The Journal of Systems & Software 204 (2023) 111787

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

An empirical study on real bug fixes from solidity smart contract projects[☆]

Yilin Wang[a], Xiangping Chen[b], Yuan Huang[c,*], Hao-Nan Zhu[d], Jing Bian[a], Zibin Zheng[c]

[a] School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China
[b] School of Journalism and Communication, Sun Yat-sen University, Guangzhou, China
[c] School of Software Engineering, Sun Yat-sen University, Zhuhai, China
[d] Department of Computer Science, University of California, Davis, United States of America

A R T I C L E I N F O

Article history:
Received 2 October 2022
Received in revised form 15 May 2023
Accepted 11 June 2023
Available online 15 June 2023

Keywords:
Bug fix
Empirical study
Smart contract
Solidity

A B S T R A C T

Smart contracts are pieces of code that reside inside the blockchains and can be triggered to execute any transaction when specifically predefined conditions are satisfied. Being commonly used for commercial transactions in blockchain makes the security of smart contracts particularly important. Over the last few years, we have seen a great deal of academic and practical interest in detecting and fixing the bugs in smart contracts written by Solidity. But little is known about the real bug fixes in Solidity smart contract projects. To understand the bug fixes and enrich the knowledge of bug fixes in real-world projects, we conduct an empirical study on historical bug fixes from 46 real-world Solidity smart contract projects in this paper. We provide a multi-faceted discussion and mainly explore the following four questions: File Type and Amount, Fix Complexity, Bug distribution, and Fix Patches. We distill four findings during the process to explore these four questions. Finally, based on these findings, we provide actionable implications to improve the current approaches to fixing bugs in Solidity smart contracts from three aspects: Automatic repair techniques, Analysis tools, and Solidity developers.

1. Introduction

Blockchain uses cryptographic proof to replace trusted third parties to ensure the correctness of the information, allowing any two willing parties to transact directly with each other in an open environment. Ethereum (Wood et al., 2014; Ethereum, 2014) is the most popular blockchain platform, which not only allows transactions with tokens but also offers storage and execution of the code, known as smart contracts. Smart contracts are at the core of Ethereum (Durieux et al., 2020), which are pieces of code that reside inside the decentralized blockchains in essence, and can be triggered to execute any task when specifically predefined conditions are satisfied (Gao, 2020). Smart contracts in Ethereum are usually written using a statically-typed high-level programming language named Solidity (Ethereum, 2021). As a young language born in 2015, Solidity is not safe enough and is exposed to severe bugs (Lutellier et al., 2020), which allow attackers to steal money or cause other damages while exploiting them. For example, in April 2018, the attackers exploited the Integer Overflow bug of the USChain BEC contract to copy tokens infinitely, causing the value of BEC tokens which is worth 900 million dollars to zero (Ethereum, 2018). The attackers exploited the Reentrancy bug of SIREN AMM pools to steal nearly 3.5 million dollars worth of assets on 3 September 2021 (Dalakos, 2021). Therefore, it is necessary to study the bugs in smart contracts in order to prevent such attacks and financial losses.

Over the last few years, we have seen a great deal of both academic and practical interest in the topic of detecting bugs in smart contracts written by Solidity developed for the Ethereum blockchain (Wang et al., 2013; Luu et al., 2016; Zou et al., 2019; Perez and Livshits, 2021; Hwang and Ryu, 2020; Nguyen et al., 2021, 2013; ConsenSys, 2019). Meanwhile, Nguyen et al. (2021), Chen et al. (2020) reduce the effort to repair the bugs in smart contracts by proposing various automatic repair techniques. Although quite a few works have focused on the bugs in Solidity smart contracts, the research community has limited knowledge of the naturalness of bug fixes in Solidity smart contract projects. For example, how many Solidity files are modified when fixing bugs? What is the most common fixed element in Solidity files during real bug fixes? It is of great importance to design an empirical study to answer these questions and to enrich the knowledge of real bug fixes in Solidity smart contract projects. The results may provide future insights for us to improve existing technologies.

[☆] Editor: Laurence Duchien.
[*] Corresponding author.
E-mail addresses: wangylin28@mail2.sysu.edu.cn (Y. Wang), chenxp8@mail.sysu.edu.cn (X. Chen), huangyuan5@mail.sysu.edu.cn (Y. Huang), hnzhu@ucdavis.edu (H.-N. Zhu), zhzibin@mail.sysu.edu.cn (Z. Zheng).

https://doi.org/10.1016/j.jss.2023.111787
0164-1212/© 2023 Elsevier Inc. All rights reserved.

In this paper, we focus on understanding bug fixes from real-world Solidity smart contract projects. We extract the bug fixes from the history of 46 Solidity smart contract projects and then conduct a multi-faceted empirical study to analyze them. In general, our research questions and their findings are as follows.

(1) **File Type and Amount.** We explore the types and the number of files involved during bug fixes to understand the distribution and complexity of bugs at the file level. Our results show that the distribution of bugs at the file level in Solidity smart contract projects is very complicated. Nearly 40% of the bug fixes in Solidity files modify two or more Solidity source files. What is more, not only Solidity files but also the other source code files have quite a few bugs. About 92% of the 6146 bug fixes involve at least one the other source code file.

(2) **Fix Complexity.** We explore the modifications of the Solidity code during bug fixes to understand the fix complexity in Solidity files. There are 19 element kinds that may be modified in Solidity files during bug fixes. Additionally, the fix actions taken to these elements contain *addition*, *deletion*, and *change*. Specifically, we explore the modified elements, the element kinds, and the fix actions taken to Solidity files during bug fixes. Our results show that *Comment* is the most common fixed element during bug fixes in Solidity files. What is more, the majority (59%) of the Solidity files are multi-element modifications during bug fixes. *EventDefinition* and *EmitStatement* are the most relevant elements in the multi-element modifications with a correlation coefficient of 0.55. In general, a Solidity file involves an average of 2.5 code element kinds modification and 7.4 fix actions to code elements during bug fixes.

(3) **Bug Distribution.** We try to analyze the distribution of the bugs that can be detected by the current analysis tools Mythril and Slither in the historical bug-fixed versions of the Solidity files. Our results show that nearly 20.5% of the Solidity files in our dataset have been exposed to bugs. They involve 14 categories of bugs. *Arithmetic* and *Unused-return* are the two most common bugs.

(4) **Fix Patches.** Based on the Bug Distribution, we want to find out how many bugs have been fixed, how many bugs have been newly introduced, and how developers fix them during the real bug fixes. The results show that the developers may not put much attention to fixing the bugs reported by the tools completely or avoid introducing them again. Meanwhile, Mythril and Slither perform poorly in detecting *Reentrancy* bugs in practice.

Contributions. Our contributions in this paper can be summarized as follows.

(1) We provide a bug-fix dataset (commit-level) of 46 real-world Solidity smart contract projects.

(2) We provide a multi-faceted discussion of bug fixes in real-world Solidity smart contract projects.

(3) We analyze the bugs and their fixes in Solidity files from the history of 46 Solidity smart contract projects on GitHub.

(4) We provide actionable implications based on our results and findings for researchers to improve the automatic repair techniques and analysis tools.

The rest of this paper is structured as follows: Section 2 introduces the data extraction process and research questions we use to conduct the study. Section 3 presents the results of our empirical study and the actionable implications based on our findings. After that, Section 4 discusses the related work. Section 5 introduces the threats to validity. Finally, Section 6 gives the conclusions of this work. To facilitate research and application, our replication package and the dataset are available at https://github.com/echowyl8/Real-Bug-Fixes-in-Smart-Contracts-Projects.

Table 1
Dataset statistics.

Name	Amount
The number of commits	43,354
The number of bug-fix commits	6,146
The number of all the files	7,306
The number of the Solidity files	3,545
The lines of all the files	2,686,281
The lines of the Solidity files	241,445

2. Methodology

2.1. Dataset

To analyze the bug fixes from real-world Solidity smart contract projects, we first collect 50 projects from GitHub (GitHub, 2021) in September 2021. In addition to downloading these 50 projects, we also download their commit metadata available. Then, we identify the bug fixes from these commit metadata. The specific data collection and screening are as follows.

Search Key. We collect the projects with the applied search key "contract" and "solidity" or "javascript". In this paper, we only explore the smart contract on Ethereum written by solidity. Because most of the smart contract platforms are JavaScript frameworks, we also consider the projects related to the keywords "contract" and "javascript". We select 25 projects in each of the two search results. All the 50 projects have been manually checked to assure they are Solidity smart contract projects. We invite three volunteer graduate students to conduct the validation process. They have an average of 1.5 years of development experience in Solidity smart contracts. Specifically, two participants independently determine whether the projects are related to Solidity smart contracts or not by reading the readme files of these projects and checking the Solidity files under their directories. If both of them agree that the project is related to Solidity smart contract projects in Ethereum, then the project is chosen. If there is a discrepancy between them, the rest participant determines the final result.

Popularity. The number of stars of a project is a proxy for its popularity on GitHub. Starring a project shows appreciation from the users (Wen et al., 2022). We select the projects from the search results based on the number of stars.

Bug Fix. We follow previous work (Lutellier et al., 2020) to extract bug fixes from the commit history of these 50 projects. We determine the commits as bug fixes if there are keywords "fix", "bug", and "patch" in their commit messages and filter commits using anti-patterns "rename", "clean up", "refactor", "merge", "misspelling", "compiler warning". According to the investigation of this approach in Lutellier et al. (2020), it has a 93% accuracy to extract bug fixes.

After filtering out projects with zero bug fix commits, we finally get 6146 bug fixes from 46 projects for further study. These commits are created from February 2016 to September 2021. Fig. 1 shows the number of non-bug-fix commits and bug-fix commits in the 50 projects. *contracts-solidity* is the project with the most commits and bug-fix commits, which has 4836 commits including 710 bug-fix commits. As shown in Table 1, there are 6146 bug-fix commits in our dataset from these 46 projects. These projects involve 7306 files with the total lines adding up to 2,686,281. Among them, there are 3545 Solidity files with the total lines adding up to 241,445.

2.2. Research questions

In order to understand the distribution of bugs and bug fixes in real-world Solidity smart contract projects. We consider the following four research questions. Their results can help developers

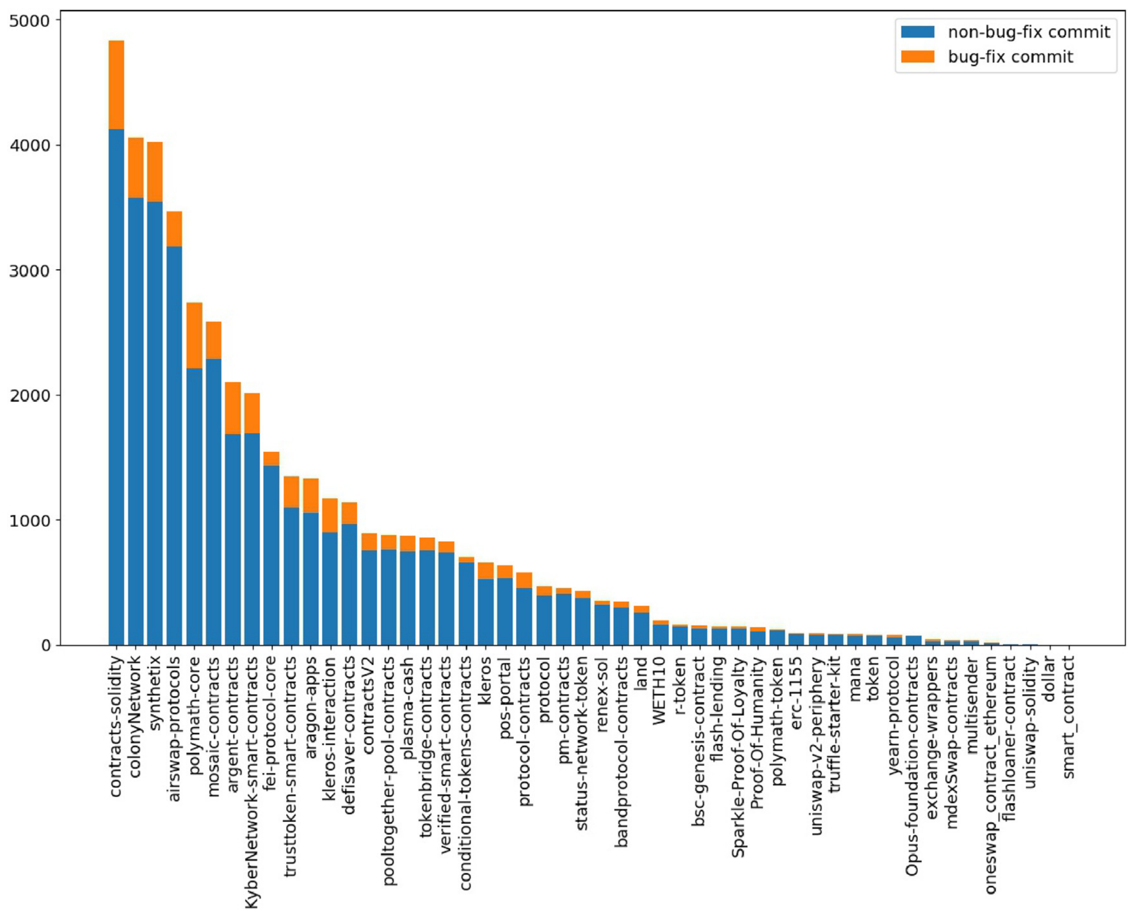


Fig. 1. The number of non-bug-fix commits and bug-fix commits of each project.

understand the real bug fixes during the maintenance process of Solidity smart contract projects better.

File Type and Amount. Recently, several automated smart contract repair approaches have been proposed (Yu et al., 2020; Nguyen et al., 2021; Chen et al., 2020). But they only modify one Solidity file so that the bugs which are not related to the Solidity files or involve more than one file could not be fixed by these automatic repair techniques. It is still unknown how many bugs in projects cannot be fixed by the limitations of these automatic repair techniques. So, in this part, we explore the types and the number of files involved during bug fixes to understand the distribution and complexity of bugs at the file level and to enrich the knowledge of bug fixes in Solidity smart contract projects. What is more, knowing the types and the number of files modified during bug fixes from real-world Solidity smart contract projects may help us to inspire novel approaches for finding, locating, and repairing bugs.

The problems and goals in this part we study are as follows.

RQ1. What types of files are involved when fixing bugs? We try to find out the file types that are modified during bug fixes in real-world Solidity smart contract projects. It can enrich our knowledge of bug distribution at the file level in Solidity smart contract projects.

RQ2. How many Solidity files are modified during a fix? We try to find out the number of Solidity files modified during a bug fix. It can help us understand the impact of the bugs and the dependency among the Solidity files.

RQ3. How many Solidity files are necessary to be added or deleted to fix bugs? We try to find out the number of Solidity files added and deleted during a bug fix. It can help us to understand the need for adding and deleting Solidity files during a bug fix.

Fix Complexity. All the code elements such as *PragmaDirective*, *FunctionDefinition*, *RequireStatement* and *VariableDeclaration* in Solidity files can be modified during bug fixes. The non-code element *Comment* also can be the modified object. Three types of fix actions can be taken to these elements during the fix: *addition*, *deletion*, and *change*. For example, if the developers add a *RequireStatement* during a bug fix. We consider that they take a fix action of the *RequireStatement* element. We want to explore the most common fixed element and the most common fix action taken to it. An Analysis of it can help us understand the bug fixes from real-world projects and find out the most susceptible element to bugs in practice. Further, we try to find out the element kinds and the number of fix actions that are involved during bug fixes to understand the complexity of fixes in Solidity files. The problems and goals in this part we study are as follows.

RQ4. What is the most common fix operation during bug fixes? In this question, we try to find out the most common fixed element and the most common fix action taken to it during the bug fixes. Such analysis can help us enrich the knowledge of the fixed objects and provide suggestions for future research of fixes.

RQ5. How many element kinds are modified in a Solidity file during bug fixes? We try to find out the number of element kinds that are involved in a Solidity file during bug fixes. It can help us understand the complexity of fixed objects during bug fixes at the code level in Solidity files.

RQ6. How many fix actions are taken to a Solidity file during bug fixes? We try to find out the number of fix actions taken to a Solidity file during bug fixes. It can directly show us the complexity of the fix in a Solidity file.

Bug Distribution. Quite a few automated analysis tools have been proposed to detect the bugs in Solidity smart contracts. In

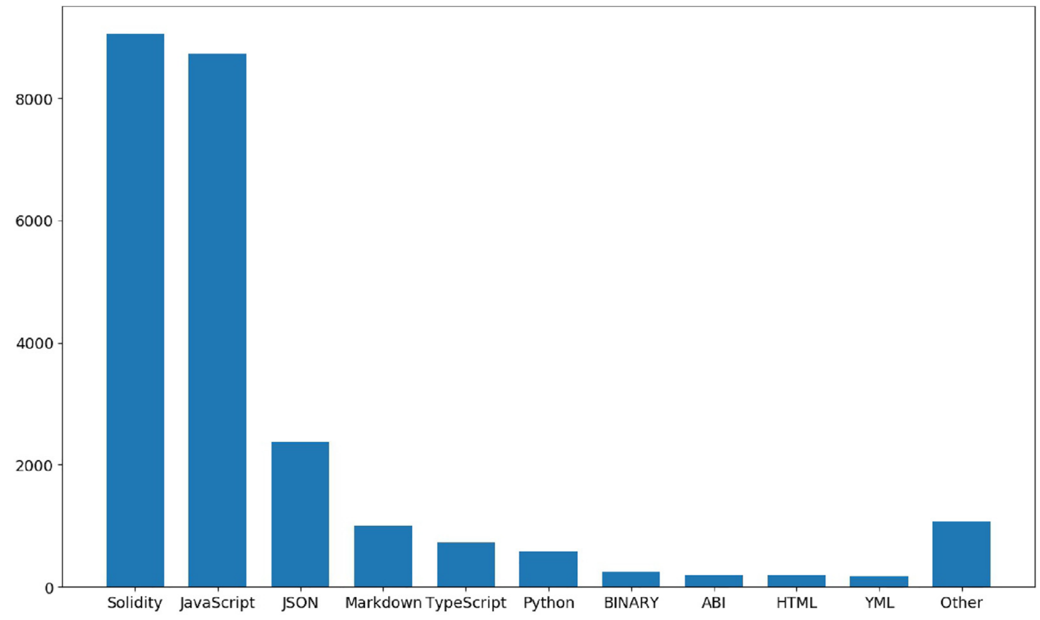


Fig. 2. The types of modified files during bug fixes.

this part, we try to use automated analysis tools to analyze the distribution of the bugs in the bug-fix versions of the smart contract projects. Such analysis can help us enrich the knowledge of bugs in real-world Solidity smart contract projects. The problems and goals in this part we study are as follows.

RQ7. What are the specific bugs reported by Mythril and Slither? We try to analyze the specific bugs that are reported by Mythril and Slither. It can help us understand the distribution of the specific categories of bugs in real-world Solidity smart contract projects.

RQ8. How many bugs are reported by Mythril and Slither and how many Solidity files do they exist in? We try to find out the number of bugs and the number of Solidity files that have been exposed to bugs from the historical bug-fix versions of the projects.

Fix patches. Real bugs and their patches collected from real-world projects are critical for research in the repair of smart contracts. In this part, we want to find out the fixes of these bugs reported by Mythril and Slither, including the number of fixed bugs, non-fixed bugs, newly introduced bugs, and how they are fixed. Such analysis can help us understand the detailed fixed information of these bugs in real-world Solidity smart contract projects. The problems and goals in this part we study are as follows.

RQ9. How many bugs have been fixed and how many bugs have been newly introduced? We try to find out the number of the bugs that have been fixed and newly introduced during the bug fixes. It can help us to understand the detailed fixed information of these bugs in the maintenance of real-world projects.

RQ10. How do developers fix these bugs? We try to manually explore the human-written patches for these bugs.

3. Empirical results

In this section, we present the results of the empirical study and provide actionable implications based on our findings for researchers.

3.1. File type and amount

RQ1. What types of files are involved when fixing bugs?

We calculate the types of files that are involved during a bug fix. Fig. 2 shows the top 10 types of modified files in the 6146 bug fixes. As we can see from the results, except for Solidity files, JavaScript is the most common modified source code file type during bug fixes. TypeScript and Python are also among the common fixed code file types. What is more, there are almost as many bugs in JavaScript files as there are in Solidity files. The result is quite different from that in traditional projects like Java projects. Zhong and Su (2015) conduct an empirical study on bug fixes from five popular Java projects. They find the most common modified files are Java files, and the other source code files are much fewer. It indicates that the bug fixes in smart contract projects are more complex than that in traditional projects. Because not only Solidity files but also the other source code files like JavaScript have quite a few bugs.

There are two general purposes for developers to use the other source code files in an Ethereum smart contract project. One is that the developers use programs written in various high-level languages such as JavaScript, TypeScript, and Python to develop clients to interact with Ethereum. In this way, to implement deploying and working with smart contracts, and to integrate with client nodes on the Ethereum network. Another is that the developers use these files for automated testing to validate the functionality of smart contracts from the perspective outside the blockchain. Both of the two usages require various operations to the code in the smart contract, which means that the bugs in these files may make these operations fail. For example in Listing 1, a test file *transferManager.js* tries to test the function *cachedPrices* in a Solidity file named *TokenPriceStorage*. But *TokenPriceStorage* does not have the function *cachedPrices*, which fails the test. This bug fix is as follows, where the non-fixed code starts with <, and the fixed starts with >.

```
const TokenPriceStorage = require("../build/
    TokenPriceStorage");
let tokenPriceStorage = await deployer.deploy(
    TokenPriceStorage);
< const tokenPriceSet = await tokenPriceStorage.
    cachedPrices(erc20First.contractAddress);
```

```
---
> const tokenPriceSet = await tokenPriceStorage.
    getTokenPrice(erc20First.contractAddress);
```

Listing 1: A fixed example in a JavaScript file.

The test file calls a function that does not exist and thus causes a bug. The fixed code changes the function call from *cachedPrices* to *getTokenPrice*, which is a function implemented in the Solidity file *TokenPriceStorage*. As we can see from this example, the bug in the test file makes the test fail and thus the functionality of the smart contracts cannot be tested. Similarly, the bugs in the deployment files may make the contract deployment fail and thus the project cannot work properly. It indicates that we should pay attention to not only the bugs in Solidity files but also the other source code files that operate the Solidity files.

Among the non-source files, the most common modified files are JSON documents and Markdown files. The JSON files are used to exchange data and hold the project configuration. Solidity compilers use JSON files to capture the output for each compiled contract to store the contract metadata, including the link to the libraries, the deployed bytecode, the ABI, and so on. By using the information in these JSON files, developers can easily create objects in other languages to represent the smart contract and then use its functions. In our study, most of the modified JSON files are the latest outputs of the fixed smart contracts. While the Markdown files are manuals and tutorials which are used to explain something about the projects. The bugs in these non-source files may result in the usage of wrong data or operations by developers.

Further, we compare the numbers of Solidity files, the other source code files (except the Solidity files), and non-source files known as configuration files and natural language documents that are modified during a bug fix. Then, we calculate the percentage of bug fixes according to the number of these three kinds respectively. The results are shown in Fig. 3. Its horizontal axes show the number of modified files in a bug fix and the vertical axes show the percentage of corresponding bug fixes. As shown in Fig. 3, the number of bug fixes decreases with the increase in the number of modified files. In total, about 80% of these bug fixes commits modify no more than one Solidity source file and nearly 20% involve two or more Solidity files. Specifically, nearly 47% (2887) of the 6146 bug fixes do not involve any Solidity files and 32% of the 6146 bug fixes involve one Solidity file. However, about 92% of the 6146 bug fixes involve at least one the other source code file, which is much higher than that of Solidity files. It indicates directly that bugs occur more frequently in the other source code files than in Solidity files. The developers mainly use the other source code files to interact with Ethereum and deploy the Solidity smart contracts, and test the functionality of the contracts. The bugs in these files may fail the deployment and testing of smart contracts, which may cause severe consequences. The high percentage of bug fixes that involve the other source files and the important functionality of these files inspire us to propose some novel tools in the future to help developers find the bugs in these files conveniently. In total, we can see that the distribution of bugs in smart contract projects is very complex, involving different source code files.

RQ2. How many solidity files are modified during a fix?

In RQ1, we find that nearly 47% (2887) of the 6146 bug fixes do not involve any Solidity files. In this RQ, we try to explore the number of modified Solidity files during the rest 3259 bug fixes that involve Solidity files. To answer this question, we calculate the number of modified Solidity files during the 3259 bug fixes. Fig. 4 shows the results. Its horizontal axes show the number of modified Solidity files and the vertical axes show the percentage of the corresponding bug fixes. As shown in Fig. 4, most bug fixes in Solidity files from these projects modify only one Solidity source file. But there are nearly 40% of the 3259 bug fixes that need to modify two or more Solidity source files. It shows the dependence of the Solidity files and indicates the complexity of the bug fixes at the Solidity file level. Current smart contract repair approaches (Yu et al., 2020; Nguyen et al., 2021; Chen et al., 2020) can only repair one independent Solidity file, so these bugs could not be repaired by nowadays automatic program repair techniques. We can understand the bug fixes at the Solidity file level from this result and see the limitations of nowadays automatic repair techniques.

RQ3. How many solidity files are necessary to be added or deleted to fix bugs?

To answer this question, we calculate the number of Solidity files added and deleted during the 3259 bug fixes that involve Solidity files. Then, we calculate the percentage of the bug fixes according to the number of Solidity files added and deleted and present the results in Fig. 5. The horizontal axes show the number of Solidity files added and deleted, while the vertical axes show the percentage of the corresponding bug fixes. Generally, about 90% of the 3259 bug fixes do not need to add new Solidity files. Even if it needs to add files, one file (6%) is quite enough. Almost all the 3259 bug fixes that involve Solidity files do not need to delete Solidity files. Since most of the bug fixes that involve Solidity files do not need to add extra Solidity files or delete existing Solidity files, it is reasonable for us to focus on fixing codes in Solidity files.

The results lead to the following finding.

Finding.1 The distribution of bugs in Solidity smart contract projects is very complex. About 92% of the 6146 bug fixes involve at least one the other source code file. Nearly 40% of the 3259 bug fixes that involve Solidity files modify two or more Solidity source files.

3.2. Fix complexity

After the discussion of file types and amount during bug fixes, in this section, we explore the complexity of these fixes at the Solidity code level.

RQ4. What is the most common fix operation during bug fixes?

In this paper, we use fix operation <*e*, *a*> to describe the fix actions on different elements, where *e* means an element and *a* means one action of *addition*, *deletion*, and *change*. Then, we calculate the number of fix operations in Solidity files to find out the most common one. We use AST to analyze and extract the fix actions to the code elements during bug fixes. The source code of each Solidity file is represented as an abstract syntax tree (AST) and we compare the ASTs of the origin Solidity file and the fixed Solidity file to get the fix operations during fixes. We do not consider the Solidity files that are added or deleted during bug fixes. For example, we count the number of *FunctionDefinition* nodes before and after bug fixes to judge the addition and deletion of functions. Then, we consider the subnodes of the *FunctionDefinition* nodes (the *name*, the *parameters*, the *modifier*, the *visibility*, the *stateMutability*, the *returnParameters*) to judge whether the *FunctionDefinition* is changed or not. According to the grammar of Solidity (Ethereum, 2021) and the Solidity parser in Python (ConsenSys, 2021), we finally define 19 kinds of code elements. As for the *Comment*, we get their fix operations by comparing their number and content before and after the bug fix in Solidity files. Fig. 6 shows the number of fix operations during bug fixes in Solidity files. Its vertical axis shows the names of the elements we define. Its horizontal axis shows the number of fix actions on *addition*, *deletion*, and *change*.

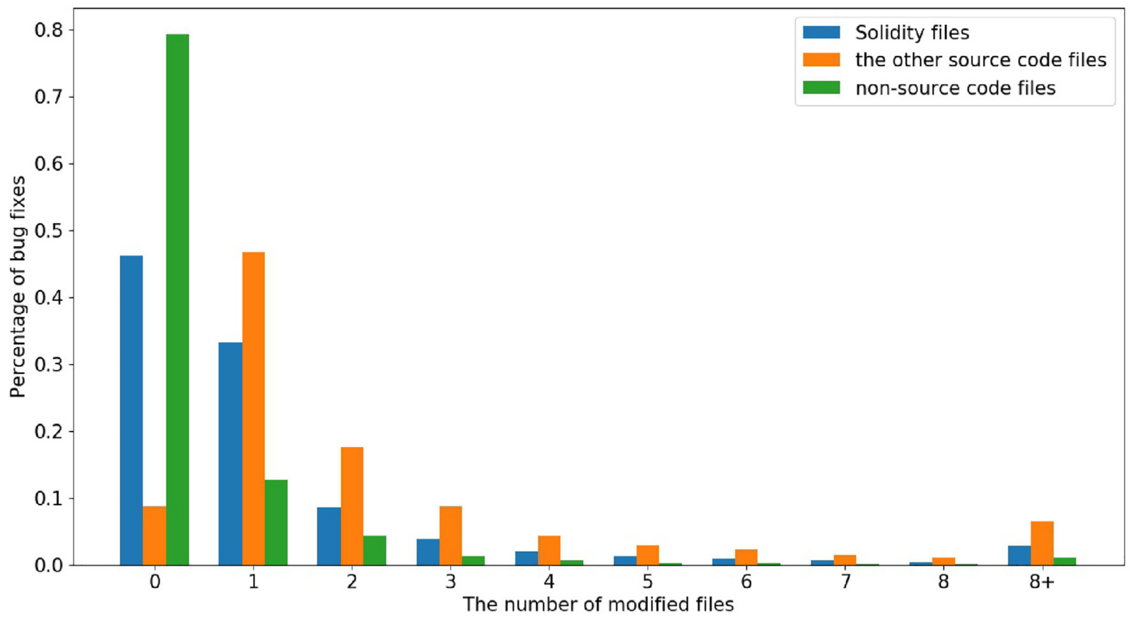


Fig. 3. The number of files modified during a bug fix.

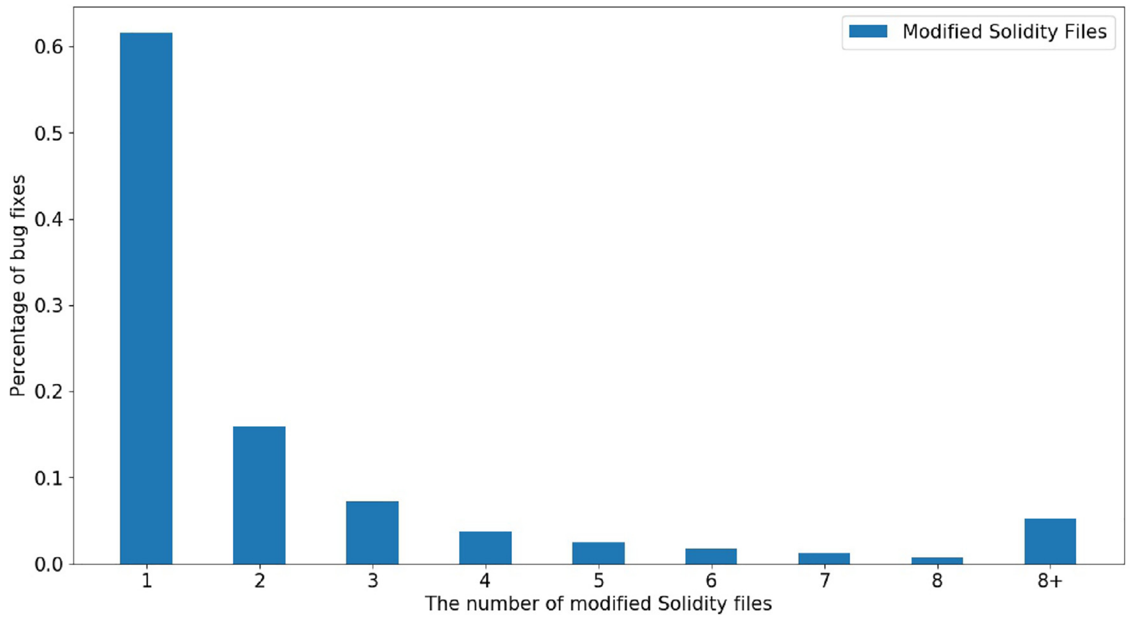


Fig. 4. The number of Solidity files modified during a bug fix.

As is shown in Fig. 6, non-code element *Comment* is the most common fixed element in our dataset. This result is not surprising, since it has the same result in traditional Java projects (Zhong and Su, 2015). <*Comment, addition*> is the most common fix operation during bug fixes in Solidity files. It is surprising that the number of comments that are added during bug fixes is so large. In Java projects (Zhong and Su, 2015), the developers change the code comments much more frequently than adding them during bug fixes. But it is quite different in Solidity smart contract projects, the number of comments that are added during bug fixes is much larger than that are changed. It seems to indicate that developers like to add comments to the Solidity code during bug fixes. The maximum number of comments added in Solidity files appears in *mosaic-contracts* with the commit hexsha *4cca31ea4b2bc206f39a8553afba93a31e5c1bc7*, where the bug fix adds 435 lines of comments in one Solidity file. We manually check this modification and notice that most of the comments are added to explain the elements written before. The developers do not write comments to the code while they are implementing it but add them during bug fixes. There exists one research to help generate comments for Solidity files. Yang et al. (2022) propose an information retrieval-based code comment generation method for the functions of smart contracts written in Solidity. Since there are so many comments added to the code during bug fixes, it may still have space to improve the comment generation method in Solidity files.

ExpressionStatement is the most common fixed code element. <*ExpressionStatement, change*> is the most fix operation in code elements during bug fixes in Solidity files. The inside node *expression* of *ExpressionStatement* is complicated, including lots of types like *FunctionCall*, *BinaryOperation*, and so on. *FunctionCall* may invoke complicated functions. *BinaryOperation* may operate

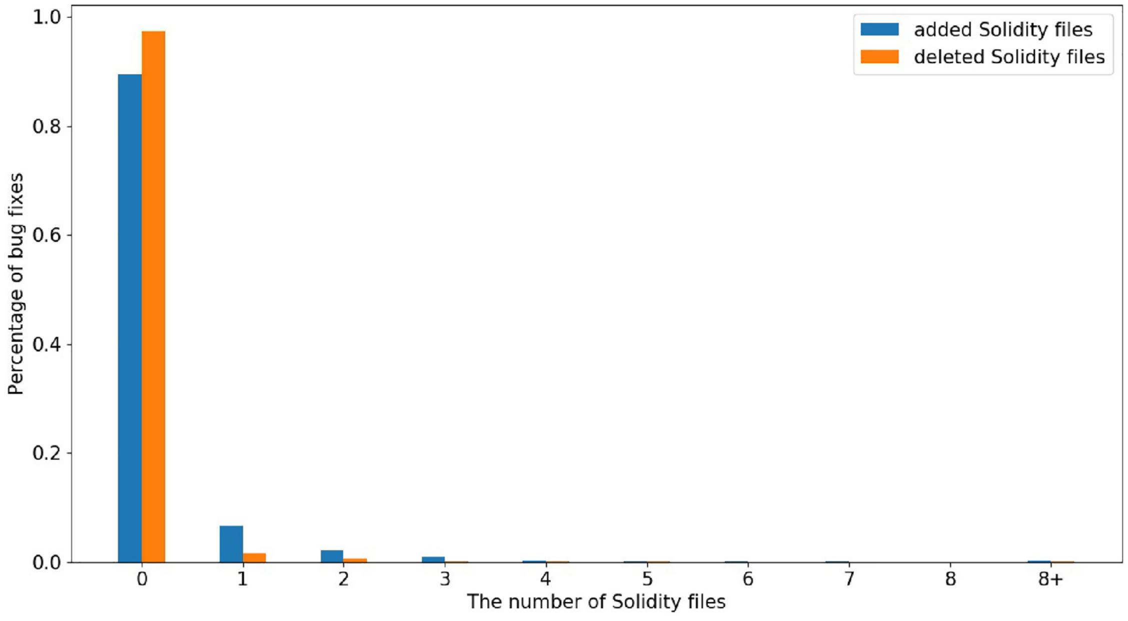


Fig. 5. The number of added and deleted Solidity files during a bug fix.

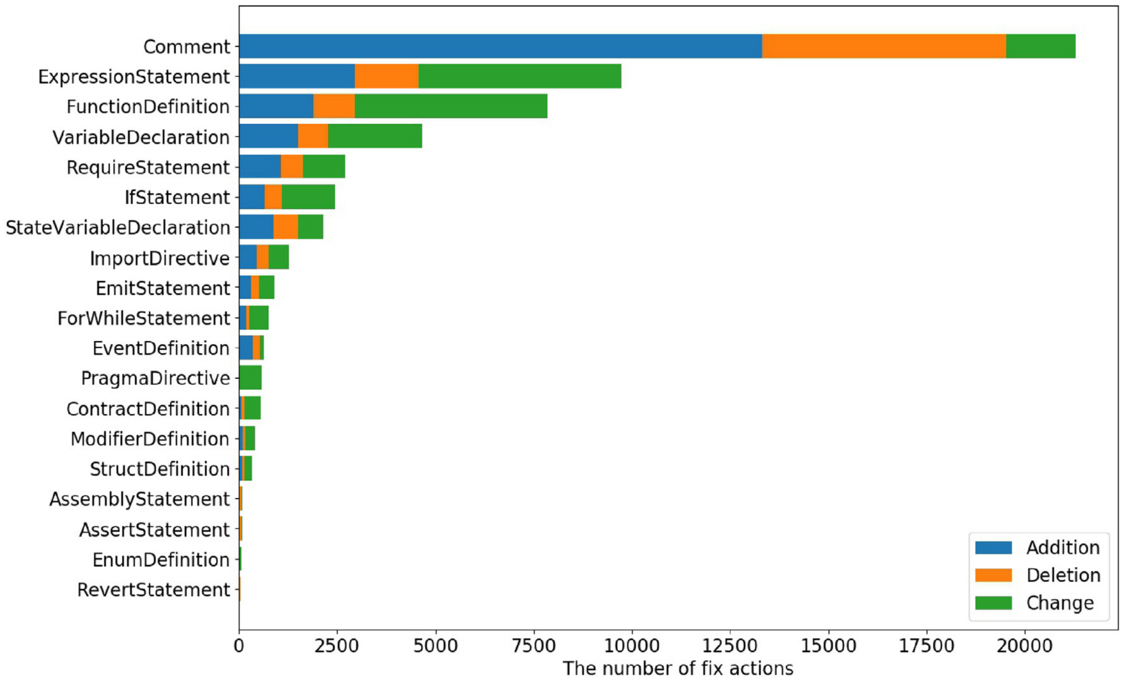


Fig. 6. Fix operations in Solidity files during bug fixes.

complicated objects. The complexity of the subnodes in *ExpressionStatement* makes it much easier than the other elements like *EmitStatement* that only has one type of subnode to be exposed to bugs and thus become the most common fixed code element. *FunctionCall* is the most common fixed type in the subnodes of *ExpressionStatement*. Maybe in the future, we can explore the invoked functions in *FunctionCall* to learn more knowledge of the functions called during bug fixes in Solidity files. For example, we can analyze the bug fixes in the functions called from the open-source Solidity libraries. The result may give us suggestions when we fix the same type of bugs. Similarly, *FunctionDefinition* also have complicated subnodes. <*FunctionDefinition, change*> ranks the second among the fix operations in code elements. *FunctionDefinition* has 6 types of subnodes that can be changed during bug fixes. We explore them to find out the most common fixed one. As is shown in Fig. 7, *parameter* is the most common fixed one in the changes of *FunctionDefinition*. Specifically, nearly 51% of the fix operations <*FunctionDefinition, change*> modify the parameters of functions. For example in Listing 2, where non-fixed code statements start with <, the fixed statements start with >, and *c* means change. To fix the bug, the name of the parameter *_uri* in the function *constructor* is changed into *uri_*. All the usages of *_uri* also have to be modified. It is meaningful for future directions of IDE support to highlight the function calls while changing the

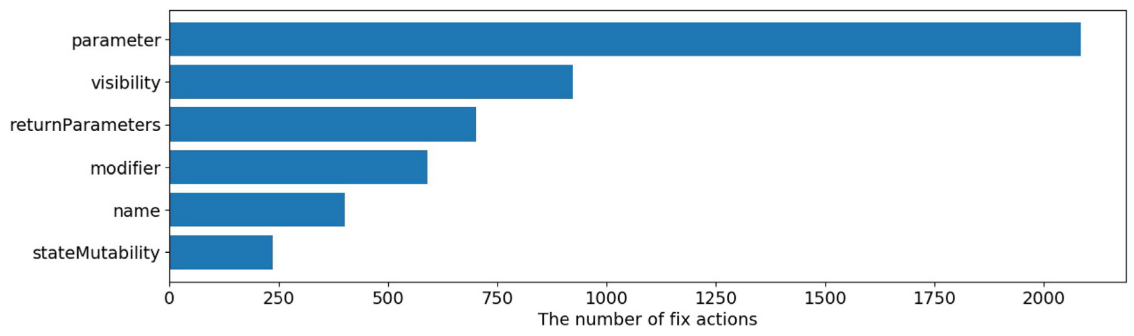


Fig. 7. Changes of *FunctionDefinition*.

functions and even automatically change the function calls to make it easier for developers to keep code consistent.

```
8c8
<   constructor(string memory _uri)
---
>   constructor(string memory uri_)
10,11c10,11
<     ERC1155(_uri)
<     NetworkAgnostic(_uri, ERC712_VERSION,
    ROOT_CHAIN_ID)
---
>     ERC1155(uri_)
>     NetworkAgnostic(uri_, ERC712_VERSION,
    ROOT_CHAIN_ID)
```

Listing 2: An example of modifying the parameter of a function.

```
3c3
<  pragma solidity >=0.6.2 <0.8.0;
---
>  pragma solidity 0.7.6;
```

Listing 3: An example of *Pragma Change*.

RQ5. How many element kinds are modified in a Solidity file during bug fixes?

In RQ4, we have defined 19 kinds of elements that may be modified during bug fixes. In this RQ, we want to explore the number of element kinds that are involved in a Solidity file during bug fixes. First, we cluster Solidity files according to the number of element kinds that are modified during bug fixes and present the results in Fig. 8. As is shown, during the bug fixes in Solidity files, 41% of the Solidity files involve only one kind of element, which means that nearly 59% of the Solidity files involve multi-element modification. More specifically, 20% of the Solidity files involve two elements, and 12% involve three-element. The number of Solidity files decreases with the increase in the number of element kinds. In general, a Solidity file involves 2.5 code element kinds on average during bug fixes. Further, we explore the kinds of elements that are modified. We cluster the Solidity files that only involve one kind of element and present the results in Fig. 9. *Comment* ranks the first. The result directly shows that quite a few bug fixes in Solidity files are taken only to fix the bugs in these code comments. It does not need to modify any code elements. Future research can explore it to help address the problems in these comments. *FunctionDefinition* ranks the second among the single-element modification. Some modifications of the *FunctionDefinition* may not cause the modifications of other code elements, such as the change of *visibility* and *modifier*. *PragmaDirective* ranks the third. Quite a few Solidity files only involve the *PragmaDirective* during bug fixes, which means the developers only change the versions of Solidity compiles. For example in Listing 3, to fix the bug in a Solidity file, the version of the pragma compile is changed from *>=0.6.2 < 0.8.0* to *0.7.6*. The version of the Solidity is changed from a range to an exact one. Most of the Solidity files only involving *PragmaDirective* take the fix in the same way. It indicates that we have better use the exact version of compile to avoid the bugs caused by the versions of compiles.

As for the multiple-element modification, we use the Kendall (1938) to analyze the correlation between elements. We show the results in Fig. 10, where blue represents the positive correlation. The bluer it is, the stronger the positive correlation is. The results show that *EventDefinition* and *EmitStatement* are the most relevant with a correlation coefficient of 0.55. *ImportDirective* and *ContractDefinition* rank the second with the 0.50. Event is used to store the arguments passed in transaction logs, which is called by *EmitStatement*. So it is reasonable that the modification of *EventDefinition* usually appears with the modification of *EmitStatement*. It is meaningful for future IDE support to highlight *EventDefinition* and its corresponding *EmitStatement* to help the developers to manage the code. *ImportDirective* is used to load other Solidity files for manipulation. Our result shows that the developers usually inherit the contracts in these imported Solidity files to fix bugs. The development between inherited and new contracts is also worth being researched.

RQ6. How many fix actions are taken to a Solidity file during bug fixes?

We calculate the number of fix actions that are taken to a Solidity file during bug fixes and cluster the Solidity files according to it. The result is shown in Fig. 11. More specifically, we calculate the number of fix actions taken to all the elements and only to the code elements in a Solidity during bug fixes respectively. As is shown in Fig. 11, nearly 31% of the Solidity files are taken only one fix action to code elements. Combined with the result in RQ5, 41% of the Solidity files modify one kind of element, we can conclude the result that most of the Solidity files that involve one element kind only need to be modified by one action. It is also what the existing repair techniques have done to fix. The result also shows that 68% of the Solidity files need to be modified by several fix actions. The average number of fix actions to code elements and all the elements including *Comment* in a Solidity file during bug fixes are 7.4 and 9.8. In RQ5, we notice that a Solidity file involves 2.5 code element kinds on average during bug fixes. It means that the developers need to take an average of three actions to one code element kind in a Solidity file during bug fixes. This result clearly shows the limitations of current automatic repair techniques, since the majority of fixes produced by them only take one action to one code element in a Solidity file.

Finding.2 *Comment* is the most common fixed element during bug fixes in Solidity files. Nearly 59% of the Solidity files involve multi-element modification. *EventDefinition* and *EmitStatement* are the most relevant elements in the multi-element modification with a correlation coefficient of 0.55. On average, a Solidity file involves 2.5 code element kinds modification and 7.4 fix actions to code elements during bug fixes.

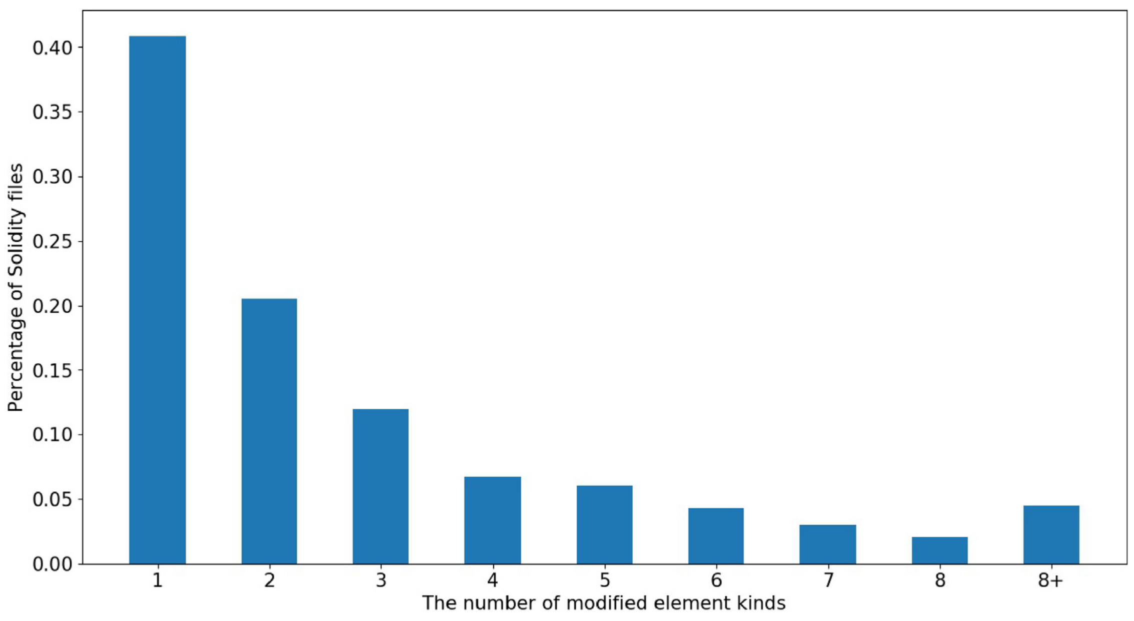


Fig. 8. The number of element kinds modified in a Solidity file during bug fixes.

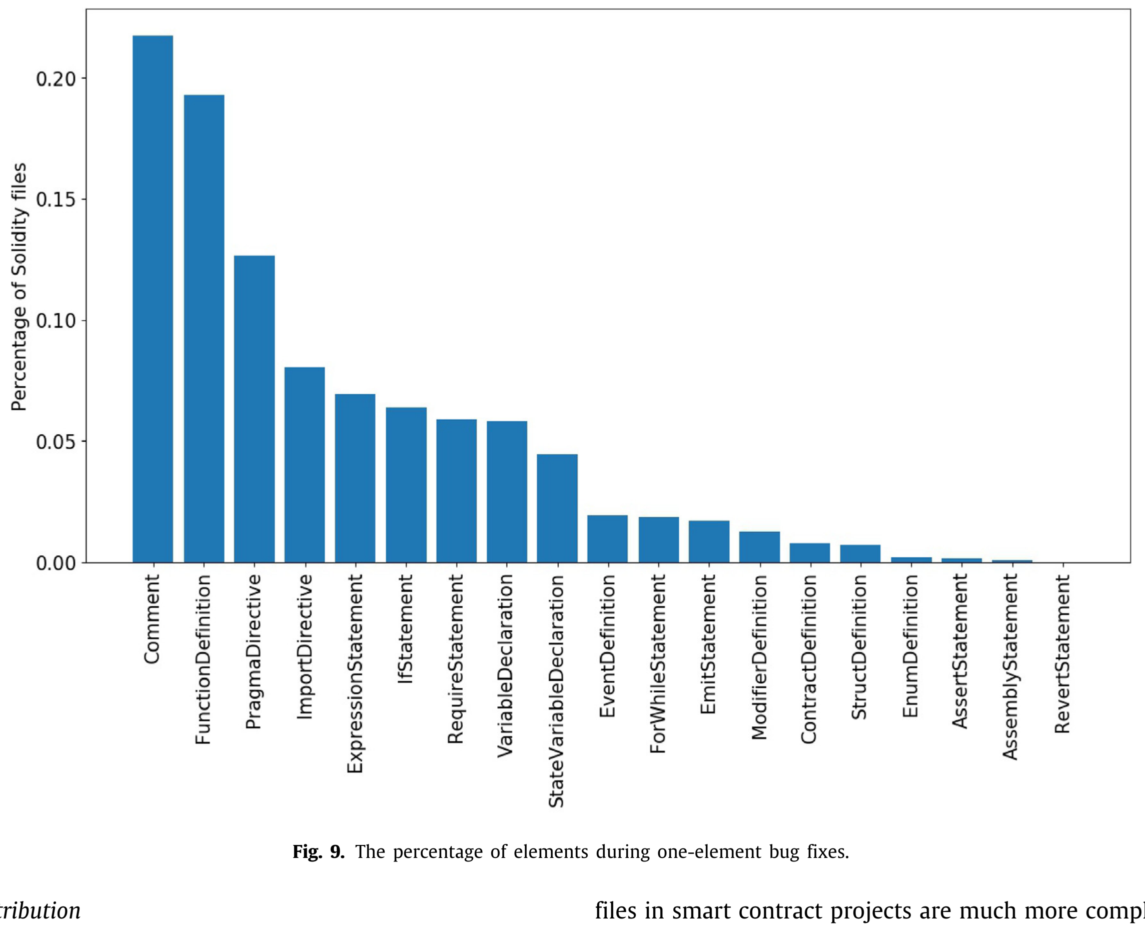


Fig. 9. The percentage of elements during one-element bug fixes.

3.3. *Bug distribution*

In this part, we try to analyze the distribution of the bugs that can be detected by analysis tools in the historical bug-fixed versions of the Solidity files. However, there is one problem we need to consider before our study. Due to the limitations of current analysis tools to detect bugs in Solidity files, we cannot analyze Solidity files in these smart contract projects directly. The Solidity files in smart contract projects are much more complicated than those on Ethereum. These Solidity files may import Solidity files under other directories or online Solidity files to achieve certain functions. But current analysis tools to detect bugs in Solidity files cannot identify and import the related Solidity files automatically. To solve this problem, we must import the dependent content of the Solidity files first. Then, use the analysis tools to detect the Solidity files with full imported content. In practice, we take

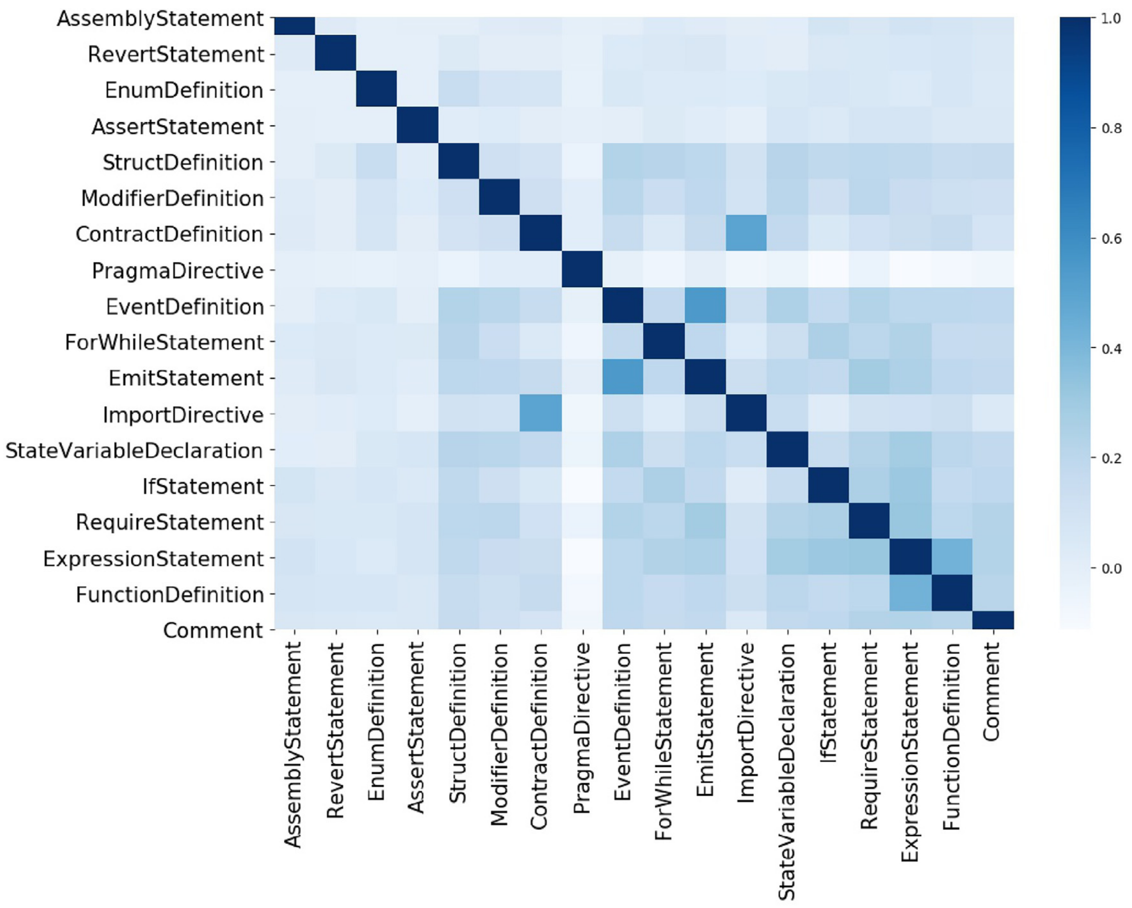


Fig. 10. The correlation between elements.

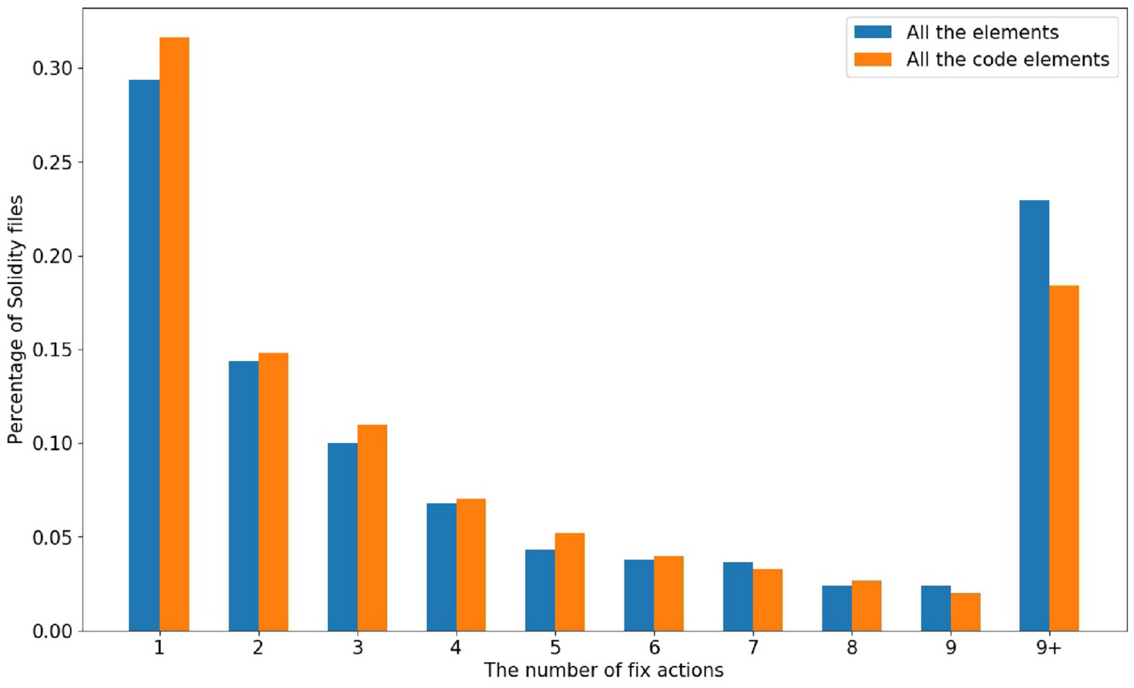


Fig. 11. The number of fix actions taken to a Solidity file during bug fixes.

three steps to solve this problem. First, we utilize the abstract syntax tree (AST) to obtain the imported files and the inheritance relationship among contracts from the Solidity files. Second, we use Topological Sorting (Kahn, 1962) to get the inheritance order of the contracts. Then, we extract the contracts from the Solidity files and merge them according to the inheritance order. Based

Table 2
The description of the 14 categories of bugs.

Category	Description
Arithmetic	*Integer Overflow* (CWE, 2022a), and *Integer Underflow* (CWE, 2022b).
Reentrancy	Reentrant function calls make a contract behave in an unexpected way (Durieux et al., 2020).
Front-Running	Two dependent transactions that invoke the same contract are included in one block (Durieux et al., 2020).
Access Control	The developers do not restrict or incorrectly restrict the access of functions, use of tx.origin, or make reckless use of delegatecall (Durieux et al., 2020).
Bad Randomness	Malicious miner biases the outcome (Durieux et al., 2020).
Time manipulation	The timestamp of the block is manipulated by the miner (Durieux et al., 2020).
Denial of Service	The contract is overwhelmed with time-consuming computations (Durieux et al., 2020).
Locked-ether	Ethereum smart contracts can receive Ether. But the received funds might get locked permanently into the contract (Perez and Livshits, 2021).
Uninitialized variables	Uninitialized variables.
Strict Balance Equality	Use of strict equalities that can be easily manipulated by an attacker (Feist et al., 2019).
Shadow variables	Shadow variables occur within a single contract when there are multiple definitions on the contract and function level (SmartContractSecurity, 2020).
Unmatched ERC-20 Standard	The function name, parameter types, and return value do not strictly follow the ERC20 standard, for example, miss return values or miss some functions (Chen et al., 2020).
Unused-return	The return value of an external call is not stored in a local or state variable (Feist et al., 2019).
*A*rbitrary-send	Unprotected call to a function sending Ether to an arbitrary address, or when *msg.sender* is not used as from in *transferFrom*.

on this method, we revert to the specific bug fix commits to obtain the Solidity files under different versions of the projects. We finally pack 116,410 Solidity files under the 3259 bug-fix commits of the projects.

We use Mythril (Mueller, 2018) and Slither (Feist et al., 2019) to detect bugs in Solidity files. According to the evaluation of nine current analysis tools (Durieux et al., 2020), the tool Mythril has the best accuracy among them. But Mythril is not powerful enough to replace all the tools. So we follow the suggestion in Durieux et al. (2020) and use the combination of Mythril and Slither.

RQ7. What are the specific bugs reported by Mythril and Slither?

Considering the impact and severity of the bugs, we do not take into account the informational bugs reported by Mythril or Slither. The two tools detect 27 types of bugs in our dataset in total. Mythril and Slither use different descriptions for bugs. To facilitate our analysis, we first need to have a unified description of these bugs. We classify these bugs according to the taxonomy presented in the DASP10 (N.C.C. Group, 2018). As for the bugs that DASP10 does not cover, we classify them according to their characteristic. We finally divide these 27 types of bugs into 14 categories. Table 2 presents detailed information about the categories.

RQ8. How many bugs are reported by Mythril and Slither and how many Solidity files do they exist in?

We identify the bugs by *<category, project, function, code>*. Then, we calculate the number of bugs reported by the two tools. There are some categories like *Reentrancy* and *Access Control* that can be detected by both Mythril and Slither. We consider the bugs that are both reported by Mythril and Slither to reduce false positives. As for those categories that can only be detected by one tool like *Arithmetic* and *Time manipulation*, we directly calculate the number of bugs detected by the tool. Table 3 shows the number of bugs identified by Mythril and Slither. Each row in Table 3 represents a bug category. The Total column shows the number of bugs.

As Table 3 is shown, *Arithmetic* is the most common bug category, which refers to the bugs *Integer Overflow* and *Integer Underflow*. *Integer Overflow* and *Integer Underflow* are common types of bug in many programming languages (Perez and Livshits, 2021). But they can be used by hackers to attack and steal money from smart contracts in the context of Ethereum. The *Unused-return* is the second most common category in our dataset. It means that developers lack a check on the return value of a function. Without the check for the return value, developers cannot detect unexpected states and conditions of these functions (CWE, 2022c). The code in the functions may resume executing even if the invoked function throws an exception. An attacker could force the function to fail or otherwise return a value that is not expected, then the subsequent program logic could lead to unexpected results.

There are 3545 Solidity files with different names from the 3259 bug-fix versions of the 46 smart contract projects in total. After removing the duplicates, we find that nearly 20.3% (720) of the 3545 Solidity files have been exposed to bugs.

Finding.3 Nearly 20.3% of the 3545 Solidity files in our dataset have been exposed to bugs. There are 14 categories of bugs reported by Mythril and Slither. *Arithmetic* and *Unused-return* are the two most common bugs.

3.4. Fix patches

As we introduced in Bug Distribution, we check out 3259 bug fix commits of the 46 projects and finally pack 116,410 Solidity files under these versions. In this part, we use these historical

Table 3
Bugs identified per category by Mythril and Slither.

Category	Mythril	Slither	Total
Arithmetic	803	/	803
Reentrancy	859	942	141
Front-Running	52	0	52
Access Control	20	17	8
Bad Randomness	27	0	27
Time manipulation	/	58	58
Denial of Service	/	418	418
Locked-ether	/	51	51
Uninitialized variables	/	364	364
Strict Balance Equality	/	94	94
Shadow variables	/	178	178
Unmatched ERC-20 Standard	/	30	30
Unused-return	/	470	470
Arbitrary-send	/	119	119
Total	1,761	2,741	2,813

versions of Solidity files to determine whether the bugs that are reported by Mythril and Slither have been fixed or not, how many bugs have been newly introduced, and how they are fixed.

RQ9. How many bugs have been fixed and how many bugs have been newly introduced?

We use <*category, project, function, code*> as the unique identifier IDs for bugs to facilitate our tracking of these bugs. It is worth noting that the code specifically refers to those that are capable of detecting bugs. During the process of fixing bugs, the developers may modify or add new code, then these changes may cause the identifier IDs of the bugs to change. For example, the developers change the code statement of the unique identifier ID, which causes the identifier ID to change from <*category, project, function, code*> to <*category, project, function, code_c*>. In this case, we consider the original bugs <*category, project, function, code*> to be fixed, but the developers may have introduced a new bug by *code_c*. Hence, simply monitoring the number of fixed and non-fixed bugs does not accurately reflect the level of significance that developers place on these bugs. To address this, we calculate the number of bugs that have been newly introduced by developers during the bug fixes. Combining the two results, we consider that if the developers put much attention to fixing the bugs reported by the tools, then the fix rate of the bugs should be high and the number of newly introduced bugs should be small.

When analyzing whether a bug has been fixed or not, we first identify the version file where the bug first appeared. Then, starting from this version file, we analyze the later version files with the results from Mythril and Slither to determine whether the bug has been fixed. Once the bug does not exist in the next versions of the Solidity file, we consider that it has been fixed. For those bugs that exist from the initial version where the bug first appeared to all of the later versions, we consider them to be non-fixed. We compare the time of bug-fix commits to determine whether the bugs are newly introduced or not. If it has not appeared in a previous commit, we assume that it is a newly introduced bug. Fig. 12 illustrates the distribution of fixed, non-fixed, and newly introduced bugs, while its horizontal axes show the abbreviation of bugs and the vertical axes show the number of bugs.

By comparing the results in Fig. 12, we notice that for most of the 14 categories of bugs, the number of newly introduced bugs exceeds that of the fixed bugs. For some bugs, such as *Uninitialized variables* and *Time manipulation*, the number of newly introduced bugs is close to that of fixed bugs. It shows that the developers may not put much attention to fixing them completely or avoid introducing them again. Because if the developers attach great importance to fixing the bugs reported by the tools completely and avoiding introducing them again, the number of newly introduced bugs should be much smaller than that of the fixed ones. It seems that although there are serious smart contract security incidents (Dalakos, 2021; Ethereum, 2018) that are exploited by the bugs like *Arithmetic* in smart contracts. Maybe these bugs are common in Solidity files, and the developers consider that it is rare and hard to exploit them. So they do not pay much attention to fixing these bugs completely or avoid introducing them strictly. However, *Access Control* has a different trend from the other categories of bugs. It has the highest fixed percentages and the smallest number of newly introduced bugs, which indicates that developers may pay much attention to fixing them completely and avoid introducing them again when they are fixing this category of bugs. This phenomenon inspires us to think about the priority to fix bugs in the future. There have been many analysis tools proposed to detect bugs in Solidity files. Although they can detect quite a few bugs, developers may not pay much attention to fixing all of them completely. These tools may provide suggestions for developers to fix the bugs with a high fixed percentage and low introduction number like *Access Control* first.

RQ10. How do developers fix these bugs?

In this part, we try to explore how the developers fix those fixed bugs in RQ9 by manually checking the fix in Solidity files. More specifically, we find the two versions of the Solidity file before and after the bug fix, and then compare their difference to determine the specific fix. Considering the size of the samples, we only select the top 4 bug categories in terms of the total number of fixed bugs. Then, we randomly select a sample of 50 in each of these four categories for manual inspection, and a total of 200 samples. We invite three volunteer graduate students to participate in the manual check process. They have an average of 1.5 years of development experience in Solidity smart contracts. Two of them check the human-written patches independently. If there is a discrepancy between them, the rest participant determines the final result.

During our study, we find that these fixed instances can be roughly divided into three categories: one is that the original bug is modified into another one. Developers may change the *code* and *function* of <*category, project, function, code*> to fix the functionality of the smart contracts rather than fix the bug. In this case, we consider that the original bug is modified into a newly introduced bug. The second one is that the bug is deleted. The third is that the bug is truly fixed.

Reentrancy. Although we have only considered the *Reentrancy* bugs that are both reported by Mythril and Slither to reduce the false positives. But among our manual check, we find that most of these samples are still false positives. Most of the 50 samples are related to the function *addr.transfer()*, which is marked as a *Reentrancy* bug by Slither. *addr.transfer()* has a built-in gas limit of 2300 to prevent Reentrancy problems (Ren et al., 2021b). Some of the published tools mark it as true bugs during their evaluation. But it cannot be exploited in practice.

Arithmetic. There are 9 false positives in our samples of *Arithmetic*. 18 of the rest samples are modified into the newly introduced bugs. 5 bugs are fixed by deletion. Most of the 18 truly fixed samples are fixed by using a vetted safe math library for arithmetic operations like the example in Listing 4.

```
<  totalSupply += msg.value;
---
>  totalSupply = safeAdd(totalSupply, msg.value)
    ;
```

Listing 4: A fixed example of *Arithmetic*.

Uninitial Vairbales. *Uninitial Vairbales* is caused by using uninitialized storage variables, uninitialized state variables, and uninitialized local variables. These uninitialized variables sometimes

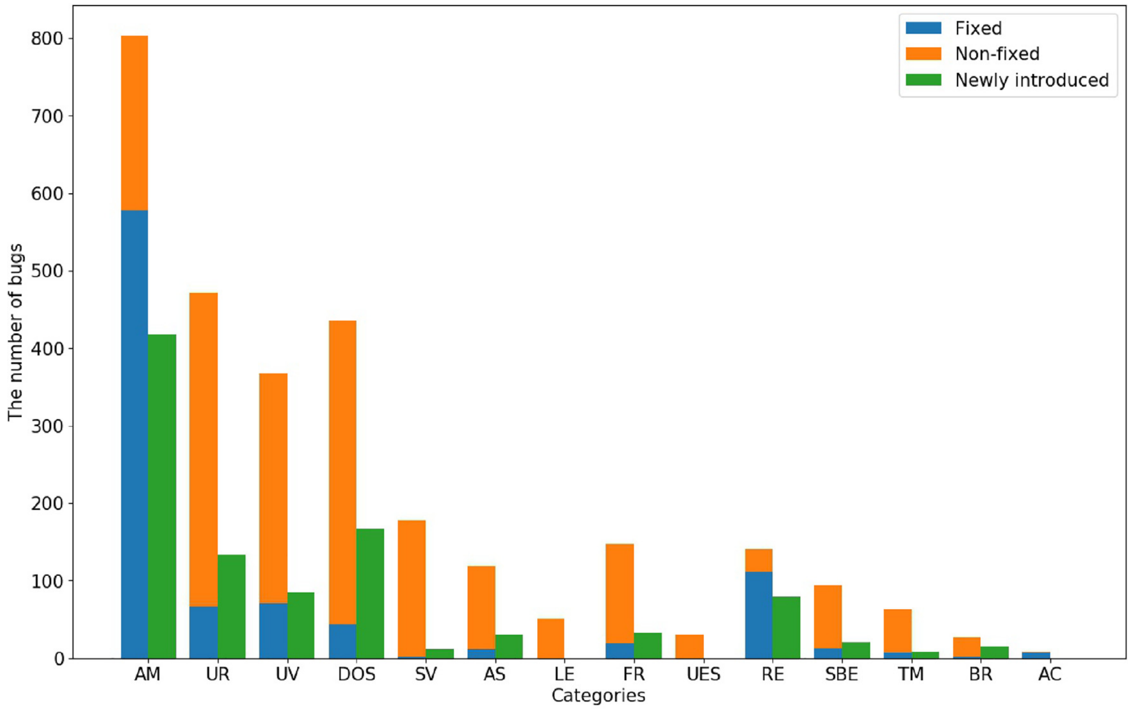


Fig. 12. The number of fixed, non-fixed, and newly introduced bugs during bug fixes. For each category: AM means *Arithmetic*, UR means *Unused-return*, UV means *Uninitialized variables*, DOS means *Denial of Service*, SV means *Shadowing variables*, AS means *Arbitrary send*, LE means *Locked-eth*, FR means *Front-Running*, UES means *Unmatched ERC20 Standard*, SBE means *Strict Balance Equality*, TM means *Time manipulation*, RE means *Reentrancy*, BR means *Bad Randomness*, AC means *Access Control*.

may cause financial loss. For example, the developer defines a state variable of address but fails to initialize it. Once the developers use it as the destination to transfer, the Ethers are sent to the address *0x0* and are lost. However, among the 50 samples that we select from *Uninitial Vairbales*, only 18 of them are truly fixes like the example in Listing 5, where the developer set the exact value to initialize the variable *liquidationIncentivePercent*. 23 of them are deleted and the rest 9 are modified into the newly introduced bugs.

```
<   uint256 public liquidationIncentivePercent;
---
>   uint256 public liquidationIncentivePercent =
     5 * 10**18; // 5\% collateral discount
```

Listing 5: A fixed example of *Uninitialized variables*.

Unused-return. *Unused Return* is caused by the lack of a check on the return value of a function. Among the 50 samples we select, only 6 of them are truly fixed by developers. All of these truly fixed bugs are ensured the return values of the function calls by *RequireStatement* like the example in Listing 6.37 of the rest are modified into the newly introduced bugs. 7 bugs are deleted.

```
<  token.transfer(transaction.receiver,
    _amountReimbursed);
---
>  require(token.transfer(transaction.receiver,
    _amountReimbursed) != false, "The transfer
    function must not failed.");
```

Listing 6: A fixed example of *Unused-return*.

Finding.4 The developers may not put much attention to fixing the bugs reported by the tools completely or avoid introducing them again. Mythril and Slither perform poorly in detecting *Reentrancy* bugs in practice.

3.5. Discussion

In this section, we provide the actionable implications based on our findings for researchers in Solidity smart contracts.

Automatic Repair Techniques. Our findings provide two actionable implications for researchers in the automatic repair techniques of smart contracts. First, it is meaningful to improve the code comment generation method for Solidity files. *Comment* is the most common fixed element. *<Comment, addition>* is the most common fix operation in Solidity files during bug fixes, which indicates the space to research the code comments in Solidity files. The researcher can explore the outdated comments and missing comments in Solidity files. Second, current automatic repair techniques need to extend to multiple-element modifications to fix more bugs. 59% of the Solidity files involve multi-element modification. Most of the current automatic repair techniques are restricted to one code element, which is not enough to fix the bugs in practice. It is meaningful for automatic repair techniques to learn the relationship between the code elements like *EventDefinition* and *EmitStatement*, and *ImportDirective* and *ContractDefinition* to fix more bugs automatically.

Analysis Tools. Our results and findings provide three actionable implications of the analysis tools of bugs in Solidity files. First, our results show the complex distribution of bugs at the file level in Solidity smart contract projects. Nearly 40% of the bug fixes in Solidity files modify two or more Solidity source files. 92% of the 6146 bug fixes involve at least one the other source code file. So it is meaningful for researchers to propose analysis tools to check the code link among the Solidity files and to help developers find out the bugs in the other source code files. For example, considering the main usage of the other source code files, researchers can check the function calls from Solidity files used in the other source code files and provide suggestions to modify the other source code files in time while modifying the corresponding code in the Solidity files. It can help developers to determine the impact of bugs more quickly at the different language file levels and reduce the bugs in the other source code files caused by code inconsistencies. Second, although the current analysis tools of bugs in Solidity files can detect quite a few bugs, developers may not pay much attention to fixing all of them completely or avoid introducing them again. So the analysis tools can provide the priorities to fix bugs for developers. For example, the analysis tools can suggest fixing the bugs with a high fixed percentage and low introduction rate like Access Control first. Third, it is important to reduce the false positives in *Reentrancy* reported by analysis tools. Although we have only considered the *Reentrancy* bugs that are both reported by Mythril and Slither to reduce the false positives. But most of the samples we select are still false positives, which are hard to exploit in reality. It seems that these analysis tools need to improve the precision rate to detect *Reentrancy* in reality.

Solidity developers. Our results and findings provide the following recommendations to Solidity developers. First, it is better to use the exact version of the Solidity compile. Our results find that some of the bug fixes only involve the *PragmaDirective* during bug fixes. It means the developers only change the versions of Solidity compiles. What is more, most of these fixes change the version of the Solidity compile from a range to an exact one. Therefore, it is recommended that developers accurately specify the version of the compiler at the beginning to reduce the possibility of bugs caused by incorrect versions in the future. Second, to improve the security of smart contracts, use safe library functions and *RequireStatement* for condition checks as much as possible. During our exploration of human-written patches, we find that using safe library functions and *RequireStatement* is the most common way to fix the bugs. So, we suggest developers can learn from these instances and use the same way to avoid some security bugs at first.

4. Related work

Multiple studies that focus on bug fixing and smart contracts have been published in recent years. This section summarizes existing studies in relation to our work.

Empirical Study on Bug Fixes. Yin et al. (2011) present a comprehensive characteristic study on incorrect bug fixes from large operating system code bases and a mature commercial OS. Their results show that the bug-fixing process can also introduce errors, which leads to buggy patches that further aggravate the damage. Nguyen et al. (2013) present a study of the repetitiveness of code changes in the evolution of 2841 Java projects. Their results show that the repetitiveness of changes could be very high at small sizes and decreases exponentially as size increases. What is more, repetitiveness is higher and more stable in cross-project settings than in within-project ones. Fixing changes repeat similarly to general changes. Zhong and Su (2015) have conducted an empirical study on thousands of real-world bug fixes from five popular Java projects to analyze the links between the nature of bug fixes and automatic program repair. They summarized two key ingredients of automatic program repair: fault localization and faulty code fix, which provide useful guidance and insights for improving the state-of-the-art of automatic program repair. Campos and de Almeida Maia (2017) explore the underlying patterns in bug fixes mined from software project change histories in Java repositories. They characterized the prevalence of the five most common bug-fix patterns in bug fixes. The results showed that developers often forget to add IF preconditions in the code. Bernardi et al. (2018) try to find out the relation between the bug-inducing and fixing phenomenon and the lack of written communication between committers in open-source projects. They perform an empirical study on four open-source projects and find that increasing the level of communication between fix-inducing committers could reduce the number of fixes induced in a software project. Wen et al. (2019) conduct the first systematic empirical study to understand the correlations, in terms of code elements and modifications, between a bug's inducing and fixing commits. Their results show that leveraging the information of bug-inducing commits can significantly boost the performance of existing automated fault localization and program repair techniques. Wang et al. (2020) aim to investigate the effects of developers' familiarity with bugs on the efficiency and effectiveness of bug fixing. They conduct an empirical study on 6 well-known Apache Software Foundation projects with more than 9000 confirmed bugs. They find that familiarity with bugs has complex effects on bug fixing: the developers may fix the bugs introduced by themselves more quickly, but they are more likely to introduce future bugs when fixing the current bugs.

Smart Contract Security. In recent years, we have seen a great deal of academic and practical interest in the topic of bugs in smart contracts. To alleviate the problem of insecure smart contracts, researchers have developed various analysis tools and verification frameworks to detect bugs. Oyente (Luu et al., 2016) is a symbolic execution tool to find potential security bugs in smart contracts on the Ethereum system. Mythril (ConsenSys, 2019) uses static analysis, taint analysis, and concolic execution to detect a variety of security bugs in smart contracts. ContractFuzzer (Jiang et al., 2018) is a fuzzing framework to detect the bugs of Ethereum smart contracts. Securify (Tsankov et al., 2018) converts contract bytecode to Datalog and extracts semantic facts. Then it transforms the bugs into a series of compliance and violation patterns to search for unsafe patterns in smart contracts. SmartCheck (Tikhomirov et al., 2018) checks XML-based intermediate representation of Solidity source code against XPath patterns to detect bugs (Ren et al., 2021a). Pied-Piper (Tsinghua University, 2019) is a static analysis tool that constructs CFG based on bytecode and extracts semantic facts to detect potential backdoors hidden in smart contracts (Ren et al., 2021a). SmartEmbed (Gao, 2020) is a deep learning-based approach for detecting bugs in smart contracts.

To reduce the effort of fixing the bugs of smart contracts, various approaches have been proposed to automatically repair programs. Yu et al. (2020) present a gas-aware automated smart contract repair approach. The repair algorithm is search-based, and it breaks up the huge search space of candidate patches down into smaller mutually exclusive spaces that can be processed independently. The repair approach considers gas usage of contracts when generating patches for detected bugs. Nguyen et al. (2021) and Zhang et al. (2020) propose approaches by analyzing the bytecode to fix potentially vulnerable smart contracts automatically. Nguyen et al. (2021) propose a tool called SGUARD, which first collects a finite set of symbolic execution traces of the smart contract and then performs static analysis on the collected traces to identify potential bugs. Then it applies a specific fixing pattern for each type of bug in the source code. Zhang et al. (2020) extract bytecode-level semantic information and utilize them to transform insecure contracts into secure ones. Our study shows the distribution of bugs in real-world Solidity smart contract projects and builds the foundation for the automatic repair of smart contracts.

Empirical Study on Smart Contract. Chen et al. (2020) conduct the first empirical study by collecting smart-contract-related posts from Ethereum StackExchange as well as real-world smart contracts to understand and characterize smart contract defects. Pinna et al. (2019) perform a comprehensive empirical study of smart contracts deployed on the Ethereum blockchain. Their empirical results show the features of smart contracts and smart contract transactions within the blockchain, the role of the development community, and the source code characteristics.

The study contributes to understanding the interaction between Smart Contracts and Blockchain and to the knowledge of the main characteristics of contracts written in Solidity. Wan et al. (2021) perform a mixture of qualitative and quantitative studies with software practitioners who have experience in smart contract development to understand practitioners' perceptions and practices on smart contract security. Their results show that smart contract practitioners tend to have a higher awareness of security than practitioners in other software areas. Perez and Livshits (2021) focus on finding how many of the vulnerable contracts have been exploited. They survey the vulnerable contracts reported by six recent academic projects and find that, despite the amounts at stake, no more than 2% of them have been exploited since deployment. Hwang and Ryu (2020) conduct an empirical study on Solidity patches and live contracts to understand the current security status of real-world smart contracts. Their results show that many Solidity developers are unaware of the importance of Solidity patches. Durieux et al. (2020) present an empirical evaluation of 9 state-of-the-art automated analysis tools to obtain an overview of the current state of automated analysis tools for Ethereum smart contracts. Zou et al. (2019) perform an exploratory study to understand the current state and potential challenges developers are facing in developing smart contracts on blockchains compared to traditional software development. Our study provides a multi-faceted discussion on bug fixes that are extracted from the history of real-world Solidity smart contract projects.

5. Threats to validity

Threat to internal validity is related to the tools we use. There are roughly two categories to analyze the bug in smart contracts: static and dynamic analysis. Static analysis tools catch bugs or vulnerabilities without the need to deploy smart contracts, while dynamic analysis tools work in the opposite way. Static analysis tools have been the main focus of research (Perez and Livshits, 2021). In this paper, we use the static analysis tool Mythril and Slither to analyze smart contracts. A potential threat to the internal validity is related to the fact that there may be some false positives in our results and they may cause some bias. Our future research agenda is to reduce the influence of these false positives.

Threat to external validity is related to the commits that are extracted only from open-source Solidity smart contract projects. Our results may not generalize to commercially developed projects or smart contract projects using different programming languages.

6. Conclusion

Despite numerous efforts in detecting and repairing the bugs in smart contracts, little is known about bug fixes in Solidity smart contract projects. In this paper, we provide a multi-faceted discussion of bug fixes in real-world Solidity smart contract projects. We conduct an empirical study to explore the File type and amount, Fix complexity, Bug distribution, and Fix patches in the historical bug-fixed versions of the 46 Solidity smart contract projects. We finally distill 4 findings and provide actionable implications from three aspects for researchers to improve the current approaches and propose novel approaches for the bug fixes in Solidity smart contracts.

CRediT authorship contribution statement

Yilin Wang: Visualization, Investigation, Writing – original draft. **Xiangping Chen:** Conceptualization, Methodology. **Yuan Huang:** Writing – review & editing. **Hao-Nan Zhu:** Visualization, Investigation. **Jing Bian:** Methodology. **Zibin Zheng:** Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

The work described in this paper is supported by the Key-Area Research and Development Program of Guangdong Province (2020B010164002) and Guangdong Basic and Applied Basic Research Foundation (2023A1515010746), National Natural Science Foundation of China (61976061). Yuan Huang is the corresponding author.

References

Bernardi, M.L., Canfora, G., Di Lucca, G.A., Di Penta, M., Distante, D., 2018. The relation between developers' communication and fix-inducing changes: An empirical study. J. Syst. Softw. 140, 111–125.
Campos, E.C., de Almeida Maia, M., 2017. Common bug-fix patterns: A large-scale observational study. In: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM, IEEE, pp. 404–413.
Chen, J., Xia, X., Lo, D., Grundy, J., Luo, X., Chen, T., 2020. Defining smart contract defects on ethereum. IEEE Trans. Softw. Eng..
ConsenSys, 2019. Mythril. URL https://github.com/ConsenSys/mythril-classic.
ConsenSys, 2021. Python-solidity-parser. URL https://github.com/ConsenSys/python-solidity-parser.
CWE, 2022a. Integer overflow. URL https://cwe.mitre.org/data/definitions/190.html.
CWE, 2022b. Integer underflow. URL https://cwe.mitre.org/data/definitions/191.html.
CWE, 2022c. Unchecked return value. URL https://cwe.mitre.org/data/definitions/252.html.
Dalakos, 2021. SIREN incident report. URL https://medium.com/siren/siren-incident-report-264e57f16d7.
Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P., 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 530–541.
Ethereum, W., 2014. Ethereum whitepaper. Ethereum. URL: https://ethereum.org. (Accessed 07 July 2020).
Ethereum, 2018. BeautyChain integer overflow. URL https://etherscan.io/token/0xc5d105e63711398af9bbff092d4b6769c82f793d.
Ethereum, 2021. Solidity — solidity 0.8.0 documentation. URL https://docs.soliditylang.org/.
Feist, J., Grieco, G., Groce, A., 2019. Slither: A static analysis framework for smart contracts. In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain. WETSEB, IEEE, pp. 8–15.
Gao, Z., 2020. When deep learning meets smart contracts. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. pp. 1400–1402.
GitHub, 2021. GitHub API. URL https://developer.github.com/v3/.
Hwang, S., Ryu, S., 2020. Gap between theory and practice: An empirical study of security patches in solidity. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 542–553.
Jiang, B., Liu, Y., Chan, W.K., 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 259–269.
Kahn, A.B., 1962. Topological sorting of large networks. Commun. ACM 5 (11), 558–562.
Kendall, M.G., 1938. A new measure of rank correlation. Biometrika 30 (1/2), 81–93.
Lutellier, T., Pham, H.V., Pang, L., Li, Y., Wei, M., Tan, L., 2020. Coconut: Combining context-aware neural translation models using ensemble for program repair. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 101–114.
Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A., 2016. Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269.
Mueller, B., 2018. Smashing ethereum smart contracts for fun and real profit. HITB SECCONF Amsterdam 9, 54.

N.C.C. Group, 2018. Decentralized application security project. URL https://dasp.co.

Nguyen, H.A., Nguyen, A.T., Nguyen, T.T., Nguyen, T.N., Rajan, H., 2013. A study of repetitiveness of code changes in software evolution. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 180–190.

Nguyen, T.D., Pham, L.H., Sun, J., 2021. SGUARD: Towards fixing vulnerable smart contracts automatically. In: 2021 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 1215–1229.

Perez, D., Livshits, B., 2021. Smart contract vulnerabilities: Vulnerable does not imply exploited. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 1325–1341.

Pinna, A., Ibba, S., Baralla, G., Tonelli, R., Marchesi, M., 2019. A massive analysis of ethereum smart contracts empirical study and code metrics. IEEE Access 7, 78194–78213.

Ren, M., Ma, F., Yin, Z., Fu, Y., Li, H., Chang, W., Jiang, Y., 2021a. Making smart contract development more secure and easier. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1360–1370.

Ren, M., Yin, Z., Ma, F., Xu, Z., Jiang, Y., Sun, C., Li, H., Cai, Y., 2021b. Empirical evaluation of smart contract testing: What is the best choice? In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 566–579.

SmartContractSecurity, 2020. Shadowing state variables. URL https://swcregistry.io/docs/SWC-119.

Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y., 2018. Smartcheck: Static analysis of ethereum smart contracts. In: Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain. pp. 9–16.

Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., Vechev, M., 2018. Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 67–82.

Tsinghua University, 2019. Pied-piper: Revealing the backdoor threats in smart contracts. URL https://github.com/renardbebe/BackdoorDetector.

Wan, Z., Xia, X., Lo, D., Chen, J., Luo, X., Yang, X., 2021. Smart contract security: A practitioners' perspective. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering. ICSE, IEEE, pp. 1410–1422.

Wang, C., Li, Y., Chen, L., Huang, W., Zhou, Y., Xu, B., 2020. Examining the effects of developer familiarity on bug fixing. J. Syst. Softw. 169, 110667.

Wang, B., Passos, L., Xiong, Y., Czarnecki, K., Zhao, H., Zhang, W., 2013. Smartfixer: Fixing software configurations based on dynamic priorities. In: Proceedings of the 17th International Software Product Line Conference. pp. 82–90.

Wen, F., Nagy, C., Lanza, M., Bavota, G., 2022. Quick remedy commits and their impact on mining software repositories. Empir. Softw. Eng. 27 (1), 1–31.

Wen, M., Wu, R., Liu, Y., Tian, Y., Xie, X., Cheung, S.-C., Su, Z., 2019. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 326–337.

Wood, G., et al., 2014. Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper 151 (2014), 1–32.

Yang, G., Liu, K., Chen, X., Zhou, Y., Yu, C., Lin, H., 2022. CCGIR: Information retrieval-based code comment generation method for smart contracts. Knowl.-Based Syst. 237, 107858.

Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., Bairavasundaram, L., 2011. How do fixes become bugs? In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. pp. 26–36.

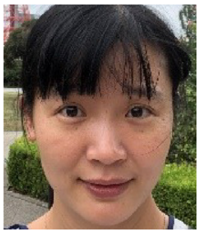
Yu, X.L., Al-Bataineh, O., Lo, D., Roychoudhury, A., 2020. Smart contract repair. ACM Trans. Software Eng. and Methodol. (TOSEM) 29 (4), 1–32.

Zhang, Y., Ma, S., Li, J., Li, K., Nepal, S., Gu, D., 2020. Smartshield: Automatic smart contract protection made easy. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 23–34.

Zhong, H., Su, Z., 2015. An empirical study on real bug fixes. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 1. IEEE, pp. 913–923.

Zou, W., Lo, D., Kochhar, P.S., Le, X.-B.D., Xia, X., Feng, Y., Chen, Z., Xu, B., 2019. Smart contract development: Challenges and opportunities. IEEE Trans. Softw. Eng. 47 (10), 2084–2106.

Yilin Wang is a postgraduate student at Sun Yat-sen University. Her research interest includes software engineering, code analysis and comprehension, and mining software repositories.



Xiangping Chen is an associate professor at Sun Yat-sen University. She got her Ph.D. degree from Peking University in 2010. Her research interest includes software engineering and mining software repositories.

Yuan Huang received a Ph.D. degree in computer science from Sun Yat-sen University in 2017. He is an associate professor at Sun Yat-sen University. He is particularly interested in software evolution and maintenance, code analysis and comprehension, and mining software repositories.

Hao-Nan Zhu is a postgraduate student at the University of California, Davis. His research interest includes software engineering, code analysis, and fault detection.

Jing Bian received a B.Sc. degree in automation, an M.Sc. degree in computational mathematics, and the Ph.D. degree in physics from Sun Yat-sen University, Guangzhou, China, in 1988, 2001, and 2006, respectively. She is currently a ViceProfessor at Sun Yat-sen University. Her current research interests include the design and analysis of algorithms, blockchain, electronic commerce, and social networks.

Zibin Zheng received a Ph.D. degree from the Chinese University of Hong Kong, in 2011. He is currently a Professor with the School of Software Engineering, Sun Yat-sen University, China. He published over 300 international journal and conference papers, including 9 ESI highly cited papers. His research interests include blockchain, artificial intelligence, and software reliability. He was a recipient of several awards, including the Top 50 Influential Papers in Blockchain of 2018, the ACM SIGSOFT Distinguished Paper Award at ICSE2010, the Best Student Paper Award at ICWS2010.