2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

# Code Cloning in Solidity Smart Contracts: Prevalence, Evolution, and Impact on Development

Ran Mo, Haopeng Song, Wei Ding, Chaochao Wu
*School of Computer & Hubei Provincial Key Laboratory of Artificial Intelligence and Smart Learning*
Central China Normal University, China
moran@ccnu.edu.cn, {haopeng, wding, wcc}@mails.ccnu.edu.cn

***Abstract*—In recent years, the development of Solidity smart contracts has been increasing rapidly in popularity. Code cloning is a common coding practice, and many prior studies have revealed that code clones could negatively impact software maintenance and quality. However, there is little work systematically analyzing the nature and impacts of code clones in solidity smart contracts. To bridge this gap, we investigate the prevalence, evolution, and bug-proneness of code clones in solidity smart contracts, and further identify the possible reasons for these clones' occurrences. With our evaluation of 26,294 smart contracts with 97,877 functions, we have found that code clones are highly prevalent in smart contracts. Additionally, on average, 32.01% of clones co-evolve, indicating the need for careful management to avoid consistency issues. Surprisingly, unlike in traditional software development, code clones in smart contracts are rarely involved in bug fixes. Finally, we identify three main factors that affect the occurrences of clones. We believe our study can provide valuable insights for developers to understand and manage code clones in solidity smart contracts.**

***Index Terms*—Solidity Smart Contracts, Code Cloning, Clone Evolution, Blockchain.**

## I. Introduction

A smart contract is a digital agreement on a blockchain, which will be executed automatically once certain terms and conditions are satisfied. It is a fundamental part of decentralized applications. In recent years, smart contracts have rapidly grown in popularity. Thousands of smart contracts are being developed and deployed every day. Recent research [1] has indicated that the global blockchain market is expected to reach $248.9 billion by 2029, with smart contracts serving as a key driver of this growth.

Like traditional software projects, as smart contracts evolve, developers may make poor coding or design decisions due to reduced development time and increased complexity. This can cause difficulties in the understanding, management, and maintenance of smart contracts [2]–[5]. Code cloning refers to the existence of similar or identical code fragments in a program. It is a common coding practice, and numerous studies have presented the prevalence of code clones in software development [6], [7]. On the one hand, code cloning can enhance developers' productivity and efficiency by copying and pasting operations or reusing frameworks or coding patterns. On the other hand, code cloning can also negatively impact software development and maintenance and requires careful consideration [8]–[10].

Similarly, smart contracts can also contain code clones. Moreover, smart contracts typically require highly reliable and secure code compared to traditional software projects, because once deployed on the blockchain, smart contracts cannot be easily modified. Moreover, the decentralized nature of smart contracts demands high transparency and verifiability, making the issue of code cloning more complex and significant in smart contract development [11], [12]. Therefore, some researchers have examined code clones in smart contracts. For example, Kondo et al. [11] presented that 79.2% of the studied smart contracts contain clones. Khan et al. [12] extended the study by analyzing function-level clones, and reported a 30.13% overall clone ratio. Chen et al. [13] presented that 26% of contract code blocks were reused, and summarized commonly reused contracts. Sun et al. [14] showed that about 50% of all subcontracts contain more than 90% similar functions.

However, most of the existing studies mainly examine the presence of code clones in smart contracts, there is little work comprehensively investigating code cloning in smart contracts, including the prevalence, evolution, and bug-proneness of code clones in smart contracts, not mention to the possible reasons for clones' occurrences. To bridge this gap, we conducted an empirical study on code cloning in solidity smart contracts. In this case, we leveraged NiCad to detect code clones in the contracts. NiCad is a widely used clone detector and supports Solidity language [12]. Based on the detected code clones in smart contracts, we analyze 1) the prevalence of code codes; 2) the evolution of code clones; 3) the bug proneness of code clones; and 4) the factors associated with occurrences of code clones.

Through our analyses of 26,294 solidity smart contracts with 97,877 functions, we presented that: 1) Code clones are highly prevalent in smart contracts, with 52.44% to 95.8% of functions (67.43% on average) containing code clones. This indicates that code reuse is a common practice in smart contract development. The percentages of cloned code lines range from 2.78% to 21.61% (5.12% on average), suggesting that a significant portion of smart contract code may be duplicated or highly similar. 2) A notable portion (32.01% on average) of clones co-evolve, meaning the code clones in smart contracts are not independent in maintenance. They should be carefully monitored and managed to avoid inconsistency issues. Meanwhile, we revealed that co-evolution is primarily caused by code reuse, modularity, and rapid feature development. 3) Unlike traditional software, code clones in smart contracts are rarely involved in bug fixes, meaning that cloned code in smart contracts may not be so bug-prone as in other software domains. 4) we identified three key factors associated with clones' occurrences, including feature addition and expansion, code optimization and refactoring, and high-level changes to a project's structure. The above results could provide insights into the nature and impact of code cloning in solidity smart contracts.

We believe our study has made the following contributions:

- It is the first study that systematically analyzes the prevalence, evolution, and impact of code clones in Solidity smart contracts, establishing a baseline for future research in this area.
- The analysis of clone evolution provides insights into how smart contracts grow and change over time, which can inform better development practices and tools for managing smart contract codebases.
- The finding that cloned code in smart contracts is not prone to bugs challenges the traditional wisdom about code cloning and suggests that the impact of cloning may differ in blockchain contexts.
- We identify factors influencing clone rates in smart contracts, which can guide developers in managing code reuse more effectively.

The rest of this paper is structured as follows: Section II introduces the basic concepts behind this work; Section III presents the prior studies related to our work; Section IV reports the design of our study. Section V presents our experimental methods and results; Section VII discusses the threats to the validity; Section VI discusses our findings and implications; Section VIII draws the conclusion.

## II. Background

### A. Smart Contracts

Smart contracts are computer protocols that operate on the blockchain. They are designed to automate, verify, or enforce the terms of a contract. Smart contracts use code to define rules and penalties, which are then automatically enforced. Smart contracts have features of decentralization, transparency, and immutability, which make them popular in various applications such as finance, supply chain management, and legal contracts. To create and deploy smart contracts, developers typically use programming languages (e.g., Solidity) and blockchain platforms (e.g., Ethereum). In addition, once smart contracts are deployed on the blockchain, their code and behaviors cannot be changed. Thus, writing and auditing smart contracts must be done with great care.

### B. Code Clones

Code clones are identical or similar code fragments within a program. Developers often copy and paste existing code for convenience and efficiency, which results in the creation of code clones. These clones are often categorized into the following types:

•Type 1: Completely identical code fragments, with no differences other than whitespace and comments.

•Type 2: Code fragments that perform the same function and have the same structure but differ in variable names, function names, and other identifiers.

•Type 3: Code fragments that are similar to Type 2 clones but include added, deleted, or modified statements.

Code clones can cause difficulties in software maintenance [8]–[10]. For example, cloned code increases the redundancy of the codebase, making it more challenging to understand and manage. when developers modify a clone-involved feature, they need to identify and consistently update all related cloned code fragments, otherwise, inconsistencies issues may be introduced. Therefore, it's important to effectively manage code clones for improving code quality and maintainability.

## III. Related Work

### A. Code Clone Analysis

Code clone is a common coding practice. It can lead to code redundancy, increased maintenance costs, and potential defects [9], [10], [15], [16]. Thus, researchers have proposed various studies to understand and evaluate the impact of code clones. For example, Mondal et al. [17] discussed how the presence of cloned code fragments can propagate the same defects to multiple locations, increasing the complexity of debugging and fixing. Lozano et al. [18] noted that maintaining cloned code fragments requires developers to synchronize updates across all related fragments with each modification, otherwise, inconsistency issues may arise. Zhao et al. [19] analyzed the code clones across microservices, and presented a taxonomy of the possible causes for these code clones. Mo et al. [20] analyzed the prevalence of code clones in Deep Learning software systems. The authors investigated the correlation between the likelihood of co-changes and a series of code metrics, examined the bug-proneness of co-changed clones, and presented possible reasons for the prevalence of code cloning. Ehsan et al. [21] leveraged machine learning to develop clone ranking models. Their method can help developers identify the most risky code clones and prioritize them for refactoring.

### B. Smart Contract Analysis

Due to the immutability and high-risk nature of smart contracts, code clones in smart contracts can lead to severe security vulnerabilities and economic losses [4], [22]. Therefore, smart contract analysis has become an active research area aimed at enhancing the security and reliability of smart contracts. Luu et al. [4] uses static analysis to examine the syntax and logical structure of the code, allowing potential security issues to be identified without executing the code.Jiang et al. [23] presented tools such as Mythril and Slither, which are commonly used for static analysis of smart contracts to detect common security issues such as reentrancy attacks, integer overflows, and unhandled exceptions. Kalra et al. [24] used model-checking tools like VeriSmart to verify the state transition logic of smart contracts, ensuring their correctness under various conditions. Hildenbrandt et al. [25] utilized formal verification techniques to ensure the correctness of smart contract logic during the design phase, which reduced runtime risks and vulnerabilities.

In our work, we aim to analyze the prevalence, evolution, and impact of code clones in solidity smart, and reveal the possible causes of code clones.

## IV. Study Design

In this section, we present our research objectives, questions, and how we collected data.

### A. Objectives

In this study, we provide a fundamental understanding of code clones in solidity smart contracts. Specifically, we first examine the prevalence of code clones in solidity smart contracts. Then we analyze how code clones evolve. Next, we study the relationships between code clones and bug-proneness. Finally, we explored the possible reasons for the occurrences of code clones.

### B. Research questions

To achieve our objectives, we seek to answer the following research questions:

**RQ1: How prevalent are code clones in solidity smart contracts?**

Smart contract developers often reuse existing contract code to reduce development time and enhance code reliability. Meanwhile, the modular nature of the Solidity language and the frequency of code reuse, also make code cloning very common. Therefore, in this question, we quantify the extent of code cloning in solidity smart contracts. This provides a foundational understanding of the phenomenon's scale and importance in this domain.

**RQ2: To what extent do the code clones co-evolve?**

For a clone pair, if the involved code fragments never demand to evolve consistently, indicating these fragments are independent in their maintenance. If there exist co-evolved clone pairs, this means that the code fragments need to be co-added, co-changed, or co-deleted as the software evolves. This could cause maintenance difficulties since whenever a code fragment is added or modified, its cloned code should be added or modified consistently to avoid the risk of errors [9], [10], [15], [16], [20]. Thus, in this question, We investigate the co-evolution of code clones across different versions of smart contracts, examining possible reasons for their co-additions, co-changes, and co-deletions.

**RQ3: Are code clones in solidity smart contracts prone to bugs?**

In this question, we examine the potential relationship between code clones and the introduction of bugs. We aim to identify whether these clones pose a high potential of bug-proneness to smart contracts.

**RQ4: What factors are associated with the occurrence of code clones in solidity smart contracts?**

We examine various project characteristics, development practices, and environmental factors that may contribute to the occurrences of code clones in solidity smart contracts. This analysis will help identify the possible reasons for code cloning in this specific context.

By addressing these research questions, we aim to provide a comprehensive understanding of code cloning in Solidity smart contracts, its evolution, potential risks, and underlying causes. This knowledge will be valuable for improving development practices and enhancing the quality of smart contracts.

### C. Data Collection

**Subjects.** In this paper, we collect data from a wide range of projects: 1) Etherscan: Etherscan is the blockchain explorer for the Ethereum blockchain, offering access to all smart contracts deployed through the Ethereum network [26]; 2) OpenZeppelin: OpenZeppelin provides a series of audited and secure smart contract libraries widely used to build reusable contract components, especially in DeFi projects. OpenZeppelin contracts can serve as benchmarks for code quality and security [27]; 3) Synthetix: Synthetix is a decentralized finance platform built on Ethereum with its smart contract code publicly available, representing high complexity and professional-grade financial application development [28].

The selected projects also satisfied the following two conditions: 1) *Language.* Our study focuses on smart contracts in Ethereum, which is one of the most widely used blockchain platforms, with over 3 million smart contracts deployed on its network. Solidity is the primary programming language for the Ethereum smart contract. Therefore, we selected the smart contracts written in Solidity. If the number of Solidity lines of code (LOC) in a project accounts for more than 60% of the project's total LOC, and GitHub indicates that Solidity is the primary language of the project, we classify it as a Solidity project; 2) *Stars.* The number of stars a project has often indicates its popularity. We selected projects with at least 500 stars to avoid toy, training, or tutorial projects.

To the end, we collected 26,294 smart contracts with 97,877 functions[1], where OpenZeppelin has 7 releases, containing 259 - 806 functions, 4,045 in total, Synthetix has 32 releases, containing 1,238 - 2,720 functions, 61,888 in total, and Etherscan has 31,944 functions. We have shared all the collected data on our website [2].

**Detection.** *1) Code Clone Detection.* Given the smart contracts, we use Nicad to detect code clones. Nicad is a widely used clone detection tool, and it supports Solidity language [12]. We configure the settings of Nicad as follows: The minimum and maximum number of lines is set to 3 and 2500; The granularity is set to function level. For similarity thresholds, we used NiCad's default settings: 100% for Type-1 and Type-2 clones, and 70% for Type-3 clones. This default setting has been widely used in prior studies and could maximize the precision and recall of detection [8], [12], [17], [20]. We adjusted the minimum line count from the typical 5 lines to 3 lines, since we observed that over 90% of cloned

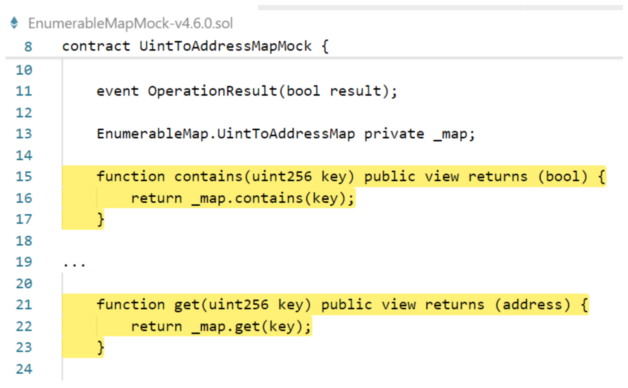
[1]Our data is updated on May 7, 2024

[2]https://dingling258.github.io/code-cloning-in-solidity/

code fragments in our analyzed contracts are less than 5 lines long. *2) Co-evolved Clone Detection.* Given the detected code clones, we further identify three types of co-evolved clones:

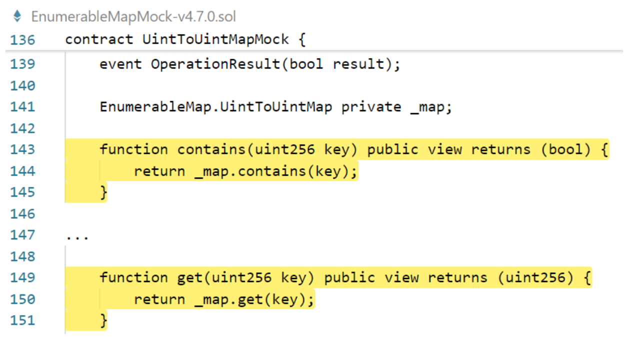
**Co-added Clone.** Assume a code clone, $C$ is detected in version V1, and the same code fragments are added into more .sol files (Solidity-based smart contracts) in version V2. As a result, $C$ is still detected as a clone in V2, but contains more clones code fragments. We consider $C$ to be a co-added clone, since the cloned code fragments are kept added into it as the smart contract evolves.

Figure 1 shows the example of a co-added clone. We can observe that, in version v4.6.0, there is a code clone in contract *UnitToAddressMapMock* of *SafeCastMock.sol* file. In v4.7.0, the same code fragments were added into another contract *UnitToUnitMapMock*.


EnumerableMapMock-v4.6.0.sol

```
contract UintToAddressMapMock {

    event OperationResult(bool result);

    EnumerableMap.UintToAddressMap private _map;

    function contains(uint256 key) public view returns (bool) {
        return _map.contains(key);
    }

...

    function get(uint256 key) public view returns (address) {
        return _map.get(key);
    }

```

(a) A code clone


EnumerableMapMock-v4.7.0.sol

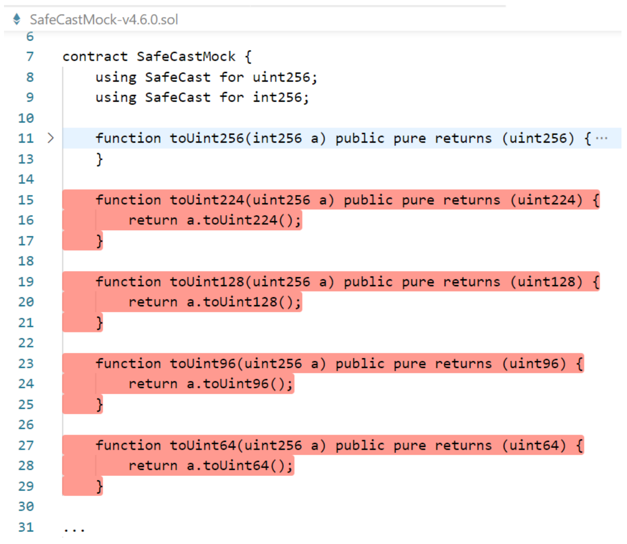
```
contract UintToUintMapMock {
    event OperationResult(bool result);

    EnumerableMap.UintToUintMap private _map;

    function contains(uint256 key) public view returns (bool) {
        return _map.contains(key);
    }

...

    function get(uint256 key) public view returns (uint256) {
        return _map.get(key);
    }
```

(b) Added code fragments in later version

Fig. 1: An example of co-added clone

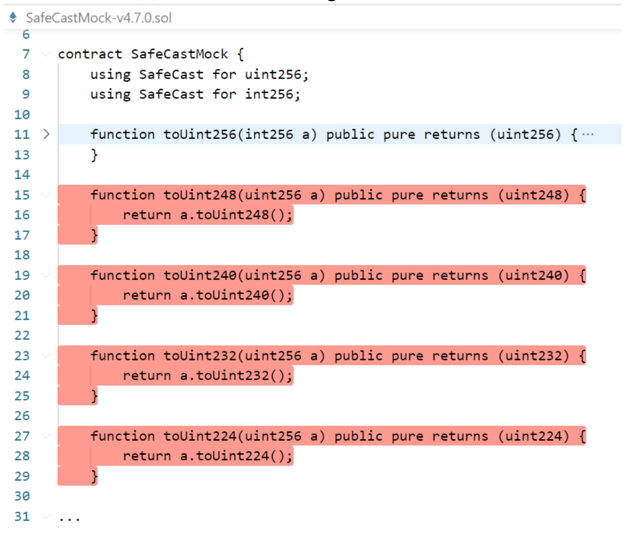
**Co-changed Clone.** Assume there is a code clone, $C$, in version V1. Then the code fragments in $C$ undergo the same or similar changes, and they are still detected as a pair of clones in version V2. We consider $C$ to be a co-changed clone, since the cloned code fragments are changed together as the smart contract evolves. Figure 2 depicts a co-changed clone. We can observe that, from version v4.6.0 to v4.7.0, code fragments of the clone in contract *SafeCastMock* have been changed together.

**Co-deleted Clone**: In version V1, two or more code fragments identified as a clone pair are simultaneously deleted in the subsequent version, causing this type of clone cluster to no longer exist or reduce the number of clone code fragments.


SafeCastMock-v4.6.0.sol

```

contract SafeCastMock {
    using SafeCast for uint256;
    using SafeCast for int256;

    function toUint256(int256 a) public pure returns (uint256) {···
    }

    function toUint224(uint256 a) public pure returns (uint224) {
        return a.toUint224();
    }

    function toUint128(uint256 a) public pure returns (uint128) {
        return a.toUint128();
    }

    function toUint96(uint256 a) public pure returns (uint96) {
        return a.toUint96();
    }

    function toUint64(uint256 a) public pure returns (uint64) {
        return a.toUint64();
    }

...
```

(a) Cloned code fragments in v4.6.0


SafeCastMock-v4.7.0.sol

```

contract SafeCastMock {
    using SafeCast for uint256;
    using SafeCast for int256;

    function toUint256(int256 a) public pure returns (uint256) {···
    }

    function toUint248(uint256 a) public pure returns (uint248) {
        return a.toUint248();
    }

    function toUint240(uint256 a) public pure returns (uint240) {
        return a.toUint240();
    }

    function toUint232(uint256 a) public pure returns (uint232) {
        return a.toUint232();
    }

    function toUint224(uint256 a) public pure returns (uint224) {
        return a.toUint224();
    }

...
```

(b) Cloned code fragments in v4.7.0

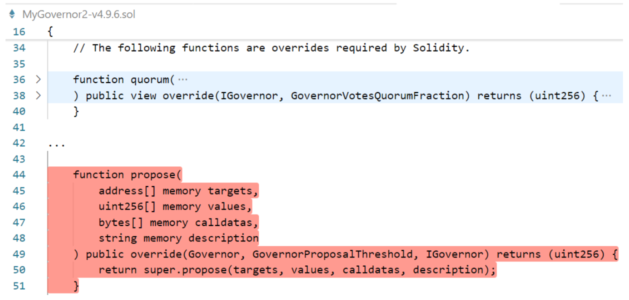
Fig. 2: An example of co-changed clone

Figure 3 shows an example, where there are two cloned code fragments in version v4.9.6. These two fragments are deleted simultaneously, thus this clone is not existent in the later versions.

## V. Experiments

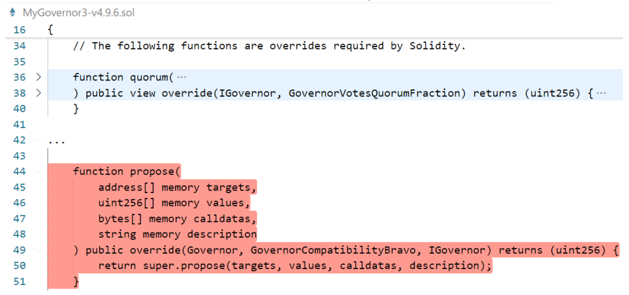
**RQ1: How prevalent are code clones in solidity smart contracts?**

To answer this question, we conducted a suite of quantitative analyses as shown in Table I. Column FUNC means the number of functions in a project. Column CF is the number of functions with code clones. PCF shows the ratio of CF to FUNC, indicating how likely the functions of smart contracts contain code clones. Column PLOC shows the number of lines of code in each project. Column CLOC is the total LOC of detected code clones. Column POC is the ratio of CLOC to


MyGovernor2-v4.9.6.sol

```
{
    // The following functions are overrides required by Solidity.

    function quorum(⋯
    ) public view override(IGovernor, GovernorVotesQuorumFraction) returns (uint256) {⋯
    }

...

    function propose(
        address[] memory targets,
        uint256[] memory values,
        bytes[] memory calldatas,
        string memory description
    ) public override(Governor, GovernorProposalThreshold, IGovernor) returns (uint256) {
        return super.propose(targets, values, calldatas, description);
    }
```

(a) Clone code fragment 1


MyGovernor3-v4.9.6.sol

```
{
    // The following functions are overrides required by Solidity.

    function quorum(⋯
    ) public view override(IGovernor, GovernorVotesQuorumFraction) returns (uint256) {⋯
    }

...

    function propose(
        address[] memory targets,
        uint256[] memory values,
        bytes[] memory calldatas,
        string memory description
    ) public override(Governor, GovernorCompatibilityBravo, IGovernor) returns (uint256) {
        return super.propose(targets, values, calldatas, description);
    }
```

(b) Clone code fragment 2

Fig. 3: An example of co-deleted clone

PLOC, indicating how likely code fragments in a project are involved in code clones.

Using the first raw as an example, we can observe that there are 4,045 functions in Openzeppelin, where 2,208 of them (52.44%) contain code clones. For all the code clones, the ratios of Type1, Type2, and Type3 are 40.35%, 54.94%, and 4.71%, respectively. when considering the code lines of this project, we can see that more than 7,500 LOC are involved in clones, accounting for 7.23% of the total LOC.

Considering all three projects together, we can make the following observations: 1) from 52.44% to 95.8% functions (67.43% on average) contain code clones; 2) Compared to the others, Type3 is less likely to happen. On average, Type1 is the most prevalent, but for OpenZepplina and Synthetix project, Type2 has a higher percentage; 3) from 2.78% to 21.61% code lines (5.12% on average) are involved in clones;

**Finding 1:** Code clones commonly happen in smart contracts, with a majority of (more than 50%) functions containing clones and a considerable portion of code lines involved in cloning. This high prevalence of code clones indicates that developers often reuse code in smart contract development. When considering each clone type individually, the results have shown that Type1 and Type2 are far more common than Type3 clone in smart contracts.

**RQ2: To what extent do the code clones co-evolve?**

To answer this question, we identified the co-evolved clones between consecutive versions as introduced in section IV-C. We detected co-added, co-changed, and co-deleted clones by analyzing the commits between two adjacent versions. Then we calculated the ratios of co-evolved clones to the number of clones in the former version. Since Etherscan incorporates smart contracts daily without specific versions, we focused our analysis on OpenZepplin and Synthetix.

Table II has shown the experiment results of OpenZepplin. Now we use the first raw to illustrate the values. For the version pair of v4.5.0 and v4.6.0, 576 functions were identified in v4.5.0, and when it was updated to v4.6.0, 148 of these cloned functions were co-evolved, with 33, 7, and 108 functions containing co-added, co-changed and co-deleted clones, respectively. They account for 5.73%, 1.22%, and 18.75% of the total cloned functions. For the whole table, we can observe that the ratios of co-evolved clones are from 7.66% to 55.83%, with an average of 32.01%. Specifically, ratios of co-added clones range from 1.81% to 12.26%, with an average of 6.89%. For the co-changed clones, the largest ratio is as high as 31.64%. It means about one-third of clones have been changed consistently. The ratios range from 0 to 31.64%, with an average of 10.91%. For the co-deleted clones, the ratios range from 5.85% to 18.75%, with an average of 14.21%.

Table III has shown the experiment results of Synthetix. Compared to OpenZepplin, it has a smaller ratio of co-evolved clones. But the ratio is still notable, ranging from 0% to 25.24% with an average of 7.24%. Specifically, the ratios of co-added clones are from 0 to 10.93%, with an average of 1.92%. For the co-changed clones, most of the release pairs don't have co-changed clones. The ratios range from 0 to 3.26%, with an average of 0.49%. the ratios of co-deleted clones range from 0 to 12.17%, with an average of 4.83%.

The above experimental results have shown that code clones in solidity smart contracts are not always independent in maintenance. On the opposite, a notable portion of them need to evolve consistently. This means that code clones in smart contracts can indeed cause maintenance difficulties, as whenever a code fragment of such a clone is added, changed, or deleted, the related code should also be through the same operations to avoid inconsistency issues. Therefore, these clones should be carefully tracked and managed during their evolution.

Furthermore, We analyzed possible causes for the co-evolved clones by examining their revision history, and identified two main reasons as follows: 1) In smart contracts, due to the complexity of contracts and high-security requirements, developers tend to reuse verified code to reduce errors and enhance code reliability. This reuse often leads to the co-added code clones. Figure 4 shows the example of reusing verified code: in version v4.7.0, there are 3 clone functions in *EnumerableSetMock.sol* file. In v4.8.0, the same code fragments were added to contract *CheckpointsMock*.

2) During the iteration of smart contracts, new features may need to be added or existing features extended. Developers may copy and modify existing functions to quickly implement new features, resulting in co-added or co-changed clone codes. Figure 5 shows OpenZeppelin's v5.0.0 release notes, listing the newly added contracts and libraries. These new features, such as *AccessManager* and *GovernorTimelockAccess*, are new functionalities or extensions of existing features in the iteration of smart contracts.

TABLE I: The prevalence of code clones in solidity smart contracts

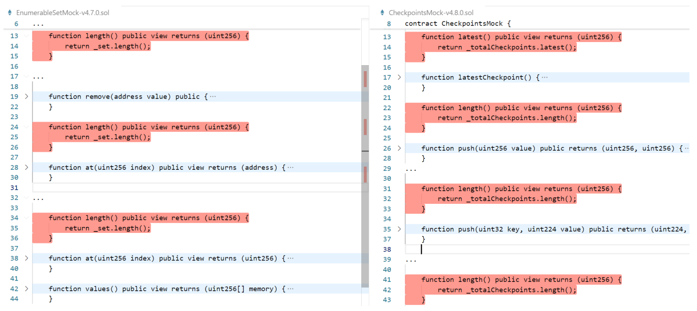
| Project | FUNC | CF | PCF | Type1 | Type2 | Type3 | Type1% | Type2% | Type3% | PLOC | CLOC | POC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Openzeppelin | 4,045 | 2,208 | 54.59% | 891 | 1213 | 104 | 40.35% | 54.94% | 4.71% | 103,842 | 7,598 | 7.32% |
| synthetix | 61,888 | 33,187 | 52.69% | 10,692 | 18,116 | 4,379 | 32.22% | 54.59% | 13.19% | 68,9917 | 152,269 | 21.61% |
| Etherscan | 31,944 | 30,603 | 95.80% | 29,509 | 863 | 231 | 96.43% | 2.82% | 0.75% | 5,066,850 | 140,412 | 2.77% |
| Average | | | 67.43% | | | | 62.26% | 30.59% | 7.14% | | | 5.12% |

TABLE II: The evolution of code clones in OpenZeppelin

| Version Pair | Cloned functions | Co-added | Co-changed | Co-deleted | Co-evolved | Co-added% | Co-changed% | Co-deleted% | Co-evolved% |
|---|---|---|---|---|---|---|---|---|---|
| v4.5.0-v4.6.0 | 576 | 33 | 7 | 108 | 148 | 5.73% | 1.22% | 18.75% | 25.69% |
| v4.6.0-v4.7.0 | 620 | 76 | 11 | 99 | 186 | 12.26% | 1.77% | 15.97% | 30.00% |
| v4.7.0-v4.8.0 | 754 | 61 | 23 | 122 | 206 | 8.09% | 3.05% | 16.18% | 27.32% |
| v4.8.0-v4.9.0 | 806 | 60 | 255 | 135 | 450 | 7.44% | 31.64% | 16.75% | 55.83% |
| v4.9.0-v4.9.6 | 496 | 9 | 0 | 29 | 38 | 1.81% | 0.00% | 5.85% | 7.66% |
| v4.9.6-v5.0.2 | 534 | 22 | 117 | 45 | 184 | 4.12% | 21.91% | 8.43% | 34.46% |
| Average | | | | | | 6.89% | 10.91% | 14.21% | 32.01% |

TABLE III: The evolution of code clones in Synthetix

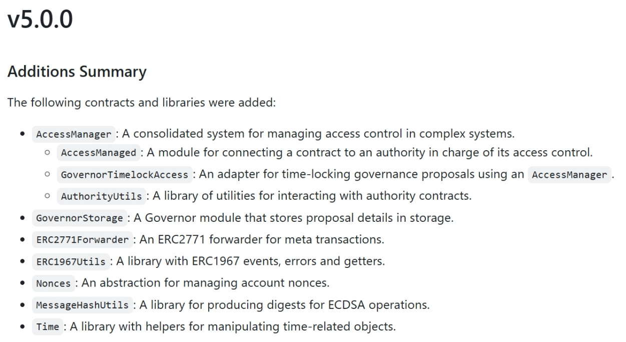
| Version Pair | Cloned functions | Co-added | Co-changed | Co-deleted | Co-evolved | Co-added% | Co-changed% | Co-deleted% | Co-evolved% |
|---|---|---|---|---|---|---|---|---|---|
| v2.51.0-v2.53.0 | 1,238 | 55 | 13 | 222 | 290 | 4.44% | 1.05% | 17.93% | 23.42% |
| v2.53.0-v2.54.0 | 1,319 | 10 | 0 | 67 | 77 | 0.76% | 0.00% | 5.08% | 5.84% |
| v2.54.0-v2.55.0 | 1,337 | 0 | 0 | 0 | 0 | 0.00% | 0.00% | 0.00% | 0.00% |
| v2.55.0-v2.56.1 | 1,338 | 0 | 0 | 0 | 0 | 0.00% | 0.00% | 0.00% | 0.00% |
| v2.56.1-v2.57.1 | 1,337 | 8 | 0 | 3 | 11 | 0.60% | 0.00% | 0.22% | 0.82% |
| v2.57.1-v2.58.0 | 1,354 | 47 | 23 | 112 | 182 | 3.47% | 1.70% | 8.27% | 13.44% |
| v2.58.0-v2.59.0 | 1,408 | 7 | 2 | 9 | 18 | 0.50% | 0.14% | 0.64% | 1.28% |
| v2.59.0-v2.60.0 | 1,421 | 22 | 36 | 156 | 214 | 1.55% | 2.53% | 10.98% | 15.06% |
| v2.60.0-v2.61.0 | 1,366 | 2 | 2 | 0 | 4 | 0.15% | 0.15% | 0.00% | 0.29% |
| v2.61.0-v2.62.0 | 1,366 | 2 | 0 | 0 | 2 | 0.15% | 0.00% | 0.00% | 0.15% |
| v2.62.0-v2.63.0 | 1,372 | 0 | 0 | 0 | 0 | 0.00% | 0.00% | 0.00% | 0.00% |
| v2.63.0-v2.64.1 | 1,372 | 150 | 19 | 276 | 445 | 10.93% | 1.38% | 20.12% | 32.43% |
| v2.64.1-v2.66.0 | 1,679 | 10 | 2 | 21 | 33 | 0.60% | 0.12% | 1.25% | 1.97% |
| v2.66.0-v2.68.2 | 1,704 | 102 | 6 | 204 | 312 | 5.99% | 0.35% | 11.97% | 18.31% |
| v2.68.2-v2.70.0 | 1,890 | 43 | 7 | 139 | 189 | 2.28% | 0.37% | 7.35% | 10.00% |
| v2.70.0-v2.72.1 | 1,954 | 88 | 14 | 132 | 234 | 4.50% | 0.72% | 6.76% | 11.98% |
| v2.72.1-v2.74.1 | 2,009 | 8 | 0 | 15 | 23 | 0.40% | 0.00% | 0.75% | 1.14% |
| v2.74.1-v2.75.2 | 2,031 | 3 | 0 | 4 | 7 | 0.15% | 0.00% | 0.20% | 0.34% |
| v2.75.2-v2.78.1 | 2,037 | 76 | 17 | 217 | 310 | 3.73% | 0.83% | 10.65% | 15.22% |
| v2.78.1-v2.80.5 | 2,175 | 213 | 71 | 265 | 549 | 9.79% | 3.26% | 12.18% | 25.24% |
| v2.80.5-v2.82.2 | 2,471 | 4 | 0 | 0 | 4 | 0.16% | 0.00% | 0.00% | 0.16% |
| v2.82.2-v2.84.4 | 2,481 | 32 | 22 | 302 | 356 | 1.29% | 0.89% | 12.17% | 14.35% |
| v2.84.4-v2.86.1 | 2,199 | 118 | 27 | 209 | 354 | 5.37% | 1.23% | 9.50% | 16.10% |
| v2.86.1-v2.88.1 | 2,374 | 50 | 1 | 38 | 89 | 2.11% | 0.04% | 1.60% | 3.75% |
| v2.88.1-v2.90.1 | 2,525 | 12 | 0 | 3 | 15 | 0.48% | 0.00% | 0.12% | 0.59% |
| v2.90.1-v2.92.1 | 2,555 | 12 | 0 | 3 | 15 | 0.47% | 0.00% | 0.12% | 0.59% |
| v2.92.1-v2.94.1 | 2,585 | 29 | 0 | 6 | 35 | 1.12% | 0.00% | 0.23% | 1.35% |
| v2.94.1-v2.96.1 | 2,655 | 4 | 0 | 0 | 4 | 0.15% | 0.00% | 0.00% | 0.15% |
| v2.96.1-v2.98.2 | 2,662 | 14 | 0 | 0 | 14 | 0.53% | 0.00% | 0.00% | 0.53% |
| v2.98.2-v2.100.0 | 2,696 | 11 | 2 | 31 | 44 | 0.41% | 0.07% | 1.15% | 1.63% |
| v2.100.0-v2.101.2 | 2,720 | 12 | 30 | 448 | 490 | 0.44% | 1.10% | 16.47% | 18.01% |
| Average | | | | | | 1.92% | 0.49% | 4.83% | 7.24% |



```
...
    function length() public view returns (uint256) {
        return _set.length();
    }

...

    function remove(address value) public {···
    }

    function length() public view returns (uint256) {
        return _set.length();
    }

    function at(uint256 index) public view returns (address) {···
    }

...

    function length() public view returns (uint256) {
        return _set.length();
    }

    function at(uint256 index) public view returns (uint256) {···
    }

    function values() public view returns (uint256[] memory) {···
    }
```

```
contract CheckpointsMock {
    function latest() public view returns (uint256) {
        return _totalCheckpoints.latest();
    }

    function latestCheckpoint() {···
    }

    function length() public view returns (uint256) {
        return _totalCheckpoints.length();
    }

    function push(uint256 value) public returns (uint256, uint256) {··
    }
...

    function length() public view returns (uint256) {
        return _totalCheckpoints.length();
    }

    function push(uint32 key, uint224 value) public returns (uint224,
    }

...

    function length() public view returns (uint256) {
        return _totalCheckpoints.length();
    }
```

Fig. 4: Code reuse and resulting code clones



v5.0.0

Additions Summary

The following contracts and libraries were added:

- `AccessManager` : A consolidated system for managing access control in complex systems.
    - `AccessManaged` : A module for connecting a contract to an authority in charge of its access control.
    - `GovernorTimelockAccess` : An adapter for time-locking governance proposals using an `AccessManager` .
    - `AuthorityUtils` : A library of utilities for interacting with authority contracts.
- `GovernorStorage` : A Governor module that stores proposal details in storage.
- `ERC2771Forwarder` : An ERC2771 forwarder for meta transactions.
- `ERC1967Utils` : A library with ERC1967 events, errors and getters.
- `Nonces` : An abstraction for managing account nonces.
- `MessageHashUtils` : A library for producing digests for ECDSA operations.
- `Time` : A library with helpers for manipulating time-related objects.

Fig. 5: OpenZeppelin v5.0.0 Release Notes

**Finding 2:** The experimental results show that co-evolved clones are a common and important development practice in smart contracts. A considerable portion of clones, especially in OpenZeppelin, evolve together, with co-evolved clones averaging 32.01%. This co-evolution is mainly driven by the demand for code reuse, modularity, and rapid feature development. While this practice can improve reliability and development speed, it also introduces maintenance challenges. Developers of smart contracts should monitor the co-evolved clones and ensure their consistency to reduce potential vulnerabilities in the evolving contract ecosystem.

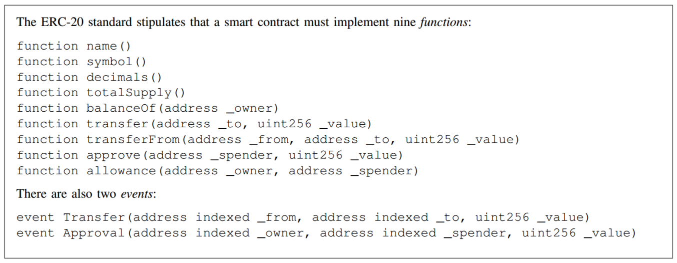
**RQ3: Are code clones in solidity smart contracts prone to bugs?**

To answer this question, we examined whether and to what extent the code clones are involved in bugs. To proceed with this analysis, we first extracted the commits from each project's revision history, then identified bug-fixing commits by using the keyword-matching method [29]–[31]. Specifically, if a commit's message contained the keyword "fix", we classified it as a bug-fixing commit. Next, we examined the code fragments in this commit to determine whether they were related to code clones. If we found that a code fragment was part of a code clone and had been changed as part of the bug fix, we considered that it was involved in bugs.

Table IV reports the experimental results. We can observe that, for both OpenZeppelin and Synthetix projects, there are no cloned code fragments involved in the bug-fixing commits. This phenomenon differs from studies that analyzed code clones of projects from other domains. Numerous studies have shown that code clones are related to bug-proneness [20], [32], [33]. For example, Mondal et al [32] presented that cloned code would bring bugs due to inadequate context. Chatterji et al. [33] studied the impact of code clones on bug fixes and presented that fixing bugs in cloned code became more difficult. Mo et al [20] showed that cloned code fragments had a high potential of involving in bugs.

To analyze why the code clones in smart contracts are rarely involved in bug fixes, we further examined the commits and code snippets of the smart contracts. We finally discovered some possible reasons specific to the implementations of smart contracts as follows:

*Highly Standardized and Simplified Logic.* a) Solidity smart contracts often involve high-risk operations such as financial transactions, and their code is written following strict security standards and simplified logic. b) This highly standardized code, when cloned, tends to have fewer inherent errors, making the copied code blocks less likely to introduce new bugs. c) In the analyzed projects, such as ERC20 token contracts [34] and fundamental modules of decentralized finance (DeFi) protocols, the cloned code is almost always a verified and tested standard implementation. Figure 6 presents the nine core functions and two events mandated by the ERC-20 standard, representing the standardized implementation of ERC20 token contracts. Figure 7 shows the core architecture of the *Governor.sol* governance contract, which serves as an abstract foundation extended through specific modules, including key functions such as vote counting and vote retrieval.

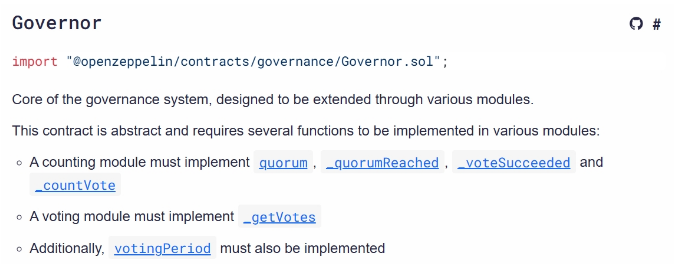

The ERC-20 standard stipulates that a smart contract must implement nine *functions*:

```
function name()
function symbol()
function decimals()
function totalSupply()
function balanceOf(address _owner)
function transfer(address _to, uint256 _value)
function transferFrom(address _from, address _to, uint256 _value)
function approve(address _spender, uint256 _value)
function allowance(address _owner, address _spender)
```

There are also two *events*:

```
event Transfer(address indexed _from, address indexed _to, uint256 _value)
event Approval(address indexed _owner, address indexed _spender, uint256 _value)
```

Fig. 6: Standard implementation of ERC20 token contracts


Governor

```
import "@openzeppelin/contracts/governance/Governor.sol";
```

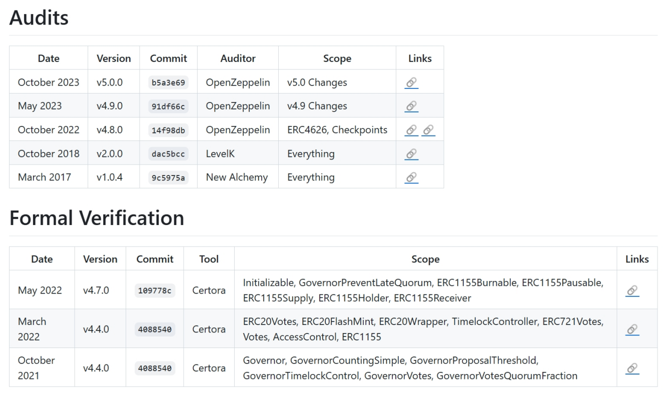
Core of the governance system, designed to be extended through various modules.

This contract is abstract and requires several functions to be implemented in various modules:

- A counting module must implement `quorum`, `_quorumReached`, `_voteSucceeded` and `_countVote`
- A voting module must implement `_getVotes`
- Additionally, `votingPeriod` must also be implemented

Fig. 7: Governor: A Standard implementation of DeFi protocol modules

*Immutability of Smart Contracts.* Once deployed on the blockchain, the logic of smart contracts cannot be changed [35]. This requires developers to perform rigorous testing and review before deployment to ensure code correctness. For example, developers have conducted multiple audits and tests on the project OpenZeppelin before its deployment. Thus even cloned codes have undergone thorough review and testing, which reduces the occurrence of bugs in such clones. For instance, projects like Synthetix and Aave conduct multiple audits and tests before deployment [36], [37]. Figure 8 illustrates the comprehensive audits and formal verifications conducted on the OpenZeppelin project at various time points from 2017 to 2023.


Audits

| Date | Version | Commit | Auditor | Scope | Links |
|---|---|---|---|---|---|
| October 2023 | v5.0.0 | b5a3e69 | OpenZeppelin | v5.0 Changes | 🔗 |
| May 2023 | v4.9.0 | 91df66c | OpenZeppelin | v4.9 Changes | 🔗 |
| October 2022 | v4.8.0 | 14f98db | OpenZeppelin | ERC4626, Checkpoints | 🔗 🔗 |
| October 2018 | v2.0.0 | dac5bcc | LevelK | Everything | 🔗 |
| March 2017 | v1.0.4 | 9c5975a | New Alchemy | Everything | 🔗 |

Formal Verification

| Date | Version | Commit | Tool | Scope | Links |
|---|---|---|---|---|---|
| May 2022 | v4.7.0 | 109778c | Certora | Initializable, GovernorPreventLateQuorum, ERC1155Burnable, ERC1155Pausable, ERC1155Supply, ERC1155Holder, ERC1155Receiver | 🔗 |
| March 2022 | v4.4.0 | 4088540 | Certora | ERC20Votes, ERC20FlashMint, ERC20Wrapper, TimelockController, ERC721Votes, Votes, AccessControl, ERC1155 | 🔗 |
| October 2021 | v4.4.0 | 4088540 | Certora | Governor, GovernorCountingSimple, GovernorProposalThreshold, GovernorTimelockControl, GovernorVotes, GovernorVotesQuorumFraction | 🔗 |

Fig. 8: OpenZeppelin Multiple Audits and Formal Verifications

*Testing and Verification Techniques.* Smart contract developers often employ advanced testing techniques such as formal verification, symbolic execution, and fuzzing to detect potential vulnerabilities and ensure code reliability. These techniques effectively identify issues in standardized, cloned code segments [38], [39], which could reduce the likelihood of bugs in cloned code. In our studied projects, particularly in OpenZeppelin, we observed the systematic use of the Hardhat and Chai testing frameworks. Hardhat provides a robust development environment with built-in testing capabilities, while Chai offers extensive assertion functions for thorough test case development. Our analysis of OpenZeppelin's testing directory presents extensive unit testing practices and well-structured test suites for contract components. This mature testing infrastructure, combined with standardized development practices and thorough code review processes in clone-related implementations, could significantly reduce the occurrence of bugs in cloned code segments. Such a systematic testing approach can benefit standardized contract patterns, as test cases can effectively validate both the original and cloned implementations.

**Finding 3:** Through a detailed analysis of these projects, we found that bug fixes are usually unrelated to code clones. In other words, clones in smart contracts are not prone to bugs. This finding contradicts the common perception in traditional software development that code clones have a higher potential of inducing bugs. The unique characteristics of smart contract development, including standardized logic, immutability, and rigorous testing, contribute to the reduced bug-proneness of cloned code in this domain. This implies that code cloning in smart contracts might be a more acceptable practice compared to traditional software development, as long as the cloned code comes from well-verified and standardized sources.

TABLE IV: Total, Bug-fixing, and Clone-involved Commits

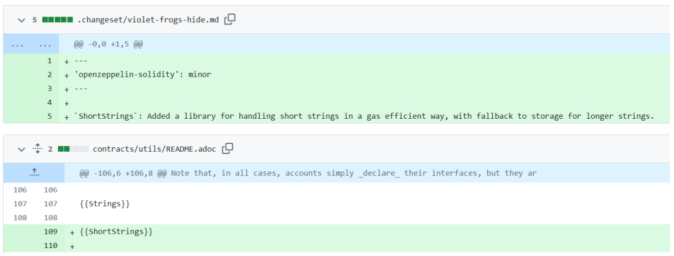
| Project | History Period | Total | Bug-fixing | Clone-involved |
|---|---|---|---|---|
| OpenZeppelin | v4.5.0-v5.0.2 | 792 | 131 | 0 |
| Synthetix | v2.90.1-v2.101.2 | 1090 | 130 | 0 |

**RQ4: What factors are associated with the occurrences of code clones in solidity smart contracts?**

To explore the possible reasons for code clones in smart contracts, we examined related code and commit records, particularly focusing on different version updates of the projects as follows: 1) Used NiCad to detect clones for each version and recorded the clone ratios; 2) Marked versions with significant ratio changes and extracted corresponding update announcements/descriptions from GitHub; 3) Formulated possible reasons for clone ratio changes based on update announcements and commits; 4) Examined corresponding code fragments and commits to analyze these reasons. Through the above systematic analysis, we identified three instances where the ratios of clones have a nontrivial change (decrease or increase) as follows:

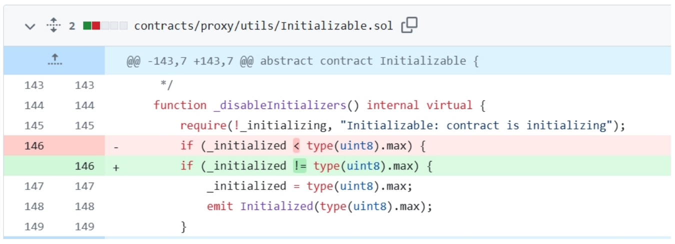
**OpenZeppelin v4.8.0 to v4.9.0** The POC (percentage of cloned code lines) decreased from 9.75% to 4.96%, and the PCF (percentages of cloned functions) decreased from 61.54% to 42.34%. In this period, OpenZeppelin incorporated a few revisions, including enhanced security measures (*ReentrancyGuard* and *ERC1967Upgrade*); extensions for existing components (*ERC721Wrapper, EnumerableMap, Governor*, and *Strings*); Supports for Ethereum Improvement Proposals (*EIP-5313* and *EIP-4906*); Optimizations for storage and data handling (*StorageSlot* and *ShortStrings*), and some core components (*EIP712, SafeERC20, Initializable*, and *TimelockController*). We found the specific reasons for the decrease are as follows:

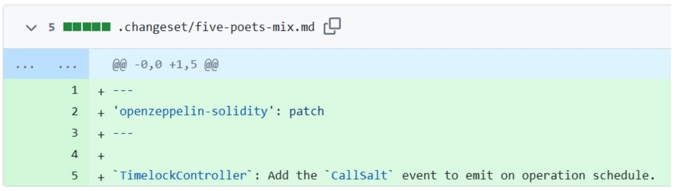
*Adding new libraries.* By introducing new libraries, like *ShortStrings*, developers could replace multiple instances of similar code with centralized, reusable functions. Figure 9


```
5 .changeset/violet-frogs-hide.md
@@ -0,0 +1,5 @@
+ ---
+ 'openzeppelin-solidity': minor
+ ---
+
+ `ShortStrings`: Added a library for handling short strings in a gas efficient way, with fallback to storage for longer strings.
```
```
2 contracts/utils/README.adoc
@@ -106,6 +106,8 @@ Note that, in all cases, accounts simply _declare_ their interfaces, but they ar

  {{Strings}}

+ {{ShortStrings}}
+
```
Fig. 9: Added a library for handling short strings

shows a changeset describing the new *ShortStrings* library. This library enhances gas efficiency by handling short strings and provides a fallback to storage for longer strings. This approach allows developers to replace multiple instances of similar code with centralized, reusable functions, thereby optimizing the code structure.

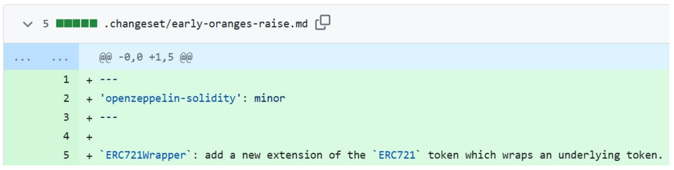
*Optimizations.* Developers implemented various optimizations, particularly for *Initializable* and *TimelockController*, which could reduce code duplication by streamlining existing functionalities. Figure 10 shows the optimization of the


```
2 contracts/proxy/utils/Initializable.sol
@@ -143,7 +143,7 @@ abstract contract Initializable {
         */
        function _disableInitializers() internal virtual {
            require(!_initializing, "Initializable: contract is initializing");
-           if (_initialized < type(uint8).max) {
+           if (_initialized != type(uint8).max) {
                _initialized = type(uint8).max;
                emit Initialized(type(uint8).max);
            }
```
Fig. 10: The optimization in version update


```
5 .changeset/five-poets-mix.md
@@ -0,0 +1,5 @@
+ ---
+ 'openzeppelin-solidity': patch
+ ---
+
+ `TimelockController`: Add the `CallSalt` event to emit on operation schedule.
```
Fig. 11: Create ProposalSalt event for TimelockController

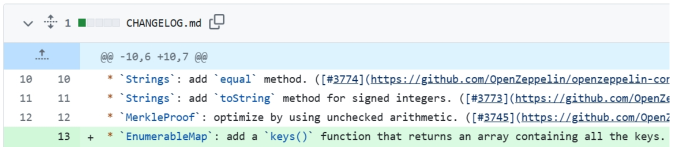
*disableInitializers* function in *Initializable* contract. Figure11 presents the creation of the *ProposalSalt* event for *TimelockController* contract. Both reduce code redundancy and streamline functionality.

*Enhanced abstraction.* New methods and extensions, such as the *ERC721Wrapper* and *EnumerableMap's keys()* method, potentially provided higher-level abstractions, which could eliminate the need for repetitive code in multiple contracts.



```
5 .changeset/early-oranges-raise.md
@@ -0,0 +1,5 @@
+ ---
+ 'openzeppelin-solidity': minor
+ ---
+
+ `ERC721Wrapper`: add a new extension of the `ERC721` token which wraps an underlying token.
```

Fig. 12: Add ERC721 Wrapper



```
1 CHANGELOG.md
@@ -10,6 +10,7 @@
   * `Strings`: add `equal` method. ([#3774](https://github.com/OpenZeppelin/openzeppelin-cor
   * `Strings`: add `toString` method for signed integers. ([#3773](https://github.com/OpenZe
   * `MerkleProof`: optimize by using unchecked arithmetic. ([#3745](https://github.com/OpenZ
+  * `EnumerableMap`: add a `keys()` function that returns an array containing all the keys.
```

Fig. 13: Add keys() accessor to EnumerableMaps

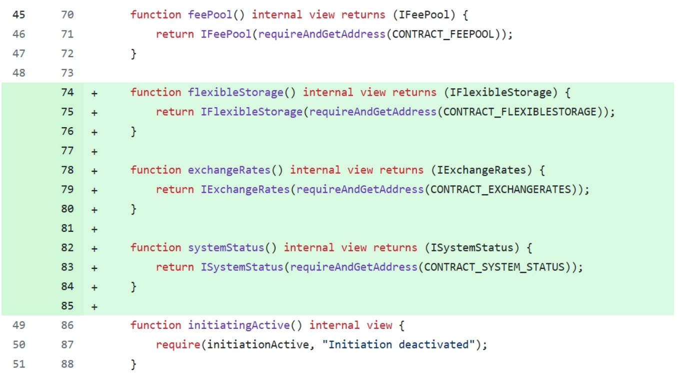
Figure 12 and 13 present enhancements in abstraction through new methods and extensions. Figure12 shows the addition of the ERC721Wrapper, a new extension that wraps an underlying ERC721 token. Figure13 shows the addition of the *keys()* method to *EnumerableMap*, which returns an array containing all the keys.

**Synthetix v2.66.0 to v2.68.2** The POC increased from 19.77% to 25.07%, and the PCF increased from 48.77% to 56.14%. According to the description in Github [3], the main updates in this version include: adding a new spot Synth and new futures markets on L2; Updating the base exchange rate and the ISynthetix and IExchanger interfaces; Updating the Optimism bridge to enable cross-chain Synth transfers. These updates introduced several new features and interface modifications, leading to an increase in code clones. We found the specific reasons are as follows:

*Adding new feature modules.* When new feature modules, such as new spot Synths and futures markets, are added, a significant amount of new code would be introduced. This can increase the code clone rate since these feature modules often share similar code logic or interface implementations. Figure 14 shows code additions in *BaseSynthetixBridge.sol*, where new feature modules, such as new spot Synths and futures markets, have been introduced. These feature modules often share similar code logic or interface implementations, which increases the code clone rate.

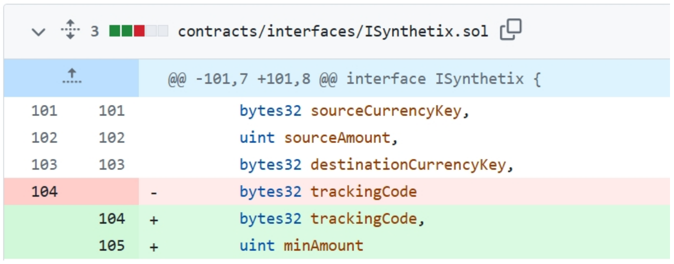
*Updating interfaces:* When updating the *ISynthetix* and *IExchanger* interfaces to include new parameters and functionalities, it often requires applying these changes in multiple places, which can also bring code clones. Figure 15 shows the update of the *ISynthetix* interface, adding *trackingCode* and *minAmount* parameters. Figure 16 presents the update of the *IExchanger* interface, similarly adding *trackingCode* and *minAmount* parameters. These updates often need to be applied in multiple places, potentially leading to code clones.

[3]https://github.com/Synthetixio/synthetix/releases/tag/v2.68.2



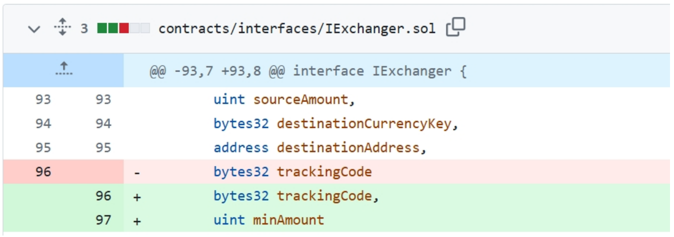
```
      function feePool() internal view returns (IFeePool) {
          return IFeePool(requireAndGetAddress(CONTRACT_FEEPOOL));
      }

+     function flexibleStorage() internal view returns (IFlexibleStorage) {
+         return IFlexibleStorage(requireAndGetAddress(CONTRACT_FLEXIBLESTORAGE));
+     }
+
+     function exchangeRates() internal view returns (IExchangeRates) {
+         return IExchangeRates(requireAndGetAddress(CONTRACT_EXCHANGERATES));
+     }
+
+     function systemStatus() internal view returns (ISystemStatus) {
+         return ISystemStatus(requireAndGetAddress(CONTRACT_SYSTEM_STATUS));
+     }
+
      function initiatingActive() internal view {
          require(initiationActive, "Initiation deactivated");
      }
```

Fig. 14: Code Additions in BaseSynthetixBridge.sol



```
3 contracts/interfaces/ISynthetix.sol
@@ -101,7 +101,8 @@ interface ISynthetix {
          bytes32 sourceCurrencyKey,
          uint sourceAmount,
          bytes32 destinationCurrencyKey,
-         bytes32 trackingCode
+         bytes32 trackingCode,
+         uint minAmount
```

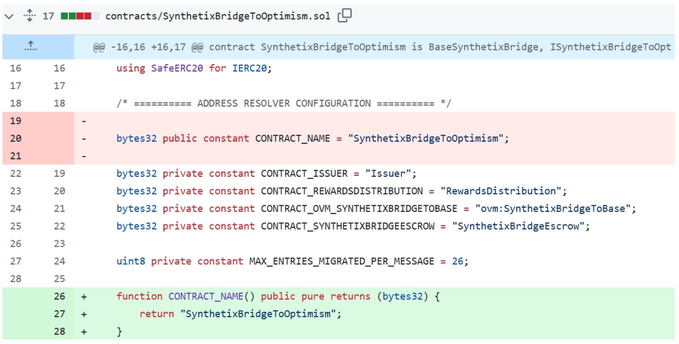
Fig. 15: The update of ISynthetix interfaces



```
3 contracts/interfaces/IExchanger.sol
@@ -93,7 +93,8 @@ interface IExchanger {
          uint sourceAmount,
          bytes32 destinationCurrencyKey,
          address destinationAddress,
-         bytes32 trackingCode
+         bytes32 trackingCode,
+         uint minAmount
```

Fig. 16: The update of IExchanger interfaces

*Implementing cross-chain functionality.* To implement cross-chain Synth transfer functionality, developers need to add similar code logic in multiple contracts to ensure consistency and reliability of cross-chain operations, which increases the code clone rate. Figure 17 shows the update of



```
17 contracts/SynthetixBridgeToOptimism.sol
@@ -16,16 +16,17 @@ contract SynthetixBridgeToOptimism is BaseSynthetixBridge, ISynthetixBridgeToOpt
      using SafeERC20 for IERC20;

      /* ========== ADDRESS RESOLVER CONFIGURATION ========== */
-
-     bytes32 public constant CONTRACT_NAME = "SynthetixBridgeToOptimism";
-
      bytes32 private constant CONTRACT_ISSUER = "Issuer";
      bytes32 private constant CONTRACT_REWARDSDISTRIBUTION = "RewardsDistribution";
      bytes32 private constant CONTRACT_OVM_SYNTHETIXBRIDGETOBASE = "ovm:SynthetixBridgeToBase";
      bytes32 private constant CONTRACT_SYNTHETIXBRIDGEESCROW = "SynthetixBridgeEscrow";

      uint8 private constant MAX_ENTRIES_MIGRATED_PER_MESSAGE = 26;

+     function CONTRACT_NAME() public pure returns (bytes32) {
+         return "SynthetixBridgeToOptimism";
+     }
```

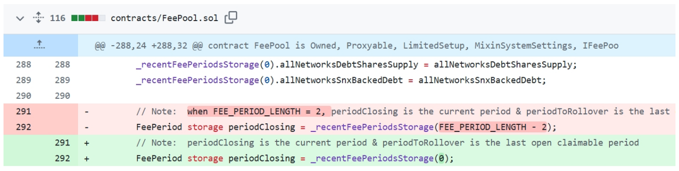
Fig. 17: The update of Optimism bridges

*SynthetixBridgeToOptimism* contract to implement cross-chain

Synth transfer functionality. To ensure the consistency and reliability of cross-chain operations, developers need to add similar code logic in multiple contracts, which increases the code clone rate.

**Synthetix v2.82.2 to v2.84.4** The POC decreased from 25.37% to 21.9%, and the PCF decreased from 56.71% to 54.75%. The main update in this version is the improvement of the *FeePool* module, which enables automatically burning the collected *sUSD* fees at the end of the fee period. This reduces user debt without requiring manual claiming of the synthetic assets. This change led to a notable decrease in the code clone rate for the following reasons:

*Refactoring code.* The improvement of the *FeePool* module involved refactoring the existing code, which reduces redundant code and repetitive logic, hence lowering the code clone rate. Figure 18 shows the refactoring of the *FeePool*

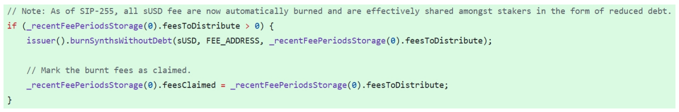


```
116 contracts/FeePool.sol
@@ -288,24 +288,32 @@ contract FeePool is Owned, Proxyable, LimitedSetup, MixinSystemSettings, IFeePoo
            _recentFeePeriodsStorage(0).allNetworksDebtSharesSupply = allNetworksDebtSharesSupply;
            _recentFeePeriodsStorage(0).allNetworksSnxBackedDebt = allNetworksSnxBackedDebt;

-           // Note:  when FEE_PERIOD_LENGTH = 2, periodClosing is the current period & periodToRollover is the last
-           FeePeriod storage periodClosing = _recentFeePeriodsStorage(FEE_PERIOD_LENGTH - 2);
+           // Note:  periodClosing is the current period & periodToRollover is the last open claimable period
+           FeePeriod storage periodClosing = _recentFeePeriodsStorage(0);
```

Fig. 18: The refactoring of FeePool module

module. The refactored code simplifies the logic by directly setting *periodClosing* to the first period in the recent fee periods storage, which can remove unnecessary complexity and redundant logic.

*Simplifying functionality.* It involved automating processes to replace manual operations, such as burning *sUSD* fees automatically. This not only simplified the code logic but also reduced possible duplicate implementations, which in turn decreased the code clone rate. Figure 19 shows the



```
// Note: As of SIP-255, all sUSD fee are now automatically burned and are effectively shared amongst stakers in the form of reduced debt.
if (_recentFeePeriodsStorage(0).feesToDistribute > 0) {
    issuer().burnSynthsWithoutDebt(sUSD, FEE_ADDRESS, _recentFeePeriodsStorage(0).feesToDistribute);

    // Mark the burnt fees as claimed.
    _recentFeePeriodsStorage(0).feesClaimed = _recentFeePeriodsStorage(0).feesToDistribute;
}
```

Fig. 19: The addition of a call to call issuer burnSynthsWithoutDebt

introduction of automated processes, such as automatically burning *sUSD* fees, to replace manual operations. The code changes indicate that, according to SIP-255, all sUSD fees are now automatically burned and effectively redistributed among stakers by reducing debt. This enhancement simplifies the code logic and reduces duplicate implementations, which decreases the code clone rate.

**Finding 4:** The occurrence of code clones in smart contracts is closely associated with the nature of development activities and project evolution. We identified three main factors influencing clone rates: 1) Feature addition and expansion. Introducing new features or expanding existing ones often increases the rate of code clones, especially when these features share similar logic or interface implementations; 2) Code optimization and refactoring. Optimizing and refactoring code usually result in a decrease in code clones. This includes consolidating similar functions, simplifying logic, and introducing more generalized implementations; 3) high-level changes: changes to a project's structure, such as adding new libraries or changing core interfaces, can significantly impact the clone rates. These changes can increase clones (if the interfaces are implemented across multiple contracts) or decrease them (if the abstractions reduce repetitive code).

## VI. Discussion

Based on our experimental results, we now discuss the implications of the findings and list some lessons learned to guide future analysis of code clones in smart contracts.

Finding 1 illuminates the high prevalence of code clones in Solidity-based smart contracts. Code clones are common in the studied smart contracts, with an average of 67.43% of functions containing code clones and 5.12% of code lines involved in clones. This indicates that, due to the specific development natures of smart contracts, such as the need for standardized and secure implementations, there is a high potential for code reuse and cloning. The prevalence of Type1 and Type2 clones indicates that developers often reuse code without significant modifications, which potentially results from the need for consistency and reliability in blockchain applications

Finding 2 reveals the significant co-evolution of code clones in smart contracts. The results show that a notable portion of clones (32.01% on average in OpenZeppelin) co-evolve, meaning they are added, changed, or deleted together as the software evolves. This co-evolution primarily occurs due to the need for code reuse, modularity, and rapid feature development in smart contracts. The high rate of co-evolution indicates the importance of managing these clones to maintain consistency and reduce potential risks in the evolving contract ecosystem.

Finding 3 challenges the traditional wisdom about the bug-proneness of code clones. Unlike studies in other software domains, our study found that code clones in smart contracts are rarely involved in bug fixes. This special result can be attributed to the highly standardized and simplified logic of smart contracts, their immutability, and the rigorous testing and verification techniques used in their development. This finding suggests that code cloning in smart contracts may be a more acceptable practice compared to traditional software development, as long as the cloned code comes from well-verified and standardized sources.

Finding 4 identifies key factors associated with the occurrence of code clones in smart contracts. The results reveal that clone rates are affected by three main factors: feature addition and expansion, code optimization and refactoring, and high-level changes to project structure. These factors can increase or decrease clone rates depending on how they are implemented. This insight provides valuable guidance for developers and project managers in understanding and managing code clones throughout the development lifecycle of smart contracts.

We believe our findings have the following implications: 1) Smart contract developers need to establish robust best practices for managing code clones, given their high prevalence (67.43% of functions containing clones) and co-evolution patterns (32.01% on average). These practices should define scenarios where code cloning is acceptable, particularly when reusing well-verified, standardized implementations. Guidance should be provided for tracking and maintaining clones throughout the contract lifecycle, with special attention to maintaining consistency during co-evolution events. 2) Tool developers should consider integrating clone detection and management features into the smart contract development environment. These tools should not only identify clones but also help developers monitor clone co-evolution patterns, verify clone consistency, and manage clone-related changes across multiple contracts. 3) Researchers should further explore the long-term impacts of clones in smart contracts, particularly in terms of maintainability and security. Additionally, they can also investigate how different types of clones (Type-1, Type-2, and Type-3) affect these aspects differently.

## VII. Threats to Validity

In this section, we will discuss potential threats to the validity of this study.

Construct Validity. We used the NiCad tool to detect code clones in Solidity smart contracts because it has been widely used in previous studies. However, the configuration of NiCad may affect our detection results. To mitigate this issue, we referred to the configurations used in previous studies [6], [19] to ensure high precision and recall.

Internal Validity. Our analysis of commit messages and code changes to identify reasons for clone rate changes might cause bias. We mitigate this by adopting a three-person validation. Two annotators with 1.5 years of experience in smart contract development independently examined the commits and code changes, and disagreements would be resolved through discussion supervised by a third researcher with 4 years of experience. We used Cohen's Kappa coefficient to assess the agreement of our analysis, which resulted in a value of 0.78, indicating that the raters had relatively high consistency.

External Validity. Although our research dataset covers multiple representative Solidity projects and a large number of smart contracts, the scale and diversity of the dataset are still limited. We acknowledge that further studies on different projects and platforms would be beneficial for the generalizability of our results.

## VIII. Conclusion

This study provides a comprehensive analysis of code cloning in Solidity-based smart contracts. It provides valuable insights into the prevalence, evolution, and impact of code clones in smart contracts.

Our experimental results presented that code clones are highly prevalent in smart contracts, with an average of 67.43% of functions containing clones and 5.12% of code lines involved in cloning, suggesting the importance of understanding and managing code clones in blockchain applications. In addition, we observed that a significant portion of clones co-evolve, meaning that developers should well manage and maintain these clones to avoid inconsistency issues in the evolving contracts. Furthermore, unlike traditional software, code clones in smart contracts are rarely involved in bug fixes. Finally, we identified three key factors associated with the occurrence of code clones in smart contracts. We believe the findings of this study can guide developers and project managers in better understanding and managing code clones throughout the development lifecycle of smart contracts.

## Acknowledgment

This work is supported by the Knowledge Innovation Program of Wuhan-Shuguang Project under grant No. 2022010801020280.

## References

[1] MarketsandMarkets, "Blockchain market by component (platforms, services (managed, professional), provider, type (public, private, hybridconsortium), deployment mode (cloud, on-premises), organization size, vertical and region - global forecast to 2029," 2024, retrieved from https://www.marketsandmarkets.com/Market-Reports/blockchain-technology-market-90100890.html.

[2] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, "Smart contract development: Challenges and opportunities," *IEEE transactions on software engineering*, vol. 47, no. 10, pp. 2084–2106, 2019.

[3] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *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 6*. Springer, 2017, pp. 164–186.

[4] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[5] M. Wohrer and U. Zdun, "Smart contracts: security patterns in the ethereum ecosystem and solidity," in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2018, pp. 2–8.

[6] C. K. Roy and J. R. Cordy, "An empirical study of function clones in open source software," in *2008 15th Working Conference on Reverse Engineering*. IEEE, 2008, pp. 81–90.

[7] M. Mondal, M. S. Rahman, R. K. Saha, C. K. Roy, J. Krinke, and K. A. Schneider, "An empirical study of the impacts of clones in software maintenance," in *2011 IEEE 19th International Conference on Program Comprehension*, 2011, pp. 242–245.

[8] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[9] R. Koschke, "Survey of research on software clones," in *Duplication, Redundancy, and Similarity in Software*, ser. Dagstuhl Seminar Proceedings (DagSemProc), R. Koschke, E. Merlo, and A. Walenstein, Eds., vol. 6301. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2007, pp. 1–24. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/DagSemProc.06301.13

[10] C. J. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, December 2008. [Online]. Available: https://doi.org/10.1007/s10664-008-9076-6

[11] M. Kondo, G. A. Oliva, Z. M. J. Jiang, A. E. Hassan, and O. Mizuno, "Code cloning in smart contracts: a case study on verified contracts from the ethereum blockchain platform," *Empirical Softw. Engg.*, vol. 25, no. 6, p. 4617–4675, nov 2020. [Online]. Available: https://doi.org/10.1007/s10664-020-09852-5

[12] D. V. Khan, I. David and S. McIntosh, "Code cloning in smart contracts on the ethereum platform: An extended replication study," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2006–2019, April 2023.

[13] X. Chen, P. Liao, Y. Zhang, Y. Huang, and Z. Zheng, "Understanding code reuse in smart contracts," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 470–479.

[14] K. Sun, Z. Xu, C. Liu, K. Li, and Y. Liu, "Demystifying the composition and code reuse in solidity smart contracts," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 796–807. [Online]. Available: https://doi.org/10.1145/3611643.3616270

[15] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 485–495.

[16] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[17] M. Mondal, C. K. Roy, and K. A. Schneider, "Connectivity of co-changed method groups: a case study on open source systems." in *CASCON*, 2012, pp. 205–219.

[18] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *2008 IEEE International Conference on Software Maintenance*. IEEE, 2008, pp. 227–236.

[19] Y. Zhao, R. Mo, Y. Zhang, S. Zhang, and P. Xiong, "Exploring and understanding cross-service code clones in microservice projects," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ser. ICPC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 449–459. [Online]. Available: https://doi.org/10.1145/3524610.3527925

[20] R. Mo, Y. Zhang, Y. Wang, S. Zhang, P. Xiong, Z. Li, and Y. Zhao, "Exploring the impact of code clones on deep learning software," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–34, 2023.

[21] O. Ehsan, F. Khomh, Y. Zou, and D. Qiu, "Ranking code clones to support maintenance activities," *Empirical Software Engineering*, vol. 28, no. 3, p. 70, 2023.

[22] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*, 2018, pp. 9–16.

[23] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 259–269.

[24] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: analyzing safety of smart contracts." in *Ndss*, 2018, pp. 1–12.

[25] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu *et al.*, "Kevm: A complete formal semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018, pp. 204–217.

[26] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2017, pp. 442–446.

[27] Consensys, "Ethereum smart contract security best practices," 2024, available at: https://consensys.github.io/smart-contract-best-practices/.

[28] L. Ante, "Smart contracts on the blockchain–a bibliometric analysis and review," *Telematics and Informatics*, vol. 57, p. 101519, 2021.

[29] Y. Gil and G. Lalouche, "On the correlation between size and metric validity," vol. 22, p. 2585–2611, 2017.

[30] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 428–439. [Online]. Available: https://doi.org/10.1145/2884781.2884848

[31] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 1–12.

[32] M. Mondal, B. Roy, C. K. Roy, and K. A. Schneider, "Investigating context adaptation bugs in code clones," in *2019 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 2019, pp. 157–168.

[33] D. Chatterji, J. C. Carver, N. A. Kraft, and J. Harder, "Effects of cloned code on software maintainability: A replicated developer study," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 112–121.

[34] P. Cuffe, "The role of the erc-20 token standard in a financial revolution: the case of initial coin offerings," 2018.

[35] A. M. Antonopoulos and G. Wood, *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[36] S. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. Knottenbelt, "Sok: Decentralized finance (defi)," in *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, 2022, pp. 30–46.

[37] L. Zhou, K. Qin, C. F. Torres, D. V. Le, and A. Gervais, "High-frequency trading on decentralized on-chain exchanges," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 428–445.

[38] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 557–560.

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