Empirical Software Engineering (2025) 30:141
https://doi.org/10.1007/s10664-025-10685-3

On state reverting in solidity smart contracts: Developer practices, fault categorization, and tool evaluation

Lu Liu[1] · Lili Wei[2] · Wuqi Zhang[3] · Shuqing Li[4] · Yifan Zhou[5] · Yepang Liu[5] · Shing-Chi Cheung[3] · Michael R. Lyu[4]

Accepted: 3 June 2025
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2025

Abstract

Smart contracts are computer programs deployed on blockchains to facilitate transactions. A critical aspect of smart contract security is the use of state-reverting statements (e.g., `require`, `if...revert`, `if...throw`). These statements protect transactions from abnormal behaviors or malicious attacks by reverting a contract to its previous state when certain input constraints or security properties are violated. While essential, the correct use of these state-reverting (SR) statements is nontrivial. Improper use can lead to security vulnerabilities, resulting in substantial financial losses or other severe consequences. It is, therefore, highly important to understand developers' practices of state reverting in smart contracts and the common mistakes they make. To achieve this goal, we conduct the first comprehensive empirical study on the use of SR statements and their related faults in Solidity smart contracts. First, we analyze the prevalence and purposes of SR statements in 21,414 verified contracts from popular decentralized applications (dapps) and manually examine 381 SR statements, leading to a taxonomy of their uses. Second, we collect 320 real-world state-reverting faults (SR faults) from open-source projects on GitHub and audit reports on Code4rena. We categorize the SR faults into 17 types and summarize 12 distinct fixing strategies. This knowledge can help researchers and practitioners to better understand the common usages of SR statements and learn how to prevent or cope with SR faults. Lastly, the variety of SR fault types and the presence of high-risk issues highlight the need for automated tools to identify and mitigate these faults. This further motivates us to assess the SR fault detection performance of state-of-the-art security analyzers, with the aim of understanding their capability and identifying their deficiencies. Via evaluating 12 representative tools on a benchmark comprising 243 contracts with six types of SR faults and the corresponding patched versions, we observe that existing tools exhibit limited capabilities in detecting SR faults (the average detection rate is 14.4%). This result underscores the need for more advanced security analysis tools specifically tailored for SR faults. To facilitate the development of such tools, we further provide a comprehensive analysis of three common limitations of existing tools.

Keywords Smart contract · Empirical study · Security · Vulnerability detection

Communicated by: Federica Sarro

Extended author information available on the last page of the article

Published online: 17 July 2025

1 Introduction

Smart contracts are computer programs stored on blockchains to execute transactions. Ethereum, the largest blockchain platform supporting smart contracts, processes millions of transactions on a daily basis (Ethereum daily transactions chart 2024). A blockchain transaction involves either transferring crypto assets between addresses or interacting with a deployed smart contract, resulting in a state change. To protect transactions from abnormal behaviors or malicious attacks, the states of the underlying smart contracts often need to be reverted, when certain input constraints or security properties are violated (Solidity error handling: Assert, require, revert and exceptions 2024). For example, Solidity, one of the most popular programming languages for implementing smart contracts, provides developers with *state-reverting* statements (SR statements) (Solidity error handling: Assert, require, revert and exceptions 2024; Ethereum improvement proposal eip-140: Revert instruction 2024; Liu et al. 2021; Liao et al. 2023) that revert all state changes when transactions fail. These statements include `require`, `if...revert`, and `if...throw`, as demonstrated in Listing 1. While all three statements serve the purpose of reverting states, `require` and `if...revert` also return unused gas to transaction senders.

```
1 if (msg.sender != owner) {revert();}
2 require (msg.sender == owner);
3 if (msg.sender != owner) {throw;}
```

Listing 1: Examples of SR statements

Despite the widespread usage of smart contracts and the importance of SR statements in handling state reverting when transactions fail, there is a lack of comprehensive study on how developers use SR statements in practice. To bridge this research gap, we conduct a large-scale empirical study. As shown in Fig. 1, we analyze 21,414 real-world Solidity contracts from 682 decentralized applications (dapps) to investigate two research questions:

- **RQ1 (Prevalence):** Are SR statements commonly used in Solidity smart contracts?
- **RQ2 (Purpose):** What are the major purposes of using SR statements in smart contracts?

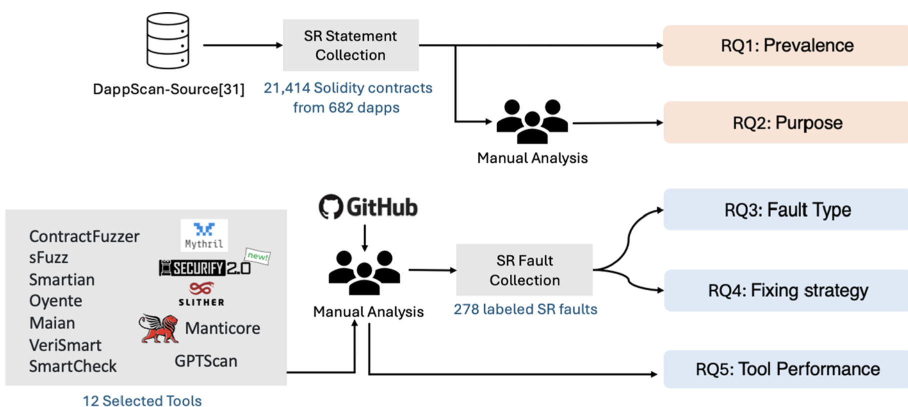


Fig. 1 Overview of the methodology

To answer RQ1, we measure the code density of SR statements in smart contracts and compared it with that of general-purpose `if` statements by analyzing the 21,414 Solidity contract files. To answer RQ2, we manually build a taxonomy of the purposes of SR statements following an inductive coding process (Seaman 1999). Surprisingly, we find that SR statements appear even more frequently than general-purpose `if` statements: on average, there is one SR statement every 42.1 lines in a Solidity contract file; comparatively, there is one `if` statement every 54.5 lines. We also find that SR statements are commonly used to perform 10 different types of authority verification and validity checks, many of which involve security-critical constraints. It is, therefore, crucial to ensure the proper use of such statements, as their incorrect usage can pose serious security threats to smart contracts.

Existing research related to state reverting in smart contracts primarily focuses on two areas: 1) examining how attackers leverage faults in smart contracts to trigger transaction reverting when outcomes are unexpected (Liao et al. 2023; He et al. 2021; Chen et al. 2022) and 2) investigating gas-related state-reverting issues (Grech et al. 2018; Ghaleb et al. 2022; Nassirzadeh et al. 2022), where running out of gas during transaction execution leads to transaction abortion, resulting in security consequences. The types of faults developers would make in using SR statements remains unknown. Without such knowledge, one cannot design automatic tools to effectively identify the inappropriate uses of SR statements or formulate good practices to help smart contract developers. This motivates us to conduct a comprehensive study of smart contract faults that stem from improper SR statement usage, which we refer to as *state-reverting faults* (SR faults) in this paper. Specifically, we investigate the following research questions:

- **RQ3 (Fault Types):** What are the common types of SR faults? What is the distribution of such faults across different types, and what are the potential security impacts of these faults?
- **RQ4 (Fixing Strategies):** How do developers fix SR faults in real-world smart contracts? Are there common fixing strategies?

To answer RQ3 and RQ4, as shown in Fig. 1, we construct a dataset of 278 real-world SR faults from open-source projects on GitHub (Github 2024). These faults are extracted from the commit histories of the 1,000 most-starred smart contract projects on GitHub. Each fault links to a commit where developers fixed the issue by adding, deleting, or modifying SR statements. Such a dataset construction process ensures that our collected issues are genuine SR faults of concern to developers. By investigating the 278 faults, we find that SR faults in smart contracts are diverse and can be categorized into 17 distinct types. To understand the fault types' relevance and potential security impact, we extract 42 SR faults of our identified types from audit reports published from Oct 2023 to Oct 2024 on Code4rena (Code4rena 2024), a leading smart contract auditing platform. Each of these 42 faults is associated with a severity tag of "high risk" or "medium risk" and includes a clear mitigation strategy. Analysis of the 42 faults reveals that our identified SR fault types are mostly concerning. Some SR faults, such as excessive slippage and unauthorized access, are high-risk faults that can pose significant threats to asset security. To answer RQ4, we further analyze our collected faults and distill 12 common fixing strategies for SR faults. The findings of RQs 3-4 can effectively guide developers in identifying, diagnosing, and mitigating SR faults. The diver-

sity of SR faults also suggests the necessity of developing automated tools to help identify and fix SR faults, especially those that impact asset security.

Lastly, given the prevalence and variety of SR faults, we are intrigued to know whether existing tools can effectively detect common SR faults. This can shed light on future research. As such, we conduct another study to investigate the last research question in our work:

– **RQ5 (Fault Detection Capability of Existing Tools):** How effective are existing smart contract security analysis tools in detecting SR faults?

To answer RQ5, we first identify 12 state-of-the-art smart contract security analyzers in different approaches, including fuzzing, symbolic execution, static analysis, and large language models. These tools can collectively detect six of the 17 SR fault types we identified. In addition, by evaluating the 12 tools on 243 SR faults of the six types, we observe that the existing tools exhibit a high false negative rate, failing to detect 85.6% of the 243 SR faults in our benchmark. We further explore the reasons behind existing tools' poor performance and summarize three common limitations of these tools. Our findings can be used to facilitate the design of more advanced security analysis tools specifically tailored for SR faults in smart contracts.

Contributions To summarize, our work makes the following three major contributions:

– To the best of our knowledge, we conduct the first empirical study on the use of SR statements in real-world smart contracts at the source-code level. Our findings can facilitate further research in the area of smart contract quality assurance and provide practical guidance to smart contract developers on the appropriate use of SR statements.
– We construct a large dataset of 320 fixed SR faults in real-world smart contracts and conduct the first in-depth study of these faults. By categorizing the SR faults into 17 distinct types, we offer a structured benchmark with rich information to help practitioners and researchers to understand SR faults and evaluate the efficacy of security analysis tools. Furthermore, we derive 12 strategies commonly adopted to address SR faults, providing practical guidance to practitioners to help them mitigate SR faults to enhance contract security and to researchers to help them design automatic SR fault repair techniques.
– We evaluate 12 state-of-the-art security analysis tools, assessing their capabilities in detecting SR faults. Our analysis on the tools' fault type coverage and fault detection capability can deliver practical recommendations for tool designers and developers on how to strengthen the fault detection mechanisms of a tool so as to improve smart contract security assurance.

While our previous work (Liu et al. 2021) provided initial insights into the use of SR statements in smart contracts based on 3,866 dapp contracts and 270 template contracts, it did not address the faults arising from their improper use or evaluate the effectiveness of existing security tools in detecting such faults. This extended study addresses these critical gaps. By analyzing a larger, more recent dataset of 21,414 contracts from 682 decentralized applications (dapps), we capture evolving practices in smart contract development, including

the influence of modular design and external libraries on SR statement usage. Unlike the original study, which focused solely on the prevalence and purposes of SR statements, this extension systematically investigates SR faults, categorizing 320 real-world instances into 17 distinct types and analyzing their security impacts. Furthermore, we evaluate the effectiveness of 12 state-of-the-art security analysis tools in detecting these faults, revealing critical gaps in current tool capabilities (average detection rate of 14.4%). This comprehensive analysis is vital for advancing the understanding of SR statements and faults, offering developers practical guidance and highlighting areas for future research in automated fault detection and repair. Specifically, this journal version extends the preliminary study in the following aspects:

- It re-examines the prevalence and purposes of SR statements using a larger, more recent dataset of 21,414 contracts from 682 dapps (Section 3).
- It expands the taxonomy of SR statement purposes by adding five subcategories to reflect the evolving usage of SR statements: *Boolean State Variable Check*, *Function Return Value Check*, *Element Existence Check*, *Type Validation Check*, and *Consistency Check*. (Section 4).
- It extends the answers to two research questions (RQ1 and RQ2) in the preliminary study based on further empirical findings.
- While the preliminary study investigated only the possible security impacts of omitting SR statements without systematically analyzing SR faults, this journal version **systematically analyzes the SR faults in smart contracts** by studying two important research questions (RQ3 in Section 5 and RQ4 in Section 6). In particular, it identifies 17 distinct SR fault types and 12 commonly adopted fixing strategies, thereby establishing **the most comprehensive taxonomy of SR faults to date**.
- To facilitate future research on automated tools for SR fault detection, this journal version additionally **evaluates 12 state-of-the-art security analysis tools, assessing their capabilities and limitations in detecting SR faults** (RQ5 in Section 7).**Data Availability.** To facilitate future research, we have made all our artifacts available on GitHub (The material for this study 2024).

2 Background and Related Work

In this section, we provide a brief overview of the background knowledge related to our research. We also summarize the related work to position our study within the literature.

2.1 Smart Contracts, SR statements, and SR Faults

2.1.1 Smart Contracts & Dapps

Smart contracts are computer programs running on blockchains like Ethereum (Ethereum yellowpaper 2024). The execution of smart contracts does not rely on a trusted third party and is fully decentralized. Dapps are decentralized applications that can offer end users various functionalities. The core logic of dapps is supported by smart contracts to meet the requirements of applications. Solidity (Solidity documentation 2024) is the most popular

high-level programming language to implement smart contracts. In this paper, we focus on the smart contracts written in Solidity.

2.1.2 Transactions

Transactions are the fundamental operations within blockchain networks, representing state changes that are securely recorded on the distributed ledger (Ethereum transactions 2024). In the context of blockchains like Ethereum, a transaction may involve transferring cryptocurrency, invoking a function within a smart contract, deploying a new contract to the network, and so on. Each transaction is cryptographically signed by the sender to ensure authenticity and is processed by the network nodes through consensus mechanisms to validate its correctness. Once validated, the transaction is included in a block and becomes immutable, ensuring a tamper-proof history of operations.

2.1.3 State-Reverting (SR) Statements and State-Reverting (SR) Faults

State-Reverting (SR) Statements The official Solidity documentation (Solidity error handling: Assert, require, revert and exceptions 2024) suggests using the following four types of statements for error handling and state reversion: `require`, `if...revert`, `assert`, and `if...throw`. Upon the occurrence of certain erroneous conditions, these statements will throw an exception and revert the blockchain and the smart contract to the state prior to the execution of the contract. The four error-handling statements can be further divided into the following two categories:

- **SR statements** refer to the `require`, `if...revert`, and `if...throw` statements that are used to check for conditions (e.g., on inputs) that should be satisfied. Before version 0.4.10, Solidity provides the `if...throw` statement for reverting state changes upon the occurrence of erroneous conditions. As the language evolves, there are two more alternatives, namely, `require` and `if...revert`, to replace `if...throw` since Solidity 0.4.10. The `if...throw` statement was officially deprecated in Solidity 0.4.13. These statements can all trigger state reverting when erroneous conditions occur. The only difference between `if...throw` and its two replacements is that `if...throw` will use up all remaining gas when errors occur, while the two replacements will refund the remaining gas to the transaction sender.
- **The assertion statement** `assert` is typically used to test for internal errors, which are not supposed to exist in well-written code. If a specified assertion is violated, it means that the contract likely has a bug, which needs to be fixed.

In our study, we focus on SR statements. Since `if...throw` statement is already deprecated, we mainly investigate the use of `require` and `if...revert` statements in real-world smart contracts. In the remainder of this paper, SR statements refer to `require` and `if...revert` statements unless otherwise specified.

State-Reverting (SR) Faults Using SR statements incorrectly can cause transactions to roll back improperly and revert all state changes in a smart contract. This can result in various

issues, ranging from inconsistent contract states to security vulnerabilities. We refer to these faults as *State-Reverting faults* (SR faults) (Liao et al. 2023) in our work.

2.2 Related Work

2.2.1 Empirical Studies on Smart Contract Security

Given the popularity of smart contracts, researchers have recently conducted multiple empirical studies of smart contract security. These studies include various systematic reviews of existing vulnerabilities and attacks on Ethereum smart contracts. For example, Atzei et al. (2017) summarize 12 types of vulnerabilities on Ethereum smart contracts and group them into three categories, namely, Solidity, EVM bytecode, and blockchain, according to the level where they are introduced. They also discuss nine attacks and three defenses of Ethereum smart contracts. Chen et al. (2020) present a survey of smart contract vulnerabilities, attacks, and defenses. They study 40 types of smart contract vulnerabilities and systematize their root causes. They include 29 types of attacks, correlate them with vulnerabilities, and summarize their consequences. They also systematize 51 types of defenses and present an in-depth analysis. Chen et al. (2020) conduct an empirical study on defining smart contract defects. They collect smart-contract-related posts from Stack Exchange (Stackexchange website 2024) and define 20 kinds of contract defects from them. Conducting an online survey and manually labeling some contracts, they study the contract defects from five perspectives: security, availability, performance, maintainability, and reusability. Zhang et al. (2023) conduct a study on 516 exploitable smart contract vulnerabilities that can lead to direct financial losses. They examine the vulnerabilities' root causes, distributions, auditing difficulties, consequences, and repair strategies. Additionally, they abstract bug models for each vulnerability type to facilitate code auditing and future research on automated tools.

Also, there are works concerning evaluating the capabilities of smart contract testing tools. Harz and Knottenbelt (2018) examine ten smart contract verification tools and introduce their verification approach, level of automation, coverage, and supported languages. Di Angelo and Salzer (2019) present a systematic survey on 27 tools for analyzing Ethereum smart contracts. They investigate the 27 tools from several aspects: availability, maturity level, methods employed, and detection of security issues. Durieux et al. (2020) present an empirical evaluation of nine state-of-the-art analysis tools using two smart contract datasets. They find that these tools detect only 42% vulnerabilities. At the same time, 93% Ethereum contracts are tagged as vulnerable, which might contain many false positives. Ghaleb and Pattabiraman (2020) introduce SolidiFI, a framework that injects synthetic vulnerabilities into smart contracts to assess security analysis tools' precision and recall. After evaluating six widely-used static analysis tools, they find that all tools report false positives and fail to detect certain vulnerabilities despite claiming they can detect such bugs. Li et al. (2024) propose a fine-grained taxonomy that includes 45 smart contract vulnerability types. Based on the taxonomy, they construct a benchmark of 788 smart contracts, uncovering 10,394 vulnerabilities. They also evaluate 8 state-of-the-art tools based on the benchmark. Additionally, Chaliasos et al. (2024) evaluate the security tools for smart contracts and DeFi applications from a practitioner's perspective, highlighting gaps between tool capabilities and developer needs. While these works provide a broad assessment of tool usability, our study delves into a specific, underexplored area: the use and misuse of state-reverting (SR)

statements in Solidity contracts. By systematically analyzing SR statements in real contracts and categorizing SR faults, we address a critical subset of vulnerabilities overlooked in existing tool evaluations and offer insights that can facilitate the development of more precise and powerful tools for improving the security of smart contracts.

In our previous work (Liu et al. 2021), we conducted a preliminary study on SR statements by investigating 3,866 dapp contracts and 270 template contracts. The primary purpose is to analyze the prevalence and purposes of SR statements. This journal version substantially extends our previous work by leveraging a larger dataset of 21,414 contracts from 682 dapps, which allows for a more comprehensive analysis of SR statement usage in modern smart contracts. To reflect the evolving use of SR statements, we expand the taxonomy of their purposes with five new subcategories (i.e., Boolean State Variable Check, Function Return Value Check, Element Existence Check, Type Validation Check, and Consistency Check). In addition, in the previous work, we focused on studying the use of SR statements but did not analyze the faults induced by the misuse of these statements. In this paper, we perform an in-depth analysis of SR faults, categorizing 320 real-world instances into 17 types and assessing their security impacts. We also evaluate 12 state-of-the-art security tools, addressing an important research gap in understanding the existing tools' efficacy for SR fault detection.

2.2.2 Studies on SR Faults and Error-handling Mechanisms in Solidity

Mitropoulos et al. (2024) conduct an extensive empirical study on using Solidity error-handling features in smart contracts, including SR mechanisms. Their study encompasses various facets including the frequency and evolution of the usage of Solidity error-handling features, the types of error-handling misuses, and the evolution of such misuses over time. Olsthoorn et al. (2022) propose an approach for automating the generation of test cases to effectively cover SR statements within smart contracts. They devise an interprocedural fitness function to measure how far a test case is from satisfying the conditions in SR statements, thereby guiding the test case generation process. In contrast, our work delves deeply into issues stemming from real-world practices and incorrect uses of SR statements. We analyze different types of faults in using SR statements and common strategies adopted by developers to fix the faults. Additionally, we assess the ability of existing smart contract security analyzers in detecting the vulnerabilities caused by misusing SR statements.

Liao et al. (2023) propose SmartState, a technique for detecting SR faults in Solidity contracts through state-dependency analysis. These vulnerabilities occur when malicious attackers exploit the SR mechanism to revert transactions when outcomes don't meet their expectations, thus gaining an advantage. Specifically, SmartState examines SR faults from a perspective different from ours. It focuses on the attacker's side-how attackers exploit the SR mechanism to trigger transaction rollbacks when outcomes are unexpected. In SmartState's scenarios, the victim contract may not contain SR statements or might be safe despite using them. EOSafe (He et al. 2021) and WASAI (Chen et al. 2022) also detect the rollback attacks based on symbolic execution and fuzzing, respectively. However, these two techniques are specifically designed for smart contracts in WASM language, and they only consider vulnerabilities that are caused by attackers maliciously rolling back the transaction to gain profits. In contrast, our study investigates faults arising from the misuse of SR statements within the victim contracts themselves. We systematically classify SR faults into 17

distinct categories. Additionally, we identify 12 different strategies for fixing these faults. By doing so, we provide a more comprehensive understanding of SR faults in smart contracts, thereby helping to improve their security and reliability in blockchain applications.

When a transaction runs out of gas during execution, it causes the contract state to revert, which can lead to security vulnerabilities. These gas-related vulnerabilities are related to SR faults. Several studies, including eTainter (Ghaleb et al. 2022), Madmax (Grech et al. 2018), and Gas Gauge (Nassirzadeh et al. 2022), examine gas-related vulnerabilities. MadMax (Grech et al. 2018) is a static analysis tool for detecting gas-focused vulnerabilities. It combines a control-flow-analysis-based decompiler with declarative program-structure queries to identify high-level domain-specific concepts related to gas-related vulnerabilities. eTainter (Ghaleb et al. 2022) is a taint analysis approach for detecting gas-related vulnerabilities. This tool is built on the key insight that gas-related vulnerabilities stem from contract codes depending on data items that are either provided or manipulated by the contract users. Compared with gas-related vulnerabilities, we study SR faults from the perspective of misusing state-reverting statements in smart contracts.

3 RQ1: Prevalence

In this section, we first measure the prevalence of SR statements in real-world smart contracts.

3.1 Data Collection

To investigate the prevalence of SR statements, we analyze the most recent large-scale dapp dataset DAPPSCAN-SOURCE (Zheng et al. 2024) (as of October 2024), which contains 21,457 Solidity files from 682 dapps. We choose DAPPSCAN-SOURCE (Zheng et al. 2024) for its comprehensive and publicly available collection of verified Solidity contracts from popular dapps. Its size (21,414 contracts) and diversity ensure statistically significant findings reflecting current development practices. Several existing works (Li et al. 2024a, b; Eshghie et al. 2024; Chen et al. 2025) have also used DAPPSCAN-SOURCE as their experimental dataset. From this dataset, we successfully retrieve 21,414 verified Solidity files, with 1,858,717 lines of code (LOC) in total and an average of 86.8 LOC per file. Table 1 provides the basic statistics of the 21,414 contracts. We exclude 5,374 interface contracts and 786 abstract contracts, of which the functions are not implemented. After removing them, our dataset contains 15,254 unique smart contracts, which we use for a detailed examination of SR statements.

Table 1 Summary of the Dapp Contract Dataset

Metric	Value
Total Solidity Files	21,414
Total Interface Contracts	5,374
Total Abstract Contracts	786
Total Contracts for Analysis	15,254
Total Lines of Code (LOC) for Analysis	1,858,717
Average LOC per File	121.9

3.2 Results

To answer RQ1, we identify all the SR statements in the 15,254 contracts and compute the code density of these statements. Following existing practices (Hu et al. 2021; Cheng et al. 2024), we identify SR statements using an open-source Solidity parser (Solidity-parser 2024) built on top of a robust ANTLR4 grammar, extracting all `require` and `if... revert` instances while excluding comments and strings to avoid false positives. Following existing practices (Yuan et al. 2012; Harty et al. 2021), we compute code density for SR statements as LOC/LOS, where LOC is the lines of code of a contract and LOS is the lines of SR statements. Similarly, we compute the code density for general-purpose `if` statements and `if...throw` statements for comparison. It is worth mentioning that we only count the lines that involve conditions in the `if` statements, not their bodies. Besides, we separately analyze general-purpose `if`, `if...throw`, and `if...revert` statements. When an `if` statement is used with `throw` or `revert`, we do not consider it as a general-purpose `if` statement since it is used to revert state changes.

Finding 1 In our analyzed smart contracts, SR statements (49.7%) occur more frequently than general-purpose *`if`* statements (34.1%).

Table 2 presents the detailed results. The *#Contracts (Ratios)* column shows both the number of contracts containing each statement type and their corresponding percentages in the dataset. The *#Statements* column lists the total number of each statement type in the dataset. The *Code Density* column shows the average code density per contract for each type, calculated by dividing the total lines of code by the number of each statement type in the dataset. As shown in the table, SR statements are more frequently used than general-purpose `if` statements. Of the 15,254 contracts, 7,580 (49.7%) contain SR statements, with a breakdown of 6,899 (45.2%) contracts containing `require` statements, 703 (4.6%) containing `if...revert` statements, and 68 (0.4%) containing `if...throw` statements. Comparatively, 5,197 (34.1%) contain general-purpose `if` statements. On average, there is one SR statement per 42.1 lines, while general-purpose `if` statements appear once per 54.5 lines.

Understanding the semantic and functional roles of SR statements is important for assessing their security impact in Solidity smart contracts. As our analysis in RQ2 (Section 4) will show, SR statements are predominantly employed for enforcing security-critical constraints, such as authority verification and various validity checks. For instance, 13.1% of SR statements perform address authority checks, ensuring that only authorized entities can execute sensitive operations. Validity checks, including number range and address verification, protect contracts by validating inputs and preventing invalid state interactions. These purposes

Table 2 Distribution and Density of Different Types of Conditional Statements in Smart Contracts

Statement Type	# Contracts (Ratios)	# Statements	Code Density
SR statement	7,580 (49.7%)	44,103	42.1
- `require` statement	6,899 (45.2%)	42,419	40.5
- `if...revert` statement	703 (4.6%)	1,684	49.3
`if...throw` statement	68 (0.4%)	416	4,468.1
General-purpose `if` statement	5,197 (34.1%)	34,130	54.5

highlight the pivotal role of SR statements in maintaining security properties beyond their widespread adoption. For a comprehensive taxonomy and detailed discussion of these purposes, please refer to Section 4 (RQ2).

Finding 2 The percentage of contracts containing SR statements dropped from 94.3% in 2021 to 49.7% in 2024, likely due to modular design practices and increased use of external libraries. However, the frequency of using SR statements within contracts remains steady, indicating their continued importance for security enforcement in specific contracts.

Interestingly, compared to the data from our conference paper (Liu et al. 2021), we observe a significant drop in the percentage of contracts containing SR statements. Our previous work (Liu et al. 2021) analyzed 3,866 contracts, finding that 3,647 (94.3%) contained SR statements and 3,399 (87.9%) contained general-purpose `if` statements. However, in our analysis of 15,254 contracts in this journal extension, only 49.7% contain SR statements and 34.1% contain general-purpose `if` statements. One possible reason for this shift could be the trend toward increased modularity (Modularity, the way forward to building defi systems 2023; Customization and dynamic structuring of smart contracts in solidity 2023). Specifically, the average lines of code (LOC) for Dapp contracts dropped from 763.5 in 2021 (Liu et al. 2021) to 121.9 in 2024. This suggests that developers are increasingly splitting functionality across multiple smaller contracts instead of combining everything into a single monolithic contract. This modular approach improves code readability, reduces coupling, and enhances maintainability. By dividing functionalities across various contract files, developers may avoid duplicating logic and, as a result, reduce the need for SR statements within individual files.

Another key factor is the growing reliance on external libraries that encapsulate validation logic. For instance, the use of OpenZeppelin (Openzeppelin 2024), a popular library providing secure contract implementations, has increased significantly. In 2021, 1,332 out of 3,667 contracts (36.3%) imported OpenZeppelin libraries. In 2024, the percentage rises to 50.6% (7,725 out of 15,254 contracts). OpenZeppelin includes built-in validation checks (e.g., `require` statements) within its functions and modifiers, such as onlyOwner in the Ownable contract or overflow checks in SafeMath. As a result, contracts using these libraries can leverage pre-built security mechanisms without needing to implement redundant SR statements.

Furthermore, the average number of modifier definitions per contract has decreased from 2.70 in 2021 (9,895 modifiers across 3,667 contracts) to 0.38 in 2024 (5,774 modifiers across 15,254 contracts). This suggests that developers are defining fewer custom modifiers, likely because they are increasingly using modifiers provided by libraries like OpenZeppelin. For example, instead of writing a custom onlyOwner modifier, contracts can inherit from Ownable and use the library's implementation.

Despite the drop in the percentage of contract files containing SR statements, for contracts that still include SR statements, the density of SR statements remains stable - 42.1 per contract in 2024 compared to 49.7 in 2021. This consistency indicates that developers still rely heavily on SR statements to enforce conditions in smart contracts and ensure the security of the blockchain applications. Although the raw count of SR statements has decreased, the overall validation coverage is maintained or improved due to the adoption of secure libraries and modular design. Therefore, the decrease in SR statement usage does not imply

reduced security but rather a shift toward more efficient and secure development practices leveraging well-tested external solutions.

Finding 3 SR statement density moderately correlates with contract complexity, particularly in terms of size and structural complexity.

We further examine the relationship between SR statement density and contract complexity through correlation analysis. Following established methodology (Hegedűs 2018), we analyze six key complexity metrics: SLOC (Source Lines of Code), LLOC (Logical Lines of Code), CLOC (Comment Lines of Code), NF (Number of Functions), WMC (Weighted Method Complexity), and NL (Nesting Level). SR density is calculated as the ratio of SLOC to SR statements. We evaluate both Pearson and Spearman correlations (Pearson 1920; Spearman 1961) to capture linear and non-linear relationships, respectively. We visualize the results through scatter plots in Fig. 2. Our analysis reveals that SR density demonstrates significant correlation with structural complexity metrics, e.g., SLOC (Spearman ρ =0.44), WMC (ρ=0.39), and NL (ρ=0.49). This indicates that more complex contracts systematically incorporate more safety checks. The relatively low correlation with NF (Pearson r=0.12, Spearman ρ=0.36) suggests that SR implementation is driven more by control flow complexity than function count. The weak correlation with CLOC (Pearson r=0.18, Spearman ρ=0.51) indicates that SR usage decisions are largely independent of documentation practices.

These findings suggest that developers naturally increase safety checks as contract complexity grows, though not linearly. The stronger Spearman correlations compared to Pearson values indicate a non-linear scaling pattern that SR density grows more rapidly in moderately complex contracts but plateaus in very large ones. This pattern highlights potential

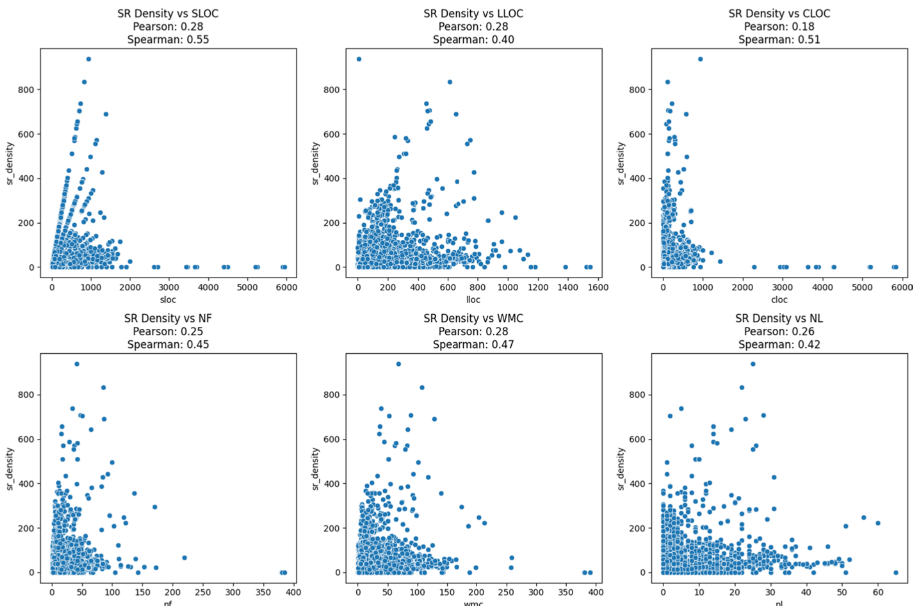


Fig. 2 Correlation Between SR Statement Density and Code Complexity

gaps in SR coverage for highly complex contracts, suggesting opportunities for automated tools to enforce more comprehensive safety checks.

Finding 4 0.40% of our analyzed smart contracts are still using the deprecated `if... throw` statements, which may cause unnecessary financial loss to users.

As explained in Section 2.1.3, `if..throw` statements can also help revert contract states but using them would incur additional costs of gas, thereby causing unnecessary financial loss to the contract users. As a result, `require` and `if...revert` statements are introduced in Solidity 0.4.10 as replacements and `if..throw` is officially deprecated since Solidity 0.4.13 in 2017. However, our analysis reveals that 68 contracts (0.40%) still use `if...throw` statements (in 2021, this percentage was 8.6% (Liu et al. 2021)). Furthermore, six smart contracts (0.04%) use a mix of `if...throw` and `require` statements. To understand this phenomenon, we examine the Solidity versions used in 15,254 contracts. The result shows that 153 contracts (1.0%) still use Solidity versions older than 0.4.10. These contracts can only use the deprecated `if...throw` statements to revert contract states, potentially causing unnecessary gas costs to users who interact with these contracts.

Answer to RQ1: *SR statements are more frequently used in smart contracts than general-purpose `if` statements. The percentage of contracts containing SR statements has decreased from 2021 to 2024, likely due to the modular design of contracts and the frequent uses of external libraries. However, SR statement density within contracts that involve SR statements remains steady.*

Implication: *SR statements play an essential role in assuring the correct execution of transactions. Researchers working on smart contract quality assurance and security analysis should pay more attention to such statements as inappropriately using them may lead to abnormal contract behaviors or financial losses.*

4 RQ2: Purpose

In this section, we investigate the SR statements to understand the purposes of using them in real-world smart contracts.

4.1 Analysis Process

To understand the purposes of using SR statements, we manually analyze the collected smart contracts with the following two steps:

Statement selection Given that the dataset contains 44,103 SR statements, manually analyzing all of them is impractical. For our study, we randomly select 381 of these SR statements, which represent the entire set with a confidence level of 95% and a 5% margin of error. These 381 SR statements comprise 406 clauses in total.

Purpose identification To analyze the purpose of these 381 SR statements, we follow the open coding procedure (Seaman 1999) to create a taxonomy inductively using a bottom-up

approach. Two authors examine all sampled SR statements and their corresponding contracts to understand the purposes of using these SR statements. Specifically, they conduct two rounds of labeling of the 381 sampled statements, refining the taxonomy by alternating between categories and SR statements. When there are labeling conflicts, a third author is invited to participate in the discussion and resolve the differences. Through independent labeling and group discussions, we refine the taxonomy from our previous work and obtain the final results, which are then checked and confirmed by all authors. During the process, we use Cohen's Kappa score (Cohen 1960) to measure inter-rater agreement in the two rounds of labeling. In the first round, the score is 0.72. In the second round, the score reaches 0.92, demonstrating a much stronger agreement between the authors on the taxonomy.

4.2 Purpose taxonomy

Table 3 presents our identified purposes of using SR statements. As we can see, the purposes are organized into two categories, each of which is further divided into subcategories. In our previous work (Liu et al. 2021), we identified seven main purposes: Authority Verification (Address Authority Check, Token Verification) and Validity Check (Logic Check, Range Check, Overflow/Underflow Check, Arithmetic Check, Address Validity Check). This study extends the Validity Check with five new subcategories: *Boolean State Variable Check*, *Function Return Value Check*, *Element Existence Check*, *Type Validation Check*, and *Consistency Check*. Furthermore, we merge the following two subcategories of the *Address Authority Check* category: *Equal to a specific address* and *Within a specific address list*. The reason is that these two kinds of checks essentially serve the same purpose. It is also worth noting that all of the subcategories in the taxonomy are mutually exclusive, meaning that a clause in an SR statement can be classified into only one of them. To facilitate understanding, Table 3 provides illustrative examples collected from our dataset. In the following, we discuss the findings of this study.

Finding 5 SR statements are commonly used to perform ten types of authority verifications or validity checks.

Table 3 Purposes of using SR statements in smart contracts

Category	Subcategory	Description	Illustrative Example	Count	Ratio
Authority Verification	Address Authority Check	Verify whether a contract address is authorized, i.e., has permission for further operations.	`require(msg.sender == address(nonFungibleContract));`	53	13.1%
	Token Verification	Check whether a given token ID is authorized, i.e., whether it is within the mappings of addresses.	`require(_exists(tokenId), "Query for nonexistent token");`	6	1.5%
Validity Check	Number Range Check	Check whether the value of a numeric variable falls within a specific range.	`require(underlyingBalance > 0, "Not have any liquidity deposit");`	153	37.7%
	Address Validity Check	Check whether a contract address is valid.	`require(owner != address(0));`	55	13.5%
	Boolean State Variable Check	Verify the validity of the value of specific boolean state variables using logical operators.	`require(!_mintingFinished);`	50	12.3%
	Function Return Value Check	Check the return value of a specific function call.	`require(token().transferFrom(msg.sender, address(this), _amount));`	42	10.3%
	Overflow/Underflow Check	Check if the variable's value exceeds the range of its declared data type.	`require((z = x + y)>= x);`	24	5.9%
	Element Existence Check	Verify the existence and proper initialization of a required entry (such as an ID or mapping key).	`require(lookup[chainId] != bytes32(0), "LayerZero: chainId does not exist");`	12	3.0%
	Type Validation Check	Ensures that a variable or input matches its expected type.	`require(coverCurr == "ETH", "Pool: Unexpected asset type");`	7	1.7%
	Consistency Check	Verify the alignment of properties between two items by comparing their corresponding values.	`require(records.length == values.length, "Input lengths must match");`	4	1.0%

Authority Verification 59 of the 406 clauses in the 381 SR statements are for *Authority Verification*, aiming to check whether a given contract address or token ID is authorized by the contract owner for the sake of security:

- *Address Authority Check* verifies if a given address is authorized. There are two methods for performing address authority checks. The first method compares the given address to a specified address for equality. The second method checks if the given address is included in a list of authorized addresses. This subcategory accounts for 13.1% of SR statement clauses.
- *Token Verification* involves checking whether a given token ID is authorized. Tokens are value counters stored in a contract, represented as mappings of addresses to account balances. Token verification determines if the token ID exists within these mappings. 1.5% of the clauses in the 381 SR statements perform token verification.

Validity Check 347 of the 406 clauses in the 381 SR statements are for validity checks. 0Generally, validity checks are performed to examine if certain runtime values are valid, i.e., satisfying pre-defined conditions. Unlike the static validity checks performed by SMTChecker (Smtchecker and formal verification 2024) at compile time, SR statements enforce runtime condition checks, serving as a critical safeguard for validating external inputs whose correctness cannot be statically verified. For example, the `require` statement at line 4 in Listing 4 checks a condition using `now` to get the current time, which cannot be analyzed by SMTChecker at compile time. We observe eight sub-categories of validity checks:

- *Number Range Check* determines whether a numeric value falls within a specific range or equals an expected value. This is the largest category, with 37.7% of SR statement clauses falling into it.
- *Address Validity Check* ensures a contract address is valid. It differs from the address authority check discussed earlier, which verifies if an address is authorized (an authorized address must be valid, but a valid address may not be authorized). A common check in this subcategory examines whether a contract address equals `address(0)` in an Ether transfer function. If an Ether transfer is allowed to the address zero, the Ether would be irretrievably lost. To prevent such scenarios, address validity checks are crucial. These checks constitute 13.5% of SR statement clauses.
- *Boolean State Variable Check* uses logical operators to verify the current value of a Boolean state variable within a program's execution flow. This represents the third largest category, comprising 12.3% of SR statement clauses.
- *Function Return Value Check* verifies the return value of a specific function call. These checks commonly appear in SR statement conditions, particularly when validating results from low-level function calls. 10.3% of SR statement clauses belong to this subcategory.
- *Overflow/Underflow Check* determines whether a runtime value fits within the allowed boundaries of a data type. This check constitutes 5.9% of SR statement clauses. It is worth noting that the above-mentioned number range checks are different from the overflow/underflow checks here. The ranges in the number range checks are application-specific, whereas the value boundaries of a data type are prescribed.

- *Existence Check* verifies the presence of a specific element, record, or value within a data structure or system before proceeding with further operations. It ensures that a required entry (such as an ID or mapping key) exists and is properly initialized, preventing potential errors or security risks associated with non-existent or uninitialized data. This subcategory accounts for 3.0% of SR statement clauses.
- *Type Validation Check* ensures that a variable or input conforms to an expected type before further execution. It makes up 1.7% of SR statement clauses.
- *Consistency Check* verifies that the properties of two different items are aligned by comparing their corresponding values. This subcategory accounts for 1.0% of SR statement clauses.
- *Arithmetic Check* examines whether the value of a variable violates common constraints in arithmetic operations, such as division by zero or modulo zero. These checks are less frequent compared to the above categories, accounting for 1.0% of SR statement clauses.

From the results, we can see that number range check, address validity check, and address authority check are the most common purposes of using SR statements in smart contracts, with boolean state variable check and function return value check being the fourth and fifth most frequent purposes, respectively. It shows that developers primarily use SR statements to ensure that inputs remain within safe boundaries and meet logical conditions. This aligns with Solidity documentation (Solidity error handling: Assert, require, revert and exceptions 2024), which recommends using SR statements for input validation. While our study confirms the common usage of SR statements, it also reveals additional usages of SR statements (e.g., Boolean State Variable Check), which are not highlighted in the documentation, offering a more comprehensive taxonomy for understanding developers' diverse motivations for using SR statements. Besides, the prominence of address authority and validity checks underscores the importance of verifying trusted entities and valid addresses, which is essential in a decentralized environment where contracts interact with various accounts and external contracts.

Answer to RQ2: *SR statements are commonly used to perform authority verification and validity checks, many of which involve security-critical constraints.*

Implication: *Since SR statements often check the runtime status of smart contracts against security-critical constraints, it is crucial to ensure the proper use of such statements. Future research can focus on studying the vulnerabilities induced by various misuses of SR statements and proposing detection or repair techniques to combat such vulnerabilities.*

5 RQ3: Fault Types

Our first and second RQs reveal that SR statements are frequently used in smart contracts to perform authority verification and validity checks, many of which involve security-critical constraints. Given the pivotal role of SR statements in smart contracts, understanding their associated faults is important. RQ3 focuses on understanding the categories, distribution, and security impacts of faults induced by various misuses of SR statements, which we refer to as *state-reverting faults* (SR faults).

5.1 Data Collection & Fault Analysis

To collect SR faults, we analyze the relevant code commits submitted to open-source smart contract projects on GitHub. Specifically, we aim to construct a dataset containing pairs of SR faults and their corresponding fixes from the commit histories of Ethereum smart contract repositories on GitHub.

5.1.1 Mining GitHub Repositories

We start with the top 1,000 smart contract repositories on GitHub, according to their number of stars, which is an indicator of project popularity and impact. We use the GitHub search API (Github rest api 2022) to search the repositories to look for commits that contain changed Solidity files and the concerned code changes include adding, deleting, or modifying SR statements. Specifically, we use the query `language:Solidity extension:sol path:*.sol` to identify commits modifying lines with `require`, `revert`, or `throw`. The search process returns 4,335 commits. After excluding irrelevant commits using the following three rules, we retain 1,896 commits for further analysis.

- *The change includes only adding/deleting files.* It is unlikely that such file-level changes are made to fix SR faults, which essentially arise from improper condition checking or error handling.
- *The changed lines of code (LOC) in a file is over 200 lines.* Such a change is likely a composite change that contains complex modifications more than a bug fix, making it very difficult to pinpoint and isolate the code related to SR faults for our study.
- *The commit contains only typo fixing.* These commits solely address typographical errors in the code or documentation. They do not involve functional changes, logic alterations, or bug fixes.

We then sample 320 commits for manual review to identify patterns in commit messages that may indicate the presence of SR faults in the code. We determine the sample size based on a confidence level of 95% and a 5% margin of error (Junk 1999) to ensure that the selected commits provide a robust representation of the population of all the 1,896 commits. After sampling, two authors conduct a detailed review of these 320 commits, examining cases where SR statements are added, deleted, or modified to fix bugs. This review includes an analysis of both commit messages and code differences. Specifically, checking commit messages helps us identify commits that likely address SR faults by revealing developers' intent through keywords and phrases (e.g., ”fix,” ”bug,” ”revert”), while analyzing code differences verifies that these commits involve actual changes to SR statements, ensuring their relevance to our study. After reviewing these 320 commits, we observe that the following keywords in the commit messages are highly correlated with SR faults: *fix*, *check*, *optimize*, *improve*, *require*, *revert*, and *assert*. These keywords, alone or in combination, indicate SR fault fixes. We then use them for filtering the remaining 1,696 commits. A commit is kept only if its commit message contains any of the above keywords (case-insensitive and we perform keyword matching after stemming). After conducting the keyword search, we are left with 537 commits.

5.1.2 Refining Dataset

After the previous filtering steps, the extracted commits may still contain noises unrelated to SR faults, such as code refactoring, functionality improvement, or gas optimization. To refine the dataset, two of the authors further filter the extracted commits through manual analysis. Specifically, for each commit, they check the available data, including the commit message, project specification, relevant audit reports, issue reports, pull requests, code, and developer comments, and exclude commits unrelated to the fixing of SR faults. Any conflicts that arise during this process are discussed and resolved by introducing another author as an arbitrator. To assess the agreement between the two authors during the process of To assess the agreement between the two authors during the process of refining the dataset, we use Cohen's Kappa score (Cohen 1960) as the indicator. The κ value for the dataset refinement process is 0.82, indicating almost perfect agreement. Finally, after the refinement step, our dataset contains 301 commits.

5.1.3 Manual Labeling

We conduct a manual review of the refined dataset to label each entry with its appropriate fault type based on symptoms and to identify fixing strategies. Three authors are involved in this process. We adopt the open coding procedure, a widely used data labeling methodology (Seaman 1999; Stol et al. 2016; Chen et al. 2021), to classify and label the commits. The process comprises three steps.

Individual Classification Two authors with over four years of experience in smart contract research participate in this process. They follow the open coding procedure to inductively create categories for SR faults by analyzing the extracted commits. Specifically, the two authors read and analyze the commits multiple times to understand the context of faults. They provide descriptive phrases to indicate the fault symptoms and how a fault is fixed. During this process, they retrieve all available information of each commit, including the commit message, code, developer comments, pull requests, issue reports, audit reports, and project specification (if any). Additionally, they consult external resources to aid in the manual labeling process, including the SWC registry (Swc registry 2024), which is widely used by developers and researchers to classify smart contract weaknesses, and the EEA Specification (Eea ethtrust security levels specification 2024), a guideline for reviewing smart contracts from a security perspective. They also refer to several academic papers (Zhang et al. 2023; Zhou et al. 2023; Ghaleb and Pattabiraman 2020) to establish the categories.

Next, the two authors construct taxonomies for SR faults by grouping similar ones into categories. This grouping process is iterative, as they continuously refine the taxonomies by going back and forth between categories and commits. If there is any ambiguity or uncertainty about a commit, they mark it for further discussion. Similar to the earlier dataset refinement step, we also adopt Cohen's Kappa score κ(Cohen 1960) to measure the agreement between the two authors. The κ value for the first round of classification is 0.47, suggesting a moderate level of disagreement (Cohen 1960).

Discussions for Resolving Conflicts In some cases, there are ambiguities in classifying faults between the two authors participating in the above individual classification process. To resolve these differences, a third author with over four years of experience in smart contract research is invited to join discussions and analyze the conflicting cases. For each conflicting case, the three authors would engage in discussions until a consensus is reached. After resolving conflicts arising in the individual classification process, the three authors settle on a coding schema, which guides the classification in the following iterations. Leveraging the agreed coding schema, the two authors go through all the commits again to ensure that all the buggy-fixed pairs are selected and classified based on the agreed schema. The κ value for the second classification round is 0.78, which is much higher than the first round, indicating a greater degree of agreement on the classification and labeling results (Cohen 1960).

Iterative Reviews for Reaching Agreements Following the second round of classification, the three authors convene to discuss and review all cases where agreement has not yet been reached. During these discussions, any remaining labeling conflicts are thoroughly analyzed and resolved. This iterative process continues until consensus is achieved for all commits, ensuring that there are no unresolved labeling conflicts. Once all cases have been agreed upon, the final labeling results are reviewed and approved by all authors of this paper, confirming the robustness and reliability of the classification process.

Following our methodology, we classify 278 out of 301 commits into 17 distinct SR fault types and identify common fixing strategies for each type. The remaining 23 commits pose classification challenges due to either complex, intertwined code changes within single files that are difficult to isolate or insufficient contextual information (commit messages, comments, or project specifications) for fault analysis. While we exclude these complex cases from our taxonomy to ensure clarity and reliability, we acknowledge that this could potentially introduce bias if these cases represent a unique category of SR faults. To address this limitation, we provide detailed documentation of these cases and have made our complete dataset publicly available for further research (The material for this study 2024).

5.2 Results of Fault Categorization

Finding 6 We observe 17 SR fault types, indicating the diversity of SR faults in smart contracts.

Following the methodology presented in Section 5.1, we identify 17 SR fault types. Table 4 presents all the identified types of SR faults in smart contracts. Among these faults, *Integer underflow/overflow*, *unprotected Ether withdrawal*, *unchecked call return value*, and *reentrancy* adhere to the definitions in the SWC registry (Swc registry 2024). *Atomicity violation*, *erroneous accounting*, and *ID uniqueness violation* are inherited from Zhang et al.'s work (Zhang et al. 2023). Note that for SWC 123 Requirement Violation (Swc 123: Requirement violation 2024), the definition suggests two possible issues when a requirement is violated: 1) A bug exists in the contract that provides the external input; 2) The condition used to express the requirement is too strong. The first situation is out of our scope, as we focus on faults in contracts containing SR statements rather than those providing external inputs. The second situation, while falling into the scope of our work, fails to reveal the specific causes of the faults. Therefore, we divide the concerned faults in our dataset into

Table 4 Summary of SR Fault Types

Fault Type	SWC	# Faults in the GitHub Dataset
Integer Underflow/Overflow	101 Swc 101: Integer overflow and underflow (2024)	67
Unprotected Ether Withdrawal	105 Swc 105: Unprotected ether withdrawal (2024)	59
Unchecked Call Return Value	104 Swc 104: Unchecked call return value (2024)	38
Missing Zero Address Check	-	33
Reentrancy	107 Swc 107: Reentrancy (2024)	27
Temporal Property Violation	-	13
Consistency Violation	-	9
Resource Sufficiency Violation	-	8
Missing Account Type Verification	-	6
Atomicity Violation	-	3
Time Constraint Violation	-	3
Non-asset-related Unauthorized Access	-	3
Lack of Property Check for External Call	-	3
Erroneous Accounting	123 Swc 123: Requirement violation (2024)	3
Excessive Slippage	123 Swc 123: Requirement violation (2024)	1
Number Rounding Error	123 Swc 123: Requirement violation (2024)	1
ID Uniqueness Violation	-	1

three different types, namely, erroneous accounting, excessive slippage, and number rounding error, in order to provide a more granular classification of faults based on their underlying causes. This detailed classification not only aids in understanding the exact nature of an SR fault, but also provides developers with targeted guidance for mitigating these issues. Additionally, well-defined fault types enable automated tools to detect and categorize faults more effectively, ultimately leading to improved contract reliability and security. In the following, we introduce each of the 17 SR fault types one by one.

Integer Underflow/Overflow (UO) An overflow or underflow occurs when an arithmetic operation produces a value that exceeds the maximum or minimum value of a data type. This can happen due to missing or improper range checks. Identifying the extreme values that can trigger these issues is challenging, especially in complex operations. It is the most common type of faults in our dataset and there are in total 67 such SR faults. The Solidity compiler has addressed integer underflow/overflow issues since v0.8.0 in 2020 through built-in checks and the SafeMath library (Solidity v0.8.0 breaking changes 2020). However,

many older contracts still in use may be vulnerable (Li et al. 2024). In 2023, integer underflow/overflow issues remain to be the second most common type of smart contract vulnerabilities (Owasp smart contract top 10 2023). Moreover, some developers may choose to implement their own arithmetic functions for optimization purposes, bypassing compiler checks (for example, by using the *unchecked* keyword (Li et al. 2024)). In such cases, understanding and addressing potential overflow/underflow issues remains essential.

```
1  contract Voting {
2      function createProposal(bytes32 proposalHash) external {
3          proposals[proposalCount] = Proposal(proposalHash, uint128(block.number - 1));
4          proposalCount += 1;
5      }
6      function vote(uint128 votingPower, uint256 proposalId, Ballot ballot) public {
7  +       require(proposals[proposalId].created != 0, "Proposal does not exist");
8          proposals[proposalId].votingPower += votingPower;
9      }
10     function executeProposal(uint256 proposalId, address[] memory targets, bytes[] memory
           calldata){
11         for (uint256 i = 0; i < targets.length; i++) {
12             (bool success,) = targets[i].call(calldata[i]);
13             require(success, "Call failed");}
14         delete proposals[proposalId];
15     }
16 }
```

Listing 2: A temporal property violation example

Unprotected Ether Withdrawal (UE) Smart contracts often involve transactions of crypto assets (like tokens and NFTs) between entities (such as contract addresses or users). It is crucial to perform permission checks during these processes to verify the authorization and legitimacy of participants. When proper permission checks for Ether transfers are absent, it is called *unprotected Ether withdrawal*. This vulnerability can be exploited by unauthorized attackers to gain contract ownership or withdraw valuable assets without permissions, leading to significant financial losses. There are 59 such SR faults, making UE the second most common type.

Unchecked Call Return Value (UC) Solidity provides low-level call methods that operate on raw addresses, such as `address.call()` and `address.delegatecall()`. These low-level methods do not raise exceptions and only return *false* when an exception occurs. Without verifying the return value of such a call, if the call fails unintentionally or if an attacker forces the call to fail, it can result in unexpected behavior in the subsequent program execution. Therefore, it is important to verify the return value of these low-level functions to handle potential failures. The lack of checking the return value of low-level functions is referred to as *unchecked call return value*. This fault type is the third most common in our dataset.

Missing Zero Address Check (ZA) The zero address (`0x0`) is a special address in Ethereum. Obtaining the private key of the zero address account is cryptographically considered impossible. Thus, the zero address acts like a black hole. Any tokens sent to it cannot be recovered. If a zero address check is not performed for critical addresses (e.g., the ownership address, the token transfer address, etc.), contract ownership or transferred tokens could be lost forever. Thus, it is crucial to conduct a zero address check to ensure the destination address is not the zero address. This type of fault is referred to as *missing zero address check*.

Reentrancy (RE) A *reentrancy* attack occurs when a function makes an external call to an untrusted contract, and then the untrusted contract makes a recursive call back to the original function, potentially breaking the atomicity of the function execution. The reentrancy attack is one of the most destructive attacks in Solidity smart contracts. The most well-known reentrancy attack, the DAO attack (Understanding the dao attack 2023), resulted in a loss of 60 million US dollars. Our GitHub dataset contains 27 instances of reentrancy, highlighting both the severity and prevalence of this type of vulnerabilities.

```
1  contract TokenSwap{
2      address public immutable WETH;
3      function estimateSwap(address beefyVault, address tokenIn) public view{
4  +       checkWETH();
5          (, IUniswapV2Pair pair) = _getVaultPair(beefyVault);
6          bool isInputA = pair.token0() == tokenIn;
7          require(isInputA || pair.token1() == tokenIn);
8          (uint256 reserveA, uint256 reserveB,) = pair.getReserves();
9          (reserveA, reserveB) = isInputA ? (reserveA, reserveB) : (reserveB, reserveA);
10     }
11 +   function checkWETH() public view returns (bool isValid) {
12 +       isValid = WETH == router.WETH();
13 +       require(isValid, "WETH address not matching Router.WETH()");}
14 }
```

Listing 3: A consistency violation example

Temporal Property Violation (TP) Smart contracts often encapsulate temporal properties within their business logic, i.e., the functions involved in a business flow need to be invoked in a specific order. Take a decentralized autonomous organization (DAO) contract as an example (Listing 2 An temporal property violation example 2024). The complete business flow involves creating a new proposal, voting on the proposal, and executing the proposal if enough votes are received. Each proposal should follow this life cycle. This process requires the correctness and validity of certain critical states within a business flow to ensure its normal execution. For instance, it is necessary to check whether a proposal has already been created before proceeding with voting and execution. However, developers may forget to check the correct value of state variables during a part of the business flow, or they may incorrectly set the value of state variables. This can lead to a violation of the temporal property of a business flow, giving malicious attackers opportunities to exploit it, such as voting on a non-existent proposal. We refer to this type of fault as *temporal property violation*. A comprehensive understanding of the contract's business logic is essential for identifying and fixing such SR faults.

Consistency Violation (CV) A consistency check determines whether the values of the properties of two different items are equal. For example, in a token swap contract(Listing 3), a consistency check should verify that the current WETH address matches the WETH address of a Uniswap router. This check ensures the contract's environment aligns with its expectations. We refer to the improper use of a consistency check as a *consistency violation*.

Resource Sufficiency Violation (RS) It is often necessary to check for adequate resources (e.g., gas or balance) to complete a transaction or operation within a smart contract. This ensures smooth execution and prevents potential runtime errors due to insufficient resources. For example, many smart contracts involve asset transfers between accounts. Before executing a transfer, it is essential to check that the sender has sufficient balance. Additionally, it is necessary to ensure that there is a sufficient amount of resources to be traded or oper-

ated. In smart contracts involving asset trading, such as NFT trading, ensuring that the trade amount does not exceed the quantity of available assets is critical. Otherwise, overselling could occur, negatively affecting the profit of the participants. In our GitHub dataset, we observe eight cases of *resource sufficiency violations*, where developers fail to properly check resource sufficiency, resulting in undesirable consequences.

Missing Account Type Verification (MA) The Ethereum platform provides two types of accounts: externally owned accounts (EOAs) and contract accounts. EOAs are controlled by Ethereum users with private keys, and can only be used for Ether transfers. In contrast, contract accounts are smart contracts that can perform various actions, such as transferring or creating new tokens. Both types of accounts have an associated Ethereum address. When performing operations linked to Ethereum addresses, developers should determine the account type (Liao et al. 2023). This is particularly crucial in decentralized finance (DeFi) applications. For instance, it's essential to verify that tokens involved in swaps and other transactions are smart contracts adhering to token standards like ERC-20, rather than wallet addresses. We refer to the lack of such checks as *missing account type verification*.

```
1  contract Machine{
2      function rentMachine(address _pendingRenter, uint256 _startTime, uint256 _endTime,
        uint256 _price) public onlyOwner {
3          require(_pendingRenter != address(0));
4  +       require(now <= _startTime && now <= _endTime && _startTime <= _endTime);
5          rentalPrice = _price;
6          pendingRenter = _pendingRenter;
7      }
8  }
```

Listing 4: A time constraint violation example

```
1  contract Referral{
2      function isCircularReference(address referrer, address referee) internal returns(bool){
3          address parent = referrer;
4          for (uint i; i < levelRate.length; i++) {
5              if (parent == referee) return true;
6              parent = accounts[parent].referrer; }
7          return false;
8      }
9      function addReferrer(address payable referrer) internal {
10 +       require(!isCircularReference(referrer,msg.sender), "Referee can't be in referrer's
     upline");
11         Account storage userAccount = accounts[msg.sender];
12         Account storage parentAccount = accounts[referrer];
13         userAccount.referrer = referrer;
14         parentAccount.referredCount = parentAccount.referredCount.add(1);
15     }
16 }
```

Listing 5: A non-asset-related unauthorized access example

Atomicity Violation (AV) Multiple business flows, or transaction sequences, may interleave and interfere with each other by accessing the same state variables (Zhang et al. 2023). Certain business flows require business-level atomicity, which ensures that specific state variables can be accessed by only one business flow at a time. Failing to adhere to this atomicity is known as *atomicity violation* (Zhang et al. 2023).

Time Constraint Violation (TC) Certain smart contracts incorporate business flows that include time constraints, meaning that the execution of specific functions must adhere to a set timeframe, or the function call will not succeed. For example, in a contract shown in Listing 4 (An example of fixing others faults by adding a timestamp check 2018), it man-

ages the usage of a machine. The `rentMachine` function is used to allow the contract owner to rent out a machine by specifying the rental period (start and end time). For successful machine rental, it is important to verify that the rental period starts after the current time and ends after the start time. However, the original function fails to restrict the selection of a past rental period or an illogical time frame in which the start time exceeds the end time, thereby permitting the booking of a machine for an invalid duration. The improper setting of time constraints is called *time constraint violation*.

Non-asset-related Unauthorized Access (NU) Access control restricts permissions on who can call a smart contract function. Lack of access control on asset-related functions can lead to serious consequences, providing malicious attackers with opportunities to invoke functions without authorization, like draining assets from a contract account. Apart from unauthorized access to asset-related functions, many other non-asset-related functions necessitate access control, such as assigning contract ownership or identifying qualified participants. The absence of access control in these non-asset-related functions can also result in severe consequences. We refer to it as *non-asset-related unauthorized access*. Such unauthorized access control cases are not directly related to asset transfer operations (e.g., token or Ether transfers) or explicit code destruction operations (e.g., *selfdestruct*), thus making them difficult to detect. For instance, in Listing 5 (An example of fixing access control faults by adding a blacklist address check 2023), the contract includes a referral system that rewards those who introduce new referees. However, the original contract does not have a check to confirm that a new referee is not one of the referrers, thus violating the integrity and fairness of the referral system.

```
1  contract MultiAMBErc20ToErc677{
2      function relayTokens(ERC677 token, address receiver, uint256 _value) internal {
3  +       uint256 balanceBefore = token.balanceOf(address(this));
4          setLock(true);
5          token.transferFrom(msg.sender, address(this), _value);
6          setLock(false);
7  +       uint256 balanceDiff = token.balanceOf(address(this)).sub(balanceBefore);
8  +       require(balanceDiff <= _value);
9  -       bridgeActionsOnTokenTransfer(token, msg.sender, receiver, _value);
10 +       bridgeActionsOnTokenTransfer(token, msg.sender, receiver, balanceDiff);
11     }
12 }
```

Listing 6: A lack of property check for external call example

```
1  Contract Vault{
2      uint256 public totalAmountDeposited = 0;
3      uint256 internal constant RATIO_MULTIPLY_FACTOR = 10**6;
4      function provideLiquidity(uint256 amount, uint256 minOutputAmount) external nonReentrant{
5          require(amount > 0, 'Cannot stake zero token');
6          uint256 receivedETokens = getNrOfETokensToMint(amount);
7  +       require (receivedETokens >= minOutputAmount, "Insufficient Output");
8          totalAmountDeposited = amount + totalAmountDeposited;
9          _mint(msg.sender, receivedETokens);
10         require(stakedToken.transferFrom(msg.sender, address(this), amount));
11     }
12     function getNrOfETokensToMint(uint256 amount) internal view returns (uint256) {
13         return (amount * RATIO_MULTIPLY_FACTOR) / (1 * RATIO_MULTIPLY_FACTOR);
14     }
15 }
```

Listing 7: An excessive slippage example

Lack of Property Check for External Call (LP) Smart contracts enable cross-contract calls, which allow a contract to interact with other deployed contracts. This feature is useful for retrieving information from another contract or executing a function in another contract.

However, when performing an external call, the external function is uncontrollable, and the return data is thus not guaranteed. Therefore, to avoid any unexpected behavior, developers need to check the return data from the external call. In practice, developers may assume that the external function is benign and neglect to check the return data. We refer to such a fault as *lack of property check for external call*. One example is shown in Listing 6 (Property check for external call example 2021). The developers initially overlook the potential fee deduction from the external call `token.transferFrom` (line 5) and directly send the input token amount for further execution. This could lead to a loss on the contract owner's side.

Erroneous Accounting (EA) Smart contracts often involve formula calculations, particularly in those scenarios with complex business models like decentralized finance (DeFi) applications. These formulas, used for calculating interests, yields, and so on, can be quite intricate. They involve complex arithmetic operations with multiple variables and are prone to errors. The incorrect implementations of underlying business model formulas are called *erroneous accounting* (Zhang et al. 2023). While such faults may seem minor, they can result in significant financial losses (Defi money market compound overpays millions in comp rewards in possible exploit 2021). Therefore, when dealing with complex business model formulas, developers need to exercise caution to prevent erroneous accounting.

Excessive Slippage (ES) Slippage refers to the difference between the price a user expects to pay or receive for buying or selling tokens and the actual price at which the transaction is executed. A major cause of slippage is the mismatch between demand and available liquidity (the volume of buy and sell orders at a given time). In centralized exchanges, slippage occurs when the order book cannot fulfill the trade request at the initial price, thus requiring moving up or down the book to complete the trade. For example, if the liquidity of a traded asset is insufficient or the spread across the order prices is uneven and irregular, it results in price slippage. Different from centralized exchanges, decentralized exchanges use liquidity pools instead of spread order books. Here, slippage happens when the liquidity pool lacks a sufficient quantity of the paired tokens. In practice, traders are usually allowed to set up the maximum allowed slippage (slippage tolerance) for their trades to prevent significant losses due to slippage. Consider an example in our dataset (Listing 7 Slippage protection example 2022). When adding liquidity to a pool, the calculated number of tokens, representing the amount of asset deposited, must be greater than or equal to a specified minimum. If not, the transaction will be reverted to prevent unexpected slippage. For such cases, developers should properly implement SR statements, as otherwise price slippage will happen.

```
1  contract Tranche{
2      uint128 public valueSupplied;
3      function prefundedDeposit(address _destination) public{
4          (uint256 shares, uint256 usedUnderlying, uint256 balanceBefore) = position.
               prefundedDeposit(address(this));
5          uint256 holdingsValue = (balanceBefore*usedUnderlying)/shares;
6  -       require(valueSupplied<=holdingsValue);
7  +       require(valueSupplied<=holdingsValue + 2); //The +2 allows for small rounding errors
8          uint256 adjustedAmount;
9          if (valueSupplied>0 && holdingsValue>valueSupplied) {
10             adjustedAmount = usedUnderlying-((holdingsValue-valueSupplied)*usedUnderlying)/
                   interestSupply;
11         } else {
12             adjustedAmount = usedUnderlying;}
13         valueSupplied = uint128(valueSupplied+adjustedAmount);
14         interestToken.mint(_destination, usedUnderlying);}
15 }
```

Listing 8: A number rounding error example

Number Rounding Error (NR) Solidity only supports integers, while floating-point or fixed-point numbers are not supported. If not carefully designed, the precision loss in the integer division may accumulate and result in unexpected contract states. For example, consider a tranche contract (Number rounding protection example 2021) shown in Listing 8. This contract allows users to deposit an underlying token and rewards them with interest and a different principal token reflecting their share in the tranche (function `prefundedDeposit`). If floating-point arithmetic is used to calculate the payout (line 6), improper handling of these calculations can cause a loss of precision. This could result in unfair distributions, potentially harming the benefits of participants. Since number rounding errors are inherent, contract developers should handle these calculations properly (line 7) to prevent precision loss from floating-point arithmetic. This will help avoid unfair distributions or potential fund loss for individuals or the host.

ID Uniqueness Violation (IU) Many contracts involve operations where an entity (such as a user or contract account) manages an asset (like an NFT token) (Zhang et al. 2023). Depending on the contract's specifications, a unique ID field may be required to represent an entity or asset. For instance, in a lottery contract, each lottery ticket should have a unique number ID. If developers do not ensure the uniqueness of this ID field, it results in double payment to lottery winners. By definition, when a contract's function involves implementing non-fungible tokens or assets with a uniqueness feature, developers should be particularly vigilant about *ID uniqueness violation*.

5.3 Relevance and Security Impact of SR Faults

The above observations are made by studying only the GitHub dataset. To validate the relevance and investigate the security impact of our identified SR fault types, we further extract SR faults of our identified types from audit reports published from Oct 2023 to Oct 2024 on Code4rena (Code4rena 2024), a leading smart contract auditing platform. Using Code4rena data is a common practice in smart contract research (Zhang et al. 2023; Xi et al. 2024; Sun et al. 2024). Following Zhang et al.' work (Zhang et al. 2023), we evaluate exploitable vulnerabilities using Code4rena's severity ratings, given the platform's established reliability. Specifically, we collect SR faults for fault types not covered by the SWC registry or where the number of faults is fewer than 10 in our GitHub dataset. This ensures that our findings can extend beyond public repositories of open-source projects. Moreover, Code4rena employs a comprehensive tagging system that categorizes vulnerabilities into three severity levels: high, medium, or low. Such a classification allows for a precise evaluation of the potential security impact associated with SR faults.

To collect issues related to SR faults on Code4rena, we use specific keywords and labels to filter bug reports. We design a set of search keywords that match the characteristics of our interested SR faults. These keywords are listed in Table 5. We apply them to filter audit reports on Code4rena based on two essential criteria:

– *Categorized under security-relevant labels.* Code4rena allows auditors to label bugs by type and severity. We focus on labels indicating critical contract behavior, including "high severity" and "medium severity". We exclude "low risk and non-critical" issues

and “gas optimization” issues. This ensures that the identified issues are impactful and related to contract security.
– *Contain recommended mitigation steps.* The suggested fix allows us to understand the fault and validate that the fault is an SR fault. We exclude issues lacking recommended mitigation information for our analysis. For issues with mitigation steps, we thoroughly review their description, proof-of-concept, and suggested fixing strategy or developers' fixes to confirm they are SR faults.

Finding 7 Although most SR faults (83.3%) are medium-risk, high-risk issues like excessive slippage and unauthorized access pose serious threats to asset security. This demonstrates the need for stronger protection of contracts against SR faults.

With the above process, we collect 42 SR faults. These faults are collected from 99 audit reports dated from Oct 2023 to Oct 2024. Table 5 shows that 7 of these faults are classified as high-risk (Code4rena github repository 2024), which can cause significant security breaches, including direct theft or loss of assets. The remaining 35 faults are categorized as medium-risk (Code4rena github repository 2024). While these medium-risk issues may not directly threaten assets, they could affect contract functionality or availability. Among

Table 5 Searching Keywords and Results for SR faults on Code4rena

Fault Type	Keywords (Case-Insensitive)	#High	#Mid	#Total
Consistency Violation	consistency, consistent/ inconsistent	0	3	3
Resource Sufficiency Violation	sufficient/insufficient, exceed	0	9	9
Missing Account Type Verification	account type, EOA	0	1	1
Atomicity Violation	atomicity	0	0	0
Time Constraint Violation	timing, time constraint	1	1	2
Non-asset-related Unauthorized Access	access control	2	0	2
Lack of Property Check for External Call	return value	0	2	2
Erroneous Accounting	accounting, incorrect calculation	1	1	2
Excessive Slippage	slippage	2	16	18
Number Rounding Error	rounding	1	1	2
ID Uniqueness Violation	unique/uniqueness, ID	0	1	1

“#High” indicates the number of faults tagged as “High risk”. “#Mid” indicates the number of faults tagged as"Medium risk". “#Total” refers to the total number of faults in this category

the faults, the most frequently occurring issues are *Excessive Slippage*, with 18 instances, including 2 high-risk cases. This suggests that slippage errors, although often classified as medium risk, have the potential to critically affect asset transfers, leading to substantial losses if slippage tolerance limits are improperly enforced. *Resource Sufficiency Violation* is the next most common fault type. We observe a total 9 such faults and the issues are all considered to have a medium risk. Another notable fault type is *Non-asset-related Unauthorized Access*, with 2 high-risk instances, reflecting the importance of access control to prevent unauthorized interactions with contracts.

Answer to RQ3: *SR faults in smart contracts are diverse, encompassing 17 distinct types. Most SR faults not included in SWC are medium-risk faults. However, there also exist some high-risk SR faults like excessive slippage and unauthorized access, which can pose significant threats to asset security.*

Implication: *The diversity of SR fault types and the presence of high-risk issues underscore the importance of targeted protection mechanisms for critical SR checks. Future research should prioritize the development of automated tools to identify and mitigate high-risk SR faults, especially those that directly impact asset security.*

6 RQ4: Fixing Strategies

This section presents an empirical analysis of smart contract fault fixes in real-world applications. We aim to identify and categorize repair strategies observed in practice, providing a descriptive account of current approaches. While we focus on documenting and classifying existing approaches rather than evaluating their effectiveness, this analysis can serve as a foundation for future research on best practices and automated repair techniques. Specifically, we summarize the fixing strategies for each fault type using both the GitHub and Code4rena datasets. We identify 12 fixing strategies through a systematic open coding procedure (Seaman 1999). Two authors independently analyze code diffs from 1,896 commits that addressed SR faults in Solidity smart contracts. They examine changes such as adding new checks, modifying existing conditions, or removing unnecessary SR statements, allowing patterns to emerge organically from the data. These changes are grouped into preliminary categories and iteratively refined through repeated analysis and discussion. In cases of ambiguity, a third author is consulted to ensure consensus. This process continues until all observed fixes could be classified under a distinct strategy, resulting in a comprehensive and reliable set of common fixing practices. The Cohen's Kappa score (Cohen 1960) for the process is 0.88, indicating almost perfect agreement. We detail these identified strategies and demonstrate some real-world examples of faults along with their corresponding fixes. Figure 3 presents the distribution of fixing strategies for each type of SR faults. The X-axis represents each fault type. The Y-axis shows the frequency of each fixing strategy.

Finding 8 We identify 12 common strategies for addressing SR faults. These strategies vary according to the specific faults. The most common strategies are fixing logical comparison checks and adding whitelist address checks, accounting for 35.3% and 17.8% of all the studied fixes, respectively.

Add mutex lock In 19 out of 27 *reentrancy* issues, mutex locks are added to vulnerable code snippets to place a lock on the smart contract state. This locking mechanism prevents the function from being called repeatedly before the first call is completed.

Change state update statement position In the remaining eight of the *reentrancy* issues, the corrective action involves adjusting the order of state updates within the functions. This ensures that all state changes precede external calls, adhering to the checks-effects-interactions pattern (Checks-effects-interactions pattern 2024) and eliminating reentrancy opportunities.

Add low-level call return value check All 38 cases of *unchecked call return value* are fixed by introducing checks that verify the success of the call. This ensures that if the call fails, the entire transaction is reverted to maintain contract integrity.

Add whitelist address check 55 instances of *unprotected ether withdrawal* faults are fixed by implementing whitelist checks. These checks ensure that only addresses authorized by the contract logic can initiate transfers, thus preventing unauthorized withdrawals. For example, when performing fund transfer operations, SR statements should be implemented to verify if the `msg.sender` is authorized to carry out the operations. In a `transfer-Batch` function (Unprotected ether withdrawal exapmle 2021), which handles asset transfers, an SR statement "`require(msg.sender == owner);`" is added to ensure only the contract owner can invoke the functions.

Add blacklist address check Additionally, four fixes for *unprotected ether withdrawal* involve validating whether the user is blacklisted. *Non-asset-related unauthorized access* faults are also fixed by adding missing blacklist address checks. For example, in Listing 5, where the contract manages a referral system that rewards the referrer for bringing in new referees, a check "`require(!isCircularReference(referrer, msg.sender));`" is added in the `addReferrer` function. This function records new referees and their corresponding referrers. Without this check, a referee could potentially become their own referrer (either directly or indirectly) through a series of referrals, leading to an exploitable loop where rewards could be unfairly generated. This check ensures that the new

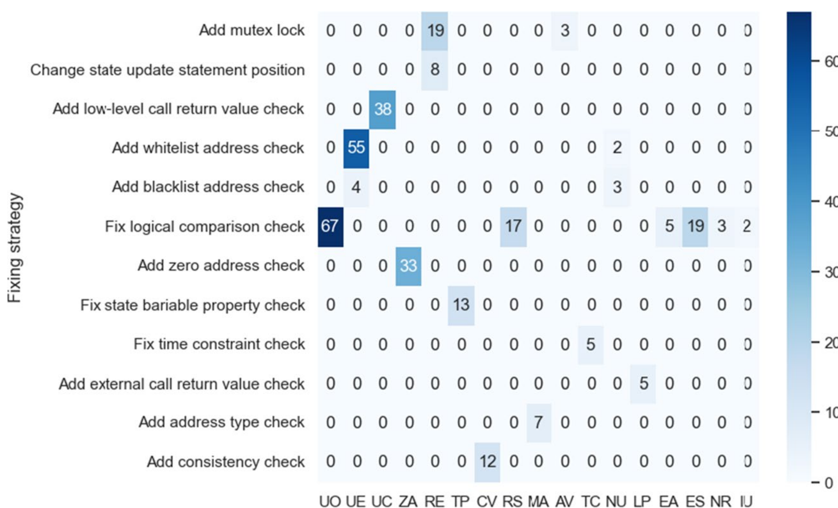


Fig. 3 Fixing strategies for SR faults

referee is not one of the referrers, thus preserving the integrity and fairness of the referral system.

Fix logical comparison check Fixing logical comparison checks is a common strategy for remedying SR faults, including adding logical comparison checks, modifying logical comparison checks, and deleting logical comparison checks. The purpose behind these checks can vary depending on the fault addressed. *Excessive slippage* faults can be resolved by adding logical comparison checks. Take the `vault` contract (referenced in Listing 7) for instance. In the original version, it lacks a check to ensure that the number of `eTokens` received meets the minimum amount anticipated by the liquidity provider (`minOutputAmount`). This could result in situations where users receive fewer `eTokens` than expected due to slippage or fluctuations in the conversion rate between the initiation and execution of the transaction. Developers address this by adding a `require` statement (line 7) that checks if the number of `eTokens` to be received is at least the `minOutputAmount` specified by the user. This is a common practice in DeFi platforms to protect users from slippage. This check is essential to ensure that users' deposits are not undervalued due to market instability or manipulation. *Erroneous accounting* faults and *number rounding error* faults can also be mitigated by implementing logical comparison checks. These checks serve as safeguards against accounting mistakes and number rounding errors. For example, the number rounding error in Listing 8 is fixed by correcting the logical comparison check within a `require()` statement (line 7). This adjustment introduces a small tolerance (`+2`) to accommodate minor rounding errors during the calculation of the holding value. It is also the primary strategy for fixing *underflow/overflow* issues.

Add zero address check Adding a zero address check is a prevalent strategy in the *missing zero address check* faults. Developers fix this kind of faults by adding zero address checks for critical addresses. For example, in this contract (Zero address check example 2021), it manages a reward system that claims reward tokens and sends them to the users. In the original contract, users could delegate to the zero address (`address(0)`), which could result in the loss of governance rights, as delegating to the zero address is essentially equivalent to burning the rights or rewards. The developer addresses this issue by adding the `require(delegate!= address(0))` check to the function, ensuring the `delegate` parameter is not the zero address. This fix helps to maintain the integrity of the reward system and ensures that the delegation of rewards or voting rights is done deliberately.

Fix state variable property check Issues in *temporal property violation* can be addressed by correcting property checks. For example, the code snippet in Listing 2 presents a simplified version of a voting contract (An temporal property violation example 2024). This voting contract facilitates a voting system where proposals are created, voted on, and executed if certain conditions are met. It includes three main functions: `createProposal`, `vote`, and `executeProposal`. The `createProposal` function is designed for creating new proposals. The `vote` function allows token holders from specified voting vaults to cast their votes on active proposals. If a proposal has sufficient votes and is valid (matches the proposal hash), the contract attempts to execute the proposal. These three functions reflect the temporal property of the contract and should be executed in sequence to align with a

proposal's life cycle. However, the original contract violates the temporal property of a proposal in the `vote` function. It allows votes on non-existent proposals or on proposals that have already been executed or expired without checking the `created` property. To address this, developers add a new `require()` check in line 7 to ensure that votes can only be cast on proposals that have been properly created. This prevents participants from interacting with proposals that have not been initialized or have been removed, which could potentially lead to unexpected outcomes or wasted gas if the function calls are made to non-existent proposals. Thus, it maintains the integrity of the voting process, ensuring the contract's correct temporal property, and guarantees a fair and reliable governance mechanism.

Fix time constraint check *Time Constraint Violation* faults are addressed by correcting time constraint verification. For example, in a machine contract (Listing 4), which manages the usage of a machine, a function `rentMachine` is intended to allow the contract owner to rent out a machine. The original function does not prevent setting a rental period in the past or an illogical timeframe where the start time is after the end time. To fix this, developers add a timestamp check to ensure that the current time (`now`) is before or equal to the start time `_startTime` and the start time `_startTime` is less or equal to the end time `_endTime`. This ensures the rental period is set for a valid and upcoming timeframe, preventing the renting out of a machine for an invalid time duration.

Add external call return value check *Lack of property check for external call* faults are typically corrected by introducing return value checks for external calls. An example is shown in Listing 6. This contract is designed for a token bridge scenario where tokens from one network are locked in a contract and a corresponding amount is minted or unlocked on another network. The `relayTokens` function is designed to relay tokens from a user to another contract while ensuring that the correct amount, after fees, is transferred. In the original contract, there is no consideration for tokens that deduct a fee. If such a token is used, the amount received by the contract would be less than the `_value` specified, because the fee would be subtracted during the transfer. To address this, the developers introduce a new check at line 8 that accommodates the possibility of fee deduction. The new check ensures that the increase in the contract's balance (`balanceDiff`) does not exceed the transferred value (`_value`) and the `balanceDiff` is then sent to the `bridgeActionsOnTokenTransfer` function (line 10).

```
1  contract FixedRateExchange{
2  +   function isContract(address addr) public view returns(bool){
3  +    uint32 size;
4  +    assembly {size := extcodesize(addr)}
5  +     return (size > 0); }
6
7      function createExchagePairs(address datatoken, address[] memory addresses) external
           onlyRouter returns (bytes32 exchangeId) {
8          require(addresses[0] != address(0), "Invalid baseToken: zero address");
9  +       require(isContract(addresses[0]), "Invalid baseToken: EOA");
10         require(datatoken != address(0), "Invalid datatoken: zero address");
11         require(addresses[0] != datatoken, "Invalid datatoken: equals baseToken");
12         exchanges[exchangeId] = Exchange({
13             datatoken: datatoken,
14             baseToken: addresses[0]
15         });
16     }
17 }
```

Listing 9: A missing account type verification example

Add address type check *Missing account type verification* faults can be resolved by adding or correcting the verification for an address to confirm it is a contract account address. Typically, this check is performed on an address before introducing new features or functions for it, such as creating a new exchange pair for a token address. For example, in a decentralized exchange (Dex) contract(Listing 9), it allows the creation of new exchange pairs between a base token and a data token. The original contract does not have a mechanism to validate if the base token's address is a contract address. In the revised version, developers add a check (the `isContract` function) to confirm if an address is a contract account address. The boolean return value is then passed to a `require()` statement to prevent non-contract account addresses. By including this check, developers ensure that only valid contracts can be used as the base token in the exchange pair. This is vital to prevent potential errors or attacks that could happen if a regular wallet address is used instead of a token contract.

Add consistency check Adding consistency check is a typical way to fix *consistency violation* faults. For example, in a token swap contract (Listing 3), it interacts with a Uniswap V2 Router (Uniswap v2 router 2024) to facilitate token swaps between two tokens. The Uniswap V2 router enables users to directly exchange ERC20 tokens through automated liquidity pools, without the need for traditional market makers or order books. The original contract assumes that the address stored in `WETH` aligns with the `WETH` address used by the router. If these addresses do not match, the contract may not function correctly as it would be using the wrong `WETH` address for swaps. To fix the fault, developers introduce the `checkWETH` function as a prerequisite check. This function contains a `require()` statement to validate that the contract's `WETH` address matches the `WETH` address of the Uniswap router.

Answer to RQ4: *To address SR faults, developers often use 12 different strategies tailored to specific fault types. Among these strategies, the most prevalent ones are correcting logical comparison checks (35.3%) and adding whitelist address checks (17.8%).*

Implication: *The variety of strategies for addressing SR faults highlights the complexity of ensuring robust SR checks in smart contracts. This suggests that future work should focus on developing systematic guidance and automated tools to assist developers in selecting and applying appropriate SR fault mitigation strategies, particularly for critical tasks such as logical validation and access control.*

7 RQ5: Fault Detection Capability of Existing Tools

SR statements ensure transaction atomicity by determining whether state changes should occur based on specific conditions. As SR statements often validate critical conditions, faulty or missing implementations of these statements can expose contracts to severe security risks. Besides, when improperly implemented, these statements can also lead to unnecessary gas consumption. This combination of risks and complexities makes SR faults particularly important to study. Although faults like reentrancy, integer overflow/underflow, and access control have been widely studied (Di Angelo and Salzer 2019; Luu et al. 2016; Jiang et al. 2018; Feist et al. 2019; Ren et al. 2021; Ghaleb et al. 2023; Chaliasos et al. 2024; Wu et al. 2024) and there exist various tools for their detection, it is unknown whether these tools can perform well when they are applied to detect different types of SR faults. It is important to investigate the existing tools' performance in detecting SR faults as this can shed light on future design or improvement of smart contract security analyzers. To this

end, we investigate 12 state-of-the-art smart contract analyzers to evaluate their detection capabilities for SR faults. We present this study below.

7.1 Tool Selection

To collect representative security analyzers for our study, we conduct a literature review. We follow the widely-adopted guidelines (Brereton et al. 2007) to select the state-of-the-art smart contract analyzers. First, we search papers published in top-tier conferences and journals between 2016 and 2024. Ethereum, the first blockchain platform supporting smart contracts, was launched in July 2015 (Ethereum launches 2015). To capture the earliest academic responses to this technological advancement, we select 2016 as the starting point of our literature review. We extend the review through 2024 to include the most recent research available at the time of this study. Specifically, we use *contract* and *Ethereum* as search keywords and search for publications in all CORE A/A* journals/conferences in software engineering and security fields with research codes: 4612, 4604, and 0803 (Field of research code 2024). We collect 223 research papers as an initial paper list. We then read the abstract of each paper and apply the following exclusion criteria to remove irrelevant papers:

- *Not focusing on the Ethereum blockchain platform.* We exclude papers that target other blockchain platforms, e.g., EOS (Eos platform 2024), Hyperledger Fabric (Hyperledger fabric 2015).
- *Empirical study, literature review, and measurement study papers.* For example, Chen et al. (2020) conduct a comprehensive survey that systematically studies faults, attacks, and defenses of Ethereum systems security. Since such papers do not propose fault detection techniques for Ethereum smart contracts, we consider them out-of-scope.
- *Not aiming to detect smart contract faults.* For example, Hu et al. (2021) propose a technique to automatically generate user notice for Ethereum smart contract functions, which is beyond the research scope of our work.

After applying the exclusion criteria, we have 53 papers closely related to detecting smart contract faults. We then conduct a snowballing process (Wohlin 2014) to ensure that relevant papers and academic tools that might have been missed during the keyword search process are also included in our survey. This yields 21 additional publications related to fault detection for smart contracts. In total, we collect 74 relevant papers.

Our goal is to select tools that are general, popular, and well-maintained. It allows for an unbiased comparison of tool performance across different fault types and ensures that the research conclusions are not overly tailored to a narrow problem domain (e.g., specific SR faults). Based on the publications, we select tools by following these criteria:

- *Available and open-source.* Access to the tool and its source code allows us to independently evaluate the tools and understand their strengths and weaknesses in detecting faults. With this criterion, we exclude 25 tools.
- *No additional input needed.* We focus on tools that accept source code or bytecode directly. This enables a more direct comparison between tools that operate under similar input conditions, ensuring a fair and meaningful evaluation of their capabilities. We exclude tools like Horus (Ferreira Torres et al. 2021) and SmartState (Liao et al. 2023),

which rely on transaction history to identify faults. We also exclude SCType (Zhang 2024), as it requires user annotations as input. Seven tools are excluded based on this criterion.
- *Support detecting at least one SR fault type.* We exclude tools that do not detect SR faults, as our goal is to assess tools' capabilities in identifying such faults. 15 tools are excluded based on this criterion.
- *Support Solidity.* As our dataset only contains contracts in Solidity, only tools that support Solidity are considered. We exclude tools like EOSafe (He et al. 2021) and WASAI (Chen et al. 2022). After this filtering, we retain 27 tools.
- *Well-maintained*. We follow Chaliasos et al.'s definition (Chaliasos et al. 2024) of maintained tools as those that have developer commits within the past year. We exclude 12 inactively maintained tools based on this criterion[1].
- *Popularity and relevance*. In the final step, we consider several factors that reflect the tools' impact, adoption, and relevance in the research community and among practitioners to prioritize tools. Specifically, we investigate ① the number of SR fault types supported by each tool, ② the number of citations on Google Scholar (Google scholar searching engine 2024) (if a related paper exists), ③ the number of GitHub stars the project has received, and ④ the tool's publication year. We give equal weight to these four factors.

Based on the above criteria, we end up with selecting the following tools: ContractFuzzer (Jiang et al. 2018), sFuzz (Nguyen et al. 2020), SmarTian (Choi et al. 2021), Oyente (Luu et al. 2016), Mythril (Mythril 2017), Maian (Nikolić et al. 2018), Manticore (Mossberg et al. 2019), VeriSmart (So et al. 2020), Securify2 (Tsankov et al. 2018), SmartCheck (Tikhomirov et al. 2018), Slither (Feist et al. 2019). These 11 tools fall into three categories: fuzzing, symbolic execution, and static analysis. Additionally, large language models have gained increasing attention in recent years and have been applied to various fault detection tasks. For example, GPTScan (Sun et al. 2024) is a state-of-the-art LLM-based tool that combines large language models with static analysis for smart contract fault detection. As it supports at least one of our identified SR fault types, we also include it in our experiments. Table 6 provides an overview of these tools.

Finding 9 The 12 tools that we study can support detecting only six of the 17 types of SR faults, underscoring significant gaps in fault detection capabilities. While no single tool can support detecting all six types of faults, Manticore, Securify2, Slither, and SmartCheck demonstrate a broader coverage than their peers, each supporting four SR fault types.

As we can see from Table 6, the 12 tools support the detection of six SR fault types. Among them, we observe that Manticore, Manticore, Securify2, Slither, and SmartCheck are capable of detecting most types of faults. Each one can identify four different types. Most tools fail to detect the missing zero address check faults. In fact, the detection of such faults is not complicated in most situations - it requires a check to ensure an address is not zero before a

[1] It is worth noting that we include Oyente, SmartCheck, and ContractFuzzer despite their lack of active maintenance. This is because they are still frequently used in evaluations of recent research (Li et al. 2024; Chaliasos et al. 2024; Wu et al. 2024; Wang et al. 2024).

significant operation, such as an Ether transfer. This underscores a common weakness in the current smart contract security analyzers and is an area for improvement.

7.2 Performance of Existing Tools

To evaluate the tools' performance on detecting the six types of faults in our GitHub dataset, we perform experiments on a server equipped with 176 vCPUs (Intel(R) Xeon(R) Gold 6238 CPU @ 2.10GHz x4) and 251 GB RAM, running Ubuntu 20.04.6 LTS (64-bit). For each tool, we select the latest version (see the "Tool (Version)" column in Table 6) for evaluation and comparison. To ensure that the execution results are fair and stable, we follow the default setting of each tool in our experiments. The detailed setting for the parameters of each tool can be found on our GitHub repository (The material for this study 2024).

Following existing studies (Liu et al. 2019; Wang et al. 2022), we evaluate the *fault detection rate (FDR)* and *false positive rate (FPR)* of the tools:

– ***Fault detection rate (FDR).*** We leverage the buggy contract versions, each containing one SR fault, to evaluate the fault detection rate of a tool t. For each fault, if a tool t correctly identifies the fault in the buggy contract and does not report it in the patched contract, we consider the reported warning a true positive. Otherwise, we consider that the tool fails to detect the fault. We use (1) to calculate the fault detection rate for t.

Table 6 Summary of smart contract security analysis tools

Tool (Version)	Method	# Citations	# Stars	Publication	Fault Type					
					UO	UE	UC	ZA	RE	ES
ContractFuzzer (1.0)	Fuzzing	230	776	ASE'18 (Jiang et al. 2018)					✓	
sFuzz (ce87440)	Fuzzing	269	87	ICSE'20 (Nguyen et al. 2020)	✓				✓	
SmarTian (1.0)	Fuzzing	112	139	ASE'21 (Choi et al. 2021)	✓	✓			✓	
Oyente (9dc0a9)	SE	2508	1.3k	CCS'16 (Luu et al. 2016)	✓	✓			✓	
Mythril (0.24.3)	SE	/	3.8k	White Paper (Mythril 2017)	✓	✓	✓		✓	
Maian (3965e30)	SE	555	742	ACSAC'18 (Nikolić et al. 2018)		✓				
Manticore (0.3.7)	SE	380	3.7k	ASE'19 (Mossberg et al. 2019)	✓	✓	✓		✓	
VeriSmart (36d191e)	SE	91	167	SP'20 (So et al. 2020)	✓	✓				
Securify2 (def1e30)	SA	1058	580	CCS'18 (Tsankov et al. 2018)	✓	✓	✓		✓	
SmartCheck (7b02010)	SA	785	356	WETSEB'18 (Tikhomirov et al. 2018)	✓	✓	✓		✓	
Slither (0.10.0)	SA	577	5.2k	WETSEB'19 (Feist et al. 2019)		✓	✓	✓	✓	
GPTScan	LLM	52	50	ICSE'24 (Sun et al. 2024)						✓

In the "Method" column, "SE" stands for symbolic execution, "SA" for static analysis, and "LLM" for LLM-based technique.

– ***False positive rate (FPR).*** We leverage the patched contract versions to evaluate the false positive rate of each tool. For each fault, if a tool t reports a warning of the fault when analyzing the patched contract version, we consider the warning as a false positive. We use (2) to calculate the false positive rate for t.

$$FDR(t) = \frac{\text{\# True positives reported by t on buggy contract versions}}{\text{\# Buggy contracts experimented on t}} \quad (1)$$

$$FPR(t) = \frac{\text{\# False positives reported by t on patched contract versions}}{\text{\# Patched contracts experimented on t}} \quad (2)$$

Finding 10 Existing security analyzers show very limited capability in detecting SR faults, collectively achieving a fault detection rate of 14.4%. Slither has the highest detection rate, while nine tools fail to detect any faults.

Table 7 shows the fault detection rate of each tool (the second last column). The table shows significant variability in the performance of the 12 evaluated tools, with many falling short of expectations. Among the tools analyzed, Slither stands out, detecting the highest number of issues across multiple fault types, leading to a fault detection rate (FDR) of 15.3%. SmartCheck detects 7 faults in the category unchecked call return value (UC), showing a FDR of 3.7%. SmarTian detects 5 faults in the categories of unchecked call return value (UC) and unprotected Ether withdrawal (UE), showing an FDR of 3.3%. Other conventional tools like sFuzz and Oyente fail to detect any faults in our dataset. The LLM-based tool GPTScan, which in theory supports the detection of excessive slippage faults, fails to detect any case of the 19 excessive slippage faults in our dataset. Overall, the tool achieves a detection rate of 14.4% (35 out of 243), which falls short of meeting the stringent security needs of smart contracts.

Finding 11 The 12 evaluated tools exhibit low false positive rates (1.6%) when identifying SR faults. Their poor performance is mainly due to the substantial false negatives.

Table 7 also shows the false positive rate of each tool (the last column). As we can see, although these tools exhibit weak detection capabilities across all categories of SR faults, they have a relatively low false positive rate (4 out of 243, 1.6%). We observe only four cases of false positives from the detected SR faults, which are due to the over-approximation of Slither in the detection of reentrancy. Slither reports a contract exposing reentrancy once an event is emitted after an external call (Detection documentation of reentrancy vulnerabilities by slither 2023). The heuristic can lead to false positives. Listing 10 shows an example of the false positive cases reported by Slither. In the patched version, although there is already an `nonReentrant` modifier in line 10, which contains a `require()` statement (line 3) to restrict the number of function callers at one time to prevent the reentrancy exploit, Slither still mistakenly considers it as vulnerable to reentrancy. The reason is that in the patched contract, there is still an event emitted (line 17) after an external call (lines 12-16). Therefore, Slither reports a false positive for such a case.

Table 7 Summary of tool execution results

Tool	Supported Faults	Fault Type						FP	FDR	FPR
		UO	UE	UC	ZA	RE	ES			
ContractFuzzer	27	-	-	-	-	0	-	0	0.0%	0.0%
sFuzz	94	0	-	-	-	0	-	0	0.0%	0.0%
SmarTian	153	3	2	-	-	0	-	0	3.3%	0.0%
Oyente	153	0	0	-	-	0	-	0	0.0%	0.0%
Mythril	191	0	0	0	-	0	-	0	0.0%	0.0%
Maian	59	-	0	-	-	-	-	0	0.0%	0.0%
Manticore	191	0	0	0	-	0	-	0	0.0%	0.0%
VeriSmart	126	0	0	-	-	-	-	0	0.0%	0.0%
Securify2	191	0	0	0	-	0	-	0	0.0%	0.0%
SmartCheck	191	0	0	7	-	0	-	0	3.7%	0.0%
Slither	157	-	6	5	5	8	-	4	15.3%	2.5%
GPTScan	19	-	-	-	-	-	0	0	0.0%	0.0%
Total	243	3/67	8/59	11/38	5/33	8/27	0/19	4	14.4%	1.6%

"FP" means the number of false positives

7.3 Tool Deficiencies Analysis

We observe that the tools generate high false negatives, indicating that they are far from being practically useful for combating SR faults. To facilitate future research, we further investigate the reasons behind the low fault detection rate.

We first identify and analyze the situations where the selected tools fail to analyze the subject contracts or generate reports. Following the practices of existing work (Li et al. 2024), we investigate the number of contracts where the tool fails due to exceeding a time limit and the number of contracts that could not be scanned due to compilation errors.

```
1  contract Loan{
2      bool private _notEntered = true;
3      modifier nonReentrant() {
4          require(_notEntered);
5          _notEntered = false;
6          _;
7          _notEntered = true;
8      }
9      function acquireLoan(BaseWallet _wallet, bytes32 _loanId)
10 +         nonReentrant {
11         require(cdpManager.owns(uint256(_loanId)) == address(_wallet));
12         invokeWallet(address(_wallet), address(cdpManager),
13             abi.encodeWithSignature("give(uint256,address)", uint256(_loanId), address(this))
14         ); // External call
15     }
16 }
```

Listing 10: An example of a cross-contract reentrancy fault

Finding 12 Timeout (24.4%) and compilation errors (33.8%) hinder the effective contract analysis of the tools.

As demonstrated in Table 8, in total, only 41.8% (648 out of 1,552) of the contracts can be executed successfully. Among the tools evaluated, GPTScan and VeriSmart perform best, with success rates of 100.0% and 73.8%, respectively. Compared to dynamic execution and static analysis tools, the LLM-based tool GPTScan excels with its freedom from timeout issues and ability to operate without compilation. However, tools like sFuzz and Securify2

Table 8 Analysis result of each tool

Tool	Successfully	Execution Failure		TP	FN	FDR	Tool Limitation		
	Analyzed	Timeout	Compilation				IP	CC	PP
ContractFuzzer	9/27	10	8	0	9	0.00%	7	-	2
sFuzz	0/94	36	58	0	0	N/A	0	-	-
SmarTian	43/153	59	51	5	38	11.6%	12	-	26
Oyente	62/153	41	50	0	62	0.00%	-	47	15
Mythril	94/191	60	37	0	94	0.00%	-	53	41
Maian	44/59	3	12	0	44	0.00%	-	12	32
Manticore	64/191	87	40	0	64	0.00%	-	0	64
VeriSmart	93/126	0	33	0	93	0.00%	-	55	38
Securify2	6/191	69	116	0	6	0.00%	-	0	6
Slither	92/157	14	51	24	68	26.1%	-	41	27
SmartCheck	122/191	0	69	7	115	5.7%	-	67	48
GPTScan	19/19	0	0	0	19	0.00%	-	-	19

“Timeout” refers to contracts where the tool fails due to exceeding the time limit. “Compilation” refers to the number of contracts that could not be scanned because of compilation errors. “TP” indicates the number of true positives. “FN” indicates the number of false negatives. “FDR” is calculated by dividing the number of true positives by the number of SR faults in successfully analyzed contracts. 2. “IP”, “CC”, and “PP” stand for Inefficient Path Exploration, Lack of Support for Cross-Contract Analysis, and Over Reliance on Unsound Pre-defined Patterns

significantly underperform, with sFuzz failing to scan any contract successfully and Securify2 successfully scanning only 6 out of 191 contracts. Compilation errors and timeouts are major reasons of these failures. Compilation errors pose a significant challenge, especially for Securify2, which fail to compile 116 contracts, severely limiting its effectiveness. Timeout issues are particularly prevalent in tools like Manticore (87 timeouts) and SmarTian (59 timeouts). Recent empirical studies corroborate these findings (Li et al. 2024; Wu et al. 2024).

The compilation process, a prerequisite for contract analysis, is a common point of failure. We observe that several tools are tightly coupled to specific Solidity compiler versions (Li et al. 2024), leading to incompatibilities with contracts developed using different Solidity versions. For instance, Oyente is unable to analyze smart contracts with Solidity versions higher than v0.4.19, while Securify2 is limited to Solidity versions between 0.5.x and v0.6.x. Manticore fails to provide full support for certain opcodes with a Solidity version greater than 0.4.x. Consequently, the limited support for Solidity versions leads to compilation issues, preventing the tools from successfully scanning the contracts. Additionally, some tools face challenges with intermediate representation compatibility. For example, Securify2, which relies on the SlitherIR intermediate representation, faces challenges when converting EVM bytecode to the static single assignment (SSA) form of SlitherIR.

Timeout is another significant issue. We use the default settings for each tool in our experiment, and some tools like Manticore and SmarTian, frequently encounter timeouts. For example, Manticore’s comprehensive emulation of the Ethereum environment, while allowing for complex contract interactions, significantly increases computational overhead. This results in a higher likelihood of timeouts compared to tools with more streamlined execution models.

To isolate and analyze tool failures due to inherent algorithmic limitations in fault detection rather than simple compatibility issues, we calculate the Fault Detection Rate (FDR)

(as shown in Table 8) based only on successfully analyzed contracts. This metric helps us understand each tool's true detection capabilities and algorithmic deficiencies. Our analysis revealed obvious variations in fault detection effectiveness across different tools. Slither demonstrated strong detection capabilities with a 26.1% FDR (24 true positives from 92 analyzed contracts), showcasing the efficacy of its pattern-based analysis. SmartCheck showed moderate effectiveness with an 11.6% FDR (5 true positives from 43 contracts). Oyente, Mythril, and Manticore achieved 0% FDR, suggesting fundamental limitations in their detection algorithms rather than compatibility issues. Similarly, the LLM-based GPTScan analyzed all 19 contracts but detected no faults (0% FDR). These findings indicate that the primary challenge for most tools is their core detection mechanisms rather than compatibility constraints.

Finding 13 Existing analyzers produce a significant number of false negatives when detecting SR faults. The major limitations include inefficient path exploration (especially for fuzzing techniques), lack of support for cross-contract analysis, and over reliance on predefined patterns.

To understand why existing techniques perform poorly, we analyze all false negative (FN) cases for each tool. Two authors independently examine each case by reviewing the smart contract code, investigating the specific fault, and studying the tool's detection logic and documentation where available. A third author resolves any disagreements between the two authors. Through this analysis, we identify three key limitations that affect the tools' ability to detect faults: inefficient path exploration, the lack of support for cross-contract analysis, and the reliance on unsound pre-defined patterns. Table 8 shows the number of FN cases for each tool due to different limitations.

Inefficient path exploration Many SR faults, such as overflow/underflow (UO) and excessive slippage (ES), require specific conditions or transaction sequences to manifest. For instance, an overflow/underflow fault might occur only when a variable exceeds its type boundary after a series of arithmetic operations (e.g., Listing 11, where `performReputationCalculation` requires preconditions in prior functions to be satisfied). Fuzzing tools like sFuzz and SmarTian rely on random or heuristic input generation, which often fails to explore deep or conditional execution paths necessary to trigger these faults. These fuzzing tools typically lack systematic path coverage due to their reliance on random input generation, which is inefficient for contracts with complex logic or multi-transaction dependencies. Besides, the generated test inputs may not be effective enough to trigger the fault locations due to hard-to-satisfy preconditions. Consequently, many SR faults remain undetected because fuzzing tools are unable to systematically navigate through the intricate and conditional paths within the contract's execution flow. As the faults are commonly related to extreme value checks of variables and Ether transfer as discussed in Section 5, there is often a set of preconditions (e.g., numerical operations of state variables, checks of the identity of function callers) before reaching the vulnerable locations. For example, in a contract in Listing 11, to trigger an integer overflow/underflow within the `performReputationCalculation` function, the analyzer needs to satisfy a set of data constraints in the three functions preceding the `performReputationCalculation` function (i.e., the `require()` statements at lines 15-17, 21, and 25), which cannot be satisfied by SmarTian.

```
1  contract ReputationMiningCycle{
2      function respondToChallenge() public{
3          checkKey();
4          proveBeforeReputationValue();
5          proveAfterReputationValue();
6          performReputationCalculation();
7          ...
8      }
9      function performReputationCalculation() internal{
10 +        if (amount + agreeValue < agreeValue) {
11 +            require(disagreeValue == 2**256 - 1);}
12         ...
13     }
14     function checkkey() internal{
15         require(reputationLog[number].user == userAddress);
16         require(reputationLog[number].colony == colonyAddress);
17         require(reputationLog[number].skillId == skillId);
18         ...
19     }
20     function proveBeforeReputationValue() internal {
21         require(impliedRoot == jrh);
22         ...
23     }
24     function proveAfterReputationValue() internal {
25         require(jrh==impliedRoot);
26         ...
27     }
28 }
```

Listing 11: A complicated data constraint example

To address this limitation, targeted fuzzing strategies can be employed (Olsthoorn et al. 2022). These strategies focus on known critical paths and use constraint-based fuzzing to generate inputs that are more likely to trigger state-reverting scenarios. Fuzzing can also be combined with symbolic execution for systematic path exploration. Symbolic execution treats variables as symbolic expressions and solves constraints to reach deep code locations (e.g., `performReputationCalculation`). To reduce its high computational cost, optimizations like path pruning (discarding infeasible paths) and state merging (combining redundant states) can be applied. Another potential improvement direction is feedback-driven input refinement. This involves implementing adaptive fuzzing with runtime feedback through code coverage or variable value monitoring. For UO, inputs that push variables toward type boundaries are prioritized. For ES, inputs are adjusted based on slippage-related metrics (e.g., price deltas). Techniques like grey-box fuzzing (e.g., AFL-style Afl fuzzer 2013) can iteratively refine inputs, increasing the likelihood of triggering conditional faults without exhaustive enumeration.

Lack of support for cross-contract analysis SR faults such as reentrancy (RE), unchecked call return value (UC), and lack of property check for external call (LP) often arise from interactions between contracts, where the state or behavior of one contract can influence the execution path of another. A classic example is reentrancy, where a malicious contract can repeatedly call back into the vulnerable contract before state updates are complete. While tools like Slither and Oyente can analyze individual contracts effectively, they struggle with cross-contract vulnerabilities due to their isolated analysis approach. This limitation creates a significant security gap, as they cannot detect issues that arise from complex interactions between multiple contracts. To illustrate this, consider the `acquireLoan` function that calls an external `give` function in Listing 10. Popular analysis tools like Oyente, Mythril, and Slither miss potential vulnerabilities in this scenario due to their lack of inter-procedural analysis capabilities. In contrast, tools like Manticore and Securify2 take different approaches to address this challenge. Manticore provides comprehensive analysis by simu-

lating the entire Ethereum environment, while Securify2 treats all external calls as potentially malicious.

The root causes are twofold. First, cross-contract analysis presents significant computational challenges. Tools must model complex interactions, including inter-contract call graphs, state dependencies, and execution paths. In blockchain systems like Ethereum, where contracts can interact with any address, the complexity grows exponentially. While tools like Manticore attempt to address this through full environmental symbolic execution, their performance overhead makes them impractical for analyzing large-scale systems. Second, most analysis tools (including Slither, Oyente, and Mythril) focus on single-contract analysis to maintain efficiency. Instead of tracking state across contract boundaries, they rely on static pattern matching or limited symbolic execution within individual contracts. In contrast, Securify2 takes a conservative approach, treating all external calls as potentially malicious. This improves detection of SR faults but increases false positives and reduces precision. These different approaches show how tools must balance comprehensive analysis against practical usability.

To address the limitation, future research can consider extending symbolic execution engines for inter-contract modeling. This involves enhancing symbolic execution tools (e.g., Oyente, Mythril) to model inter-contract calls and track state dependencies across contracts. To mitigate computational intensity, employ path pruning techniques (e.g., based on call depth or relevance to SR statements) and limit analysis to critical external calls identified via static pre-analysis. Another potential improvement direction is a hybrid approach that combines heuristic analysis with dynamic verification would provide more robust results. This method starts with static pattern detection to identify potential vulnerabilities, followed by targeted dynamic analysis through fuzzing or symbolic execution to verify findings. This layered approach balances efficiency with accuracy in detecting cross-contract vulnerabilities.

Over reliance on pre-defined patterns Existing tools that adopt heuristic methods rely on pre-defined patterns to identify faults. However, these patterns often fail to capture the complexities of smart contract interactions, especially in sophisticated contracts with multiple parties or intricate logic. For example, tools like Slither and Mythril use static analysis with pre-defined patterns to detect Unprotected Ether Withdrawal (UE), such as missing `onlyOwner` modifiers. However, these tools do not explicitly model token-related logic in their vulnerability patterns. For instance, a token transfer function without access control might go undetected if the pattern does not account for token-specific operations (e.g., ERC-20 `transfer`). This limitation stems from an oversimplified heuristic that overlooks domain-specific behaviors, leaving UE faults in token-based contracts hidden.

GPTScan also suffers from over-reliance on predefined patterns. To mitigate hallucinations in large language models, it implements overly strict pre-defined patterns to identify vulnerable functions. For example, it only selects functions containing specific terms like “calculateSwap”, “calculateLiquidity”, “addLiquidity”, “removeLiquidity”, or “slipLimit” for further verification of excessive slippage faults. This restrictive approach leads to a high false negative rate. Securify2 detects unprotected Ether withdrawal faults by checking if the `call` function, which transfers non-zero Ether, can be executed independently of the caller

(msg.sender), meaning there's no precheck on the msg.sender before transfer operations. However, in the buggy version of the example shown in Listing 12, the execution of the call function (line 8) requires a precheck on the msg.sender's account balance (line 3). That is, the execution of the call function depends on the function caller. This scenario contradicts Securify2's model of unprotected Ether withdrawal faults, which requires the call function to be independent of the caller (msg.sender). Consequently, the fault goes undetected.

Besides, inaccurate modeling of code elements in heuristic patterns can also result in false negatives when detecting faults. When these elements are not modeled correctly, the system fails to identify certain faults. For instance, our examination of sFuzz's source code reveals that it monitors only additions and subtractions for integer overflows while ignoring multiplications. As a result, sFuzz cannot detect faults that involve multiplication operations.

Therefore, code elements related to faults must be accurately and comprehensively modeled to enable effective fault detection. Addressing over-reliance on pre-defined patterns requires a shift toward adaptive, context-aware analysis. Tools could integrate domain-specific knowledge (e.g., token standards for Unprotected Ether Withdrawal, DeFi patterns for excessive slippage) or employ dynamic techniques like symbolic execution (e.g., Manticore) to explore beyond static heuristics. While these improvements face scaling challenges, they highlight the importance of systematic research to understand how faults evolve, bridging the gap between heuristic simplicity and real-world complexity.

```
1  contract KeepBonding{
2      function withdraw(uint256 amount, address operator) public{
3  -       require(unbondedValue[msg.sender] >= amount);
4  -       unbondedValue[msg.sender] -= amount;
5  +       require(unbondedValue[operator] >= amount);
6  +       require(msg.sender == operator || msg.sender == tokenStaking.ownerOf(operator));
7  +       unbondedValue[operator] -= amount;
8          (bool success, ) = tokenStaking.magpieOf(operator).call.value(amount)("");
9          require(success);
10     }
11 }
```

Listing 12: A missing modeling of token feature example

Answer to RQ5: *Our evaluation reveals that existing tools have very limited support for detecting SR faults. They encounter serious usability issues (timeout and compilation errors) and exhibit substantial false negatives when applied to analyze real-world SR faults. The primary reasons for the false negatives include inefficient path exploration, lack of cross-contract analysis, and over reliance on pre-defined patterns.*

Implication: *The limited detection capabilities and high false negative rates highlight the need for more advanced security analysis tools specifically tailored for SR faults. Future research should focus on developing tools with enhanced path exploration ability, support for cross-contract analysis, and robust handling of complex contract structures to improve SR fault detection and mitigate security risks in smart contracts.*

8 Threats to Validity

The validity of our study results may be subject to several potential threats, which we have carefully considered:

Subjectivity of researchers A primary threat to validity lies in our manual classification and labeling of purposes of using SR statements and the category of SR faults, which could be subject to bias or errors. To mitigate this threat, we invite three individuals with over four years of research experience in the field of smart contracts to conduct the manual check. During the labeling process, two authors independently label buggy-patched contracts for two rounds, and the third author is involved in resolving the conflicts. Any disagreements are discussed during the process until a consensus is reached. Further details about our classification and labeling process can be found in Section 4 and Section 5.1.3. Besides, our study provides a comprehensive taxonomy of SR faults and fixing strategies, but it does not evaluate the effectiveness of these strategies. Future research should focus on assessing the robustness and optimality of these approaches, potentially developing guidelines or automated tools that recommend effective fixes for specific fault types.

Selection and settings of security analysis tools Another threat to validity lies in the tools used in the evaluation section. To mitigate potential selection bias, we conduct a comprehensive review of state-of-the-art tools. We consider multiple factors, such as the number of supported vulnerability types, the impact of the paper and the GitHub project, and so on, as outlined in Section 7.1. We select 12 popular, relevant, and available open-source tools for evaluation. Due to the limited capabilities of existing tools, we are only able to perform tool evaluations using 6 of the 17 identified SR fault types. Future research may focus on designing automated solutions to detect the remaining 11 fault types via leveraging SR-specific rules or techniques such as symbolic execution, allowing detection across the complete SR fault taxonomy. Empirical studies can then make broader contributions by covering more types of SR faults and a wider range of tools. The setup of these tools presents another potential threat. To minimize execution bias, we follow the documentation and default settings used in the tool papers. When execution issues arise, we actively communicate with the tool designers to seek solutions.

Selection of data sources One threat to RQ1 is the lack of Solidity version stratification in our longitudinal analysis. Our dataset includes 1% of contracts written in pre-0.4.10 Solidity versions that use only `if...throw` for state reverting. This version distribution shift between 2021 and 2024 may have influenced the observed decline in SR statement usage. Another potential threat to RQ2 lies in the use of random sampling to select the 381 SR statements for manual analysis. While this approach ensures statistical representativeness with a 9% confidence level and 5% margin of error, it may not guarantee proportional representation across smart contract application domains. Consequently, our findings on SR statement purposes could overlook domain-specific nuances in less common categories. Nevertheless, our current strategy adheres to established methodological standards, which help provide a robust foundation for our findings. The sample size of 381 SR statements ensures a high degree of statistical confidence, minimizing the likelihood that major patterns of usage were missed. Future studies could consider using stratified sampling to provide a more balanced representation of SR statement usage across diverse contract types. For RQ2, we do not explore the relationship between SR statement purposes and contract complexity. The complexity of contracts may affect how SR statements are used for different purposes. While our study provides a broad taxonomy based on a large and diverse dataset, future research could investigate this aspect to better understand SR statement usage patterns.

Another potential threat of our study lies in the dataset's representativeness. Our primary data source consists of faults from the top 1,000 most-starred Ethereum smart contract projects on GitHub. While these popular projects provide valuable insights, they may underrepresent beginner-level mistakes due to their exposure to community scrutiny and testing. To address this threat and ensure broader coverage, we supplement our dataset with 42 real-world smart contract faults identified through Code4rena audit reports. This dual-source approach combines both popular open-source projects and professional audit findings, providing a more comprehensive view of common vulnerabilities. However, future research could further enhance representativeness by including samples from less popular repositories to capture a wider spectrum of development expertise.

For RQ3, we rely on Code4rena's severity classifications for SR faults without assessing inter-rater reliability. More specifically, although Code4rena uses a multi-auditor review process, we have not independently verified the consistency of these classifications due to the lack of available data. Future research could address this by conducting an inter-rater reliability analysis or comparing Code4rena's ratings against other audit platforms to verify their reliability.

9 Conclusion

In this work, we present the first comprehensive study on SR statements and SR faults in Solidity smart contracts. Through an intensive analysis of 21,414 real-world smart contracts, we demonstrate that SR statements are prevalent. They are commonly used to check the runtime status of smart contracts against security-critical constraints. We thoroughly examine 320 real-world SR faults in smart contracts, categorizing them into 17 fault types for a nuanced understanding. We also analyze various strategies for fixing these faults, identifying 12 distinct approaches. To evaluate the detection capabilities of existing tools, we assess 12 state-of-the-art security analyzers designed to identify machine-auditable faults, revealing their weaknesses in analyzing SR faults. Our research not only offers valuable insights for future research on state-reverting faults and smart contract security but also provides practical guidance for smart contract developers and tool designers.

Author Contributions All authors contributed to the conception and design of the study, data collection, analysis, and interpretation of results. All authors reviewed and approved the final manuscript.

Funding This research is supported by the Hong Kong RGC/GRF (Grant No. 16205821) and the National Natural Science Foundation of China (Grant No. 62372219).

Data Availibility Statement All artifacts related to this study are available on GitHub (The material for this study 2024).

Declarations

Ethical approval Not applicable.

Informed consent Not applicable.

Conflict of Interest The authors declare that they have no conflict of interest.

Clinical Trial Number Not applicable.

References

Afl fuzzer (2013) https://lcamtuf.coredump.cx/afl/technical_details.txt

An example of fixing access control faults by adding a blacklist address check. (2023) https://github.com/thundercore/referral-solidity/commit/a44f0e7728974435b63c8b8ef9e323243f006884

An example of fixing others faults by adding a timestamp check. (2018) https://github.com/blockchain-IoT/Motoro/commit/db5e43f30b1f51d47f193ab32fb9f9da2e4141c9

An temporal property violation example (2024) https://github.com/delvtech/council/commit/861e7600bcd5fe3cacca4f84e9f25c72f9c21075

Atzei N, Bartoletti M, Cimoli T (2017) A survey of attacks on ethereum smart contracts (sok). In Matteo Maffei and Mark Ryan, editors, Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, volume 10204 of Lecture Notes in Computer Science, Springer, pages 164–186. https://doi.org/10.1007/978-3-662-54455-6_8

Brereton P, Kitchenham BA, Budgen D, Turner M, Khalil M (2007) Lessons from applying the systematic literature review process within the software engineering domain. J Syst Softw 80(4):571–583. https://doi.org/10.1016/j.jss.2006.07.009

Chaliasos S, Charalambous MA, Zhou L, Galanopoulou R, Gervais A, Mitropoulos D, Livshits B (2024) Smart contract and defi security tools: Do they meet the needs of practitioners? In: Proceedings of the 46th IEEE/ACM international conference on software engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024, ACM, pages 60:1–60:13. https://doi.org/10.1145/3597503.3623302

Chaliasos S, Charalambous MA, Zhou L, Galanopoulou R, Gervais A, Mitropoulos D, Livshits B (2024) Smart contract and defi security: Insights from tool evaluations and practitioner surveys

Checks-effects-interactions pattern (2024) https://github.com/fravoll/solidity-patterns/blob/master/docs/checks_effects_interactions.md#checks-effects-interactions

Chen H, Pendleton M, Njilla L, Xu S (2020) A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. ACM Comput Surv (CSUR) 53(3):1–43

Chen J, Xia X, Lo D, Grundy J, Luo X, Chen T (2020) Defining smart contract defects on ethereum. IEEE Trans Softw Eng 48(1):327–345

Chen Q, Xia X, Hu H, Lo D, Li S (2021) Why my code summarization model does not work: Code comment improvement with category prediction. ACM Trans Softw Eng Methodol (TOSEM) 30(2):1–29

Cheng Y, Yang H, Li Z, Tian L (2024) Characterizing, detecting, and correcting comment errors in smart contract functions. In: 2024 IEEE International conference on software services engineering (SSE), IEEE, pages 282–292

Chen J, Shao Z, Yang S, Shen Y, Wang Y, Chen T, Shan Z, Zheng Z (2025) Numscout: Unveiling numerical defects in smart contracts using llm-pruning symbolic execution. IEEE Trans Softw Eng

Chen W, Sun Z, Wang H, Luo X, Cai H, Wu L (2022) Wasai: uncovering vulnerabilities in wasm smart contracts. In: Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis, pages 703–715

Choi J, Kim D, Kim S, Grieco G, Groce A, Cha SK (2021) Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In: 2021 36th IEEE/ACM international conference on automated software engineering (ASE), IEEE, pages 227–239

Code4rena (2024) https://code4rena.com

Code4rena github repository (2024) https://github.com/code-423n

Cohen J (1960) A coefficient of agreement for nominal scales. Educ Psychol Measurement 20(1):37–46

Customization and dynamic structuring of smart contracts in solidity (2023). https://medium.com/@solidity101/ customization-and-dynamic-structuring-of-smart-contracts-in-solidity-d429da7d386e

Defi money market compound overpays millions in comp rewards in possible exploit (2021). https://www.coindesk.com/tech/2021/09/30/defi-money-market-compound-overpays-15m-in-comp-rewards-in-possible-exploit/

Detection documentation of reentrancy vulnerabilities by slither (2023) https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3

Di Angelo M, Salzer G (2019) A survey of tools for analyzing ethereum smart contracts. In: 2019 IEEE international conference on decentralized applications and infrastructures (DAPPCON), IEEE, pages 69–78

Durieux T, Ferreira JF, Abreu R, Cruz P (2020) Empirical review of automated analysis tools on 47, 587 Ethereum smart contracts ICSE ’20: 42nd International Conference on Software Engineering Seoul South Korea 27 June 19 July 2020 João F. Ferreira and Rui Abreu and Pedro Cruz, Gregg Rothermel and Doo-Hwan Bae. pages 530–541. ACM. https://doi.org/10.1145/3377811.3380364
Eea ethtrust security levels specification (2024) https://entethalliance.org/specs/ethtrust-sl/v2/
Eos platform (2024) https://eos.com/platform/
Eshghie M, Åryd V, Artho C, Monperrus M (2024) Solidiffy: Ast differencing for solidity smart contracts. arXiv:2411.07718
Ethereum daily transactions chart (2024) https://etherscan.io/chart/tx
Ethereum improvement proposal eip-140: Revert instruction (2024) https://eips.ethereum.org/EIPS/eip-140
Ethereum launches (2015) https://blog.ethereum.org/2015/07/30/ethereum-launches
Ethereum transactions (2024) https://ethereum.org/en/developers/docs/transactions/
Ethereum yellowpaper (2024) https://ethereum.github.io/yellowpaper/paper.pdf
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, pages 8–15
Ferreira Torres C, Iannillo AK, Gervais A, State R (2021) The eye of horus: Spotting and analyzing attacks on ethereum smart contracts. In: International conference on financial cryptography and data security, Springer, pages 33–52
Field of research code (2024) https://researchonline.jcu.edu.au/view/subjects/subjects.html
Ghaleb A, Pattabiraman K (2020) How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In: Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, pages 415–427
Ghaleb A, Rubin J, Pattabiraman K (2022) etainter: detecting gas-related vulnerabilities in smart contracts. In: Proceedings of the 31st ACM SIGSOFT International symposium on software testing and analysis, pages 728–739
Ghaleb A, Rubin J, Pattabiraman K (2023) Achecker: Statically detecting smart contract access control vulnerabilities. In: 45th IEEE/ACM International conference on software engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023, IEEE, pages 945–956. https://doi.org/10.1109/ICSE48619.2023.00087
Github (2024) https://github.com/
Github rest api (2022) https://docs.github.com/en/rest?apiVersion=2022-11-28
Google scholar searching engine (2024) https://scholar.google.com/
Grech N, Kong M, Jurisevic A, Brent L, Scholz B, Smaragdakis Y (2018) Madmax: Surviving out-of-gas conditions in ethereum smart contracts. Proceed ACM Program Lang 2(OOPSLA):1–27
Harty J, Zhang H, Wei L, Pascarella L, Aniche M, Shang W (2021) Logging practices with mobile analytics: An empirical study on firebase. arXiv:2104.02513
Harz D, Knottenbelt W (2018) Towards safer smart contracts: A survey of languages and verification methods. arXiv:1809.09805
Hegedűs, P (2018) Towards analyzing the complexity landscape of solidity based ethereum smart contracts. In: Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain (pp 35–39)
He N, Zhang R, Wang H, Wu L, Luo X, Guo Y, Yu T, Jiang X (2021) {EOSAFE}: security analysis of {EOSIO} smart contracts. In: 30th USENIX security symposium (USENIX Security 21), pages 1271–1288
Hu X, Gao Z, Xia X, Lo D, Yang X (2021) Automating user notice generation for smart contract functions. In: 36th IEEE/ACM International conference on automated software engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021, IEEE, pages 5–17. https://doi.org/10.1109/ASE51524.2021.9678552
Hyperledger fabric (2015) https://www.hyperledger.org/use/fabric
Jiang B, Liu Y, Chan WK (2018) Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, pages 259–269
Junk T (1999) Confidence level computation for combining searches with small statistics. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 434(2–3):435–443
Liao Z, Hao S, Nan Y, Zheng Z (2023) Smartstate: Detecting state-reverting vulnerabilities in smart contracts via fine-grained state-dependency analysis. In: Proceedings of the 32nd ACM SIGSOFT International symposium on software testing and analysis, pages 980–991
Li Z, Li W, Li X, Zhang Y (2024a) Guardians of the ledger: Protecting decentralized exchanges from state derailment defects. IEEE Trans Reliability
Li Z, Li W, Li X, Zhang Y (2024b) Stateguard: Detecting state derailment defects in decentralized exchange smart contract. In: Companion Proceedings of the ACM Web Conference 2024, pages 810–813, 2024

Liu Y, Wang J, Wei L, Xu C, Cheung SC, Wu T, Yan J, Zhang J (2019) Droidleaks: a comprehensive database of resource leaks in android apps. Empir Softw Eng 24(6):3435–3483. https://doi.org/10.1007/s10664-019-09715-8

Liu L, Wei L, Zhang W, Wen M, Liu Y, Cheung SC (2021) Characterizing transaction-reverting statements in ethereum smart contracts. In: 36th IEEE/ACM International Conference on automated software engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021, IEEE, pages 630–641. https://doi.org/10.1109/ASE51524.2021.9678597

Li K, Xue Y, Chen S, Liu H, Sun K, Hu M, Wang H, Liu Y, Chen Y (2024) Static application security testing (sast) tools for smart contracts: How far are we? Proceed ACM Softw Eng 1(FSE):1447–1470

Luu L, Chu DH, Olickel H, Saxena P, Hobor A (2016) Making smart contracts smarter. In: Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016, pages 254–269. ACM. https://doi.org/10.1145/2976749.2978309

Mitropoulos C, Kechagia M, Maschas C, Ioannidis S, Sarro F, Mitropoulos D (2024) Charting the evolution of solidity error handling. arXiv:2402.03186

Modularity, the way forward to building defi systems (2023). https://www.euler.finance/blog/modularity-the-way-forward-to-building-defi-systems

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

Mythril (2017) https://github.com/ConsenSys/mythril

Nassirzadeh B, Sun H, Banescu S, Ganesh V (2022) Gas gauge: A security analysis tool for smart contract out-of-gas vulnerabilities. In: The International Conference on mathematical research for blockchain economy, pages 143–167. Springer

Nguyen TD, Pham LH, Sun J, Lin Y, Minh QT (2020) sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, pages 778–788

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

Number rounding protection example (2021) https://github.com/delvtech/elf-contracts/commit/cb909af9b9843abd477980ba6023fb97e053005c

Olsthoorn M, van Deursen A, Panichella A (2022) Guiding automated test case generation for transaction-reverting statements in smart contracts. In: 2022 IEEE International conference on software maintenance and evolution (ICSME), IEEE, pages 163–174

Openzeppelin (2024) https://docs.openzeppelin.com/openzeppelin/

Owasp smart contract top 10 (2023) https://owasp.org/www-project-smart-contract-top-10/

Pearson K (1920) Notes on the history of correlation. Biometrika 13(1):25–45

Property check for external call example (2021) https://github.com/omni/tokenbridge-contracts/commit/1166e00c5b976295914d625504f3b53b7bfc945c

Ren M, Yin Z, Ma F, Xu Z, Jiang Y, Sun C, Li H, Cai, Y (2021) 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, pages 566–579

Seaman CB (1999) Qualitative methods in empirical studies of software engineering. IEEE Trans Softw Eng 25(4):557–572

Slippage protection example (2022) https://github.com/Equalizer-Finance/equalizer-smart-contracts-v1/commit/10a9fcf30f13ac2bdeb7dfaf87665426844128d4

Smtchecker and formal verification (2024) https://docs.soliditylang.org/en/latest/smtchecker.html

So S, Lee M, Park J, Lee H, Oh H (2020) Verismart: A highly precise safety verifier for ethereum smart contracts. In: 2020 IEEE Symposium on security and privacy (SP), IEEE, pages 1678–1694

Solidity documentation (2024) https://docs.soliditylang.org/en/v0.8.25/

Solidity error handling: Assert, require, revert and exceptions (2024) https://docs.soliditylang.org/en/latest/control-structures.html

Solidity v0.8.0 breaking changes (2020) https://docs.soliditylang.org/en/latest/080-breaking-changes.html

Solidity-parser 0.1.1, (2024). https://pypi.org/project/solidity-parser/

Spearman C (1961) The proof and measurement of association between two things

Stackexchange website (2024) https://stackexchange.com/

Stol KJ, Ralph P, Fitzgerald B (2016) Grounded theory in software engineering research: a critical review and guidelines. In: Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, pages 120–131. ACM. https://doi.org/10.1145/2884781.2884833

Sun Y, Wu D, Xue Y, Liu H, Wang H, Xu Z, Xie X, Liu Y (2024) Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24, New York, NY, USA. Association for Computing Machinery. https://doi.org/10.1145/3597503.3639117

Swc 101: Integer overflow and underflow (2024) https://swcregistry.io/docs/SWC-101

Swc 104: Unchecked call return value (2024) https://swcregistry.io/docs/SWC-104

Swc 105: Unprotected ether withdrawal (2024) https://swcregistry.io/docs/SWC-105

Swc 107: Reentrancy (2024) https://swcregistry.io/docs/SWC-107

Swc 123: Requirement violation (2024) https://swcregistry.io/docs/SWC-123

Swc registry (2024) https://swcregistry.io/

The material for this study (2024) https://github.com/Liuluuuu/SRFaults

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, pages 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, pages 67–82

Understanding the dao attack (2023) https://www.coindesk.com/learn/understanding-the-dao-attack/

Uniswap v2 router (2024) https://docs.uniswap.org/contracts/v2/reference/smart-contracts/router-02

Unprotected ether withdrawal exapmle (2021) https://github.com/tatumio/smart-contracts/commit/3cbf025e6c468822821dcf443300981850c1abd4

Wang Y, Wang Y, Wang S, Liu Y, Xu C, Cheung SC, Yu H, Zhu Z (2022) Runtime permission issues in android apps: Taxonomy, practices, and ways forward. IEEE Trans Softw Eng 49(1):185–210

Wang Z, Chen J, Wang Y, Zhang Y, Zhang W, Zheng Z (2024) Efficiently detecting reentrancy vulnerabilities in complex smart contracts. arXiv:2403.11254

Wohlin C (2014) Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: Proceedings of the 18th international conference on evaluation and assessment in software engineering, pages 1–10

Wu S, Li Z, Yan L, Chen W, Jiang M, Wang C, Luo X, Zhou H (2024) Are we there yet? unraveling the state-of-the-art smart contract fuzzers. In: Proceedings of the IEEE/ACM 46th international conference on software engineering, pages 1–13

Xi R, Wang Z, Pattabiraman K (2024) Pomabuster: Detecting price oracle manipulation attacks in decentralized finance. In: 2024 IEEE Symposium on Security and Privacy (SP), IEEE, pages 3923–3942

Yuan D, Park S, Zhou Y (2012) Characterizing logging practices in open-source software. In: Proceedings of the 2012 international conference on software engineering, pages 102–112

Zero address check example (2021) https://github.com/delvtech/council/commit/7d01b7de58569109755b84af38524884fe68a1c8

Zhang B (2024) Towards finding accounting errors in smart contracts. In: Proceedings of the IEEE/ACM 46th international conference on software engineering, pages 1–13

Zhang Z, Zhang B, Xu W, Lin Z (2023) Demystifying exploitable bugs in smart contracts. In: 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023, IEEE, pages 615–627. https://doi.org/10.1109/ICSE48619.2023.00061

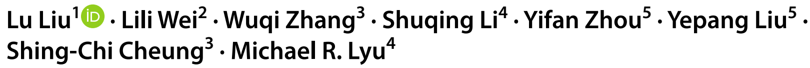
Zheng Z, Su J, Chen J, Lo D, Zhong Z, Ye M (2024) Dappscan: building large-scale datasets for smart contract weaknesses in dapp projects. IEEE Trans Softw Eng

Zhou L, Xiong X, Ernstberger J, Chaliasos S, Wang Z, Wang Y, Qin K, Wattenhofer R, Song D, Gervais A (2023) Sok: Decentralized finance (defi) attacks. In: 44th IEEE Symposium on Security and Privacy (SP). IEEE. pp 2444–2461. https://doi.org/10.1109/SP46215.2023.10179435

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Authors and Affiliations


Lu Liu[1] · Lili Wei[2] · Wuqi Zhang[3] · Shuqing Li[4] · Yifan Zhou[5] · Yepang Liu[5] · Shing-Chi Cheung[3] · Michael R. Lyu[4]

✉ Yepang Liu
liuyp1@sustech.edu.cn

✉ Shing-Chi Cheung
scc@cse.ust.hk

Lu Liu
lliubf@connect.ust.hk

Lili Wei
llweifall@gmail.com

Wuqi Zhang
wuqi.zhang@connect.ust.hk

Shuqing Li
sqli21@cse.cuhk.edu.hk

Yifan Zhou
12332419@mail.sustech.edu.cn

Michael R. Lyu
lyu@cse.cuhk.edu.hk

1 Southern University of Science and Technology, The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong

2 McGill University, Montreal, Canada

3 The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong

4 The Chinese University of Hong Kong, Ma Liu Shui, Hong Kong

5 The Chinese University of Hong Kong, Shatin, Hong Kong