An Empirical Study on Real Bug Fixes in Smart Contracts Projects

Yilin Wang, Xiangping Chen, Yuan Huang, Hao-Nan Zhu and Jing Bian

ABSTRACT

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. 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 repairing the vulnerabilities in smart contracts developed for the Ethereum blockchain. In this paper, we conduct an empirical study on historical bug fixing versions of 46 real-world smart contracts projects from Github, providing a multi-faceted discussion. In this paper, we mainly explore the following four questions: File Type and Amount, Fix Complexity, Bug distribution, and Fix Patches. By analyzing the file type, amount, and fix complexity, we find that about 80% of the bug-related commits modified no more than one solidity source file to fix bugs. Up to 80% of bugs in solidity source files can be fixed by less than three fix actions. Modification is the mostly used fix action, which involves three lines of code on average. By using the analysis tool Mythril to detect the vulnerabilities, we find that nearly 20% of the solidity files in our dataset had or have had vulnerabilities. We finally find that the developers may not put much attention to fixing vulnerabilities reported by Mythril completely or avoid introducing them again. Because vulnerabilities that have a high repair percentage usually have a high rate to be introduced again.

Keywords: Smart contract, Bug Fix

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 (2) (3) 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’s value (4), 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 (34). Smart contracts in Ethereum are typically written using a statically-typed high-level programming language named Solidity (6). As a young language born in 2015, solidity is not safe enough and is exposed to severe vulnerabilities (9), which allow attackers to steal money or cause other damage while exploiting them. For example, the attackers exploited the Reentrancy vulnerability of DAO (decentralized autonomous organization) to steal over 3,600,000 Ether, or 60 million US Dollars in June 2016 (22). In April 2018, the attackers exploited the Integer Overflow vulnerability of the USChain BEC contract to copy tokens infinitely, causing the value of BEC tokens which is worth 900 million dollars to zero (5). 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 vulnerabilities in smart contracts developed for the Ethereum blockchain (17) (23) (24) (25) (26) (27) (28) (29). Meanwhile, (22) (27) (37) reduce the effort of repairing the vulnerabilities of smart contracts by proposing various approaches to automatically repair programs. Although most of the work has focused on detecting and repairing the vulnerabilities of smart contracts in Ethereum, the research community has limited knowledge on the naturalness of bug fixes in smart contract repositories. For instance, are bug fixes all in solidity files? How many solidity files are involved when fixing bugs? How many lines of code are involved? Are the existing vulnerabilities common and representative enough? It is of great importance to design an empirical study to answer these questions.

In this paper, we focus on understanding bug fixes on real-world smart contract projects. We conduct an empirical study on the bug-related commits from the history of 46 smart contract projects from Github to analyze the bug fixes on them, providing a multi-faceted discussion. In general, we explore the following four questions in this paper.

**File Type and Amount.** File type and amount are the types and number of files that are involved when fixing bugs in a commit. According to our findings, most bug fixes in smart contract projects only involve one Solidity file. The percentage that developers add Solidity source files to the repositories to fix bugs is less than 20%.

**Fix Complexity.** We want to find out the most used fix action and the length of code for fixing bugs. This may be helpful for developers or automatic programs to repair solidity files in future studies. The findings reveal that up to 80% of bugs in Solidity source files can be fixed by less than three fix actions. Modification is the mostly used fix action, which involves three lines of code on average. Except for modification, developers tend to add code rather than delete them when performing a fix in general. The average number of lines of code added or deleted is 5.

**Bug Distribution.** The third one is the bug distribution in solidity files. We find that nearly 20% of the 3,545 solidity files in our dataset have bugs. And the number of bugs reported by Mythril is far more than that of syntax errors. Exception state and Integer Overflow are the two most common bugs.

**Fix Patches.** Based on the bug distribution, we want to find out how many and how the developers fix these vulnerabilities reported by Mythril (42) the analysis tool to detect vulnerabilities in the evolution of the projects. We also consider how many new vulnerabilities have been introduced in this part. The results show that the developers may not put much attention to fixing vulnerabilities completely or avoid introducing them again. Because vulnerabilities that have a high repair percentage usually
have a high rate to be introduced again. 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. **Contributions.** Our contributions in this paper can be summarized as follows:

- We release a bug-fixing dataset (commit-level) of 46 real-world smart contract projects.
- We provide a multi-faceted discussion of bug fixes in real-world smart contract projects.
- We analyze the vulnerabilities and their fixes in Solidity files from the bug-fixing history of smart contract projects on Github.

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. After that, Section 4 discusses the related work. Section 5 introduces the threats to validity. Finally, Section 6 gives the conclusions of this work.

2. Methodology

2.1 Dataset

In order to analyze the bug fixes on real-world smart contract projects, we collect 50 smart contract projects from GitHub (7). The specific data collection and screening are as follows.

**Programming language.** We collect the repositories 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 repositories related to the keywords "contract" and "javascript". All the 50 repositories we choose have been manually checked to assure they are projects of Ethereum smart contracts.

**Popularity.** The number of stars of a repository is a proxy for its popularity on GitHub. Starring a repository allows GitHub users to express their appreciation for the project (8). All the 50 repositories we studied are among the repositories with the highest stars from the search results.

In addition to downloading these 50 projects from Github, we also download their commit metadata available from GitHub. Then, we identify the commits that are related to bug fixes by only keeping the commits that have keywords 'fix', 'bug', 'patch' or 'resolve' (9) and filter commits using anti-patterns (9) 'rename', 'clean up', 'refactor', 'merge', 'misspelling', 'compiler warning'. After filtering out repositories with zero bug-related commits and drop duplicates, we get 46 repositories for further study. The number of commits our dataset has is 43,354, 6,146 commits are related to bug fixing, as shown in Table 1. As shown in Figure 1, contracts-solidity is the repository with the most bug-related commits, which has 4,836 commits including 710 bug-related commits.

| Table 1 |
|---------|
| Dataset statistics |
| Overall | mean | max | min |
| commit  | 43,354 | 867  | 4,836 | 1 |
| bug-fix commit | 6,146 | 133  | 710  | 4 |

2.2. Research Questions

In order to understand the distribution of bugs and bug fixes in real-world smart contract projects.

We consider the following four research questions. The results can help developers to understand the
bug fixes during the maintenance process of smart contract projects better. Future research may leverage such knowledge to repair or design new automatic repair methods to fix more bugs.

File Type and Amount. The types and number of files involved in a bug fix may affect the programmers’ operations fix and the design of automatic program repair. There are both configuration files and source code files in smart contract projects, and the two kinds of files also involve different types of files. In this part, we explore the types and the number of files during fixing bugs in a commit. We want to find out the most modified file types and the number of solidity files that are added, deleted, and modified during fixing bugs. Recently, several automated smart contract repair approaches have been proposed (22) (27) (37). 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 automatic program repair techniques. Knowing the types of files and the number of solidity files modified when programmers fix bugs may help us to maintain smart contract projects more expertly and even build the foundation for the automatic repair of smart contracts.

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

**RQ1.** What types of files are involved when fixing bugs? To explore the file types of the files that are most modified.

**RQ2.** How many solidity files are modified during a fix? To calculate the number of modified Solidity source files in a bug fix (within a commit) in all 46 repositories.

**RQ3.** How many solidity files are necessary to be added or deleted to fix bugs? To calculate the number of solidity files added or deleted during a bug-related commit.

**Fix Complexity.** Analysis of the distribution of fix actions and the code lines is essential for us to
understand the
and design automatic smart contract repair approaches. A fix action refers to addition, modification, or deletion code. In this part, we want to explore the most used operation to fix bugs in solidity files, and the lines of code it involved. The problems and goals in this part we study are as follows:

**RQ4.** How many fix actions should be performed to fulfill a fix? To explore the complexity of fix actions during fixing bugs.

**RQ5.** Which fix action is the most used one for fixing bugs and what is the length of the code it involves? To explore the most used fix action and the length of the code statements involved during fixing bugs.

**Bug Distribution.**
As we can see, many studies (37) (44) (4) have explored the vulnerabilities of smart contracts in Ethereum. Different om the data they have studied, in this part, we try to analyze the distribution of smart contract vulnerabilities from he history of 46 smart contract repositories.

**RQ6.** How many solidity files contain bugs? To explore the bug distribution in the solidity files.

**RQ7.** What are the specific bugs? To explore the specific bugs or vulnerabilities in these solidity files.

**Fix patches.** Real bugs and their patches collected from real-world projects are critical for research in the repair smart contracts. In this aspect, we want to find out how many vulnerabilities have been fixed in the code evolution and how to fix them, and how many vulnerabilities have been introduced in the bug-fixing process. By exploring the number of the fixed vulnerabilities and newly introduced in historical versions of the projects and manually inspecting the human-written patches, we can know the detailed information of vulnerabilities fixes in smart contracts, which may lead us to do more like exploring vulnerabilities that have high fixed rate or generating program patches to repair smart contracts in the future. The problems and goals in this aspect we study are as follows:

**RQ8.** How many vulnerabilities have been fixed and how many vulnerabilities have been introduced? To explore w many vulnerabilities have been fixed and introduced in the bug-fixing evolution of the solidity files.

**RQ9.** How do developers fix these bugs? To inspect the human-written patches.

### 3. Empirical Results

In this section, we will present the results of the empirical study.
3.1. File Type and Amount

**RQ1.** What types of files are involved when fixing bugs? To find out the types of files that are involved during a bug fixing, we calculate the file types of all the files. Figure 2 shows the top 10 types of modified files in 6,146 bug-related commits of 46 Repositories. More specifically, the distribution of the top 5 repositories with the most bug-related commits is shown in Table 2. To save space, we do not present file types whose percentage is less than 1% and use the suffices of file names to denote file types to identify the type of each modified file.

As we can see from the results, except for Solidity files, JavaScript is the most modified source code file type during fixes. There may be two reasons. Firstly, most of the repositories we study are developed using Truffle (18), which is the most used development framework providing automated contract testing for blockchain. Truffle offers two different ways to test, Solidity test contracts as well as JavaScript tests. Besides, the developers use Ethereum JavaScript API Web3.js (19) to interact with a local or remote Ethereum node using HTTP, IPC, or WebSocket. There may also have bugs in these 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. For example, most Solidity compilers like Remix and Truffle use JSON files to capture the output for each compiled contract. Meanwhile, they also use them to store the compilation’s artifact needed for linking a library to the file, including the link to the libraries, the bytecode, the deployed bytecode, the gas estimation, the method identifiers, and the ABI. While the Markdown files are manuals and tutorials which are used to explain something about the repositories. The bugs in these non-source files may result in the usage of wrong data or operations by developers.

**RQ2.** How many Solidity files are modified during a fix? To answer this question, we calculate the...
number of modified Solidity files during a bug fix (within a commit) in all 46 repositories by identifying the suffix of files. Figure 3 compares the rate of solidity files, the source code files (including the solidity files), and non-source files known as configuration files modified while fixing bugs within commits. Its horizontal axes show the number of modified files and the vertical axes show the percentage of the three kinds. Figure 4 shows the distribution, while its horizontal axes show the number of modified solidity files and the vertical axes show the percentage of the corresponding bug-related commits.

![Figure 3: The percentage of Solidity Files Modified During a Fix (46 repositories)](image1)

![Figure 4: Amount of Solidity Files being Modified During a Fix](image2)

From Figure 3, we can see that among the 6,146 bug-related commits extracted from these repositories, no more than 50% of them do not involve any Solidity files, and nearly 80% of them do not involve any configuration files.

More than 40% of 6,146 bug-related commits involve one code file written by other programming languages such as JavaScript or Python, from which we can see that the distribution of bugs in smart contract projects is very complex, involving different source code files.

Further, as we can see in Figure 4, about 80% of the bug-related commits modify no more than one Solidity source file. More specifically, about 33% of the bug-related commits modify only one Solidity source file. Nearly 20% of the bug-related commits modify two or more Solidity source
files. The percentage of bugs decreases in general when the number of modified source files increases. This result indicates that the dependency among Solidity source files is similar to Java source files (10).

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

We calculate the number of solidity files added or deleted during a bug-related commit. Figure 5 shows the number of solidity files added and deleted during a fix. The horizontal axes show the number of solidity files added or deleted, while the vertical axes show the percentage of the corresponding commits.

Generally, for most of the projects we study, about 80% of the bug-related commits do not add new solidity files to fix bugs. Even if it really needs to add files, one or two files are quite enough. And almost all the fixes do not need to delete files. Since most bug fixes require only one solidity file to modify, it may be helpful enough for automated program repair to focus on modifying one solidity file. The results lead to the following finding:

**Finding.1** Most bug fixes in smart contract projects only involve one Solidity file. The percentage that developers add Solidity source files to the repositories to fix bugs is less than 20%.

### 3.2. Fix Complexity

After the discussion of the file types for modifications, in this section, we explore the complexity of bug fixing in solidity files including fix actions and lines of code.

**RQ4.** How many fix actions should be performed to fulfill a fix?

In this paper, we use the `diff` command in Linux to display the differences in the files by comparing the files line by line. Diff uses certain special symbols and instructions that are required to make two files identical. By using it, we can learn the difference between the files and how to change the file to make it match the bug-fixed file. The special symbols in `diff` correspond to the fix actions in our study. Fix actions refer to adding new lines, modifying existing lines, or deleting lines. We do not consider identifying dependencies between codes in our paper. We analyze the results of the `diff` command directly.

Figure 5 shows when fixing bugs in smart contract repositories, how many fix actions have been performed in solidity files. The horizontal axes show the number of fix actions, while the vertical axes show the percentage of the corresponding commits. As shown in Figure 6, for most repositories we study, more than 20% bugs in Solidity source files can be fixed by only one fix action and up to 70% bugs in Solidity source files can be fixed by less than three fix actions.

**RQ5.** Which fix action is the most used one for fixing bugs and what is the length of the code it involves?

We calculate the percentage of the three fix actions addition, modification, and deletion in all files.
including source code files and non-source code files. The results are shown in Figure 7, where blue, orange, and green refer to addition, deletion and modification respectively. Table 3 shows the number of fix actions and the lines of source code involved when fixing bugs in solidity files and files written by other programming languages such as JavaScript. SLOCIS means the lines of source code in solidity files and AvgSLOCIS means the average lines of source code in solidity files. While calculating the lines of source code (SLOC), we do not consider the added or deleted files. As shown in Figure 7 and Table 3, modification is the mostly used fix action, which involves 3 lines of solidity code on average. Except for modification, developers tend to add code rather than delete them when performing a fix in general. The average number of lines of code while adding or deleting is 5.

![Figure 6: Fix Actions on Solidity Files During a Fix](image)

The modification includes condition strengthening, code refactoring, and so on. The addition contains inserting new semantic features such as new variable, function, and control flow to Solidity source files, while deletion removes these. Some automatic repair programs for smart contracts have an inherent limitation: since this technique relies on random program mutations such as statement addition, deletion, and modification, it is possible to generate nonsensical patches. Although the fix can pass all the given test cases, sometimes it is hard to explain and accept in the real world. D. Kim et al. (20) presented PAR, which generated patches for automatic repair programs by learning fix patterns from existing human-written patches to avoid nonsense patches. It indicates that we can learn and use the fix patches in these human-written patches to generate program patches to repair smart contracts automatically.

| Table 3 | The Number of Fix Actions and Lines of Source Code Involved in Source Code Files |
|---------|-------------------------------------------------|
| Action  | Actions in Non-Solidity | SLOC | Actions in Solidity | SLOCIS | AvgSLOCIS |
| Add     | 14,263 | 97,511 | 6,363 | 34,604 | 5.44 |
| Delete  | 9,021 | 48,550 | 4,224 | 23,510 | 5.57 |
| Change  | 73,349 | 261,174 | 30,536 | 99,554 | 3.26 |
| Total   | 96,633 | 407,235 | 41,123 | 157,668 | 3.83 |

**Finding.2** More than 20% of bugs in solidity files can be fixed by only one fix action and up to 70%
of bugs in solidity files can be fixed by less than three fix actions. Modification is the mostly used fix action, which involves 3 lines of code on average. In general, developers tend to add code rather than delete them when performing a fix. The average number of lines of code added or deleted is 5.

3.3. Bug Distribution

In this part, we try to analyze the distribution of smart contract vulnerabilities from the historical versions of the 46 smart contract repositories. We use an existing analysis tool to detect the vulnerability of solidity files. There are two problems we need to consider before our study. One is that due to the limitations of current tools to detect vulnerabilities in solidity files, we can not analyze solidity files in smart contract projects directly. The solidity files in smart contract repositories on Github 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 tools to detect vulnerabilities in solidity files can not 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 tool to detect the solidity files with imported content.

In practice, we take 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 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 on this method, we successfully package 3,545 different solidity files.

Another is how to get solidity files from the historical versions of the 46 smart contract repositories. A commit records changes to one or more files in the branch of projects. Only using the information of the modified file cannot achieve our goal, because the path of the file may be modified. So we must revert to specific commits in a repository to get the solidity files of different versions of projects. We checkout 3,259 bug-related commits involving solidity files of these 46 projects and finally get 116,410 solidity files from these versions.

Considering the analysis of existing tools (13)(4), we use Mythril (42) to detect the vulnerabilities in these smart contracts. Mythril (42) is a tool developed by ConsenSys, that relies on concolic analysis, taint analysis, and control flow checking of the EVM bytecode to prune the search space and to look for values that allow exploiting vulnerabilities in the smart contract. According to the evaluation in (4), Mythril outperforms the other 8 tools in smartBugs by the number of detected vulnerabilities.
RQ6. How many solidity files have bugs? We divide the solidity files into three categories, solidity files without bugs, solidity files with Mythril warning, and solidity files with solc compiler errors. Once there is a Mythril warning or solc error in a version of a solidity file, it will be included within that relevant category. Solidity Files with Mythril Warning have priority over Solidity Files with solc error during our classification. We drop the duplicate files and the result is shown in Figure 8. 84 solidity files have syntax errors, and nearly 20% of the 3,545 solidity files were or have been exposed to vulnerabilities.

RQ7. What are the specific bugs? We count the types and number of vulnerabilities reported by Mythril. As shown in Figure 11, there are 14 kinds in our dataset. Table 4 shows their description. Exception state and Integer Overflow are our dataset’s two most common bugs. Exception state is acceptable in most situations. Integer Overflow is a common vulnerability in smart contracts, which can be used by hackers to attack and steal money from smart contracts(5). Solidity allows parallel external invocations using the call method (someAddress.call() or ExternalContract.someMethod()). External invocations may take over the control flow. If the callee contract does not correctly manage the global states, undesirable or incorrect states may be caused and the callee contract will be attacked(37). The DAO attack is probably the most known case, which is caused by the most famous vulnerability of external invocations Reentrancy. Attackers might leverage reentrancy attacks by
using the bug of state change after the external call or Message call to an external contract in our dataset. We also explore those syntax errors deeply and find that most of these errors are caused by the version upgrade of the solidity compiler and the developer’s carelessness. An example is shown in Listing 1.

Solidity used to allow function declarations to omit their visibilities, but from v0.5.0 it throws a syntax error if no visibility is specified. Solidity enforces developers to make a decision on declaring the visibility of functions to provide more security. The developers who don’t know the compiler patches may result in such mistakes. Besides, the developers also make some common mistakes such as forgetting to write commas, writing more semicolons, and so on.

**Finding.3** After analyzing the historical versions of the solidity files in 46 smart contract repositories, we find that nearly 20% of the 3,545 solidity files in our dataset had or have had bugs. The number of vulnerabilities reported by Mythril is far more than that of syntax errors. There are 14 kinds of vulnerabilities reported by Mythril in our dataset. Exception state and Integer Overflow are the two most common bugs.
3.4. Fix patches

In this part, we mainly talk about the vulnerabilities reported by Mythril. As we introduced in Bug Distribution, we checkout 3,259 bug-related commits involving solidity files of these 46 projects and finally get 116,410 solidity files from these versions. In this part, we use these historical versions of solidity files to determine whether the vulnerabilities are fixed or not and how they are fixed, and how many new vulnerabilities have been introduced.

We identify the vulnerabilities by ⟨repository, filename, function, code⟩. Then, by analyzing the vulnerabilities in the analysis results of the solidity files reported by Mythril, we get the results of the fixes. Once the vulnerability does not exist in the next versions of the solidity file, we consider that it has been fixed. Next, we try to explore how many vulnerabilities have been introduced during the fixes. By comparing the time of commits, we can know whether the vulnerabilities is newly introduced or not.

RQ8. How many vulnerabilities have been fixed and how many vulnerabilities have been introduced? The number of vulnerabilities that have been fixed is shown in Figure 10, while its horizontal axes show the abbreviation of vulnerabilities and the vertical axes show the number of vulnerabilities and non-fixed vulnerabilities.

Figure 11 shows the number of vulnerabilities have been introduced in the bug-fixing process.

In our dataset, Integer Overflow has the highest repair percentage, which is 60.1%. But it is also the type of vulnerability that has been most introduced in the fixes. The second one is Use of tx.origin with 60.0%. It is worth noting that the number of vulnerabilities of Use of tx.origin which are newly introduced is zero. This may mean that such vulnerability is more easy to fix or the developers pay more attention to this vulnerability. The third one is Exception State with 59.9%. Exception States also have many newly introduced vulnerabilities. Nearly 50% percent of Message call to external contract and State change after external call have been fixed. Both Unchecked CALL return value and Transaction order dependence have more than 40% repair rate. Unchecked SUICIDE doesn’t have a bug-fixing instance in our dataset, which has only one bug instance.

By comparing Figure 10 and Figure 11, we can see that the trends between fixed and introduced
vulnerabilities are generally similar. It shows that the developers may not put much attention to fixing them completely or avoid introducing them again. Although there are serious smart contract security incidents (22) (5) that are exploited by the vulnerabilities like Integer Overflow in smart contracts. Maybe these vulnerabilities are common in smart contracts, and the developers consider that is rare and hard to exploit them. So they do not pay much attention to completely fixing the vulnerabilities. We should attach great importance to the vulnerabilities with a high repair rate and low introduction rate such as Use of tx.origin. It shows that the developers try to fix them as completely as possible.

| Name                              | Description                                                                 |
|-----------------------------------|-----------------------------------------------------------------------------|
| Exception State                   | Exception State can be caused by type errors, division by zero, out-of-bounds array access, or assert violations. |
| Integer Overflow                  | It means a calculation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value. The attackers can use this vulnerability to cause huge economic losses. |
| Integer Underflow                 | It means that subtracting one value from another, such that the result is less than the minimum allowable integer value, which produces a value that is not equal to the correct result (77). |
| State Change After the External Call | External contracts can take over the control flow. A malicious contract calls back into the calling contract before an internal state change is performed. This may cause undesirable or incorrect states. |
| Message call to an External Contract | It means that the contract executes a message call to another contract. The called contract may not be trusted or incorrectly execute user-supplied code. |
| Transaction order dependence      | Transaction order dependence happens when the user of a smart contract assumes a particular state of a contract, which may not exist when the transaction is processed potentially leading to malicious behavior (32). |
| Multiple Calls                    | Multiple Calls is caused by multiple sends in one transaction, which may fail accidentally or deliberately. |
| Unchecked CALL Return Value       | It means that the return value of a message call is not checked. As a result, if the call fails accidentally or an attacker forces the call to fail, this may cause unexpected behavior in the subsequent program logic (43). |
| DELEGATECALL to a User-supplied Address | It means that the contract delegates execution to a contract address obtained from calldataload. The called contract may get unrestricted access to this contract’s state. |
| Call Data Forwarded with delegate-call() | It means that the contract forwards its call data via the low-level function delegatecall in its function. The callee contract may have access to the storage of the calling contract. |
| Dependence on Predictable Environment Variables | Variables like block.coinbase, block.gaslimit, block.timestamp, and block.number are predictable, which should not influence a computation that results in no calls that send Ether. |
| Unchecked SUICIDE                 | The developer does not restrict or incorrectly restricts access to the SUICIDE instruction. It seems that the SUICIDE instruction can be called without restrictions, which may be used by attackers to send Ether. |
| Use of tx.origin                  | tx.origin is a global variable in Solidity that returns the address of the account that sent the transaction. Contracts that use the tx.origin to authorize users are vulnerable to phishing attacks. |
| Ether Send                         | The developer does not restrict or incorrectly restricts functions’ access to the send Ether. It seems that the functions can be called without restrictions. |

RQ9. How do developers fix these bugs? In this part, we manually inspect the human-written patches from the fixed vulnerabilities in the above question. During our study, we find that these fixed instances can be roughly divided into two categories: One is that the original vulnerability is modified into another one. In this way, the older one is considered fixed and the new one is considered newly introduced. Another is that the vulnerability is completely fixed. Four specific repair examples are as follows, from which we may learn how people fix them.
actually in smart contract projects. Unchecked CALL Return Value There are 46 vulnerabilities of Unchecked CALL Return Value in our dataset. 10 vulnerabilities have not been fixed. 31 vulnerabilities are inter-converted. We identify the vulnerabilities by \{ repository, filename, function, code \}, so the modification of the code and function let us identify this vulnerability as brand new. By checking the change history of these vulnerabilities, we find that there are inter-converted among them. Then, we explore the 5 vulnerabilities that have been completely fixed. As for Unchecked CALL return value, we should make sure to check the return value. In the 5 fixed instances, 2 is fixed by using the require statement. An example is shown in Listing 2.

**Figure 10:** The Number of Fixed and Non-fixed Vulnerabilities. For each type: MC means Multiple Calls, EPS means Exception State, IO means Integer Overflow, IU means Integer Underflow, DPEV means Dependence on Predictable Environment Variables, MCEC means Message call to external contract, ETS means Ether Send, UCRV means Unchecked CALL Return Value, TOD means Transaction Order Dependence, US means Unchecked SUICIDE, SCAEC means State Change after External Call, CAFWD means Call Data Forwarded with delegatecall(), DTAUA means DELEGATECALL to a User-supplied Address, TX means Use of tx.origin.

Dependence on Predictable Environment Variables Dependence on predictable environment variables is caused by using predictable variables like block.timestamp, block.gaslimit, and so on. There are 22 vulnerabilities of Dependence on predictable environment variables in our dataset. 4 vulnerabilities have not been fixed. 12 vulnerabilities are inter-converted. We explore the 6 vulnerabilities that have been completely fixed. 33.3% of the fixed instances delete the code reporting vulnerabilities. A fixed example is shown in Listing 4, where non-fixed code statements
In the original source code, the developers use `block.timestamp` to modify the behavior of functions, which is deleted in the fixed code. The developers decide not to use the predictable variable to repair such bugs. As is shown in Listing 3, Integer overflow and Integer underflow is mostly fixed by using vetted safe math libraries for arithmetic operations. As for Call data forwarded with `delegatecall()`, `DELEGATECALL` to a user-supplied address, `tx.origin`, and `Ether Send`, we inspect how the developers fixed them, but we can not get the fix logic behind them. In two fixed instances, the developers just change the compile version of solidity. Then this vulnerability disappears. It indicates that we need more fixed instances or to change our tool to learn more about them. Most vulnerabilities of Multiple Calls, Transaction order dependence, Message call to external contract, and State change after external call actually do not completely fix in our dataset. They are converted into the same kind of vulnerability by different code.

**Finding.4** The developers may not put much attention to fixing vulnerabilities completely or avoid introducing them again. Because vulnerabilities that have a high repair percentage usually have a high rate to be introduced again.

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 Fixing.** Nguyen et al (28) present a study of the repetitiveness of code changes in the evolution of 2,841 Java projects. Their results show that repetitiveness of changes could be very high at small sizes and decreases exponentially as size increases. What’s more, repetitiveness is higher and more stable in crossproject settings than in within-project one. Fixing changes repeat similarly to general changes. Yin et al (39) 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. Zhong et al (10) 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 et al (40) 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 (45) 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 (41) 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 (38) 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 vulnerabilities in smart contracts. To alleviate the problem of insecure smart contracts, researchers have developed various analysis tools and verification frameworks to detect vulnerabilities. Oyente (23) is a symbolic execution tool to find potential security bugs in smart contracts on the Ethereum system. Mythril (29) uses static analysis, taint analysis, and concolic execution to detect a variety of security vulnerabilities in smart contracts. ContractFuzzer (35) is a fuzzing framework to detect the vulnerabilities of Ethereum smart contracts. Securify (30) converts contract bytecode to Datalog and extracts semantic facts. Then it transforms the vulnerabilities into a series of compliance and violation patterns to search for the unsafe patterns in smart contracts. SmartCheck (31) checks XML-based intermediate representation of Solidity source code against XPath patterns to detect vulnerability (32). Pied-Piper (33) is a static analysis tool that constructs CFG based on bytecode and extracts semantic facts to detect potential backdoors hidden in smart contracts (32). SmartEmbed (34) is a deep learning-based approach for detecting bugs in smart contracts.

To reduce the effort of repairing the vulnerabilities of smart contracts, various approaches have been proposed to automatically repair programs. Yu et al (22) 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 vulnerabilities. Nguyen et al (27) and Zhang et al (36) propose approaches by analyzing the bytecode to fix potentially vulnerable smart contracts automatically. Nguyen et al (27) 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 vulnerabilities. Then it applies a specific fixing pattern for each type of vulnerability on the source code. Zhang et al (36) extracts bytecode level semantic information and utilizes them to transform insecure contracts into secure ones. Our study shows the distribution of bugs in real-world smart contract projects and build the foundation for the automatic repair of smart contracts.

Empirical Study on Smart Contract Chen et al (37) 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 (44) 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 Contract and Blockchain and to the knowledge of the main characteristics of contracts written in Solidity. Wan et al (21) performed a mixture of qualitative and quantitative studies with software practitioners with 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 et al (25) focuses on finding how many of the vulnerable contracts have actually 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 et al (26) conducted 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 (4) 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 (24) 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 of bug fixes in real-world smart contract projects and analyzes the vulnerabilities and their fixes in solidity files from the bug-fixing history of smart contract projects.

5. Threats to Validity

Threat to internal validity is related to the tool 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 (25). In this paper, we use the static analysis tool Mythril to analyze smart contracts. Mythril was found to be the most accurate (46) to find the bugs or vulnerabilities in smart contracts. A potential threat to the internal validity is related to the fact that we only use Mythril to detect the bugs in solidity files. Although Mythril was found to be the most accurate, it is not powerful enough to replace all the tools. It is also important to reduce the false positives reported by it. Our future research agenda is to combine dynamic and static methods to detect bugs. Threat to external validity is related to the commits that are extracted only from open source smart contract projects. Our results may not generalize to commercially developed projects or projects using different programming languages.
6. Conclusion

Despite numerous efforts in detecting and repairing the vulnerabilities in smart contracts, little is known about bug fixing in smart contract repositories. In this paper, we provide a multi-faceted discussion of bug fixes in real-world smart contract projects. We conduct an empirical study to explore the file type and amount, fix complexity, bug distribution and fix patches in the bug-related commits from the history of 46 smart contract projects. We finally distill 4 findings which we believe are useful for the study in this direction.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Ethereum project yellow paper, vol. 151, pp. 1–32, 2014.

[3] E. Foundation, "Ethereum’s white paper," 2014. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Pape

[4] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 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 (ICSE ’20). Association for Computing Machinery, New York, NY, USA, 530–541. https://doi.org/10.1145/3377811.3380364

[5] BeautyChain integer overflow. 2018. https://etherscan.io/token/0xc5d105e63711398af9bbff092d4b6769c82f793d

[6] "Solidity — solidity 0.8.0 documentation." https://docs.soliditylang.org/en/v0.8.0/. Accessed: 2021-08-10.

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

[8] Wen, F., Nagy, C., Lanza, M. et al Quick remedy commits and their impact on mining software repositories. Empir Software Eng 27, 14 (2022). https://doi.org/10.1007/s10664-021-10051-z

[9] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining contextaware neural translation models using ensemble for program repair. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020). Association for Computing Machinery, New York, NY, USA, 101–114. https://doi.org/10.1145/3395363.3397369

[10] H. Zhong and Z. Su, "An Empirical Study on Real Bug Fixes," 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, 2015, pp. 913-923, doi: 10.1109/ICSE.2015.101.

[11] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. Commun. ACM 62, 12 (2019).

[12] Xiao Liang Yu, Omar Al-Bataineh, David Lo, and Abhik Roychoudhury. 2020. Smart Contract Repair. ACM Trans. Softw. Eng. Methodol. 29, 4, Article 27 (October 2020), 32 pages.
[13] Vaccà, Annalisa, Andrea Di Sorbo, Corrado Aaron Visaggio and Gerardo Canfora. “A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges.” J. Syst. Softw. 174 (2021): 110891.

[14] A. Whitaker, R. S. Cox, and S. D. Gribble, ”Configuration debugging as search: finding the needle in the haystack,” in Proceedings of the Sixth Symposium on Operating Systems Design and Implementation, 2004, pp. 77–90.

[15] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki. Generating range fixes for software configuration. In Proc. 34th ICSE, pages 58–68, 2012.

[16] S. Zhou, X. Liu, S. Li, W. Dong, X. Liao, and Y. Xiong, ”Confmapper: Automated variable finding for configuration items in source code,” in Proceedings of the 2nd International Conference on Software Quality, Reliability and SecurityCompanion, 2016, pp. 228–235.

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

[18] Truffle.https://trufflesuite.com/ [19] https://web3js.readthedocs.io/en/v1.5.2/ [20] D. Kim, J. Nam, J. Song, and S. Kim, ”Automatic patch generation learned from human-written patches,” in 2013 35th International Conference on Software Engineering (ICSE). IEEE, 2013, pp. 802–811.

[21] Z. Wan, X. Xia, D. Lo, J. Chen, X. Luo and X. Yang, ”Smart Contract Security: A Practitioners’ Perspective,” 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 1410-1422.

[22] TheDAO smart contract. http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413code.

[23] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security. 254–269 [24] W. Zou et al, ”Smart Contract Development: Challenges and Opportunities,” in IEEE Transactions on Software Engineering, vol. 47, no. 10, pp. Oct. 2021.

[25] Perez, Daniel and Benjamin Livshits. “Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited.” USENIX Security Symposium (2021).

[26] Sungjae Hwang and Sukyoung Ryu. 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 (ICSE ’20). Association for Computing Machinery, New York, NY, USA, 542–553. https://doi.org/10.1145/3377811.3380424 [27] T. D. Nguyen, L. H. Pham and J. Sun, ”SGUARD: Towards Fixing Vulnerable Smart Contracts Automatically,” 2021 IEEE Symposium
[28] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen and H. Rajan, "A study of repetitiveness of code changes in software evolution," 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2013, pp. 180-190, doi: 10.1109/ASE.2013.6693078.

[29] ConsenSys. 2019. Mythril. https://github.com/ConsenSys/mythril-
classic.

[30] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS ’18).

Association for Computing Machinery, New York, NY, USA, 67–82.

[31] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko and Y. Alexandrov, "SmartCheck: Static Analysis of Ethereum Smart Contracts," 2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), 2018, pp. 9-16.

[32] Meng Ren, Fuchen Ma, Zijing Yin, Ying Fu, Huizhong Li, Wanli Chang, and Yu Jiang. 2021. 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 (ESEC/FSE 2021). Association for Computing Machinery, New York, NY, USA, 1360–1370.

[33] Tsinghua University. 2019. Pied-Piper: Revealing the Backdoor Threats in Smart Contracts. https://github.com/renardbebe/BackdoorDetector.

[34] Zhipeng Gao. 2020. When deep learning meets smart contracts. Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. Association for Computing Machinery, New York, NY, USA, 1400–1402.

[35] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018). Association for Computing Machinery, New York, NY, USA, 259–269.

[36] Y. Zhang, S. Ma, J. Li, K. Li, S. Nepal and D. Gu, "SMARTSHIELD: Automatic Smart Contract Protection Made Easy," 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020, pp. 23-34.

[37] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo and T. Chen, "Defining Smart Contract Defects on Ethereum," in IEEE Transactions on Software Engineering, vol. 48, no. 1, pp. Jan. 2022, doi:
[38] Chuanqi Wang, Yanhui Li, Lin Chen, Wenchin Huang, Yuming Zhou, Baowen Xu, Examining the effects of developer familiarity on bug fixing. Journal of Systems and Software, Volume 169, 2020, 110667, ISSN 0164-1212, https://doi.org/10.1016/j.jss.2020.110667.

[39] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How do fixes become bugs? In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE ’11). Association for Computing Machinery, New York, NY, USA, 26–36.

[40] E. C. Campos and M. d. A. Maia, "Common Bug-Fix Patterns: A Large-Scale Observational Study," 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2017, pp. 404-413, doi: 10.1109/ESEM.2017.55.

[41] Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 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 (ESEC/FSE 2019). Association for Computing Machinery, New York, NY, USA, 326–337.

[42] Bernhard Mueller. 2018. Smashing ethereum smart contracts for fun and real profit. In 9th Annual HITB Security Conference (HITBSecConf). HITB, Amsterdam, Netherlands, 54.

[43] https://swcregistry.io/docs/SWC-104 [44] A. Pinna, S. Ibbá, G. Baralla, R. Tonelli and M. Marchesi, "A Massive Analysis of Ethereum Smart Contracts Empirical Study and Code Metrics," in IEEE Access, vol. 7, pp. 78194-78213, 2019, doi: 10.1109/ACCESS.2019.2921936.

[45] Bernardi M L, Canfora G, Di Lucca G A, et al The relation between developers’ communication and fix-Inducing changes: An empirical study[J]. Journal of Systems and Software, 2018, 140: 111-125.

[46] Reza M. Parizi, Ali Dehghantanha, Kim-Kwang Raymond Choo, and Amritraj Singh. 2018. 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 (CASCON ’18). IBM Corp., USA, 103–113.

[47] CWE-191: Integer Underflow. https://cwe.mitre.org/data/definitions/191.html [48] CWE-190: Integer Onderflow. https://cwe.mitre.org/data/definitions/190.html