook of Software Reliability Engineering*, pages 359–399, 1996.
- [47] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.
- [48] Christian Colombo, Joshua Ellul, and Gordon J Pace. Contracts over smart contracts: Recovering from violations dynamically. In *International Symposium on Leveraging Applications of Formal Methods*, pages 300–315. Springer, 2018.
- [49] Marco Conoscenti, Antonio Vetro, and Juan Carlos De Martin. Blockchain for the Internet of Things: A systematic literature review. In *2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA)*, pages 1–6. IEEE, 2016.
- [50] ConsenSys. Smart Contract Best Practices, Feb., 2020. URL: <https://github.com/ConsenSys/smart-contract-best-practices>.
- [51] Software ConsenSys. Mythril: Security analysis tool for evm bytecode., Aug., 2019. URL: <https://github.com/ConsenSys/mythril>.
- [52] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016.
- [53] Cryptokitties, Feb., 2019. URL: <https://www.cryptokitties.co/>.

- [54] Siegel David. Understanding The DAO Attack, Apr., 2018. URL: <https://www.coindesk.com/understanding-dao-hack-journalists/>.
- [55] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- [56] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. 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*, pages 79–94. Springer, 2016.
- [57] Premkumar Devanbu, Thomas Zimmermann, and Christian Bird. Belief & evidence in empirical software engineering. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 108–119. IEEE, 2016.
- [58] Monika Di Angelo and Gernot Salzer. Mayflies, breeders, and busy bees in Ethereum: smart contracts over time. In *Proceedings of the Third ACM Workshop on Blockchains, Cryptocurrencies and Contracts*, pages 1–10, 2019.
- [59] Monika Di Angelo and Gernot Salzer. Characterizing types of smart contracts in the ethereum landscape. In *International Conference on Financial Cryptography and Data Security*, pages 389–404. Springer, 2020.
- [60] Rolf Drechsler et al. *Advanced formal verification*, volume 122. Springer, 2004.
- [61] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 530–541, 2020.

- [62] Dmitry Efanov and Pavel Roschin. The all-pervasiveness of the blockchain technology. *Procedia Computer Science*, 123:116–121, 2018.
- [63] EIP. The Ethereum Improvement Proposal repository, Apr., 2020. URL: <https://github.com/Ethereum/EIPs>.
- [64] EIP150. EIP-150, May., 2020. URL: <https://blog.Ethereum.org/2016/10/13/announcement-imminent-hard-fork-eip150-gas-cost-changes/>.
- [65] EOS. Eos, Feb., 2019. URL: <https://eos.io/>.
- [66] EtherCamp. Ethercamp, Mar., 2018. URL: <https://live.ether.camp/>.
- [67] EtherChain, Mar., 2018. URL: <https://www.etherchain.org/contracts/>.
- [68] Ethereum. Ethereum.org, Jan., 2019. URL: <https://www.Ethereum.org/>.
- [69] Ethereum. Releases of solidity, Jan., 2019. URL: <https://github.com/Ethereum/solidity/releases>.
- [70] Ethereum. go-ethereum, Mar., 2018. URL: <https://github.com/Ethereum/go-Ethereum>.
- [71] Foundation Ethereum. Block validation algorithm., Apr., 2018. URL: [https://github.com/Ethereum/wiki/wiki/Block-Protocol-2.0#block-validation-algorithm/](https://github.com/Ethereum/wiki/wiki/Block-Protocol-2.0#block-validation-algorithm).
- [72] Etherscan. Ico ethereum, Apr., 2019. URL: <https://etherscan.io/directory/ICOs>.

- [73] Etherscan. Etherscan verified contract, Jan., 2020. URL: <https://etherscan.io/contractsVerified>.
- [74] Etherscan, Mar., 2018. URL: <https://etherscan.io/>.
- [75] Ethstates. Ethereum Network Status, Mar., 2020. URL: <https://ethstats.net/>.
- [76] Dobrik Eugene, Mello, and Wijaya Royyan. Gastoken.io, June., 2021. URL: <https://gastoken.io/>.
- [77] Expanse. Expanse, Feb., 2020. URL: <https://coinswitch.co/info/expanse/what-is-expanse>.
- [78] Vogelsteller Fabian and Buterin Vitalik. ERC20, Apr., 2018. URL: <https://github.com/Ethereum/EIPs/blob/master/EIPS/eip-20.md>.
- [79] Yuanrui Fan, Xin Xia, Daniel Alencar da Costa, David Lo, Ahmed E Hassan, and Shanping Li. The Impact of Changes Mislabeled by SZZ on Just-in-Time Defect Prediction. *IEEE Transactions on Software Engineering*, 2019.
- [80] Yuanrui Fan, Xin Xia, David Lo, and Ahmed E Hassan. Chaff from the wheat: Characterizing and determining valid bug reports. *IEEE transactions on software engineering*, 2018.
- [81] Christof Ferreira Torres, Mathis Baden, Robert Norvill, and Hugo Jonker. AEGIS: Smart Shielding of Smart Contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2589–2591, 2019.
- [82] Tomás Flouri, Emanuele Giaquinta, Kassian Kobert, and Esko Ukkonen. Longest common substrings with k mismatches. *Information Processing Letters*, 115(6-8):643–647, 2015.

- [83] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191, 2016.
- [84] Ethereum Foundation. Ethereum’s white paper. <https://github.com/Ethereum/wiki/wiki/White-Paper>, 2014.
- [85] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.
- [86] M Frowis and R Bohme. In code we trust? Measuring the control flow immutability of all smart contracts deployed on Ethereum. *LNCS*, 10436:357–372, 2017.
- [87] Michael Fröwis, Andreas Fuchs, and Rainer Böhme. Detecting token systems on Ethereum. In *International Conference on Financial Cryptography and Data Security*, pages 93–112. Springer, 2019.
- [88] Ying Fu, Meng Ren, Fuchen Ma, Yu Jiang, Heyuan Shi, and Jiaguang Sun. Evmfuzz: Differential fuzz testing of Ethereum virtual machine. *arXiv preprint arXiv:1903.08483*, 2019.
- [89] Zhipeng Gao, Vinoj Jayasundara, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. SmartEmbed: A Tool for Clone and Bug Detection in Smart Contracts through Structural Code Embedding. *35th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019.
- [90] Zhipeng Gao, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. Checking Smart Contracts with Structural Code Embedding. *IEEE Transactions on Software Engineering*, 2020.

- [91] GasStation. ETH Gas Station, Mar., 2020. URL: <https://ethgasstation.info/>.
- [92] Emanuel Giger, Marco D’Ambros, Martin Pinzger, and Harald C Gall. Method-level bug prediction. In *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 171–180. IEEE, 2012.
- [93] GLM. Generalized linear models, Oct., 2018. URL: <https://www.statmethods.net/advstats/glm.html>.
- [94] Giovanni Grano, Andrea Di Sorbo, Francesco Mercaldo, Corrado A Visaggio, Gerardo Canfora, and Sebastiano Panichella. 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*, pages 8–11, 2017.
- [95] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1176–1186. IEEE, 2019.
- [96] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in Ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, 2018.
- [97] Richard Greene and Michael N Johnstone. An investigation into a denial of service attack on an ethereum network. 2018.
- [98] Dominik Harz and William Knottenbelt. Towards safer smart contracts: A survey of languages and verification methods. *arXiv preprint arXiv:1809.0980*, 2018.

- [99] Ningyu He, Lei Wu, Haoyu Wang, Yao Guo, and Xuxian Jiang. Characterizing code clones in the Ethereum smart contract ecosystem. *arXiv preprint arXiv:1905.00272*, 2019.
- [100] Péter Hegedűs. Towards analyzing the complexity landscape of solidity based Ethereum smart contracts. *Technologies*, 7(1):6, 2019.
- [101] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. Kevm: A complete formal semantics of the Ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217. IEEE, 2018.
- [102] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering*, 23(1):418–451, 2018.
- [103] Rubing Huang, Weifeng Sun, Yinyin Xu, Haibo Chen, Dave Towey, and Xin Xia. A survey on adaptive random testing. *IEEE Transactions on Software Engineering*, 2019.
- [104] TonTon Hsien-De Huang. Hunting the Ethereum smart contract: Color-inspired inspection of potential attacks. *arXiv preprint arXiv:1807.01868*, 2018.
- [105] Hyperledger. Hyperledger, Feb., 2019. URL: <https://www.hyperledger.org/>.
- [106] Intel. Intel Corporation. Intelà Software Guard Extensions Evaluation SDK User’s Guide for Windows* OS., Feb., 2015. URL: <https://software.intel.com/sites/products/sgx-sdk-users-guide-windows>.

- [107] ISO. ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary. Technical report, ISO/IEC/IEEE 24765: 2017 (E), 2017.
- [108] ISO/IEC. 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*, pages 1–58, 2006.
- [109] Frankenfield Jake and Rasure Erika. Blockchain hard fork, Oct., 2019. URL: <https://www.investopedia.com/terms/h/hard-fork.asp>.
- [110] Bo Jiang, Ye Liu, and WK Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 259–269, 2018.
- [111] Bo Jiang, Ye Liu, and WK Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 259–269. ACM, 2018.
- [112] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105. IEEE, 2007.
- [113] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *25th Annual Network and Distributed System Security Symposium (NDSS'18)*, 2018.
- [114] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: Analyzing Safety of Smart Contracts. In *The Network and Distributed System Security Symposium (NDSS)*, pages 1–12, 2018.

- [115] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 75–84. IEEE, 2009.
- [116] Lucianna Kiffer, Dave Levin, and Alan Mislove. Analyzing Ethereum’s Contract Topology. In *Proceedings of the Internet Measurement Conference 2018*, pages 494–499, 2018.
- [117] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. The emerging role of data scientists on software development teams. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 96–107. IEEE, 2016.
- [118] Barbara Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33(2004):1–26, 2004.
- [119] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. *EBSE Technical Report*, 2007.
- [120] Barbara A Kitchenham and Shari L Pfleeger. Personal opinion surveys. In *Guide to advanced empirical software engineering*, pages 63–92. Springer, 2008.
- [121] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. Exploiting the laws of order in smart contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 363–373, 2019.
- [122] Johannes Krupp and Christian Rossow. Teether: Gnawing at Ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium*, pages 1317–1333, 2018.

- [123] Ao Li and Fan Long. Detecting Standard Violation Errors in Smart Contracts. *arXiv preprint arXiv:1812.07702*, 2018.
- [124] Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen. A survey on the security of blockchain systems. *Future Generation Computer Systems*, 2017.
- [125] Zixin Li, Haoran Wu, Jiehui Xu, Xingya Wang, Lingming Zhang, and Zhenyu Chen. MuSC: A Tool for mutation testing of Ethereum smart contract. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1198–1201. IEEE, 2019.
- [126] Lisk. Delegated proof of stake, Sept., 2019. URL: <https://lisk.io/academy/blockchain-basics/how-does-blockchain-work/delegated-proof-of-stake>.
- [127] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. Re-guard: finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68. IEEE, 2018.
- [128] Han Liu, Chao Liu, Wenqi Zhao, Yu Jiang, and Jiaguang Sun. S-gram: towards semantic-aware security auditing for Ethereum smart contracts. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 814–819, 2018.
- [129] LLVM. The llvm project, Jan., 2021. URL: <https://llvm.org/>.
- [130] Matthias Lohr and Sven Peldszus. Maintenance of Long-Living Smart Contracts. *CEUR Workshop Proceedings*, 2020.

- [131] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.
- [132] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. An analysis tool for smart contracts, Mar., 2018. URL: <https://github.com/melonproject/oyente>.
- [133] Walid Maalej and Hadeer Nabil. Bug report, feature request, or simply praise? on automatically classifying app reviews. In *2015 IEEE 23rd international requirements engineering conference (RE)*, pages 116–125. IEEE, 2015.
- [134] Daniel Macrinici, Cristian Cartofeanu, and Shang Gao. Smart contract applications within blockchain technology: A systematic mapping study. *Telematics and Informatics*, 35(8):2337–2354, 2018.
- [135] Bill Marino and Ari Juels. Setting standards for altering and undoing smart contracts. In *International Symposium on Rules and Rule Markup Languages for the Semantic Web*, pages 151–166. Springer, 2016.
- [136] Anastasia Mavridou and Aron Laszka. Designing secure Ethereum smart contracts: A finite state machine based approach. In *International Conference on Financial Cryptography and Data Security*, pages 523–540. Springer, 2018.
- [137] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.
- [138] Bhabendu Kumar Mohanta, Soumyashree S Panda, and Debasish Jena. An overview of smart contract and use cases in blockchain technology. In *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–4. IEEE, 2018.

- [139] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019.
- [140] Nikolai Mushegian. EIP-140, May., 2020. URL: <https://github.com/Ethereum/EIPs/issues/140>.
- [141] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.
- [142] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. *ICSE*, 2020.
- [143] Mudge Nick. Eip-2535: Diamond standard, Dec., 2020. URL: <https://eips.ethereum.org/EIPS/eip-2535>.
- [144] Mudge Nick. Eip2535: Diamond standard, Jan., 2021. URL: <https://eips.ethereum.org/EIPS/eip-2535>.
- [145] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation*, 9(1):53–58, 2014.
- [146] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 653–663. ACM, 2018.
- [147] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 653–663, 2018.

- [148] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.
- [149] Robert Norvill, Beltran Borja Fiz Pontiveros, Radu State, Irfan Awan, and Andrea Cullen. Automated labeling of unknown contracts in Ethereum. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–6. IEEE, 2017.
- [150] Anthony D Ong and David J Weiss. The impact of anonymity on responses to sensitive questions 1. *Journal of Applied Social Psychology*, 30(8):1691–1708, 2000.
- [151] Openzeppelin. Pausable documentation, June., 2021. URL: <https://docs.openzeppelin.com/contracts/2.x/api/lifecycle>.
- [152] Openzeppelin. Pausable documentation, June., 2021. URL: <https://github.com/OpenZeppelin/openzeppelin-contracts/blob/b0cf6fbb7a70f31527f36579ad644e1cf12fdf4e/contracts/security/Pausable.sol>.
- [153] OpenZeppelin. OpenZeppelin Upgradeable Smart Contract Document, Mar., 2020. URL: <https://docs.openzeppelin.com/learn/upgrading-smart-contracts>.
- [154] Openzeppelin. Openzeppelin Contracts, Feb., 2020. URL: <https://github.com/OpenZeppelin/openzeppelin-contracts>.
- [155] Oracle. The Java Virtual Machine Specification, Aug., 2020. URL: <https://docs.oracle.com/javase/specs/jvms/se8/html/index.html>.

- [156] Reza M Parizi, Ali Dehghantanha, Kim-Kwang Raymond Choo, and Amritraj Singh. 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*, pages 103–113. IBM Corp., 2018.
- [157] Reza M Parizi, Ali Dehghantanha, et al. Smart contract programming languages on blockchains: An empirical evaluation of usability and security. In *International Conference on Blockchain*, pages 75–91. Springer, 2018.
- [158] Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roșu. 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*, pages 912–915, 2018.
- [159] Daniel Perez and Benjamin Livshits. Smart contract vulnerabilities: Does anyone care? *arXiv preprint arXiv:1902.06710*, 2019.
- [160] Thomas M Pigoski. *Practical software maintenance: best practices for managing your software investment*. Wiley Publishing, 1996.
- [161] Martin Porter. An algorithm for suffix stripping. *Program*, 3 (14): 130–137, 1980.
- [162] Remix. Remix, Jan., 2020. URL: <http://remix.ethereum.org/>.
- [163] Ana Reyna, Cristian Martín, Jaime Chen, Enrique Soler, and Manuel Díaz. On blockchain and its integration with IoT. Challenges and opportunities. *Future generation computer systems*, 88:173–190, 2018.
- [164] Ripple. Ripple Coin, Aug., 2019. URL: <https://ripple.com/>.

- [165] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv preprint arXiv:1812.05934*, 2018.
- [166] Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. Writing safe smart contracts in Flint. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, pages 218–219, 2018.
- [167] SDHardFork. Spurious Dragon Hard Fork, May., 2020. URL: <https://blog.Ethereum.org/2016/11/18/hard-fork-no-4-spurious-dragon/>.
- [168] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.
- [169] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on software engineering*, 42(9):805–824, 2016.
- [170] sklearn. sklearn, June., 2021. URL: <https://scikit-learn.org/stable/>.
- [171] SmartContractSecurity. Swc registry: Smart contract weakness classification and test cases, July., 2019. URL: <https://smartcontractsecurity.github.io/SWC-registry/>.
- [172] Solidity. Releases of Solidity, Jun., 2020. URL: <https://github.com/Ethereum/solidity/releases>.
- [173] Solidity. Solidity Document, Mar., 2020. URL: <http://solidity.readthedocs.io>.
- [174] Donna Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

- [175] StackExchange. Stackexchange, Jan., 2018. URL: <https://Ethereum.stackexchange.com/>.
- [176] Nikhil Swamy, Cătălin Hrițcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. Dependent types and multi-monadic effects in F. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–270, 2016.
- [177] Ann T Tai and Leon Alkalai. On-board maintenance for long-life systems. In *Proceedings. 1998 IEEE Workshop on Application-Specific Software Engineering and Technology. ASSET-98 (Cat. No. 98EX183)*, pages 69–74. IEEE, 1998.
- [178] A Tann, Xing Jie Han, Sourav Sen Gupta, and Yew-Soon Ong. Towards safer smart contracts: A sequence learning approach to detecting vulnerabilities. *arXiv preprint arXiv:1811.06632*, pages 1371–1385, 2018.
- [179] Don Tapscott and Alex Tapscott. *Blockchain revolution: how the technology behind Bitcoin is changing money, business, and the world*. Penguin, 2016.
- [180] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.
- [181] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of Ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 9–16, 2018.
- [182] Rembacz Tomasz. Diffchecker, Aug., 2020. URL: <https://github.com/trembacz/diff-checker>.

- [183] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in Ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 664–676, 2018.
- [184] Christof Ferreira Torres, Mathis Steichen, et al. The art of the scam: Demystifying honeypots in Ethereum smart contracts. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1591–1607, 2019.
- [185] Truffle. Truffle, Jan., 2020. URL: <https://www.trufflesuite.com/>.
- [186] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82. ACM, 2018.
- [187] Pradeep K Tyagi. The effects of appeals, anonymity, and feedback on mail survey response patterns from salespeople. *Journal of the Academy of Marketing Science*, 17(3):235–241, 1989.
- [188] Chibuzor Udokwu, Aleksandr Kormiltsyn, Kondwani Thangalimodzi, and Alex Norta. The state of the art for blockchain-enabled smart-contract applications in the organization. In *2018 Ivannikov Ispras Open Conference (ISPRAS)*, pages 137–144. IEEE, 2018.
- [189] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.
- [190] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 97–106. IEEE, 2002.

- [191] Yaron Velner, Jason Teutsch, and Loi Luu. Smart contracts make Bitcoin mining pools vulnerable. In *International Conference on Financial Cryptography and Data Security*, pages 298–316. Springer, 2017.
- [192] Buterin Vitalik. Eip-55, Jan., 2016. URL: <https://github.com/Ethereum/EIPs/blob/master/EIPS/eip-55.md>.
- [193] Buterin Vitalik and Swende Martin. Eip-3298: Removal of refunds, June., 2021. URL: <https://eips.ethereum.org/EIPS/eip-3298>.
- [194] Buterin Vitalik and Swende Martin. Eip-3529: Reduction in refunds, June., 2021. URL: <https://eips.ethereum.org/EIPS/eip-3529>.
- [195] Marko Vukolić. Rethinking permissioned blockchains. In *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, pages 3–7, 2017.
- [196] Vyper. Vyper document, Mar., 2020. URL: <https://vyper.readthedocs.io>.
- [197] Zhiyuan Wan, Xin Xia, Ahmed E Hassan, David Lo, Jianwei Yin, and Xiaohu Yang. Perceptions, expectations, and challenges in defect prediction. *IEEE Transactions on Software Engineering*, 2018.
- [198] Zhiyuan Wan, Xin Xia, David Lo, and Gail C Murphy. How does Machine Learning Change Software Development Practices? *IEEE Transactions on Software Engineering*, 2019.
- [199] Haijun Wang, Yi Li, Shang-Wei Lin, Lei Ma, and Yang Liu. 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)*, pages 1–4. IEEE, 2019.

- [200] Xinming Wang, Jiahao He, Zhijian Xie, Gansen Zhao, and Shing-Chi Cheung. ContractGuard: Defend Ethereum Smart Contracts with Embedded Intrusion Detection. *IEEE Transactions on Services Computing*, 2019.
- [201] Zeli Wang, Weiqi Dai, Kim-Kwang Raymond Choo, Hai Jin, and Deqing Zou. FSFC: An input filter-based secure framework for smart contract. *Journal of Network and Computer Applications*, page 102530, 2020.
- [202] Web3. Web3.py, April., 2019. URL: <https://web3py.readthedocs.io/en/stable/>.
- [203] Web3. web3, Jan., 2020. URL: <https://web3js.readthedocs.io/en/v1.2.4/>.
- [204] Chen Weili, Zheng Zibin, Cui Jiahui, Ngai Edith, Zheng Peilin, and Zhou Yuren. Detecting Ponzi Schemes on Ethereum: Towards Healthier Blockchain Technology. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 1409–1418. International World Wide Web Conferences Steering Committee, 2018.
- [205] Wikipedia. Decentralized application, Apr., 2019. URL: https://en.wikipedia.org/wiki/Decentralized_application.
- [206] Wikipedia. Ethereum classic, Feb., 2019. URL: https://en.wikipedia.org/wiki/Ethereum_Classic.
- [207] Wikipedia. Cryptocurrency, Jan., 2019. URL: <https://en.wikipedia.org/wiki/Cryptocurrency>.
- [208] Wikipedia. Ethereum introduction, Jan., 2019. URL: <https://en.wikipedia.org/wiki/Ethereum/>.

- [209] Wikipedia. Software defects, Jan., 2020. URL: https://en.wikipedia.org/wiki/Software_defect/.
- [210] Wikipedia. Camel case, Sept., 2019. URL: https://en.wikipedia.org/wiki/Camel_case/.
- [211] Wikipedia. Proof of stake, Sept., 2019. URL: https://en.wikipedia.org/wiki/Proof_of_stake.
- [212] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Project Yellow Paper*, 2014.
- [213] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.
- [214] Gavin Wood, C Reitwiessner, A Beregszaszi, and Y Hirai. The solidity contract-oriented programming language, Mar., 2018. URL: <https://github.com/Ethereum/solidity>.
- [215] Xin Xia, Zhiyuan Wan, Pavneet Singh Kochhar, and David Lo. How practitioners perceive coding proficiency. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 924–935. IEEE, 2019.
- [216] Wu Xiong, Qingyun Zhao, Jun Zhao, Weibing Xun, Rong Li, Ruifu Zhang, Huasong Wu, and Qirong Shen. Different continuous cropping spans significantly affect microbial community membership and structure in a vanilla-grown soil as revealed by deep pyrosequencing. *Microbial ecology*, 70(1):209–218, 2015.
- [217] Meng Yan, Xin Xia, Emad Shihab, David Lo, Jianwei Yin, and Xiaohu Yang. Automating change-level self-admitted technical debt determination. *IEEE Transactions on Software Engineering*, 45(12):1211–1229, 2018.

- [218] Yiming Yang and Jan O Pedersen. A comparative study on feature selection in text categorization. In *Icml*, volume 97, page 35. Nashville, TN, USA, 1997.
- [219] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 270–282, 2016.
- [220] Tao Zhang, Jiachi Chen, Xian Zhan, Xiapu Luo, David Lo, and He Jiang. Where2Change: Change Request Localization for App Reviews. *IEEE Transactions on Software Engineering*, 2019.
- [221] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems*, 105:475–491, 2020.
- [222] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. Blockchain challenges and opportunities: A survey. *International Journal of Web and Grid Services*, 14(4):352–375, 2018.
- [223] Ence Zhou, Song Hua, Bingfeng Pi, Jun Sun, Yashihide Nomura, Kazuhiro Yamashita, and Hidetoshi Kurihara. Security assurance for smart contract. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5. IEEE, 2018.
- [224] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach D Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering*, 2019.