# Code Review Practices for Smart Contract on the Ethereum Platform

Aryan Haddady
ahaddady@uwaterloo.ca

Shaquille Pearson
s23pears@uwaterloo.ca

Sky Qiao
s4qiao@uwaterloo.ca

March 13, 2024

## 1 Introduction

In the rapid process of developing a software project, especially when numerous teams and developers collaborate, having an effective version control system is extremely important. One of the most popular such systems is Git, the version control system introduced by Linus Torvalds in 2005.

### 1.1 Code Review in Software Development

When developers make modifications or enhancements to a software's source code, their primary objective is to merge these changes seamlessly into the central repository of the project. ***Git***, as a version control system, provides constructs such as branches and forks, facilitating this integration process. Every modification in the source code can be visualized as a distinct node within a tree data structure, representing the post-modification state of the code. Upon the completion of these changes, developers propose their updates for potential integration into the primary codebase. Platforms such as GitHub term this proposal as a ***"Pull Request"***. In contrast, platforms like GitLab use the name ***"Merge Request"*** for a similar process.

### 1.2 Blockchain Technology and Smart Contracts

Blockchain technology, particularly platforms like Ethereum, has gained considerable attention and adoption in recent times. This technology allows developers to deploy specialized programs known as Smart Contracts. These contracts explicitly define the terms and conditions of transactions that take place within the network between different parties. The predominant programming language used for designing these smart contracts is ***Solidity***.

### 1.3 Objectives of this Paper

It is essential to note that Smart Contracts, though a specialized niche within the expansive spectrum of software development, are not immune to the practices and principles of traditional software development. The concepts that govern conventional software, including version control and code review, are equally applicable and crucial in the domain of Smart Contracts.

The primary focus of this research is to see how these pull requests are reviewed by the ones in charge of reviewing pull requests to see whether any new bugs or security issues were introduced by merged pull requests, and how big and important these projects have been.

We try to investigate the accuracy of these reviews, evaluate if any new vulnerabilities or potential bugs have been introduced through the merged requests, and assess the overall importance and popularity of the projects in question. Furthermore, we aspire to recommend potential strategies and best practices to enhance the code review processes within the paradigm of Smart Contract development.

## 2 Motivation

Code review is an integral part of software development, offering several benefits such as error detection and quality assurance. In traditional software projects, code review processes have been crucial in maintaining code quality. However, the domain of smart contracts on blockchain platforms presents unique challenges. According to the paper "Code Cloning in Smart Contracts on the Ethereum Platform: An Extended Replication Study" [1], many smart contracts deployed on the chain contain numerous bugs. This paper aims to investigate whether an efficient code review process can significantly reduce these bugs. To begin, it is essential to understand the differences in code review practices between traditional projects and smart contract projects.

### 2.1 Code Review: Traditional Projects vs. Smart Contract Projects

Distinct challenges arise in smart contract development compared to traditional software projects:

- **Emerging Technologies**:
  - *Smart Contract*: Solidity, as a primary language for smart contract development, is a relatively new language compared to established languages. This relative newness can lead to inadvertent errors as developers are not as familiar with it as other well-known programming languages like Python or Javascript.
  - *Traditional Projects*: Languages like JavaScript and Python have robust ecosystems, well-established practices, and a vast community, making error detection and resolution faster.
- **Unique Paradigm**:

  - *Smart Contract*: Smart contract development introduces a novel paradigm, divergent from traditional paradigms like back-end development.
  - *Traditional Projects*: Paradigms like back-end development are based on well-understood patterns and practices, reducing emerging challenges.

- **Tooling Deficit**:
  - *Smart Contract*: The emerging nature of the smart contract domain implies a lack of mature development and debugging tools.
  - *Traditional Projects*: Mature Integrated Development Environments (IDEs), debuggers, and testing frameworks exist, simplifying the development and testing process.

Given the aforementioned difficulties, flawed code review processes can lead to significant damage to a software project.

## 2.2 Consequences of Flawed Code Review: Traditional Projects vs. Smart Contracts

Problematic smart contracts can cause more severe problems compared to many traditional software systems because of:

- **Monetary Stakes**:
  - *Smart Contract*: Facilitates financial transactions. Imperfections can lead to massive financial losses. Even minor financial losses will make this new paradigm look terrible to potential customers.
  - *Traditional Projects*: Depending on its use case, might not always have immediate financial implications.

- **Trustworthiness**:
  - *Smart Contract*: Earning trust is vital for the blockchain ecosystem. Frequent security breaches can significantly damage this trust. Especially as the paradigm is relatively new and needs to appeal to new customers out there.
  - *Traditional Projects*: While trust is important, the impact of a breach might be localized depending on the application. People are already familiar with the applications of such projects as websites, etc., and issues will only hurt that specific faulty project rather than the whole paradigm.

### *Problem Statement*

Will the proper code review process effectively reduce the number of common errors in smart contracts?

# 3 Methodology

First, we need to have some projects to inspect their pull requests. We partitioned the projects into three categories with the following samples:

1. **Famous Smart Contract Projects**

    - Uniswap [2]. Uniswap, operating on Ethereum, is a decentralized exchange protocol employing automated market maker (AMM) principles for direct cryptocurrency trading from Ethereum wallets, eliminating traditional intermediaries. It uses liquidity pools to enable token exchange, with users contributing tokens and earning fees. Its innovative constant product formula maintains token ratios as traders participate, adjusting prices based on supply and demand.
    - Aave [3]. AAVE, an Ethereum-based DeFi protocol, offers decentralized lending and borrowing. Users can lend or borrow assets using collateral, bypassing traditional banks. Liquidity pools facilitate these activities, with lenders earning interest and borrowers accessing funds. AAVE's standout feature is "flash loans," enabling collateral-free short-term borrowing.
    - Compound [4]. Compound is a decentralized lending protocol built on the Ethereum blockchain, operating within the realm of decentralized finance (DeFi). Its primary function is to enable users to lend their cryptocurrency assets and earn interest, as well as borrow assets by collateralizing their holdings, all without the need for intermediaries like traditional banks.
    - Dai Stablecoin System(DSS) [5].DAI is an Ethereum-based decentralized stablecoin in the DeFi realm, aiming to provide users with a stable cryptocurrency tied to a specific fiat currency, typically the US Dollar. It maintains stability through over-collateralization, where users lock up crypto assets to receive DAI tokens. These tokens adjust collateral levels automatically, ensuring stability even during market fluctuations.
    - Chainlink [6]. Chainlink is a decentralized oracle network that bridges the gap between blockchain smart contracts and real-world data in the decentralized finance (DeFi) ecosystem. It utilizes "oracles" to fetch and deliver data from external sources to smart contracts, ensuring informed decision-making based on accurate data. This decentralized network of nodes employs cryptographic techniques and incentives to ensure data integrity.

2. **Middle Ground Smart Contract Projects**

    - Balancer [7]. Balancer is a decentralized finance (DeFi) protocol focused on automated portfolio management. Its core innovation is the ability to create and manage liquidity pools with multiple tokens, effectively serving as an automatic portfolio manager, liquidity provider, and price sensor. Balancer's uniqueness is its flexibility in allowing pools to contain multiple tokens with any given weight, thereby broadening the range of financial actions that can be automated on its platform.

- KyberSwap [8]. KyberSwap is a decentralized exchange protocol offering seamless token swaps. A distinctive feature is the KyberAI tool, which provides market analysis to assist users in making informed trading decisions. KyberSwap's uniqueness is its on-chain liquidity protocol that aggregates liquidity from diverse sources, providing the best rates and ensuring a smooth trading experience.
- Sushiswap [9]. SushiSwap is a decentralized cryptocurrency exchange known for its innovative approach to liquidity provisioning and yield farming. SushiSwap's uniqueness is its offering of cross-network asset transfers, ensuring users get optimal rates across various blockchain networks.
- Synthetix [10]. Synthetix is a decentralized finance (DeFi) protocol that facilitates the issuance and trading of synthetic assets. These assets mirror the price of real-world assets, such as crypto, forex, and commodities. Synthetix's uniqueness is its ability to offer high-leverage trading, a feature uncommon in many other DeFi platforms.
- Yearn Vaults [11]. Yearn Finance is a decentralized protocol that streamlines yield farming by automatically allocating user deposits to the most profitable DeFi lending protocols. Yearn Finance's uniqueness is its community-driven approach, with strategies being proposed, debated, and implemented by the Yearn community.

3. **Lesser-Known Smart Contract Projects**

- DODOEX [12]. Dodo is a project that tries to address the low fund utilization and instability of liquidity providers' portfolio issues by proposing a new algorithm called Proactive Market Maker (PMM) whose primary objective is to minimize risks for fund providers and stabilize their portfolio.
- Opium [13]. Opium is a protocol for creating, settling, and trading any decentralized derivative.
- KEEP [14]. Keep network is a privacy, interoperability, and censorship-resistance toolkit for developers on Ethereum.
- UMA [15]. UMA is an optimistic oracle and dispute arbitration system that securely allows for arbitrary types of data to be brought on-chain. UMA's oracle system provides data for projects including a cross-chain bridge, insurance protocols, custom derivatives, and prediction markets.
- Hegic [16]. Hegic is an on-chain options trading protocol on Ethereum. Options are financial derivatives that give buyers the right, but not the obligation, to buy or sell an underlying asset at an agreed-upon price and date.

Second, we will use ***Slither***, a state-of-the-art Solidity source analyzer [17]. Through its detectors, Slither can find vulnerabilities, inefficiencies, and other potential issues within smart contracts. Significantly, Slither can detect reentrancy, improper visibility configurations, and other prevalent issues, making it a critical component of our analysis. By using Slither on the pull requests for the mentioned smart contract projects, we can measure the amount of flagged issues. This provides insight into the existing code review process's

challenges. With this data, we'll do the statistical analysis and visual representations to discuss the results in depth.

Third, based on our analysis and the data we get, we will discuss the usefulness of the current code review process of the smart contracts, the frequency of the common errors that appear in the smart contract, the severity of these recurrent mistakes, and the potential implications they may have.

Fourth, we will propose an improved code review methodology specifically designed to preempt and address the common errors observed in smart contracts.

# 4 Results

## 4.1 Initial Investigation

After we collected the data, we noticed some interesting numbers from the data:

| **GitHub Repo** | **PR Open** | **PR Closed** | **PR Merged** | **Avg. Reviewers** | **Avg. Comments** | **Avg. Time to Merge (hrs)** | **Avg. Time to Close (hrs)** |
|---|---|---|---|---|---|---|---|
| Uniswap/v3-core | 6 | 105 | 214 | 1 | 1 | 181 | 334 |
| aave/aave-protocol | 9 | 50 | 418 | 1 | 0 | 66 | 363 |
| compound-finance/compound-protocol | 22 | 84 | 50 | 0 | 1 | 670 | 1304 |
| makerdao/dss | 12 | 65 | 154 | 1 | 0 | 152 | 831 |
| smartcontractkit/chainlink | 108 | 1990 | 7370 | 1 | 2 | 77 | 446 |
| yearn/yearn-vault | 4 | 111 | 291 | 1.11 | 1.21 | 112.32 | 588.8 |
| sushiswap/sushiswap | 14 | 139 | 697 | 0.13 | 2.27 | 69.05 | 1132.81 |
| Synthetixio/synthetix | 5 | 378 | 1613 | 1.2 | 0.83 | 109.98 | 968.59 |
| balancer/balancer-v2-monorepo | 11 | 204 | 1880 | 1.27 | 0.56 | 83.7 | 833.1 |
| KyberNetwork/smart-contracts | 21 | 216 | 728 | 1.04 | 0.27 | 91.72 | 310.91 |
| DODOEX/dodo-smart-contract | 1 | 0 | 4 | 0.4 | 0 | 8.84 | None |
| hegic/contracts-v1/ | 7 | 1 | 3 | 0.09 | 0.09 | 24.6 | 4393.23 |
| keep-network/keep-core | 55 | 347 | 2368 | 1.52 | 1.5 | 184.65 | 1180.38 |
| opiumprotocol/opium-contracts | 38 | 18 | 3 | 0.07 | 0.29 | 0.36 | 4162.06 |
| UMAprotocol/protocol | 9 | 262 | 3366 | 1.89 | 0.8 | 56.57 | 262.07 |

Table 1: GitHub Repositories and their PR statistics

We can see from the table that:

- High visibility projects often witnessed continuous upgrades, improvements, and bug fixes. This could potentially lead to longer pull request processing times due to the high volume.
- Popular projects benefited from extensive testing, but the high number of pull requests presented a challenge, potentially slowing down their resolution.
- Middle-ground projects typically had a balance of visibility and security. They often enjoyed faster pull request resolution times due to a manageable volume of requests.
- Lesser-known projects showcased the benefits of innovation and uniqueness, with a potential for faster issue resolutions. However, the risk of security vulnerabilities was also higher in some cases due to less rigorous testing.
- The vibrancy of the developer community and the ecosystem was found to significantly impact the pull request management process. Projects with a more active community tended to be more efficient in addressing pull requests.
- Security concerns varied across projects. High visibility often led to quicker identification and patching of vulnerabilities, while lesser-known projects ran the risk of undetected security threats.

It's worth noting that some projects with low pull request numbers actually have multiple sub-projects or are rapidly updating. For instance, some have already progressed to version 3, and with each new version, a new repository is created. This spreads out the pull requests and can result in lower numbers appearing on the graph.

## 4.2 Tool Analysis

One thing that we can investigate is whether Slither is able to find issues after the pull request was merged, because this way, we will know that although the changes were reviewed, they still introduced new issues in the smart contracts. Here are a few examples that were observed in the aforementioned Github repositories.

### 4.2.1 Well Known Projects

- **Compound Finance**: We analyzed the 51 merged pull requests (PRs) for the Compound protocol. These PRs represent collaborative efforts by developers to enhance the functionality, security, and performance of the Compound protocol's smart contracts. As these changes are integrated into the codebase upon approval, it's imperative to ensure that they adhere to best practices and do not introduce any vulnerabilities or inefficiencies.

  To facilitate this analysis, Slither is employed. Slither is designed to automatically identify potential issues in Solidity smart contracts. It achieves this by employing a set of predefined "checks" that inspect various aspects of the code. These checks

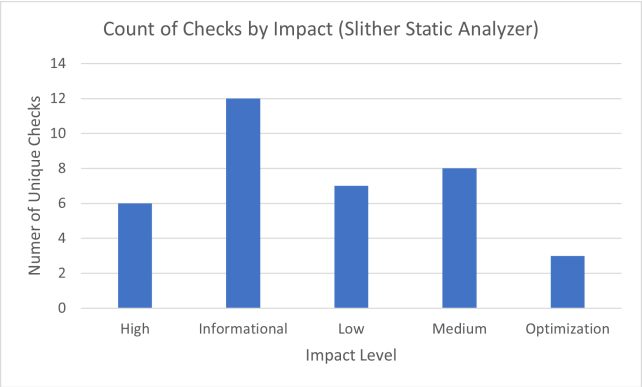


Figure 1: Slither Analysis Results

range from detecting common vulnerabilities like reentrancy attacks and uninitialized variables to identifying coding patterns that could lead to suboptimal performance or security risks.

"Impact" and "confidence" are terms used in Slither for smart contracts. "Impact" indicates the severity of a vulnerability, with levels like High, Medium, and Low, reflecting potential consequences. "Confidence" shows the reliability of the detection, ranging from High (strong evidence) to Low (requires validation). Both factors guide prioritization and resource allocation for fixing vulnerabilities in smart contracts.

Of the 51 pull requests analyzed with Slither, we found 6 unique cases of high-impact vulnerabilities in **figure 1** with at least a medium confidence level, these results are as follows:

- **Arbitrary-send-erc20**: This vulnerability is of high impact because it involves arbitrary sending of ERC-20 tokens. When arbitrary tokens are sent using the transfer function, it can lead to various security risks. For instance, if the contract lacks proper authorization or checks, an attacker could send tokens that they're not supposed to control, leading to a loss of user funds or destabilizing the token economy.
- **Controlled-delegatecall**: The use of delegatecall is a powerful but dangerous feature in Ethereum smart contracts. If not controlled carefully, it can introduce significant vulnerabilities. A high impact is assigned because misuse of delegatecall can allow an attacker to execute arbitrary code from another contract, effectively compromising the security of the entire system.
- **Controlled-array-length**: Accessing arrays without proper bounds checking can result in out-of-bounds errors, leading to unexpected behavior or crashes. Since these issues can have severe implications, especially when exploited by attackers, this vulnerability is given a high impact.
- **Uninitialized-state**: Uninitialized state variables can lead to unpredictable

behavior and potentially enable attackers to exploit unintended functionality or manipulate contract behavior. The high impact is due to the potential for data leakage, loss of funds, or overall disruption of the contract's functionality.

- **Arbitrary-send-eth**: Similar to arbitrary sending of ERC-20 tokens, sending arbitrary Ether without proper checks or authorization can lead to loss of funds or unexpected behavior. This vulnerability is assigned a high impact due to its potential financial and operational consequences.
- **Unchecked-transfer**: The unchecked use of the transfer function can lead to contract funds getting stuck if the transfer fails, which might happen for various reasons such as out-of-gas errors. Since this could potentially lead to loss of funds and contract dysfunction, it is classified as having a high impact.

The vulnerability analysis of the Compound protocol's smart contracts highlighted several high-impact vulnerabilities that could pose significant risks to the protocol's security and functionality. Notably, vulnerabilities like "Arbitrary-send-erc20" (48 instances), "Unchecked-transfer" (228 instances), "Controlled-delegatecall" (409 instances), "Controlled-array-length" (429 instances), and "Uninitialized-state" (212 instances) were identified. These counts emphasize the widespread presence of these vulnerabilities, underscoring the urgency for targeted remediation efforts to enhance the protocol's security posture and mitigate potential risks. Addressing vulnerabilities with high counts is crucial for minimizing attack vectors and ensuring the robustness of the Compound protocol.

#### 4.2.2 Middle Ground Projects

- **Kyber Network**: Within the repository of *KyberNetwork/smart-contracts*, there were a significant 1880 merged pull requests. The average time taken to merge a pull request amounted to 84 hours. Though this may appear fast, it could have been a clue to the rushed review process. One notable pull request added support for "KyberFprReserveV2 supporting WETH as quote". This introduced a new file named ***KyberFprReserveV2.sol***, which unfortunately caused the problem of missing zero-address validation check. Missing this validation could result in the permanent loss of money if they are sent to the zero address. Slither showed this problem with the log:

```
Withdrawable3.withdrawEther(uint256,address).sendTo
(utils/Withdrawable3.sol#29) lacks a zero-check on :
    - (success) = sendTo.call{value: amount}()
```

This pull request also has the problem of reentrancy vulnerability. Slither produced the following log for this problem:

```
Reentrancy in KyberFprReserveV2.doTrade
```

```
(IERC20,uint256,IERC20,address,uint256,bool)
(reserves/KyberFprReserveV2.sol#310-390):
    External calls:
    - conversionRatesContract.recordImbalance
    (destToken,int256(destAmount),0,block.number)
    (reserves/KyberFprReserveV2.sol#351-356)
    ...
    External calls sending eth:
    - weth.deposit{value: msg.value}() (reserves/KyberFprReserveV2.sol#359)
    - (success) = destAddress.call{value: destAmount}()
    (reserves/KyberFprReserveV2.sol#385)
    Event emitted after the call(s):
    - TradeExecute(msg.sender,srcToken,srcAmount,destToken,destAmount,destAddress)
    (reserves/KyberFprReserveV2.sol#389)
```

- **Yearn Vaults**: Within the *yearn/yearn-vaults* repository, the PR titled "chore: release 0.4.5 (546)" introduced a new file named ***BaseFeeOracle.sol***. However, Slither identified several problems within this file.

  One of the problems detected by Slither is the missing zero-address validation check in the methods 'setPendingGovernance' and 'setBaseFeeProvider'. This problem was detected by Slither by the following log:

```
BaseFeeOracle.setPendingGovernance(address)._governance
(BaseFeeOracle.sol#93) lacks a zero-check on :
    - pendingGovernance = _governance (BaseFeeOracle.sol#95)
BaseFeeOracle.setBaseFeeProvider(address)._baseFeeProvider
(BaseFeeOracle.sol#114) lacks a zero-check on :
    - baseFeeProvider = _baseFeeProvider (BaseFeeOracle.sol#116)
```

  Note that this repository has a long average merge time of 112 hours and it only has a small number of merged PRs (291 PRs). So with both a low number of PR merged and a long merge time, it suggests the code review process is being treated carefully. As a result, the number of detected issues appears proportionally fewer compared to other smart contract repositories.

- **Synthetix**: In the repository *Synthetixio/synthetix*, in the commit of id "d0070a7", it introduce a new file ***Migration_EnifOptimismStep1.sol***. One of the problems detected by Slither was related to setting array length with a user-controlled value, which is a quite dangerous action. Slither provided this information through the following log:

```
ExchangeRates contract sets array length
with a user-controlled value:
    - aggregatorKeys.push(currencyKey)
```

Furthermore, another detected problem was the use of multiplication after division, which could lead to precision errors. The following log from Slither shows this issue:

```
Reentrancy in PerpsV2MarketState.deleteDelayedOrder(address):
    External calls:
    - legacyState.deleteDelayedOrder(account)
    State variables written after the call(s):
    - _delayedOrderMigrated[account] = true
```

#### 4.2.3 Lesser-Known Projects

- **DODOEX** There were four merged pull requests in this repository. The number makes sense as it is a lesser-known project. In one notable instance (which was pull request number 1), a pull request was reviewed and accepted within approximately 7 hours and 45 minutes. Upon subsequent analysis with Slither, it was discovered that this merge introduced several new issues. For instance, with this merge a new function named ***sellTokenToEth*** in ***DODOEthProxy.sol*** file that introduces a reentrancy issue. The output of slither for this was

```
Reentrancy in DODOEthProxy.sellTokenToEth(address,uint256,uint256)
(contracts/DODOEthProxy.sol#123-137):
 External calls:
 - msg.sender.transfer(receiveEthAmount)
 (contracts/DODOEthProxy.sol#134)
 Event emitted after the call(s):
 - ProxySellTokenToEth(msg.sender,baseTokenAddress,tokenAmount,receiveEthAmount)
 (contracts/DODOEthProxy.sol#135)
```

Another issue introduced in this merge was the reentrancy issue in the same file with the addition of another function named ***buyTokenWithEth***. Slither logged this message this time:

```
Reentrancy in DODOEthProxy.buyTokenWithEth(address,uint256,uint256)
(contracts/DODOEthProxy.sol#139-158):
 External calls:
 - msg.sender.transfer(refund) (contracts/DODOEthProxy.sol#154)
 External calls sending eth:
 - IWETH(_WETH_).deposit{value: payEthAmount}()
 (contracts/DODOEthProxy.sol#148)
 - msg.sender.transfer(refund) (contracts/DODOEthProxy.sol#154)
 Event emitted after the call(s):
 - ProxyBuyTokenWithEth(msg.sender,baseTokenAddress,tokenAmount,payEthAmount)
 (contracts/DODOEthProxy.sol#156)
```

The reentrancy issue described here is detected by these two detectors of slither:

- **reentrancy-event**: This detector detects reentrancies that allow manipulating the order or value of emitted events. This can potentially become problematic for components that rely on the value of events.
- **reentrancy-unlimited-gas**: This detector detects reentrancy issues in general. It means that this reentrancy can lead to an over-expenditure of gas as the code does not protect from reentrancy if the gas price changes.

The table below presents a summary of issues and their respective **impact** and **confidence** resulting from running Slither before and after merging the aforementioned pull request. As can be seen, the number of issues has increased indicating that the reviewing process was not able to identify those issues.

| | Level | Before Merge | After Merge |
|---|---|---|---|
| Impact | High | 4 | 8 |
| | Medium | 7 | 14 |
| | Low | 7 | 12 |
| | Informational | 25 | 28 |
| | Optimization | 2 | 2 |
| Confidence | High | 18 | 18 |
| | Medium | 27 | 46 |
| | Low | 0 | 0 |

Table 2: Stats Before & After Merge in Pull Request #1 in DODOEX Project

- **Opium**: There were three merged pull requests in this repository. In one of them (which was pull request number 7), two new files named ***BalanceHelper.sol*** and ***PayoutHelper.sol*** were added. Both of these files introduced new issues that Slither was able to identify. For instance, in the ***PayoutHelper.sol*** file, an interface named ***IDerivativeLogic*** was imported but its functions were not implemented yet. The slither log for this issue was:

```
    IDerivativeLogic (contracts/Interface/IDerivativeLogic.sol#7-49)
    does not implement functions:
- IDerivativeLogic.allowThirdpartyExecution(bool)
 (contracts/Interface/IDerivativeLogic.sol#45)
- IDerivativeLogic.getAuthorAddress()
 (contracts/Interface/IDerivativeLogic.sol#28)
- IDerivativeLogic.getAuthorCommission()
 (contracts/Interface/IDerivativeLogic.sol#32)
- IDerivativeLogic.getExecutionPayout(LibDerivative.Derivative,uint256)
 (contracts/Interface/IDerivativeLogic.sol#24)
- IDerivativeLogic.getMargin(LibDerivative.Derivative)
 (contracts/Interface/IDerivativeLogic.sol#17)
- IDerivativeLogic.isPool()
```

```
 (contracts/Interface/IDerivativeLogic.sol#41)
- IDerivativeLogic.thirdpartyExecutionAllowed(address)
 (contracts/Interface/IDerivativeLogic.sol#37)
- IDerivativeLogic.validateInput(LibDerivative.Derivative)
 (contracts/Interface/IDerivativeLogic.sol#11)
```

The table below depicts the total number of introduced issues with the addition of these two contracts.

| | Level | BalanceHelper.sol | PayoutHelper.sol |
|---|---|---|---|
| Impact | High | 0 | 0 |
| | Medium | 0 | 0 |
| | Low | 1 | 1 |
| | Informational | 30 | 8 |
| | Optimization | 1 | 5 |
| Confidence | High | 14 | 13 |
| | Medium | 18 | 1 |
| | Low | 0 | 0 |

Table 3: Stats of Introduced Issues in Pull Request #7 in Opium Project

- **Hegic**: There were three merged pull requests in this repository. For instance, in the pull request number 11, there were changes made in files ***HegicOptions.sol*** and ***HegicERCPool.sol***, which introduced compiler errors, but the pull request was still approved. The time it took for this merge request to get approved was about 23 hours. The compile errors were the absence of the keyword *override* for two field variables. The issue was resolved 6 days later in the pull request number 13.

#### 4.2.4 Analysis

Our observations reveal that the current code review process for smart contracts mainly focuses on understanding the function of the code rather than the identification of potential problems. A significant number of comments seem to interrogate the functionality (”What is this doing?”) rather than pointing out possible security threats (”This is insecure”). This emphasis might inadvertently let some of the bugs go unnoticed during the review.

Consolidating the data across the projects, we’ve identified the top three recurring errors in the smart contracts:

- **Reentrancy-events:** This error is the most frequent and could be particularly dangerous as it indicates potential reentrancy attacks, where an adversary could potentially drain funds from a contract.
- **Unchecked-transfer:** The second most common error is transfers of funds are made without verifying the validity of the recipient’s address. This oversight may result in the permanent locking of funds.

- **Arbitrary-send-eth:** The third most common error. This indicates scenarios where Ethereum can be sent to arbitrary addresses, which might lead to unintentional or malicious fund transfers.

#### 4.2.5 Solution

To address the common issues identified in smart contract code reviews, we propose an enhanced code review methodology, that can identify the commonly observed errors. The methodology comprises the following strategies:

- **Focused Error Checklists:** Maintain a checklist based on the top recurring errors, ensuring that reviewers explicitly look for these vulnerabilities during the review process.
- **Educational Workshops:** Organize periodic training sessions for developers and reviewers to stay updated on the latest vulnerabilities and best practices in smart contract development.
- **Automated Tool Integration:** Incorporate state-of-the-art tools like *Slither* into the review process, ensuring automated checks are run before human reviews. This can help in catching common mistakes even before the review begins.

## 5 Conclusion

Smart contracts, with their unique finance-related ecosystem, make the process of code reviews even more crucial. Through this paper, we investigate the current code review practices for 15 smart contracts on Ethereum. Our study reveals that while popular and established smart contract projects tend to have rigorous review processes, the primary focus of code review is not on finding bugs. As a result, this weakens the effectiveness of code reviews in identifying and rectifying issues. The occurrence of oversight in code review process applies to small and middle ground smart contracts as well, indicating that this is an issue that happens commonly and needs more attention and work in order to get fixed.

We use the ***Slither*** tool to analyze 15 smart contracts' pull requests. Alarmingly, we identified issues being introduced into the files without the reviewer's notice. This oversight becomes especially concerning when we consider that blockchain is primarily used for financial purposes.

For the continued growth and betterment of blockchain and smart contract development, the code review process must focus on detecting bugs as well, not just the style of the code. Our research identifies common bugs in smart contracts that reviewers overlooked during the code review process. We propose an enhanced code review approach that can help the entire community towards a more secure and robust smart contract environment.

## References


[1] F. Khan, I. David, D. Varro, and S. McIntosh. Code cloning in smart contracts on the ethereum platform: An extended replication study. *IEEE Transactions on Software Engineering*, 49(4):2006–2019, April 2023.

[2] Uniswap v3. https://github.com/Uniswap/v3-core.

[3] Aave protocol. https://github.com/aave/aave-protocol.

[4] Compound. https://github.com/compound-finance/compound-protocol.

[5] Makerdao dss. https://github.com/makerdao/dss.

[6] Chainlink. https://github.com/smartcontractkit/chainlink.

[7] Balancer. https://github.com/balancer/balancer-v2-monorepo.

[8] Kyber network. https://github.com/KyberNetwork/smart-contracts.

[9] Sushiswap. https://github.com/sushiswap/sushiswap.

[10] Synthetix. https://github.com/Synthetixio/synthetix.

[11] Yearn vault. https://github.com/yearn/yearn-vault.

[12] Dodo. https://github.com/DODOEX/dodo-smart-contract.

[13] Opium protocol. https://github.com/opiumprotocol/opium-contracts.

[14] Keep network core. https://github.com/keep-network/keep-core.

[15] Uma protocol. https://github.com/UMAprotocol/protocol.

[16] Hegic. https://github.com/hegic/contracts-v1/.

[17] Trail of Bits. Slither: A solidity source analyzer. https://github.com/crytic/slither#detectors, 2021.