Detecting Code-Comment Inconsistencies in Smart Contracts by Combining LLM and Program Analysis

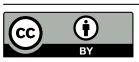
JIASHUO ZHANG, Peking University, China
JIACHI CHEN*, Zhejiang University, China
TING ZHANG, Peking University, China
YUE LI, Taiyuan University of Technology, China
DAOYUAN WU, Lingnan University, China
YANLIN WANG, Sun Yat-sen University, China
JIANBO GAO, Beijing Jiaotong University, China
TING CHEN, University of Electronic Science and Technology of China, China
ZHONG CHEN*, Peking University, China

Smart contracts have attracted rapid development and widespread application. Due to the complexity of real-world smart contracts, it is error-prone to correctly enforce all intended functionalities in code implementations, resulting in unintended functional behaviors and security issues in practice. Code-comment inconsistency detection has emerged as an important solution to these issues, which leverages the redundant functional specifications in comments to detect code implementations that violate developers' intentions. However, existing inconsistency detection solutions are typically pattern-based and limited to fixed types of inconsistencies, which prevents them from detecting the diverse inconsistencies between real-world code implementations and casually written comments. To bridge the gap, this paper presents SmartComment, the first technique that combines LLMs with program analysis techniques for detecting code-comment inconsistencies in smart contracts. SmartComment introduces an LLM-driven workflow to identify inconsistencies and incorporates various program analysis techniques into the workflow, including comment propagation and code context extraction for generating input context for inconsistency detection, as well as program variant generation and differential analysis for inconsistency confirmation. Our evaluation results show that SmartComment detects 203 valid inconsistencies from a dataset of 1,000 real-world contracts with a precision of 79.9%, highlighting its effectiveness in detecting prevalent and diverse real-world inconsistencies. Compared to previous work, SmartComment achieves both higher precision and recall, detecting over 90% of inconsistencies that existing methods fail to identify. Furthermore, an ablation experiment demonstrates the effectiveness of incorporating program analysis techniques into SmartComment, improving the F1-score from 58.7% to 81.3%.

CCS Concepts: • **Software and its engineering → Software maintenance tools**; **Software maintenance tools**; **Software defect analysis**.

*Corresponding authors.

Authors' Contact Information: Jiashuo Zhang, School of Computer Science, Peking University, Beijing, China, zhangjiashuo@pku.edu.cn; Jiachi Chen, Zhejiang University, Hangzhou, China, chenjch86@mail.sysu.edu.cn; Ting Zhang, School of Computer Science, Peking University, Beijing, China, zhangting@stu.pku.edu.cn; Yue Li, Taiyuan University of Technology, Taiyuan, China, liyue@tyut.edu.cn; Daoyuan Wu, Lingnan University, Hong Kong, China, daoyuanwu@ln.edu.hk; Yanlin Wang, Sun Yat-sen University, Zhuhai, China, yanlin-wang@outlook.com; Jianbo Gao, Beijing Jiaotong University, Beijing, China, gao@bjtu.edu.cn; Ting Chen, University of Electronic Science and Technology of China, Chengdu, China, brokendragon@uestc.edu.cn; Zhong Chen, School of Computer Science, Peking University, Beijing, China, zhongchen@pku.edu.cn.



ACM 2994-970X/2026/7-ARTFSE105
https://doi.org/10.1145/3808112

Additional Key Words and Phrases: Blockchain, Smart Contract, Code-Comment Inconsistency, LLM

ACM Reference Format:
Jiashuo Zhang, Jiachi Chen, Ting Zhang, Yue Li, Daoyuan Wu, Yanlin Wang, Jianbo Gao, Ting Chen, and Zhong Chen. 2026. Detecting Code-Comment Inconsistencies in Smart Contracts by Combining LLM and Program Analysis. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE105 (July 2026), 23 pages. https://doi.org/10.1145/3808112

1 Introduction

Smart contracts have attracted rapid development and widespread applications. Since being first introduced by Ethereum, millions of smart contracts have been deployed on-chain, attracting billions of real-world transactions [44]. Due to the prevalent real-world security incidents [15], it is essential to carefully design and thoroughly enforce the intended functionality for smart contracts to ensure that they behave as intended in all situations [40]. However, the complexity of real-world smart contracts makes it challenging and error-prone for developers to accurately implement all intended functional requirements in the code implementations. Implementations that violate the developers' intentions can lead to unexpected behaviors in practice, resulting in real-world security issues and financial losses [53, 54].

Code-comment inconsistency detection could become a promising solution for detecting code implementations that fail to align with developers' intentions [38, 39, 52]. The key observation is that comments often contain important information about the functionalities expected by developers, such as detailed function behaviors or access control rules. Therefore, cross-checking the current code implementation with the comments can help detect implementation issues that violate developers' intentions. However, despite their promising potential, there has been little prior work on detecting code-comment inconsistencies in smart contracts. To our knowledge, Hao *et al.* [13] proposed a tool called SmartCoco, which relies on pre-defined patterns to extract semantics from structured comments and checks for three fixed types of inconsistencies in the implementation, such as failing to emit an event required by the comment. However, pattern-based solutions often face fundamental challenges when handling the diverse functionalities of real-world smart contracts [8], preventing them from being effectively applied in practice.

The advent of large language models (LLMs) could bring new opportunities. With LLMs' promising capabilities in code understanding and reasoning [11, 14], it could be possible to enable the detection of various inconsistencies between the diverse functionalities of real-world smart contracts and their casual comments, especially inconsistencies that general pre-defined detection patterns cannot cover. However, previous studies have shown that directly leveraging LLMs in complex tasks like code-comment inconsistency detection could result in high false positive/negative rates [36]. Specifically, the complexity of real-world smart contracts and the scattered information related to a specific detection task can cause LLMs to become "lost in the middle" [24], leading to high false negatives. Furthermore, the limited capability of LLMs to reason through complex comment/code logic can cause them to experience misjudgement during inconsistency detection, resulting in high false positive rates in practice [6].

To address these limitations, we propose SmartComment, the first technique that combines LLMs with program analysis techniques for detecting code-comment inconsistencies in smart contracts. SmartComment leverages an LLM-based workflow to identify inconsistencies and integrates multiple program analysis techniques into the workflow. Specifically, it utilizes comment propagation and code context extraction techniques to enhance the input context for LLM-based inconsistency detection of each target function. The comment propagation process augments and aggregates comment context related to the target function by propagating comments from related code entities and refining them to the target function, and the code context extraction process extracts code entities related to the target function from extensive code implementations. For

each identified inconsistency candidate, SmartComment leverages LLM-based program variant generation to reveal the potential functional differences behind inconsistency candidates and employs both LLM-based confirmation and differential analysis to validate the inconsistencies.

We implemented SmartComment and conducted an empirical evaluation on real-world smart contracts. Specifically, we randomly sampled two datasets from 498,969 contracts collected from Etherscan [10], each designed to represent the full population with a 95% confidence level and a 5% margin of error. We then constructed the first dataset, consisting of 1,000 randomly sampled contracts, to evaluate SmartComment 's capability in detecting real-world inconsistencies. The results demonstrate that SmartComment successfully detects 203 valid inconsistencies in these contracts with a precision of 79.9%, revealing that 127 (12.7%) of these smart contracts contain at least one inconsistency. These results highlight the prevalence of code-comment inconsistencies in the wild and reveal their diversity across key functional logic, such as access control and condition checks, underscoring the necessity for effective detection tools. Second, we compiled an annotated dataset containing 200 sampled smart contracts to compare SmartComment's effectiveness with baselines in terms of precision, recall, and F1-score. The comparison with prior work SmartCoco [13] shows that SmartComment achieves both higher precision (77.1% *vs.* 20.0%) and recall (86.0% *vs.* 9.3%), and over 90% of inconsistencies it detected cannot be detected by SmartCoco. Furthermore, the ablation study demonstrates that SmartComment's comment propagation and inconsistency confirmation mechanism can effectively improve the F1-score from 58.7% to 81.3%, highlighting the importance of combining LLMs with program analysis techniques to achieve accurate results.

This paper makes the following key contributions:

- We propose SmartComment, the first technique that combines LLMs with program analysis techniques for detecting code-comment inconsistencies in smart contracts. Without relying on any pre-defined detection patterns, SmartComment enables the detection of diverse inconsistencies between the various code and comments of real-world smart contracts.
- We incorporate a set of new approaches into SmartComment. It utilizes comment propagation and code context extraction to enhance input context for inconsistency detection and employs program variant generation and differential analysis for inconsistency validation.
- We implemented SmartComment and evaluated its effectiveness on real-world smart contracts. The results demonstrate SmartComment's capability in detecting code-comment inconsistencies and reveal the prevalence and diversity of inconsistencies in the wild.
- We make the source code of SmartComment and all experimental data available at https://github.com/SmartComment-Tool/Artifact, to support further studies in this field.

2 Background and Motivation

2.1 Comments in Smart Contracts

Comments are a very important part of smart contracts. Solidity supports single-line comments ("//...") and multi-line comments ("/*...*/") to enable comment practices. When implementing smart contracts, developers often use comments to document detailed functional specifications, such as expected behaviors, input/output interfaces, access control policies, and other information. Comments not only provide essential functional specifications for developers but also offer guidelines for end-users when interacting with the smart contract [35].

2.2 Code-Comment Inconsistency Detection

Comments provide relatively independent and complementary information about the expected functionality beyond the code implementation, enabling cross-checking between the code and comments to detect implementation issues [38]. Specifically, code-comment inconsistencies refer to

```
1 function sell(address from, address to, uint256 amount) public {
2     // TAX SELLERS 25% WHO SELL WITHIN 24 HOURS
3     if (_buyTimestamp[from] != 0 && (_buyTimestamp[from] + 24 hours >= block.
      timestamp)) {
4         _fee = 15;
5     } else {// Determine fee under other situations
6         _fee = ...;
7     }
8     transferWithFee(from, to, amount, _fee); // sell the token and transfer fee
9 }
```

Fig. 1. A motivation example of SmartComment

functional differences between the current implementation and the expected functionality specified by the developer's comments. Such inconsistencies can lead to unexpected behaviors that violate the developer's intentions, compromising software reliability and maintainability in practice [33, 42].

2.2.1 Limitations of Existing Solution. The diverse functionalities of real-world smart contracts and the casual writing style of their comments pose significant challenges to code-comment inconsistency detection techniques. Existing techniques [13] often use pattern-based methods to detect inconsistencies, which employ pre-defined patterns to extract specific types of constraints from comments and identify fixed types of inconsistencies by comparing the code against these constraints. However, applying such pattern-based solutions to real-world smart contracts presents two key limitations. First, real-world comments often have casual writing styles [33], such as inline comments, acronyms, or typos, which can be challenging for patterns to cover. For example, SmartCoco [13] uses several templates to analyze comments, which can only handle certain structured comments like *"param_a must not be value_b"*. Second, these solutions typically focus only on fixed types of inconsistencies that can be characterized by low-level patterns, such as missing input parameter checks. However, those low-level patterns can fail to cover the diverse real-world inconsistencies, especially those that are semantic-related and contract-specific.

2.2.2 Motivation Example. Fig. 1 illustrates an inconsistency in a real-world smart contract, which cannot be detected by pattern-based solutions like SmartCoco [13]. The inconsistency occurs because the developers intended to set the fee at 25% when a buyer wants to resell a token in 24 hours (the comment *"TAX SELLERS 25% WHO SELL WITHIN 24 HOURS"* in line 2), but the code mistakenly sets the fee at 15% in line 4. However, since these comments do not contain structured descriptions, such as *"_fee must be 25"*, pre-defined patterns cannot extract any program constraints from the comment, and thus cannot detect this inconsistency.

The motivation example demonstrates that detecting real-world inconsistencies requires understanding diverse semantics behind the various code implementations and casually written comments, which often exceed the scope of pre-defined detection patterns or rules. Motivated by such observation, this paper investigates whether we can leverage LLM's promising capabilities in code/comment understanding to address the limitations of existing tools and enable the detection of diverse code-comment inconsistencies in practice.

3 Design of SmartComment

In this section, we present SmartComment 's overall design in Section 3.1 and describe its detailed workflow from Section 3.2 to Section 3.4.

3.1 Overview

Fig. 2 illustrates the workflow of SmartComment, where the blue blocks represent LLM-based tasks and green blocks represent program analysis tasks. SmartComment takes the source code of

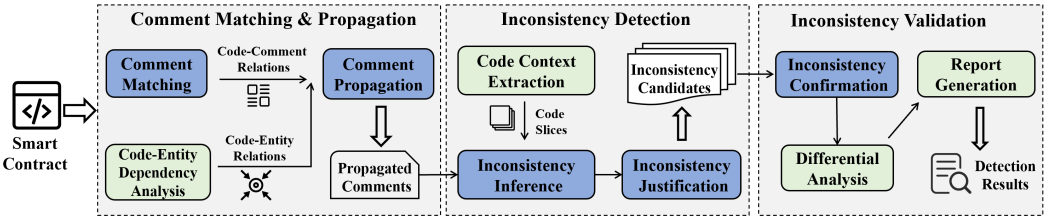


Fig. 2. The workflow of SmartComment.

(a) Comment Matching

```
You are a professional smart contracts analyzer. You need to <Comment Matching Requirements>
# <Descriptions of Code Entities> <Few-shot Examples>
# Output Format: <Format with Example>
# The contract to analyze: <Source Code>
```

(b) Comment Propagation

```
You are a professional smart contracts analyzer. You need to <Comment Propagation Requirements> # <Descriptions of Code Relations> <Few-shot Examples> # Output Format: <Format with Example>
# Comments and Code of Related Code Entities: <Comment, Code>
# Comment and Code of the target function: <Comment, Code>
```

(c) Inconsistency Inference

```
You are a professional code reviewer for smart contracts. You need to <Inconsistency Inference Requirements>
# <Description of Code-Comment Inconsistency> <Few-shot Examples>
# Output Format: <Format with example>
# Code Context: <Code Slice for the target function>
# Comment Context: <Propagated Comment for the target function>
```

(d) Inconsistency Justification

```
You are a professional code reviewer for smart contracts. You need to <Program Variant Generation Requirement>
# <Description of Code-Comment Inconsistency> <Few-shot Examples>
# Output Format: <Format with example>
# Origin Implementation: <Code, Comment>
# Inconsistency Info: <Inconsistency>
# Functions/Variables in Contract: <Declarations>
```

(e) Inconsistency Confirmation

```
You are a professional smart contract developer. You need to <Inconsistency Confirmation Requirements>
# <Description of Code-Comment Inconsistency> <Few-shot examples>
# Output Format: <Format with Example>
# Origin Implementation: <Code, Comment>
# Inconsistency Info: <Inconsistency>
# Implementation Suggested by Code Reviewer: <Program Variant>
```

Fig. 3. Prompt templates used in LLM-based tasks of SmartComment

smart contracts as input and performs a three-phase analysis to detect code-comment inconsistencies: (1) *comment matching and propagation* (Section 3.2), (2) *inconsistency detection* (Section 3.3.2), and (3) *inconsistency validation* (Section 3.4). During *comment matching and propagation*, SmartComment prepares the comment-related context used for the LLM-based inconsistency detection. SmartComment first matches comments to the specific code entities they explicitly comment on. Then, it uses static analysis to extract code entities related to each target function and applies the LLM to propagate and refine comments from these entities to the target function. During the *inconsistency detection* phase, SmartComment introduces an LLM-based inconsistency inference process to detect inconsistencies based on the code and comment context of each target function, and employs an inconsistency justification process to extract the functional differences perceived by the LLM that cause the inconsistency. During the *inconsistency validation* phase, SmartComment employs LLM-based inconsistency confirmation and differential analysis techniques to analyze each identified inconsistency candidate, filter out false positives, and finally report the results.

3.2 Comment Matching and Propagation

3.2.1 Motivation and Design Choice. Determining whether a target function aligns with the comment-specified intention requires examining not only the comments and code explicitly located within the function's code block. It also requires considering the code and comment contexts from other related code entities, such as the interfaces that the function implements. However, in complex real-world smart contracts, the code entities related to a specific function might be far apart in the source code or even in different files. Consequently, LLMs could fail to comprehensively identify all related comment information scattered across various code locations, resulting in inaccuracies in code-comment inconsistency detection.

For example, Fig. 4 shows the implementation of a *setApprovalForAll* function (line 7-12) that does not contain any explicit comments, thus appearing to have no inconsistencies. However, it actually has two code-comment inconsistencies. Line 8 and 10 provide the expected fixes to these inconsistencies. First, this function implements the *IERC721A* interface, where the developer's comment in line 15 requires a check to ensure that the *operator* cannot be the caller, which is

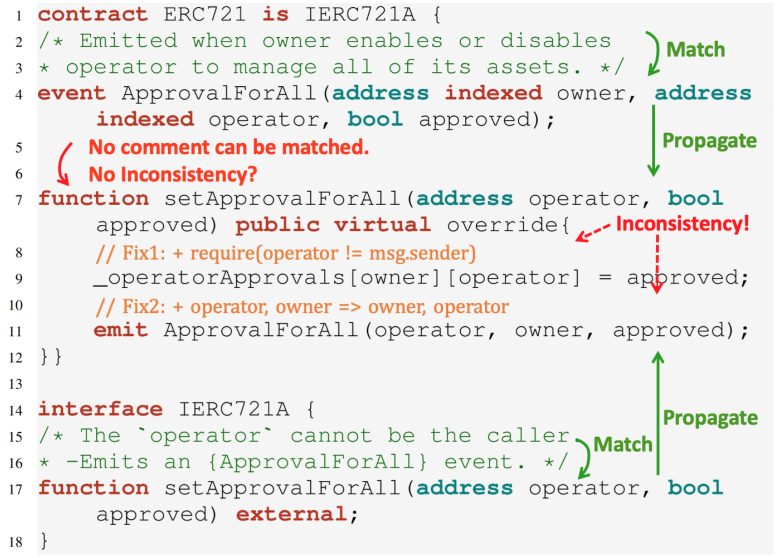


Fig. 4. A motivation example for comment propagation

missing in the current implementation. Second, the function emits an *ApprovalForAll* event with the parameters reversed. The event declaration describes that the first parameter is named "owner" and the second is named "operator" (line 4), and the comment further specifies that the "owner" must be the token owner approving the tokens (lines 2-3). However, the implementation emits the event (line 11) with the parameters reversed, leading to another inconsistency.

Motivated by this observation, SmartComment introduces a comment matching and propagation mechanism. This process aims to transform implicit code-comment matching relations in various code locations into explicit and enhanced comments for the target function, helping to prevent LLMs from getting "lost in the middle" [24] and improving the accuracy of inconsistency detection. Specifically, it first matches each comment to the explicit code entities it comments on. Then, it performs static analysis to identify code entities related to the target function and employs LLMs to propagate and refine these comments from related code entities to the target.

3.2.2 Comment Matching. Comments should first be matched to the corresponding code entities so that they can be propagated. For example, it is essential to ensure that a comment is associated with a function declaration within an interface before propagating it to other functions implementing the interface. As shown in Fig. 3a, SmartComment prompts the LLM to match each comment to the specific code entity it explicitly comment for. It provides the LLM with descriptions of code entities in smart contracts, including contract-level (contract/abstract contract/library/interface), function-level (function/modifier), statement-level (statement), and variable-level (variable/event/errors) entities, along with few-shot examples. SmartComment inputs the source code of each contract and requires the LLM to match each comment to the code entity it explicitly comment on and output comment-code matching pairs. If comments cannot be matched with a code entity during comment matching, they will be regarded as not relevant to implementation details (such as author information), and they will not be used in any subsequent processes of SmartComment.

For the example in Fig. 4, the comment matching process will first match the comment to the code entity they explicitly comments on, including the line 2-3 comment to the event *ApprovalForAll* and the line 15-16 comment to the declaration of the function *setApprovalForAll* in the interface

Table 1. Code entities extracted by SmartComment for comment propagation of the target function

Related Code Entity	Description
`Implement(d:Fd,i:It)`	The target function **implements** declaration `d` in interface `i`
`Inherit(f:Fn,c:Ct)`	The target function **inherits** function `f` in contract `c`
`Override(f:Fn,c:Ct)`	The target function **overrides** function `f` in contract `c`
`Overload(f:Fn,c:Ct)`	The target function **overloads** function `f` in contract `c`
`Call(f:Fn,c:Ct)`	The target function **calls** function `f` in contract `c`
`Use(v:Var)`	The target function **reads** or **writes** variable `v`
`Emit(ev:Ev)`	The target function **emits** event `ev`
`Raise(er:Er)`	The target function **raises** error `er`

`Ct`: {contract, library} in contracts, `It`: {interface, abstract contract} in contracts
`Fn`: {function, modifier} in contracts, `Fd`: {function declaration} in contracts
`Var`: {variable}, `Ev`: {event}, `Er`: {error} in smart contracts

IERC721A. However, no comment can be matched to the *setApprovalForAll* function in the contract *ERC721*, since its code block does not contain any explicit comments.

3.2.3 Comment Propagation. After establishing the matching relations between comments and the specific code entities they comment for, SmartComment executes a comment propagation process to enhance the comment context used in inconsistency detection for each target function by propagating comments from related code entities. Given a target function, SmartComment first leverages static analysis to identify code entities related to the target function, and then utilizes LLMs to analyze comments of these related code entities and derive refined comments for the target function. The static analysis process analyzes the control and data flow dependencies of the target function, identifying code entities that are related to the target function. Specifically, as shown in Table 1, there are eight types of code entities extracted by SmartComment for comment propagation of the target function, along with their descriptions. They cover code entities related to the implementation and execution of the target function, including the interfaces it implements, functions it inherits, overrides, or overloads, as well as the variables, events, and errors used by it, which could contain important comment information related to the target contract. For each identified code entity, its code, comment, and its identified relation with the target function are provided to the subsequent LLM-based comment propagation process. For example, comments on the interfaces can include basic properties of the target function, such as input/output specification, thus can be important for detecting inconsistencies in the target function.

The comments for these statically identified code entities could contain important user-specified information for the target function. However, not all comments from these code entities should be directly propagated to the target function. For example, if a target function overrides a function in its parent contract, it may be intended to implement functionality that is not exactly the same as that of the overridden function. Therefore, some comments for the overridden function could relate to such different functionalities. Directly propagating such comments could result in incorrect comments in the target function and lead to inaccurate inconsistency detection results.

To determine whether a comment should be propagated from a related code entity, SmartComment prompts the LLM to propagate comments from these entities and refine them to the target function. SmartComment first extracts eight types of related code entities for propagation, as shown in Table 1. For each identified code entity, its code, associated comments, and its relationship with the target function are provided to the LLM. As shown in Fig. 3b, the prompt is carefully

designed to guide the LLM in correctly propagating comments while avoiding the propagation of misleading information. For example, to handle overriding functions, the prompt explains the concept of "override" relationships in smart contracts and demonstrates that when a function overrides a parent function, some (but not necessarily all) comments from the parent can be propagated. Instead of directly propagating all comments from the parent function, the prompt provides the original code and comments of the target function and the comments and code for related code entities to the LLM, requiring it to determine whether the comments from the related entities contain information applicable to the target function, i.e., whether they should be propagated. If so, the LLMs are tasked with refining the comment to make it directly applicable to the target function, such as replacing parameter names from the overridden function with the corresponding parameter names in the target function.

SmartComment performs comment propagation from each identified related code entity. For example, for overridden cases where multiple overridden functions are available, SmartComment does not select a single overridden function to use. Instead, it attempts to propagate comments from each identified overridden function. When multiple comments with the same semantics are propagated from different code entities, the LLM deduplicates the propagated comments. SmartComment currently takes all internal information contained in the source code as input. In particular, comments in third-party libraries (e.g., OpenZeppelin [30]) are propagated in the same way as other comments in SmartComment, since they are also important for detecting code-comment inconsistencies. Furthermore, since SmartComment focuses on internal comment propagation, it currently does not provide any external information related to the comments. Such external information—white papers, documentation, and issue reports—might be integrated into SmartComment in the future to provide richer contextual information.

For the example in Fig. 4, the comment propagation process will identify two code entities related to the *setApprovalForAll* function in the contract *ERC721*, *i.e.*, the *ApprovalForAll* event and the function *setApprovalForAll* in the interface *IERC721A*. Then, it will propagate comments from these two entities and generate two new comments for the target function, *i.e.*, *"the operator cannot be the caller"* and *"ApprovalForAll(owner, operator, approved) should be emitted when the owner enables or disables the operator to manage all of its tokens"*. These propagated comments will be analyzed in the inconsistency detection process, enabling SmartComment to detect two inconsistencies in the *setApprovalForAll* function, even though it does not contain any explicit comments.

By transforming implicit code-comment relations from various code locations into explicit and enhanced comments for the target function, SmartComment provides more comprehensive comment context for the subsequent inconsistency detection process. Even functions lacking explicit comments or those that are under-commented can be checked by SmartComment by cross-checking their code with propagated comments.

3.3 Inconsistency Detection

After comment matching and propagation, SmartComment employs an LLM-based approach to detect inconsistencies between code and comments. Fig. 5 illustrates SmartComment's inconsistency detection process, which includes *code context extraction*, *inconsistency inference*, and *inconsistency justification*. The detailed steps of each phase are described as follows.

3.3.1 Code Context Extraction. The code-comment inconsistency detection task requires cross-checking two types of context: (1) the code context related to each function and (2) the comment context that specify the expected functionality of the function. The comment context has already been established through the comment matching and propagation process in Section 3.2. To generate the code-related context, SmartComment employs a code context extraction process

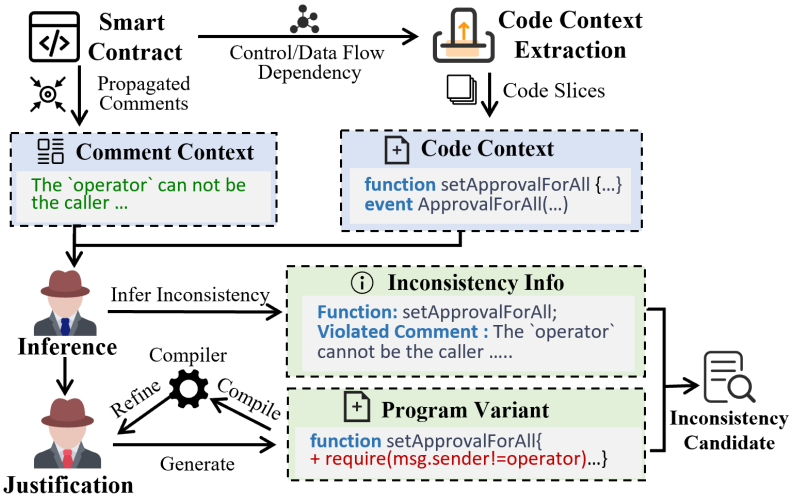


Fig. 5. SmartComment's inconsistency detection process

based on code slicing to extract code implementations related to each target function. Specifically, to generate the code context for the target function, SmartComment first transforms the source code into Abstract Syntax Trees (ASTs) and constructs a function call graph of the contract by performing static analysis on call operations within the AST. Starting with the target function, SmartComment initially includes functions that are directly invoked by the target function into the code context. It then recursively adds functions called by these newly included functions until all reachable functions are incorporated. For each included function, SmartComment utilizes the source mapping information [7] from the AST to retrieve the corresponding code locations within the smart contract and collect their source code. Additionally, SmartComment includes the declarations of contract variables, events, and errors accessed by the function or any of its callees as part of the code context. To provide complete implementation of the target function, SmartComment's code context extraction process does not introduce chunking mechanisms. This is motivated by the fact that the code context for the target function in smart contracts is typically of manageable size. Furthermore, chunking contexts could cause incomplete implementations of the target function, reducing effectiveness for subsequent inconsistency detection (Section 3.3.2) and program variant generation (Section 3.3.3). For example, if the actual code with inconsistency is chunked, SmartComment would fail to generate a comment-compatible program variant, thus introducing false negatives. In our experiments (Section 4), we calculated the token usage of the extracted code context and found no instances exceeding the Deepseek-R1's 128k-token limit (even the largest contract with 6290 lines required only 59k tokens), which validates our design.

3.3.2 Inconsistency Inference. As shown in Fig. 3c, SmartComment prompts the LLM to simulate real-world code reviewers for code-comment inconsistency detection. It provides descriptions of code-comment inconsistencies along with few-shot examples to facilitate the LLM's understanding of the task. It then provides the code and comment context as input, requiring the LLM to evaluate each comment and determine whether the current implementation correctly enforces the function requirements specified in the comments. The LLM is required to generate a yes-or-no output, indicating whether the target function contains a code-comment inconsistency. If the answer is yes, it is also required to output the specific comment that the code violated.

In line with existing inconsistency detection tools [13, 52], SmartComment aims to identify whether comments are *different* from the code (a decidable problem), rather than whether comments

are *correct* (an undecidable problem). The detected code-comment inconsistencies could include three cases: (1) correct comment, incorrect code; (2) incorrect comment, correct code; (3) incorrect comment, incorrect code, and the comment and code are inconsistent. All (1)-(3) cases should be detected and fixed: cases (1) and (3) introduce incorrect and potentially insecure code implementations, while (2) introduces incorrect comments that reduce code maintainability and understandability. To detect all three cases, SmartComment treats comments as reference specifications and faithfully reveals all functional differences in the code relative to these references. Whenever any such difference is detected, SmartComment reports it for further investigation. Furthermore, while inconsistency detection tools like SmartComment can determine that comments are *different* from the code, determining whether a comment is "correct" (and thus classifying the inconsistencies into cases (1)–(3)) is generally an undecidable task, as this depends on developers' intent, which automated tools cannot access. For example, in Fig. 1, the comment specifies that *_fee* should be 25, while the code assigns 15. There is no way to decide whether the comment is *correct* without knowing the developer's intended *_fee*, but this is still an inconsistency that should be detected.

3.3.3 Inconsistency Justification. After an inconsistency is identified, SmartComment conducts an inconsistency justification process, requesting the LLM to generate a program variant of the original function that introduces functional changes to resolve the inconsistency. The rationale behind this process is that code-comment inconsistencies essentially represent functional differences between the current code and those specified by the comments. Therefore, the program variant that the LLM proposes to fix the inconsistency can provide important justification for the inconsistency candidate it detects, which will be utilized in Sections 3.4 for inconsistency validation. As shown in Fig. 3d, SmartComment instructs the LLM to simulate the code reviewer and propose a program variant that fixes the identified inconsistency. It follows an original-program-guided variant generation process, which provides the LLM with not only the comments related to the target function but also the original implementation of the target function and the inconsistency information as input to generate comment-compatible program variants. Note that even when inconsistencies exist, the original implementation may still contain elements that align with the comments. Therefore, using the original implementation as guidance for program variant generation can improve the correctness of the generated program variants compared to generating them based solely on comments [21, 23]. Furthermore, SmartComment employs the *compile-and-refine* process [23, 49] to enhance the correctness of program variants: if a generated variant fails to compile, SmartComment provides the compilation errors to the LLM, prompting it to iteratively refine the variants. If the generated variant still fails the syntax check after the compile-and-refine process, SmartComment will not report the inconsistency candidate, because there is no justification to demonstrate the inconsistency candidate.

Fig. 5 demonstrates how SmartComment handles the example in Fig. 4. SmartComment inputs the propagated comments from Section 3.2 and the extracted code context from Section 3.3.1 for the *setApprovalForAll* function into the LLM-based inconsistency detection process. This enables the LLM to detect an inconsistency candidate, indicating that the function violates the comment *"The operator cannot be the caller"*. Then, the inconsistency justification process generates a program variant that the LLM expects to resolve the inconsistency. The variant introduces an additional check, *require(msg.sender != operator)*, to the original implementation, indicating that the absence of this check leads the LLM to report an inconsistency. The inconsistency detection and justification information will be further analyzed in Section 3.4 for inconsistency confirmation.

```
1 /* Requirements: - 'from' cannot be the zero address  ... */
2 function transferFrom(address from, address to, uint256 tokenId) private {
3     Ownership memory owner = ownershipOf(tokenId);
4     require(owner.addr == from, "ERC721A: transfer from incorrect owner");
5     // ... other checks and transfer operations
6 }
7 function ownershipOf(uint256 tokenId) internal view returns (Ownership memory){
8     for (uint256 curr = tokenId; curr >= 0; curr--){
9         Ownership memory owner = _ownerships[curr];
10        if(owner.addr != address(0)){ return owner;}
11    }
12    revert("ERC721A:unable to determine the owner");
13 }
```

Fig. 6. An example with no code-comment inconsistency.

3.4 Inconsistency Validation

3.4.1 Motivation and Design Choice. Real-world comments may not clearly specify all functionality details, leading to potential ambiguity from the code reviewer's perspective. Such incompleteness and ambiguity can cause the LLM-based inconsistency detection to provide inaccurate results, necessitating further confirmation.

For example, Fig. 6 shows a *transferFrom(from,to,tokenId)* function from an ERC721 smart contract. The comment specifies that the code has checked that *from* address cannot be the zero address, but it does not clearly specify how these checks are actually implemented, such as whether they are explicitly written in the current function or implicitly handled by subsequent functions it calls. Since there are no explicit check operations like *require(from != address(0))* in the *transferFrom* function, the LLM-based inconsistency detection flags this as an inconsistency. However, this is actually a false positive. Instead of explicitly checking *from* against *address(0)*, *transferFrom* checks that the *from* address matches *ownershipOf(tokenId)*. The *ownershipOf* function ensures it will either return a non-zero address (line 11) or revert (on line 13). Thus, a zero-value *from* address would always be reverted, making the current code already align with the comment.

To address this, SmartComment introduces an inconsistency validation process. Specifically, given an inconsistency candidate, the inconsistency justification process (Section 3.3.3) generates a program variant that the LLM expects to resolve the inconsistency. The nuanced functional changes introduced by the program variant to the original implementation provide important justification for the inconsistency candidates. Therefore, SmartComment leverages the LLM and differential analysis to evaluate whether these LLM-introduced functional changes truly align with the comments, enabling SmartComment to filter out invalid ones caused by the LLM's misunderstandings of the comments or code. Specifically, during inconsistency validation, SmartComment checks whether each candidate (1) aligns with the functionality described by the comments, and (2) exhibits different functionality from the original implementation as expected by the LLM. SmartComment first verifies (1) through LLM-based confirmation. If this check passes, static analysis is then used to identify functional differences based on static rules for (2). If static analysis does not detect differences, dynamic analysis is performed to execute the code and check for any discrepancies.

3.4.2 Inconsistency Confirmation. During the LLM-based inconsistency confirmation, SmartComment leverages the LLM to simulate a real-world developer who receives a code-comment inconsistency reported by a code reviewer and needs to determine whether this is a valid issue. As shown in Fig. 3e, SmartComment provides descriptions of code-comment inconsistency along with few-shot examples to facilitate the LLM's understanding of the inconsistency confirmation

task. Then, SmartComment inputs the inconsistency information, the original function implementation, and the LLM-generated program variant intended to resolve the inconsistency into the LLM. The LLM is required to analyze the inconsistency information and the LLM-suggested program variant provided by the code reviewer against the current code and comment, evaluate functional differences between the program variant and the current code implementation, and determine whether these differences are explicitly required by the violated comment, or if they actually stem from a misunderstanding by the code reviewer. The LLM is required to generate a yes-or-no output, indicating whether the code-comment inconsistency is valid.

3.4.3 Differential Analysis. In addition to the LLM-based inconsistency confirmation, SmartComment incorporates a differential analysis technique to validate inconsistencies. The rationale is that due to misunderstandings of code details or Solidity features, LLM may fail to recognize that the code already satisfies comment-specified requirements, leading to false positives. For example, consider the *sub(a,b)* function that performs subtraction on two *uint256* values with a comment *"Subtraction cannot overflow"*. Starting from version 0.8, the Solidity compiler [9] includes internal checks for integer overflow, so the code no longer needs to explicitly check for overflow. However, without such knowledge, the LLM might incorrectly assume that the code lacks the checks required by the comment, reporting a false positive.

To filter out false positives caused by the LLM's failure to recognize that the implementation already satisfies the comment-specified requirements, SmartComment conducts differential analysis to check whether the program variants for the inconsistency candidate indeed have different functionalities from the original implementation. It first performs a lightweight static analysis to identify functional differences that can be confirmed by static rules. The static rules check whether the variant (1) changes function interfaces, including the function name and the number and types of input and output parameters, (2) emits different events, reads or writes different state variables, uses different condition operators or operands, and (3) calls different functions/modifiers. If the static analysis cannot identify differences, SmartComment conducts a dynamic differential testing process. It deploys the original implementation and the program variant, instruments diff-checking assertions in them, and interactively sends test transactions to detect functional differences between the two versions. SmartComment instruments two types of diff-checking assertions for each test transaction: the transaction execution status (revert or not) and the return value. These assertions check the execution results of test transactions calling each public interface of the contract. Then, SmartComment employs Echidna [5], a widely used automated test generation tools, to generate test cases for triggering diff-checking assertions. If the target function is covered by the testing but no diff-checking assertions are triggered, SmartComment identifies the candidate as invalid. After inconsistency validation, SmartComment finally confirms the inconsistencies and reports them.

4 Evaluation

The goal of the evaluation is two-fold. First, we collected a dataset of real-world smart contracts to evaluate the effectiveness of SmartComment in detecting code-comment inconsistencies and compared it with existing techniques. Second, the evaluation of SmartComment on real-world smart contracts enables us to demystify code-comment inconsistencies in the wild and gain insights into their prevalence, categorization, and distribution.

4.1 Evaluation Setup

4.1.1 Research Questions. Specifically, we focus on the following three research questions.

- **RQ1.** How effective is SmartComment in detecting code-comment inconsistencies in real-world smart contracts?

- **RQ2.** What are the categories and distribution of code-comment inconsistencies in real-world smart contracts?
- **RQ3.** How effective is SmartComment compared to previous techniques for detecting code-comment inconsistencies in smart contracts?

4.1.2 Dataset. To answer these research questions, we collected two datasets comprising real-world smart contracts. We first collected all on-chain smart contracts with available verified source codes from Etherscan [10], resulting in a total of 498,969 smart contracts. From this pool, we randomly sampled 1,000 real-world smart contracts to form the first dataset, which is used in RQ1 and RQ2 to evaluate the effectiveness of SmartComment on real-world smart contracts and to investigate code-comment inconsistencies in the wild. The second dataset consists of 200 manually annotated smart contracts, which is a random subset of dataset1, originally sampled from 498,969 contracts. This dataset includes manually annotated labels indicating true/false positives and negatives of code-comment inconsistencies, which are used in RQ3 to enable evaluation metrics such as recall and F1-score for a comprehensive comparison of SmartComment with baselines. The size of both datasets is substantially larger compared to previous LLM-based studies [23, 46, 52], where dataset sizes are inherently constrained by the financial costs of LLM APIs. These two datasets provide a representation of the full set of 498,969 contracts, with a confidence level of 95% and a margin of error of 5%. In line with previous studies [26, 51], we required the smart contracts in the datasets to have at least 10 transactions to filter out potential toy contracts and ensure real-world relevance. Since SmartComment focuses on code-comment inconsistencies, we require that smart contracts contain at least one comment block, excluding those without comments. Notably, to mitigate potential subjective bias, we rely solely on objective criteria (*i.e.*, the presence of comment blocks) and avoid any manual analysis or subjective criteria for dataset inclusion or exclusion. As a result, there are smart contracts in the dataset that have comment blocks, but the comment blocks do not contain implementation-related information (*e.g.*, author-information-only comments). These smart contracts are not manually excluded, in order to faithfully represent the prevalence and distribution of code-comment inconsistencies in real-world smart contracts. Smart contracts containing only author-information comments are labeled as negative cases in dataset2, as they do not include implementation-related comments and, therefore, do not have inconsistencies.

4.1.3 Experimental Setup. We implemented SmartComment using Python 3.13. We employed deepseek-r1 [12], one of the state-of-the-art open-source models, as the base LLM in SmartComment. We chose deepseek-r1 because it is one of the most cost-efficient models while achieving comparable reasoning performance [12] to other models, such as openai-o1 [16]. The static analysis process is built upon Slither [18], and the differential testing process is built upon Echidna [5]. Given the prevalence of Slither and Echidna, they have become fairly standard base engines in recent studies [37, 50]. Their unsoundness may introduce external limitations, *e.g.*, Echidna may not achieve full coverage during differential analysis and fail to filter out false positives. However, these engines mainly play an incremental role in SmartComment. For example, Echidna is one of three inconsistency validation methods, limiting the impact of single-tool limitations. Additionally, SmartComment 's design allows easy replacement of base engines when new tools emerge.

SmartComment implements the majority voting strategy used in previous studies [21, 28] to reduce randomness in LLM outputs, running LLM-based inconsistency inference and confirmation five times and using the majority as the final result. SmartComment also sets the LLM's temperature parameter to 0 to encourage more deterministic responses. We conducted the experiments on a machine with Intel Core i9 CPU (3.0GHz), 64 GB RAM, and running Windows WSL. The source code of SmartComment, our datasets, and experiment results are all available in the online artifacts [1].

```
modifier onlyTeamOrOwner() { require(owner() == _msgSender()||inTeam(_msgSender
    ()), "Caller is not the owner or in Team."); _; }
/* This is owner only and allows a fee-free drop */
function mintToAdminV2(address _to, uint256 _qty) public onlyTeamOrOwner{
    require(_qty > 0, "Must mint at least 1 token");
    _safeMint(_to, _qty, true);}
```

(a) Access Control

```
/* @dev Internal function that burns
    token of a given account ... */
function burn(address _account,
    uint256 _amount) public {
    totalSupply = totalSupply.sub(
    _amount);
    balances[_account] = balances[
    _account].sub(_amount);
    emit Transfer(_account, address
    (0), _amount);}
```

(b) Function Interface

```
/* Restricts staking period to be
    between 90 and 365 ... Other
    Requirements ...*/
function stakeETH(uint256
    stakingPeriod) public payable {
    require(stakingPeriod >= 30 &&
    stakingPeriod <= 365,'Period
    outside of allowed range.');
    //... other checks & operations
}
```

(c) Condition Check

```
modifier onlyOwner() {
    require(msg.sender == owner); _;
}
function aabnm(address _user) public
    onlyOwner {
    require(!xMount[_user], "xx");
    xMount[_user] = true;
    // emit events as well
}
```

(d) Operation/Operands

```
/* @dev Returns the number of tokens
     in existence. Burned tokens
    will reduce the count. */
function totalSupply() public view
    override returns (uint256) {
    return _currentIndex -
    _startTokenId();
}
```

(e) Other (Function-Specific)

Fig. 7. Categories of code-comment inconsistencies detected by SmartComment

4.2 RQ1: Evaluating SmartComment's Effectiveness on Real-World Smart Contracts

To evaluate SmartComment's effectiveness in detecting code-comment inconsistencies in real-world smart contracts, we evaluated SmartComment on the first dataset and analyzed the experiment results. In summary, these smart contracts contain a total of 87,390 (mean: 87.4, median: 38.0, min: 1, max: 1,186) comment blocks (including inline and multi-line comments), 298,121 (mean: 298.1, median: 115.0, min: 1, max: 2,678) lines of comments, and 263,111 non-empty comment lines (mean: 263.11, median: 95.5, min: 1, max: 2,422). We measured the time and financial cost of SmartComment on the experiment by recording the execution time of SmartComment and calculating the number of tokens used during interactions with the LLM. SmartComment took a total of 81.7 hours and 523M tokens to analyze these contracts, with an average of 117.6 seconds and 523,433 tokens (≈0.254 USD) per contract. Such cost is comparable to other LLM-based tools [36, 52] for smart contracts, demonstrating SmartComment's efficiency and cost-effectiveness in detecting code-comment inconsistencies.

4.2.1 Evaluating SmartComment on real-world smart contracts. In total, SmartComment reported 254 code-comment inconsistencies from the contracts. To evaluate the precision of SmartComment in detecting code-comment inconsistencies, we manually analyzed the inconsistencies reported by SmartComment. In line with previous studies, two of the authors independently labeled these inconsistency reports as true positives (TPs) or false positives (FPs), with the help of the third author to resolve any possible disagreements. The results demonstrate that 203 out of the 254 reported inconsistencies are true positives and 51 are false positives. The Cohen's Kappa value

Table 2. Statistics of inconsistencies detected by SmartComment

Type of Inconsistency	Number	Inconsistency (%)	Contract (%)
Access Control	17	8.4%	1.3%
Function Interface	25	12.3%	1.6%
Condition Check	64	31.5%	5.2%
Operation/Operand	61	30.0%	3.8%
Other Func-Specific Logic	36	17.7%	3.2%
Total	203	100%	12.7%

between the two authors' results was 0.85, indicating strong agreement. This yields a precision of 79.9%, demonstrating SmartComment's effectiveness in detecting code-comment inconsistencies in real-world smart contracts. Additionally, 127 (12.7%) of the smart contracts contain at least one valid inconsistency, highlighting their prevalence in the wild. Notably, this percentage is based only on the inconsistencies detected and manually verified among SmartComment's reported results on Dataset1, *i.e.*, the true prevalence of real-world code-comment inconsistencies may be higher.

4.2.2 False Positives. After inspecting the false positives reported by SmartComment, we found that they mainly arise from the LLM's failure to understand the intended functionality behind the potentially incomplete and ambiguous comments. For example, we found a contract with a short comment *"//Public mint"* for the *mint* function, but the function actually includes a whitelist check, leading the LLM to report a false positive. However, without explicitly specifying whether "public mint" means open to everyone or only to the whitelist, it is challenging for the LLM to determine if the additional check in the code is an inconsistency or is intended by design. Future studies could leverage additional information, such as contract design documentation or human-in-the-loop feedback, to help LLMs understand the design intentions behind ambiguous comments.

4.3 RQ2: Characterizing Code-Comment Inconsistencies in the Wild

As the first LLM-driven technique for detecting code-comment inconsistencies in smart contracts, SmartComment does not rely on pre-defined patterns or limit the types of inconsistencies it can detect, enabling us to understand the prevalence, classification, and distribution of real-world code-comment inconsistencies. In RQ2, we categorize all the manually validated inconsistencies reported by SmartComment in RQ1 and evaluate the distribution of each category. For each inconsistency, we determined its category based on the type of differences between the code and comment that led to the inconsistency. Table 2 presents the categorization of real-world inconsistencies, with columns 2 to 4 showing the number of inconsistencies of each type, their proportion of all inconsistencies, and the proportion of smart contracts with this type of inconsistency, respectively.

4.3.1 Access Control. We found that 17 (8.4%) of the 203 valid inconsistencies were due to differences between the access control policies in the code and comment. Fig. 7a illustrates an inconsistency detected by SmartComment related to access control. The contract specifies that the *mintToAdminV2* function can only be called by the owner, thus requires the *OnlyOwner* modifier for access control. However, the code mistakenly uses the *onlyTeamOrOwner* modifier for access control, allowing other roles, *i.e.*, team members, to mint tokens to any users.

4.3.2 Function Interface. 25 (12.3%) inconsistencies arise from the code failing to implement the function interface as specified in the comments, including function visibility and the type of input/output parameters. Fig. 7b shows an example where the comment specifies the *burn* function should be internal, but the code declares it as public, allowing external addresses to call the function and burn tokens belonging to other users.

Table 3. Comparison results of SmartComment with SmartCoco on the Annotated dataset

Tool	TP	FP	FN	Precision	Recall	F1-score
SmartComment	**37**	**11**	**6**	**77.1%**	**86.0%**	**81.3%**
SmartComment-w/o-prop	27	14	16	65.9%	62.8%	64.3%
SmartComment-w/o-diff	37	16	6	69.8%	86.0%	77.1%
SmartComment-w/o-prop-diff	27	22	16	55.1%	62.8%	58.7%
SmartCoco [13]	4	16	39	20.0%	9.3%	12.7%

4.3.3 Condition Check. 64 (31.5%) inconsistencies were due to the code not properly checking conditions as specified in the comments, such as conditions against parameters and variables. Fig. 7c shows an example. The *stateETH* function requires that the staking period should be at least 90 days. However, when checking the range of the *stakingPeriod* variable, the code mistakenly checks if it is greater than 30 instead of 90.

4.3.4 Operation/Operand. 61 (30.0%) of the inconsistencies were due to the code not performing the specific operations required by the comments, such as updating a state variable or emitting an event, or performing them with incorrect operands. Fig. 7d shows an example. The comment for the *aabnm* function specifies that the function should emit an event after approving the user, but the function does not emit the event.

4.3.5 Other Function-Specific Logic. We also found 36 (17.7%) inconsistencies related to business logic specific to the function, thus are not covered by the four types mentioned above. For example, Fig. 7e shows a case where the comments specify that the *totalSupply* variable should be calculated by subtracting burned tokens from the total number. However, the implementation does not account for the burned tokens, resulting in a returned balance value that does not match the expected one.

4.4 RQ3: Comparing SmartComment with Baselines

4.4.1 Baselines. RQ3 aims to compare SmartComment with baseline techniques to evaluate its effectiveness and justify its design choices. To the best of our knowledge, SmartCoco [13] is the only publicly available tool for detecting code-comment inconsistencies in smart contracts. Therefore, we choose SmartCoco as the baseline for the evaluation. Liu *et al.* [25] proposes a pre-training technique to detect inconsistencies in smart contracts. However, their approach does not provide publicly available source code and focuses only on permission-related issues, preventing it from being used as a baseline. To further expand the baseline and justify the design choices of SmartComment, we introduce three ablation variants of SmartComment as additional baselines: SmartComment-w/o-prop, which removes the comment propagation process; SmartComment-w/o-diff, which removes the differential analysis-based inconsistency confirmation process; and SmartComment-w/o-prop-diff, which removes both comment propagation and differential analysis. Comparing SmartComment with these ablated versions allows us to evaluate the impact of the comment propagation and differential analysis processes on detection results, thereby justifying the effectiveness of incorporating program analysis techniques into SmartComment.

4.4.2 Labeling. The Dataset1 does not contain any labels and therefore cannot support the comparison between SmartComment and the baselines. Furthermore, the baselines produce high false positives and negatives, making a fair and meaningful comparison by cross-checking these results challenging and error-prone. Therefore, following previous studies [17, 47], we sampled an annotated dataset consisting of 200 randomly sampled real-world smart contracts. To our knowledge, this is the first annotated dataset for code-comment inconsistencies in smart contracts, which

includes annotated labels for true positives, false positives, and false negatives, thereby enabling the comparison of precision, recall, and F1-score between different tools. With the time-consuming labeling process required to identify sparsely distributed inconsistencies, the size of the dataset is considered sufficient by previous studies on smart contracts [13, 51]. To establish the ground truth, we followed the same labeling process described in Section 4.2. Two authors independently labeled the dataset, with any disagreements resolved by a third author. The Cohen's Kappa value between the two authors' results was 0.90, indicating strong agreement in their labeling.

4.4.3 Evaluating SmartComment on the annotated dataset. In total, the manual labeling process identified 43 code-comment inconsistencies in these 200 smart contracts. As shown in Table 3, SmartComment reported a total of 48 inconsistencies, with 37 of them being true positives and 11 being false positives. This demonstrates that SmartComment achieves a precision of 77.1%, a recall of 86.0%, and an F1-score of 81.3% on the manually annotated dataset. The results demonstrate that, compared to SmartCoco, SmartComment achieves improvements in both precision (77.1% *vs.* 20.0%) and recall (86.0% *vs.* 9.3%). Moreover, when compared to its ablated variants, SmartComment improves precision from 55.1% to 77.1% and recall from 62.8% to 86.0%, highlighting the effectiveness of incorporating program analysis techniques.

4.4.4 Comparing SmartComment with SmartCoco. While the results demonstrate SmartComment's effectiveness, we found that SmartCoco only achieves 20.0% precision and 9.3% recall on the dataset, with more than 90% of the inconsistencies identified by SmartComment not being detected by SmartCoco. Since SmartCoco might report repeated inconsistency instances for the same function, we merged and deduplicated these instances for the same target function. Specifically, SmartCoco can only detect four out of 43 positives in the dataset, and all four inconsistencies belonged to a specific category, *i.e.*, the lack of a check for an input parameter. This is because SmartCoco relies on structured comments and pre-defined patterns for inconsistency detection, thus can only cover a small portion of real-world inconsistencies. For example, we found all cases in Fig. 7 cannot be covered by SmartCoco's pre-defined patterns. Furthermore, we found that SmartCoco has a high false positive rate, with 16 out of 20 it reported being false positives. Manual inspection of these false positives revealed that they are caused by the pattern used by SmartCoco failing to cover the diverse real-world implementations. For example, four of the false positives were due to SmartCoco's failure to recognize wrap-up functions for *msg.sender* in real-world smart contracts, such as *_msgSender()*, leading SmartCoco to falsely report that the checks required by the comments for *msg.sender* were not implemented in the code.

4.4.5 Effectiveness of Comment Propagation. Our results demonstrate the effectiveness of SmartComment's comment propagation process. Specifically, compared to SmartComment-w/o-prop, which does not utilize comment propagation, SmartComment achieves improvements in both precision and recall, by 11.2% and 23.2% respectively. Manual analysis of the differences between SmartComment-w/o-prop and SmartComment indicates that comment propagation can help to propagate and aggregate functional requirements related to the target function in different locations, such as those specified in the overridden function's comments, thereby enabling SmartComment to reduce false negatives. Furthermore, by providing more information to code entities used by the target function, it can provide more comprehensive comment context, such as supplementing short comments (like *"see {Interface-xxx}"*) with additional details from related interfaces, thus reducing false positives caused by insufficient code/comment understanding.

4.4.6 Effectiveness of Differential Analysis. Furthermore, our results demonstrate that the inconsistency confirmation process effectively filters out false positives, improving the precision of SmartComment from 69.8% to 77.1% and that of SmartComment-w/o-prop from 55.1% to 65.9%,

```
/** @dev Leaves the contract without owner...*/
function renounceOwnership() public onlyOwner{
    emit OwnershipTransferred(_owner, address(0));
}
/*Mint amount tokens and assigns them to account*/
function _mint(address account, uint256 amount) public {
    require(msg.sender == _owner);
    _totalSupply = _totalSupply.add(amount);
    _balances[_owner]=_balances[_owner].add(amount);
    emit Transfer(address(0), account, amount);
}
```

Fig. 8. Code-comment inconsistencies detected by SmartComment in a phishing contract.

without incurring loss in recall. For example, in cases like Fig. 6, we found that both SmartComment-w/o-diff and SmartCoco fail to recognize that the current implementation already ensures that *"from"* cannot be a zero address, *i.e.*, there is no functional differences between the code and comments. However, the inconsistency confirmation process can help SmartComment to recognize that adding explicit checks such as *require(from != address(0)* does not introduce any functional differences, allowing it to filter out such false positives. Furthermore, we observed that a common root cause of FPs is the LLM fails to understand Solidity-specific syntax features, such as default return values and inherent integer overflow/underflow checking. Differential analysis can filter out such FPs by demonstrating that the intended functionalities are already achieved through these Solidity-specific syntax features.

During the evaluation of SmartComment on Dataset2, we observed cases where the LLM believed there were inconsistency candidates, but the justification process failed to generate compilable fix variants, resulting in failures in the differential analysis process. Specifically, this occurred for six inconsistency candidates (8.8% of the 68 total candidates) across five smart contracts (2.5%, 5 out of 200 contracts). We analyzed the reasons for these failures and found that they were primarily due to the LLM's limited ability to understand and generate complex contract code, including interfaces and templates (2 cases), contract-specific logic (1 case), constructors (1 case), byte operations (1 case), and opcode-level inline assembly operations (1 case). Future work could focus on enhancing the LLM's capability to better understand and generate smart contract code, thereby reducing such cases. In particular, we did not observe failures caused by the propagation of potentially irrelevant or incorrect comments. This can be attributed to the design of the original-program-guided variant generation process and the compile-and-refine procedure, which prevent the LLM from generating syntactically invalid variants to satisfy these incorrect comments.

5 Discussion

5.1 Case Study

Our results demonstrate that SmartComment can effectively detect diverse code-comment inconsistencies in smart contracts and assist developers in identifying implementation issues, such as access control and condition checks in Fig. 7. Beyond aiding developers, we found that SmartComment also has practical implications for end users. Since end users may rely solely on the comments to understand the functionality of the contract they are interacting with, SmartComment identified several malicious contracts that leverage code-comment inconsistencies to mislead users and conduct phishing attacks [3]. Fig. 8 shows a real-world contract (0x35485a8182cd2ea9b7007c35d79f5b17d3699a58) detected by SmartComment. Upon manual investigation, we found that Etherscan has labeled the contract as phishing contracts. Specifically, the comment for the *renounceOwnership* function

claims it will leave the contract without an owner, but the code merely emits an event to mislead users into believing an ownership transfer has occurred. Similarly, the *_mint* function claims to create tokens for users, but it assigns tokens to the contract owner and emits an event to falsely suggest that users' accounts have received tokens. By using SmartComment, users can avoid interacting with such malicious contracts.

5.2 Threats to Validity

Despite SmartComment's effectiveness in detecting code comment inconsistencies in real-world smart contracts, it may face certain threats to validity. The main internal threat is potential labeling errors in the manual analysis of evaluation results. To mitigate this, we implemented a double-checking process for dataset labeling, with two experts in smart contract analysis/testing carefully reviewing the manual labels. The Cohen's Kappa values for labeling processes are 0.85 and 0.90 respectively, indicating strong agreement between raters. We also made these results publicly available for external inspection [1]. Furthermore, the random sampling process of the datasets might introduce sampling bias, potentially affecting the representation of all real-world smart contracts. To address this, we selected a dataset size as large as feasible within the constraints of manual and financial costs, which is also adequate compared to previous LLM-based studies [23, 52]. Both datasets can statistically represent the full population with a 95% confidence level and a 5% margin of error. Manually increasing the dataset size, such as by sampling more contracts into dataset2, could fail to produce a statistically significant improvement at the current confidence level and interval. For example, the margin-of-error for dataset2 in Section 4.4 is 4.16%. Since RQ3 compares SmartComment with baselines, the target proportion we aim to evaluate is the proportion of smart contracts on which SmartComment produces more accurate or equivalent results compared to baselines, which is observed to be > 90% in RQ3. Accordingly, we estimated a margin-of-error of 4.16% (sample size = 200, confidence level = 95%, population size = 498,969, population proportion = 90%). The main external threat to validity lies in the randomness of the output of LLM. To mitigate this threat, we employ several countermeasures, such as following official suggestions [29] for designing prompts, using majority-voting strategies, and setting the temperature to 0, which are also commonly used in previous LLM-based studies [36, 52].

6 Related Work

6.1 Smart Contract Analysis

A substantial body of research has focused on developing static and dynamic analysis tools for smart contracts [4, 34, 45]. For example, Luu *et al.* [27] introduced Oyente, one of the first analysis tools for smart contracts, which can detect four types of issues, such as re-entrancy [22]. Shou *et al.* [34] proposed ItyFuzz, a tool that employs snapshot-based fuzzing and dataflow analysis for security analysis. Additionally, the industry has developed several analysis tools, such as the Slither [18] Echidna [5], which have been extensively utilized in practice [2].

However, these existing techniques primarily focus on analyzing code implementation [19], with little focus on the inconsistencies between code and comments. SmartCoco [13] could be the most relevant tool for detecting code-comment inconsistencies in smart contracts. However, it relies on pre-defined patterns to extract semantics from structured comments and can only detect three fixed types of inconsistencies, such as failing to emit an event required by the comments. As the first technique to combine LLMs with program analysis for detecting code-comment inconsistencies for smart contracts, SmartComment overcomes the limitations of pattern-based solutions, enabling the detection of inconsistencies between diverse real-world codes and casually written comments.

6.2 Code-Comment Inconsistency Detection

The mismatch between code implementations and developers' intentions has become a common cause of software bugs [38, 43]. Several previous studies have investigated code-comment inconsistencies in other well-studied software, such as C/C++ [38], Java [32, 39] and Rust [52], and proposed detection techniques for them. For example, Tan *et al.* [38] proposed iComment, which uses machine learning techniques to extract constraints from comments and automatically detects inconsistencies in C/C++ source codes. Their subsequent work, @tcomment [39], introduces an approach for detecting inconsistencies in the program logic related to null values. Oztas *et al.* [31] investigate a taxonomy that identifies 11 types of inline code comment smells in Python and Java projects. They propose a technique that leverages LLMs and machine learning techniques to detect each type of inline code comment smell. Yang *et al.* [48] focus on code-comment synchronization, which aims to automatically synchronize comments with code changes. They introduce a new method called *Classifying-Before-Synchronizing* for code-comment synchronization by using deep learning with help from identified categories of inconsistent code-comment samples.

Our work differs from these existing studies from two aspects. First, we focus on the new context of smart contracts, which inherently introduce different language features and coding/commenting practices [20, 41]. Therefore, SmartComment incorporates a set of new program analysis techniques tailored for smart contracts, such as comment propagation and differential analysis. Secondly, existing studies often focus on fixed types of inconsistencies [38, 39]. SmartComment achieves inconsistency detection via a new workflow, which combines LLM with program analysis techniques, allowing it to detect inconsistencies without being limited to specific types of program constraints. It enables the detection of diverse inconsistencies that may be difficult to be covered by fixed patterns, which could also provide implications for detecting inconsistencies in other software.

7 Conclusion

In this paper, we propose SmartComment, the first technique that combines LLMs with program analysis techniques for detecting code-comment inconsistencies in smart contracts. It introduces an LLM-driven workflow for inconsistency detection, which integrates multiple program analysis techniques such as comment propagation, code context extraction, and differential analysis at different steps of the workflow. Our evaluation results demonstrate that SmartComment can successfully detect 203 valid inconsistencies in 1,000 real-world smart contracts with a precision of 79.9%, revealing the prevalence and diversity of real-world code-comment inconsistencies. The comparison between SmartComment and prior work shows that SmartComment can achieve higher recall and precision and identify inconsistencies that prior work cannot detect, demonstrating its effectiveness in detecting code-comment inconsistencies.

Data Availability

The source code, dataset, and experiment results used in this paper are publicly available at https://github.com/SmartComment-Tool/Artifact .

Acknowledgement

This work was partially supported by the National Key Research and Development Program of China (2023YFB2703901), the Open Research Fund of The State Key Laboratory of Blockchain and Data Security, Zhejiang University, and the Open Research Project of the State Key Laboratory of Industrial Control Technology, China (Grant No. ICT2026B32). Daoyuan Wu's role in this research was partially supported by Stellar Academic Research Award Grant (ref. no.: 631463), Lingnan University's Direct Grant (DR26F6) and Faculty Research Grant (SDS25A13).

References

[1] Anonymous. 2025. Online Supplement Material. https://github.com/SmartComment-Tool/Artifact

[2] Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Benjamin Livshits. 2024. Smart Contract and DeFi Security Tools: Do They Meet the Needs of Practitioners?. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 60, 13 pages. doi:10.1145/3597503.3623302

[3] Liang Chen, Jiaying Peng, Yang Liu, Jintang Li, Fenfang Xie, and Zibin Zheng. 2020. Phishing scams detection in ethereum transaction network. *ACM Transactions on Internet Technology (TOIT)* 21, 1 (2020), 1–16. doi:10.1145/3398071

[4] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 227–239. doi:10.1109/ase51524.2021.9678888

[5] Crytic. 2025. Echidna: A Fast Smart Contract Fuzzer. Retrieved May 17, 2025 from https://github.com/crytic/echidna

[6] Isaac David, Liyi Zhou, Kaihua Qin, Dawn Song, Lorenzo Cavallaro, and Arthur Gervais. 2023. Do you still need a manual smart contract audit? *arXiv preprint arXiv:2306.12338* (2023).

[7] Solidity Documentation. 2024. Source Mappings. Retrieved May 17, 2025 from https://docs.soliditylang.org/en/latest/internals/source_mappings.html

[8] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*. 530–541. doi:10.1145/3377811.3380364

[9] Ethereum. 2025. The Solidity Contract-Oriented Programming Language. Retrieved May 17, 2025 from https://github.com/ethereum/solidity

[10] Etherscan. 2023. The Ethereum Blockchain Explorer. Retrieved May 17, 2025 from https://etherscan.io/

[11] Cuiyun Gao, Xing Hu, Shan Gao, Xin Xia, and Zhi Jin. 2024. The Current Challenges of Software Engineering in the Era of Large Language Models. *ACM Transactions on Software Engineering and Methodology* (2024). doi:10.1145/3712005

[12] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[13] Sicheng Hao, Yuhong Nan, Zibin Zheng, and Xiaohui Liu. 2023. SmartCoCo: Checking Comment-Code Inconsistency in Smart Contracts via Constraint Propagation and Binding. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 294–306. doi:10.1109/ase56229.2023.00142

[14] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79. doi:10.1145/3695988

[15] Nikolay Ivanov, Chenning Li, Qiben Yan, Zhiyuan Sun, Zhichao Cao, and Xiapu Luo. 2023. Security Threat Mitigation For Smart Contracts: A Comprehensive Survey. *Comput. Surveys* (2023). doi:10.1145/3593293

[16] Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720* (2024).

[17] Zhang Jiashuo, Shen Yiming, Chen Jiachi, Su Jianzhong, Gao Jianbo, Wang Yanlin, Chen Ting, Gao Jianbo, and Chen Zhong. 2025. Demystifying and Detecting Cryptographic Defects in Ethereum Smart Contracts. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*. doi:10.1109/icse55347.2025.00010

[18] Feist Josselin. 2023. Detector Documentation of Slither. Retrieved May 17, 2025 from https://github.com/crytic/slither/wiki/Detector-Documentation

[19] Kaixuan Li, Yue Xue, Sen Chen, Han Liu, Kairan Sun, Ming Hu, Haijun Wang, Yang Liu, and Yixiang Chen. 2024. Static Application Security Testing (SAST) Tools for Smart Contracts: How Far Are We? *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1447–1470. doi:10.1145/3660772

[20] Lantian Li, Yejian Liang, Zhihao Liu, and Zhongxing Yu. 2023. Understanding solidity event logging practices in the wild. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 300–312. doi:10.1145/3611643.3616342

[21] Tsz-On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. 2023. Nuances are the key: Unlocking chatgpt to find failure-inducing tests with differential prompting. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 14–26. doi:10.1109/ase56229.2023.00089

[22] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. Reguard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 65–68.

[23] Kaibo Liu, Yiyang Liu, Zhenpeng Chen, Jie M Zhang, Yudong Han, Yun Ma, Ge Li, and Gang Huang. 2024. Llm-powered test case generation for detecting tricky bugs. *arXiv preprint arXiv:2404.10304* (2024).

[24] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172* (2023). doi:10.1162/tacl_a_00638

[25] Xiaohui Liu, Sicheng Hao, Lei Yang, and Xiaohong Shi. 2024. Towards Detecting Comment-Code Inconsistency in Smart Contract via Pre-Training Techniques. In *2024 IEEE International Conference on Security, Privacy, Anonymity in Computation and Communication and Storage (SpaCCS)*. 42–49. doi:10.1109/SpaCCS63173.2024.00013

[26] Ye Liu, Yi Li, Shang-Wei Lin, and Cyrille Artho. 2022. Finding permission bugs in smart contracts with role mining. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 716–727. doi:10.1145/3533767.3534372

[27] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 254–269. doi:10.1145/2976749.2978309

[28] Wei Ma, Daoyuan Wu, Yuqiang Sun, Tianwen Wang, Shangqing Liu, Jian Zhang, Yue Xue, and Yang Liu. 2024. Combining fine-tuning and llm-based agents for intuitive smart contract auditing with justifications. *arXiv preprint arXiv:2403.16073* (2024). doi:10.1109/icse55347.2025.00027

[29] OpenAI. 2025. Best practices for prompt engineering with the OpenAI API. Retrieved May 17, 2025 from https://en.bitcoin.it/wiki/Secp256k1

[30] Openzepplin. 2023. The standard for secure blockchain applications. Retrieved May 17, 2025 from https://www.openzeppelin.com/

[31] Ipek Oztas, U. Boran Torun, and Eray Tüzün. 2025. Towards Automated Detection of Inline Code Comment Smells. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering (EASE '25)*. Association for Computing Machinery, New York, NY, USA, 1127–1137. doi:10.1145/3756681.3756988

[32] Fazle Rabbi and Md Saeed Siddik. 2020. Detecting code comment inconsistency using siamese recurrent network. In *Proceedings of the 28th international conference on program comprehension*. 371–375. doi:10.1145/3387904.3389286

[33] Pooja Rani, Arianna Blasi, Nataliia Stulova, Sebastiano Panichella, Alessandra Gorla, and Oscar Nierstrasz. 2023. A decade of code comment quality assessment: A systematic literature review. *Journal of Systems and Software* 195 (2023), 111515. doi:10.1016/j.jss.2022.111515

[34] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. Ityfuzz: Snapshot-based fuzzer for smart contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 322–333. doi:10.1145/3597926.3598059

[35] Solidity. 2025. NatSpec Format. Retrieved May 17, 2025 from https://docs.soliditylang.org/en/latest/natspec-format.html

[36] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639117

[37] Yi Sun, Zhuo Zhang, and Xiangyu Zhang. 2025. FairChecker: Detecting Fund-Stealing Bugs in DeFi Protocols via Fairness Validation. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 671–671. doi:10.1109/icse55347.2025.00134

[38] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /*icomment: bugs or bad comments?*/. *SIGOPS Oper. Syst. Rev.* 41, 6 (oct 2007), 145–158. doi:10.1145/1323293.1294276

[39] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. 2012. @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 260–269. doi:10.1109/icst.2012.106

[40] Zhiyuan Wan, Xin Xia, David Lo, Jiachi Chen, Xiapu Luo, and Xiaohu Yang. 2021. Smart contract security: a practitioners' perspective. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1410–1422. doi:10.1109/icse43902.2021.00127

[41] Ziyan Wang, Xiangping Chen, Xiaocong Zhou, Yuan Huang, Zibin Zheng, and Jiajing Wu. 2021. An Empirical Study of Solidity Language Features. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 698–707. doi:10.1109/QRS-C55045.2021.00105

[42] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 53–64. doi:10.1109/icpc.2019.00019

[43] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A Large-Scale Empirical Study on Code-Comment Inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 53–64. doi:10.1109/ICPC.2019.00019

[44] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[45] Shuohan Wu, Zihao Li, Luyi Yan, Weimin Chen, Muhui Jiang, Chenxu Wang, Xiapu Luo, and Hao Zhou. 2024. Are we there yet? unraveling the state-of-the-art smart contract fuzzers. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639152

[46] Shihao Xia, Shuai Shao, Mengting He, Tingting Yu, Linhai Song, and Yiying Zhang. 2024. Auditgpt: Auditing smart contracts with chatgpt. *arXiv preprint arXiv:2404.04306* (2024).

[47] Shuo Yang, Jiachi Chen, and Zibin Zheng. 2023. Definition and detection of defects in nft smart contracts. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 373–384. doi:10.1145/3597926.3598063

[48] Zhen Yang, Jacky Wai Keung, Xiao Yu, Yan Xiao, Zhi Jin, and Jingyu Zhang. 2023. On the Significance of Category Prediction for Code-Comment Synchronization. *ACM Trans. Softw. Eng. Methodol.* 32, 2, Article 30 (mar 2023), 41 pages. doi:10.1145/3534117

[49] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1703–1726. doi:10.1145/3660783

[50] Jiashuo Zhang, Jiachi Chen, John Grundy, Jianbo Gao, Yanlin Wang, Ting Chen, Zhi Guan, and Zhong Chen. 2025. Automated Test Generation For Smart Contracts via On-Chain Test Case Augmentation and Migration. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 633–633. doi:10.1109/icse55347.2025.00096

[51] Jiashuo Zhang, Jiachi Chen, Yiming Shen, Tao Zhang, Yanlin Wang, Ting Chen, Jianbo Gao, and Zhong Chen. 2025. When Crypto Fails: Demystifying Cryptographic Defects in Ethereum Smart Contracts. *IEEE Transactions on Software Engineering* (2025). doi:10.1109/tse.2025.3551776

[52] Yichi Zhang, Zixi Liu, Yang Feng, and Baowen Xu. 2024. Leveraging Large Language Model to Assist Detecting Rust Code Comment Inconsistency. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 356–366. doi:10.1145/3691620.3695010

[53] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. 2023. Demystifying Exploitable Bugs in Smart Contracts. ICSE. doi:10.1109/icse48619.2023.00061

[54] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. 2023. Sok: Decentralized finance (defi) attacks. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2444–2461. doi:10.1109/sp46215.2023.10179435

Received 2025-09-11; accepted 2026-03-24