SmartCoCo: Checking Comment-code Inconsistency in Smart Contracts via Constraint Propagation and Binding

Sicheng Hao
Sun Yat-sen University
Guangzhou, China
haosch@mail2.sysu.edu.cn

Yuhong Nan
Sun Yat-sen University
Guangzhou, China
nanyh@mail.sysu.edu.cn

Zibin Zheng*
Sun Yat-sen University
Guangzhou, China
zhzibin@mail.sysu.edu.cn

Xiaohui Liu
Sun Yat-sen University
Guangzhou, China
liuxh65@mail2.sysu.edu.cn

***Abstract*—Smart contracts are programs running on the blockchain. Comments in source code provide meaningful information for developers to facilitate code writing and understanding. Given various kinds of token standards in smart contracts (e.g., ERC-20, ERC-721), developers often copy&paste code from other projects as templates, and then implement their own logic as add-ons to such templates. In many cases, the consistency between code and comment is not well-aligned, leading to comment-code inconsistencies (as we call CCIs). Such inconsistencies can mislead developers and users, and even introduce vulnerabilities to the contracts. In this paper, we present SmartCoCo, a novel framework to detect comment-code inconsistencies in smart contracts. In particular, our research focuses on comments related to roles, parameters, and events that may lead to security implications. To achieve this, SmartCoCo takes the original smart contract source code as input and automatically analyzes the comment and code to find potential inconsistencies. SmartCoCo associates comment constraints and code facts via a set of propagation and binding strategies, allowing it to effectively discover inconsistencies with more contextual information. We evaluated SmartCoCo on 101,780 unique smart contracts on Ethereum. The evaluation result shows that SmartCoCo achieves good effectiveness and efficiency. In particular, SmartCoCo reports 4,732 inconsistencies from 1,745 smart contracts, with a precision of over 79% on 439 manual-labeled comment-code inconsistencies. Meanwhile, it only takes 2.64 seconds to check a smart contract on average.**

***Index Terms*—Smart contract, program comprehension, text analytics, comment-code inconsistency**

I. Introduction

Smart contracts are computer programs that are automatically executed on the blockchain. Smart contracts are widely used to build decentralized applications (DApps) such as cryptocurrencies, decentralized finance and game platforms. With the popularity of different categories of DApps, smart contract development activity has grown at an impressive rate. On Ethereum [1], one of the most popular smart contract platforms, there are more than 7.75 million smart contracts deployed in 2022 [2].

Comments are widely used in programming and software development. Previous research has shown that comments are important for code comprehension, development, and maintenance [3]. However, in many cases, due to the complexity of modern software development process (e.g., code integration and update), program code may not be perfectly aligned with comments. In this paper, we call such cases as comment-code inconsistencies (CCIs) [4]. CCIs are highly indicative of errors [5] in either the comments or code. Such inconsistencies may bring confusion to app developers or end-users, such as outdated or incorrect comments which do not affect the correctness [6]. Even worse, CCIs could be indicators of security risks and vulnerabilities, such as incorrect implementations and missing critical security checks [7].

Comment-code inconsistencies in smart contracts. CCIs are prevalent in smart contracts, as code clone, update and reuse are frequently adopted in writing smart contracts [8]. More specifically, many smart contracts are written based on pre-defined code templates, particularly for tokens with the same ERC Specifications (e.g., ERC-20 [9], ERC-721 [10]). A recent report [11] shows that, as of March 2022, there are 508,074 ERC-20-compatible tokens on Ethereum. In these smart contracts, developers make custom changes to functions of cloned contracts, but omit updating the corresponding comments, leaving a number of CCIs.

Compared to other traditional programming languages such as C and Java, CCIs in smart contracts are more likely to cause severe security implications (e.g., financial losses) [12], as smart contracts often hold valuable digital assets. For example, comments in the OpenZeppelin ERC-20 contract [13] explicitly require that *the address of token transfer cannot be zero*. If a smart contract is implemented without such a non-zero check, users may permanently freeze their tokens by mistake. Furthermore, since smart contracts can be invoked by any account on the blockchain by default, developers need to set access control strategies for different functions for security consideration [14]. Part of the comments often provide suggestions/guidance for such access control policies. In such cases, a CCI may lead to a role permission bug, allowing an adversary to easily steal the money in the contract due to the lack of appropriate access control [15]. Therefore, an effective way to automatically identify and report the potential CCIs in smart contracts is in urgent need.

Checking comment-code inconsistencies. While there are a number of previous researches [5], [6], [16]–[21] related to comment-code inconsistency detection, due to the nature of language-wise differences, these approaches can hardly be adopted to analyze CCIs in smart contracts. Particularly,

* Corresponding author

most prior works are designed for analyzing comments related to a specific type of programming language (e.g., C [16] and Java [18]). As examples, prior research focuses on specific types of CCIs such as handling interruptions [16], null values [5] and exceptions [19]. Due to the differences in running environment and language-specific features, they are either not existed, or not to be a valid concern in smart contracts.

In addition to code comments, there are a number of studies focusing on API documents [22]–[25]. Following this line of research, the most related work is DOCCON [26], which aims to find API documentation errors in Solidity smart contracts. However, DOCCON is limited to analyzing specific smart contract libraries and their API documents, rather than the general code comments in smart contract source code. Compared to API documents which are well-structured with clear textual semantics, code comments are much more diverse and semantics-vague. Even for the same constraint in different smart contracts, the comments could be written in different formats. Moreover, comments and code may not perfectly match based on locations. In other words, a potential constraint could be implemented by functions in other locations of smart contracts, requiring non-trivial analysis to establish the desired connections between comments and code.

Our work. In this paper, we present SmartCoCo, a novel Comment-Code Inconsistency (CCI) detection framework for smart contracts. SmartCoCo takes contract files as input and automatically checks the comment-code pairs for all implemented functions in the smart contract. Particularly, our research focuses on three types of comments which are security-critical (i.e., *role permission*, *parameter scope*, and *event emission*). To the best of our knowledge, SmartCoCo is the first of its kind to detect comment-code inconsistencies for smart contracts.

SmartCoCo overcomes the following two challenges in detecting CCIs for smart contracts: (1) how to effectively identify potential constraints from various code comments; (2) how to establish meaningful connections between comment constraints and contract code, and further report potential inconsistencies.

In detail, SmartCoCo first utilizes part-of-speech (POS) tagging to identify and extract comment constraints (Section III-B). In the meantime, SmartCoCo performs a light-weight program analysis to collect a set of code facts (code constraints) based on the abstract syntax tree (AST) of the contract code (Section III-C). Later, to build connections between comment constraints and contract code at the function level, SmartCoCo constructs a fact-powered call graph (as FCG), and then generates a set of comment-code pairs by propagating and binding constraints to various related functions over the FCG (Section III-D). Finally, SmartCoCo checks the comment-code pairs by inspecting whether the constraint and the corresponding entities (e.g., parameters) can be effectively mapped to contract code (Section III-E). Through the processes above, SmartCoCo reports all identified CCIs in the given smart contract.

We evaluated SmartCoCo on 101,780 real-world Ethereum smart contracts. Overall, SmartCoCo identified 4,732 CCIs from 1,745 smart contracts in our dataset. In terms of precision, we manually checked 439 unique inconsistencies and confirmed that SmartCoCo achieves a precision of 79.3%. In terms of efficiency, SmartCoCo takes only 2.64 seconds to analyze a smart contract on average. In addition, we also performed a study to show that modern Large Language Models (LLMs), such as GPT, can not accomplish this task due to the fundamental challenges in extracting and mapping critical information between comments and contract code.

In summary, this paper makes the following contributions:

- We propose SmartCoCo, a new framework to detect comment-code inconsistency for smart contracts. We applied SmartCoCo to three typical comment types in smart contracts which may lead to security implications.
- We propose a set of constraint propagation and binding mechanisms for inconsistency detection, which enable us to accurately establish connections between comments and corresponding functions.
- We thoroughly evaluated SmartCoCo on 101,780 unique smart contracts to show its effectiveness and efficiency.
- We release the prototype of SmartCoCo, as well as the dataset to benefit future research in the community [27].

The rest of the paper is organized as follows. Section II introduces the background knowledge, motivating example, and problem statements. Then in Section III, we present the design of SmartCoCo in detail. Section IV reports the comprehensive evaluation results of SmartCoCo. In Section V, we discuss the threats to validity and generality of our work. After that, Section VI discusses the related work, and Section VII concludes our research.

II. BACKGROUND AND MOTIVATION

In this section, we briefly introduce the background of smart contracts and code comments. Then, we introduce a motivating example and present the problem statement.

A. Smart Contract and its Code Comment

Smart contracts are autonomous programs running on blockchain systems. A smart contract consists of a set of contracts under the same address. Once a smart contract is deployed, all public and external functions can be invoked by other accounts/contracts on the blockchain [1]. To minimize potential attack surfaces which could be exploited by adversaries, contract developers usually design and implement specific constraints in contract code. For example, in Solidity, such constraints are implemented with statements such as `require`, or user-defined *modifiers* like `onlyOwner`. Lastly, smart contracts use `emit events` to indicate what is happening on the blockchain. Since the running environment of the whole blockchain is isolated from the outside (i.e., the off-chain environment), external off-chain programs can track the on-chain transactions by listening to the emitted events.

Comments are widely used in smart contracts for code comprehension. Comments can effectively improve code readability and facilitate contract writing. In addition to describing the functionality of code, comments in smart contracts often provide detailed information about requirements on ERC standards and security guidance. For example, comments in token contract templates like ERC-20 often explicitly describe the token standards and security suggestions. Besides, since smart contracts can be invoked by any account, there are often guidance and suggestions on access control in the corresponding comments.

Comment-code inconsistencies (CCIs) indicate that there are errors in either comments or code. These inconsistencies not only influence the code comprehension, but also introduce vulnerabilities to the code. CCIs reduce the security and reliability of smart contracts. More seriously, there have been such inconsistencies that even caused significant losses to the contract owner and users.

B. Motivating Example

We use a real-world example to illustrate an instance of comment-code inconsistency (CCI), as well as its security implication. On February 2022, the decentralized finance platform RigoBlock was hacked, and all tokens except ETH and USDT were at risk due to a protocol vulnerability [15]. The hacked tokens were valued at more than 160 ethers ($432,000). However, the cause was a small mistake made by the developer when writing the `Drago` contract. Figure 1 shows part of the code for the `Drago` contract. The content with green background color shows a consistent example, while the red background color shows an inconsistency. Specifically, the comment at line 9 requires that the following function `setMulAllowances` only allows the owner to invoke, while the external function has no access control (line 12) such as the `onlyOwner` modifier for the contract owner. As a result, adversaries can exploit this function to steal tokens. Even though the `Drago` contract has good comments that can be used to facilitate development, developers still missed the implementation.

```
1  contract Drago is Owned, SafeMath, ReentrancyGuard{
2    ///@dev Allows owner to set an allowance...
3    function setAllowance(address _token, ...)
4      external onlyOwner
5      whenApprovedProxy(_tokenTransferProxy){
6        require(setAllowancesInternal(...);
7    } }
8
9    /** @dev Allows owner to set allowances to
10   multiple approved tokens with one call. */
11   function setMulAllowances(address _token, ...)
12     external {
13       for (uint256 i = 0; i < _tokens.length; i++){
14         if (!setAllowancesInternal(...)
15           continue;
16   } }
17
18   /// @dev Allows owner to set an ...
19   function setAllowancesInternal(...)
20     internal returns (bool){
21       require(Token(_token).approve(...));
22       return true;
23   }
24   ...
25 }
```

Fig. 1: An example of comment-code inconsistency in RigoBlock. Function `setMulAllowances()` missed an access control check on the contract owner, which is not aligned with its comment (line 9-10).

C. Problem Statement

The goal of our work is to automatically detect potential CCIs in smart contracts. If comments are written, they should be consistent with the code. More specifically, our research aims to find two types of security implications related to CCIs, namely, missing implementation and confused implementation. The missing implementation refers that developers forget to implement constraints in the code, such as the motivating example. The confused implementation indicates that there are similar but incorrect implementations for comment constraints. Both situations are likely to pose security risks to smart contracts.

Our research focuses on smart contracts written in Solidity [28], as Solidity is one of the most popular programming languages designed for smart contracts. It is widely used in Ethereum and many other EVM-compatible blockchain platforms, such as TRON [29] and BNB Chain [30]. However, the design of SmartCoCo is generic to check CCIs in other smart contracts with minimal engineering efforts. More specifically, to support other smart contract languages such as Vyper [31], developers only need to extract corresponding code facts, which is independent from other modules of SmartCoCo (see Section III-C for more details).

III. Design of SmartCoCo

A. Overview

Figure 2 shows the overview of SmartCoCo. It takes smart contract source code with comments as input, and outputs the inconsistency report. The comment extractor first obtains function comments from smart contracts and generates constraints. The code fact extractor transforms source code into a set of code facts by analyzing the abstract syntax tree. Here, the code fact refers to whether the statement has, or satisfies a specific property. Then, SmartCoCo constructs a fact-powered call graph (FCG) to associate (propagate) code facts to all related functions in the call graph of the smart contract. Through the FCG, SmartCoCo generates comment-code pairs to be analyzed for related functions. Finally, SmartCoCo checks these pairs based on their semantic similarity and reports the identified inconsistencies.

SmartCoCo is designed to check CCIs in smart contracts at the function level. Particularly, our research covers three types of comments, namely, *role permission*, *parameter scope*, and *event emission*. We choose these types of comments because their corresponding implementations are prevalent in

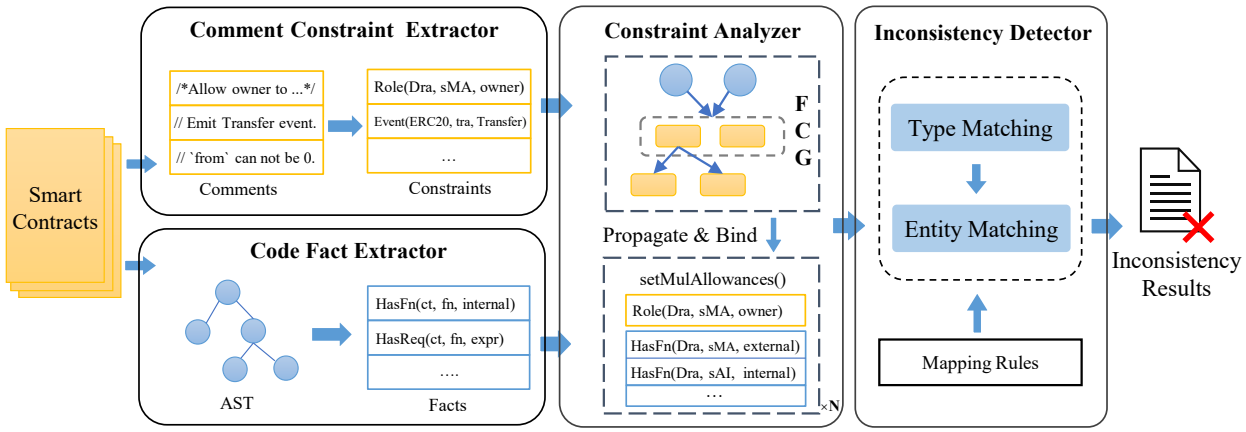

Fig. 2: Overview of SmartCoCo.

TABLE I: The three types of security-critical comments covered in our research.

Type	Example
Role Permission	**Only** available to the current CEO. **Allows** owner to set allowances to multiple tokens.
Parameter Scope	`from` and `to` **cannot be** the address(0). Threshold must be **greater than** the hardcoded min.
Event Emission	**Emits** a {Transfer} event. **Might emit** an {Approval} event.

TABLE II: Comment constraints of the three security-critical types as well as comment inheritance.

Comment Constraint	Description
Role(c:Ct, f:Fn, role:Str) Param(c:Ct, f:Fn, e:Exp) Event(c:Ct, f:Fn, e:Str, m:Bool)	Only `role` can invoke `c.f`. Function `c.f` has a parameter scope with `e`. Function `c.f` (may) emits an `e` event.
Inherit(sc:Ct, sf:Fn, ic:Ct, if:Fn)	Comments of `sc.sf` inherits from `ic.if`.
Ct: Contracts in a smart contract. Fn: Functions in a smart contract. Exp: Expressions in comments, including arithmetic and logical expressions.	

smart contract development [32], [33]. More importantly, these comments are security-critical, as they intuitively describe the pre/post conditions for smart contract function invocation. Table I shows the types and examples of the corresponding comment sentences. In addition to the three types of comments covered by our research, the design of SmartCoCo can support other types by adding new constraint specifications (see Section III-B & Section III-E) for more details.

B. Comment Constraint Extraction

SmartCoCo extracts constraints from natural language comments in the smart contracts. As mentioned earlier, due to the complexity of natural language, this task is by no means trivial, particularly because most comments are short sentences, or even terms that are semantic vagueness [34]. The key observation in our research is that comments referring to constraints usually contain certain keywords such as *only*, *allow*, and *emit*. Therefore, a list of seeding keywords can enable us to sufficiently locate those possible candidates related to constraints. Later, we employ part-of-speech to more accurately extract the actual constraints from these comments.

Comment preprocessing. SmartCoCo first collects all function comments of the smart contract. Then, it processes the raw comment text to facilitate later steps. Following the typical code-comment style used by developers [35], we treat comments above each function as its function-level comments. To this end, we build a lexical parser that binds each function with its corresponding comments according to their locations. For the collected comments, SmartCoCo first removes irrelevant words and symbols (e.g., partial NatSpec tags, punctuation and separators) at the beginning of the comment texts, this step helps to reduce the noise data which are irrelevant to our task. Then, SmartCoCo merges comments that span across multiple lines. After these steps, SmartCoCo obtains a series of triples with the `<contract, function, content>` format.

Constraint finding. SmartCoCo identifies comment constraints based on keyword matching and POS tagging. More specifically, we first use keyword-based templates to match suspicious comment sentences, and then utilize POS tagging to filter out irrelevant content. The constraint extractor analyzes the text from the previously obtained comment triples, and finally generates corresponding constraints.

Table II describes the formalized comment constraints covered in our research. Note that in addition to the three comment types aforementioned in Section III-A, there is an additional constraint type named `Inherit`, which refers to comment inheritance. The inheritance relation allows us to associate more valid constraints which are defined in other functions. Such inheritance relations can be found via specific comment texts such as `See {Contract-Function}` and `@inherit Contract`. For example, in the OpenZeppelin `ERC20` contract [13], there is a comment text `See {IERC20-transfer}` for the `transfer` function, which indicates that this function has comments inherited from its

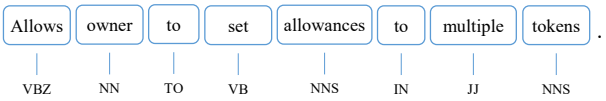

Fig. 3: The POS tags for the motivating example, which can be used to extract the role *owner* as *NN*.

interface.

By observing popular smart contracts and libraries, we find that comments on different constraints are mostly described by a limited number of keywords. For example, in smart contract comments, the keywords `allow`, `role`, `only` are sufficient to cover comments related to role permission. In the meantime, comments describing constraints are often with similar patterns and POS tagging structures, which can be further used to eliminate false positives caused by keyword-based matching. For example, the pattern *Allows? <DT>? <JJ, VBN>? <NN><TO>* can be used to match the `Role` constraint of the motivating example, as shown in Figure 3.

Constraint templates. In our research, we have summarized a total number of 20 constraint templates (patterns) to extract comment constraints, with 6 for *role permission*, 10 for *parameter scope*, 2 for *event emission*, and 2 for *comment inheritance*. These templates can be found at our project repository [27]. To achieve this, we manually reviewed more than 30 top-popular DApps and frequently used library contracts [36]. Prior research [8], [37] has shown that similar to other languages, code clone and reuse are prevalent in the smart contract. Developers tend to reuse code from mainstream DApps and library contracts (as templates) by adding new functionalities. Still, the reused code follows the same code/comment patterns as the original templates. To this end, we reviewed all comments (with around 1800 lines) from these contracts to identify and summarize the templates. Finally, to check whether the summarized patterns are sufficient, we randomly selected another 30 smart contracts from the evaluation dataset and reviewed their comments, the results showed that there were no new patterns or templates need to be added.

In addition, SmartCoCo utilizes domain-specific knowledge to check the POS tagging of specific keywords to eliminate false positives when identifying constraints. For example, *from* and *to* are usually a noun in smart contracts. Besides, some false positives can be directly filtered out as false positives. For example, *everyone* and *user* may not represent a real role. A full list of keywords used by SmartCoCo can also be found in our project repository [36].

By analyzing all the extracted comment triples, SmartCoCo generates comment constraints for each function. For example, the triple *<Drago, setMulAllowances, Allows owner to set allowance ...>* is translated to the constraint `Role(Drago, setMulAllowances, owner)`.

C. Code Fact Extraction

SmartCoCo extracts code constraints in smart contracts as a set of code facts following the specification of Datalog [38]. Code facts [39] are formal representations of whether a program statement has specific properties, or satisfies specific constraints. Code facts are true predicates with the form $P(x_1, ..., x_n)$. P is the name of the predicate, and $x_1, ..., x_n$ are its arguments. For example, the code fact `HasContract(Drago, contract)` describes a `contract` named `Drago` in this smart contract. Fact-based approaches have been widely used in program analysis [40].

TABLE III: Selected code facts for constraint propagation and inconsistency detection.

Code Fact	Description
HasContract(c:Ct, t:Ctype)	Contract `c` is type `t`.
HasInherit(c:Ct, ic:Ct)	Contract `c` inherits from Contract `ic`.
HasFunction(c:Ct, f:Fn, v:Vtype)	Contract `c` has a function named `f` with the visibility `v`.
FIsImplemented(c:Ct, f:Fn)	Function `c.f` has implementation.
FHasParam(c:Ct, f:Fn, p:List)	Function `c.f` has params `p`.
FHasMod(c:Ct, f:Fn, m:Fn)	Function `c.f` has modifiers `m`.
FHasEmit(c:Ct, f:Fn, e:Str)	Function `c.f` emits an event `e`.
FHasReq(c:Ct, f:Fn, e:Exp, m:Str)	Function `c.f` has a require expression `e` with an error message `m`.
FHasCall (sc:Ct, sf:Fn, a:List cc:Ct, cf:Fn, p:List)	Function `sc.sf` has a call with arguments `a` to the function `cc.cf` with parameters `p`.
Ct: Contracts in a smart contract.	Ctype $\in$ {contract, interface, library}
Fn: Functions in a smart contract.	Vtype $\in$ {external, public, internal, private}
List: Lists of parameters and arguments in functions and calls of a smart contract.	
Exp: Expressions in a smart contract, including arithmetic and logical expressions.	

By inspecting the way to enforce constraints in code comments, we find that developers usually explicitly write code corresponding to these constraints within the function. Therefore, it is reasonable to perform a consistency check between the comment constraints and code facts. If the comment constraint can be fully reflected by one or multiple code facts in the function, we consider the constraint is well-satisfied. Otherwise, the smart contract is possible to have a comment-code inconsistency (CCI).

Code facts for CCI inspection. To satisfy the requirements of CCI inspection, we extend the fact schemes based on previous work [26] which is used for checking incorrect implementations in smart contract code. Table III shows the summarized primary facts as well as their corresponding descriptions. We categorize these facts to the contract level and function level. The contract-level facts contain declaration information on contracts and functions, while the function-level facts contain fine-grained behaviors in the function body.

To extract code facts, SmartCoCo uses a light-weight program analysis to collect information from the abstract syntax tree (AST) of the smart contract. More specifically, SmartCoCo first compiles the smart contract to the AST. Then, it traverses the AST and builds corresponding facts according to the type and attributes of each node in the AST. For example, if SmartCoCo visits a `ContractDefinition` node, it will add a `HasContract` fact, with the contract name and type based on its attributes. Note that we treat "if" statements as either `FHasReq` or `FHasRev` facts by checking whether the

keyword `revert` is in the body. By visiting the whole AST nodes, SmartCoCo generates a set of code facts for further CCI inspection.

D. Constraint Propagation and Binding

SmartCoCo generates comment-code pairs for inconsistency detection by propagating and binding constraints in both the comments and code. To detail, it builds a set of rules for propagating constraints across the associated comments. In the meantime, it constructs a fact-powered call graph to propagate and bind corresponding code facts for each implemented function with different visibility.

Comment propagation and binding. SmartCoCo propagates constraints in comments and binds them to multiple functions across the same smart contract. In Table II, the constraint represented by `Inherit` is not a particular type but a generalization of other types. In other words, the comment constraints for a specific function can come from not only the function itself, but also from other functions such as the definition of an interface. To this end, we perform constraint propagation and binding for the comments based on two heuristic rules as shown in Figure 4. Here, $Cmt(ct, fn)$ refers to all comment constraints of the function `ct.fn`, including those constraints propagated from other functions.

$$Cmt(ct, fn) :- Cmt(ict, ifn),\ Inherit(ct, fn, ict, ifn)$$
$$Cmt(ct, fn) :- Cmt(ict, fn),\ HasInherit(ct,\ ict),\ HasContract(ict, \texttt{interface})$$

Fig. 4: Rules for comment constraint propagation.

There are two cases in which a comment constraint should be propagated: explicit propagation and implicit propagation. In explicit propagation, the two functions are associated with the `Inherit` comment constraint. In implicit propagation, the two functions share an implicit dependency relationship as the definition of an interface (`Interface IERC20`) and its implementation (`Contract ERC20`), respectively. Note that we consider propagating comments for interfaces because comments for the interface definition usually describe the same functionalities as its implementations. We do not propagate comments in functions with other relations (e.g. `override`) because such functions may have quite different functionalities.

Code propagation and binding. In addition to code comments, SmartCoCo propagates code facts across the related functions in the same smart contract. This step allows SmartCoCo to more accurately obtain the comment-code pairs for inconsistency checking.

To propagate code facts, we designed a *fact-powered call graph* (FCG) that represents the execution flow of functions in a smart contract with more fine-grained contextual information. More specifically, FCG is different from a traditional call graph in two aspects: (1) the FCG is a subset of the original call graph by eliminating functions without any code facts. (2) each node in the FCG contains additional attribute information, such as code facts related to this function. Given a fact-powered call graph $G = (V, E)$, nodes V and edges E are formally described as follows:

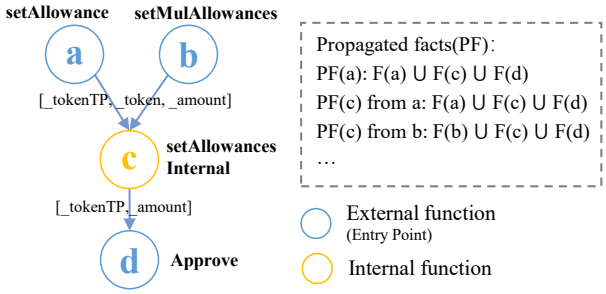

Fig. 5: The FCG and propagated code facts of the contract in our motivating example (Figure 1).

- $V = (ct,\ fn,\ param,\ code,\ cmt)$, where ct refers to contract name, fn refers to function name, $param$ refers to parameters, $code$ refers to all code facts and cmt refers to all comment constraints related to $ct.fn$.
- $V := \{V_e \vee V_i\}$, where V_e and V_i represent external and internal functions. The public and private functions are regarded as external and internal functions, respectively. This information can be obtained from the parameter `v:Vtype` in fact `HasFunction`.
- $E = (v_1,\ v_2,\ arg)$, where E is a function invocation from v_1 to v_2, with arguments arg. In SmartCoCo, each `FHasCall` fact refers to a directed edge from function `sc.sf` to function `cc.cf`.

We use an example shown in Figure 5 to illustrate the process of code fact propagation based on the FCG. In short, code facts in one node can propagate to another if they are within the same call chain (e.g., $a \rightarrow c \rightarrow d$ or $b \rightarrow c \rightarrow d$). However, whether or not a code fact can propagate from one to another is also affected by the visibility of the node (i.e., external function or internal function). To detail, (1) Nodes with external functions are the entry points for code fact propagation. For external functions such as nodes *a, b* and *d*, all code facts in their successors (children) are propagated to the external function, because the successor nodes must be called after the invocation of the external function. (2) For internal functions such as node *c*, the code facts in both its predecessors and successors are propagated to the internal function, as these nodes are within the same call chain. Note that for internal functions, the sets of code facts are context-sensitive and affected by the call chain. For example, as shown in Figure 5, the call chain of $a \rightarrow c$ and $b \rightarrow c$ will generate different code facts for node *c*, due to the contextual difference enforced by different external functions (*a* and *b*).

Besides, SmartCoCo maintains a list of mapping between the name of the parameters and arguments across function invocations. This is because while the textual similarity between parameters and arguments could be significantly different, they

are referring to the same variable (entity) when performing the CCI detection.

Finally, for each node in the FCG, the propagated code facts and comment constraints in the node are treated as comment-code pairs for inconsistency checking.

E. Inconsistency Detection

SmartCoCo utilizes a two-phase heuristic matching method to determine whether the extracted comment constraints have corresponding code facts. Specifically, for a comment constraint, SmartCoCo first identifies the possible code facts by their types, and then checks whether the variables and expressions match the comment constraint with a heuristic matching strategy. If a constraint satisfies both conditions, it is consistent. If not, we tag it as an inconsistency.

Constraint Type matching. SmartCoCo selects corresponding code facts for different comment constraints based on their types. Specific types of comment constraints are implemented by several patterns, which means specific code fact types. For example, to emit an event, the function usually has a `FHasEmit` fact. No related types of code facts indicate that the comment constraint is missing to implement. Meanwhile, SmartCoCo can filter irrelevant code facts to improve the efficiency with type matching.

To design the type-matching rules, we analyze the implementation of the comment types mentioned in this work. These rules are from the popular smart contract libraries and DApps, as well as the domain knowledge. For *role permission*, we select the `FHasReq` and `FHasMod` that are related to the `msg.sender` keyword. Although the modifiers can be regarded as a function call, they are code facts of functions in SmartCoCo. They usually contain the role name, which can be helpful for comprehension, especially when developers use variables with irrelevant names to store the addresses. The *parameter scope* constraint keeps `FHasReq` facts, while the *event emission* keeps `FHasEmit` facts.

Constraint Entity matching. After identifying constraint types, SmartCoCo checks whether certain entities extracted from comments and code are consistent. Here, an entity refers to specific attributes in the comment constraints or code facts. For example, in the `Role(Drago, setMulAllowances, owner)` constraint, the entity is `owner`. In the `FHasMod(onlyOwner)` code fact, it is the `onlyOwner`. Since the comment-code pairs have built real connections for constraints and code, we no longer need to consider the contract and function name. Due to the difference between entities in natural language and code, we need to identify whether they represent the same meaning. Ideally, the entities in the comments are equal to the variable or function names. However, in practice, they are often different.

Fortunately, we find that the description of entities in comments and code are relatively similar, as the proposed constraints are related to implementation details rather than functionality descriptions. Therefore, we adopt a heuristic-based matching scheme. Specifically, entities such as numbers and expressions should be exactly equivalent between comments and code. For string entities, we consider them to satisfy one of the following conditions as being matched:

- *Similar at the character-level*: We use Levenshtein distance [41] to measure the character-level similarity between entities in comments and code. Here, instead of assigning a fixed distance as the threshold, we use similarity ratio to check whether the two entities are sufficiently similar as in prior research [42]. Our empirical analysis showed that a ratio of 0.8 is a good fit for this task.
- *Abbreviations*: The comment constraint and variable names exactly satisfy the prefix or suffix relationship since abbreviations are frequently used.

In addition to the basic matching mechanisms, SmartCoCo integrates several additional domain knowledge for the mapping process. For example, considering that `OnlyOwner` are widely used in role permission of smart contracts, if a contract has only one role and utilizes this way to achieve access control, we regard them equally. Since smart contracts have some domain implementation patterns, such knowledge can be helpful in checking consistent comment-code pairs.

IV. Evaluation

In evaluation, we attempt to answer the following research questions (RQs):

- **RQ1:** What is the prevalence of security-related comment-code inconsistencies in smart contracts?
- **RQ2:** What is the effectiveness of SmartCoCo in detecting comment-code inconsistencies?
- **RQ3:** What is the performance in checking a smart contract with proposed constraints?
- **RQ4:** Can large language models check CCIs identified by SmartCoCo?

Experimental setup. We implemented the prototype of SmartCoCo in Python 3.10. We utilized Slither [43] as the AST parser and Standford CoreNLP [44] as the POS tagger to improve the quality of constraint extraction. The evaluation was conducted on a Ubuntu 22.04 LTS machine with one Xeon(R) Gold 5218R CPU and one RTX 3090 GPU. We ran SmartCoCo with 20 processes. Since the CoreNLP server may respond in different time, we repeated the experiments three times.

Dataset. We built our dataset based on a recent empirical study [45], which contains 139,424 unique smart contracts on the Ethereum mainnet. To the best of our knowledge, it is one of the most comprehensive dataset of smart contracts collected from Etherscan [46]. Note that each of these smart contracts has a unique address, and may contain more than one contract. After excluding contracts with very old versions (less than 0.4.11) and compilation errors, the dataset consists of 101,780 different Solidity smart contracts. We evaluated SmartCoCo on this full dataset [27].

TABLE IV: Distribution of extracted comment constraints from the large-scale dataset.

Type	# Smart Contract	# Comment Constraint
Role Permission	29,963	45,725
Parameter Scope	11,582	144,653
Event Emission	21,462	137,992
Comment Inheritance	10,810	90,746
ALL	39,372	419,116

A. Prevalence of comment-code inconsistencies

We first report the prevalence of comments and constraints on the full dataset, and then report the identified CCIs and their distributions.

Extracted comments and constraints. Code comments are indeed prevalent in smart contracts as well as the proposed comment constraints. In the 101,780 smart contracts, there are 74,926 containing comments, which accounts for nearly 74% of the full dataset. Moreover, these contracts contain a total of 1,818,665 function comment blocks. To illustrate the constraints proposed in this paper, we counted the number of constraints in different types. Table IV reports the summary of the comment constraints and their distributions. The result shows that more than 52% contracts with comments are with at least one of the proposed constraints. For the rest 48% contracts, most of them are simple contracts with comments describing copyrights and functionalities. A few of the smart contracts are with fine-grained, statement-level constraints (e.g., price calculation constraints), which are out of the scope of SmartCoCo. In particular, SmartCoCo extracts 419,116 comment constraints in 39,372 smart contracts.

Identified CCIs and distributions. Table V shows the analysis results of SmartCoCo in the 39,372 smart contracts. It reports the number of comment-code consistencies (CCCs) and CCIs, as well as the number of affected smart contracts (SCs). SmartCoCo detects 4,732 inconsistencies in 1,745 smart contracts as well as 291,143 consistencies in 34,639 smart contracts, respectively. Note that a single smart contract may have multiple CCIs with different types.

The number of smart contracts with CCCs and CCIs (Table V) is smaller than the number of smart contracts with constraints (Table IV) due to several reasons: Firstly, most missed constraints are related to interfaces without actual implementations. Besides, some comments are related to functions as templates, which are not reachable in the smart contract in practice. Lastly, the Comment Inheritance constraint does not directly relate to any CCC or CCI.

TABLE V: Distribution of identified CCCs and CCIs.

Type	#SCwCCC	#CCC	#SCwCCI	#CCI
Role Permission	25,951	39,781	482	697
Parameter Scope	10,940	129,171	296	507
Event Emission	14,981	122,191	995	3,528
ALL	34,639	291,143	1,745	4,732

RQ1: SmartCoCo finds 4,732 inconsistencies from 1,745 smart contracts, which occupies 4.4% of the smart contracts with comment constraints in our dataset.

TABLE VI: Reported precision of SmartCoCo over the manual-labeled CCIs.

Type	# CCI	# TP	# FP	% Precision
Role Permission	194	145	49	74.7%
Parameter Scope	146	116	30	79.5%
Event Emission	99	87	12	87.9%
ALL	439	348	91	79.3%

B. Effectiveness of SmartCoCo

In this section, we report the precision of SmartCoCo on a set of manual-labeled CCIs. Since our goal is to find the inconsistencies, the precision here refers to the number of real inconsistent comment-code pairs among all pairs reported by SmartCoCo.

Evaluation setup. To prepare the ground truth for evaluating the effectiveness of SmartCoCo, we selected 439 unique CCIs for manual inspection. Since code reuse is widespread in smart contracts, many contracts and functions have similar comments and structures. However, they are actually unique on the contract level due to the differences in the whole smart contracts. Therefore, we removed possible duplicates of CCIs based on the contract name, function name, and the number of code facts. This can avoid the bias caused by counting the redundant CCIs of multiple instances. For example, we found that there were more than 1,000 missing implementations on event emission in `decreaseAllowance` and `increaseAllowance` functions which are cloned from ERC-20 templates. Two domain experts working on smart contracts manually inspected the CCIs. If the two experts give different results over the same instance, there will be an additional round of discussion until they reach an agreement.

Results. We evaluated SmartCoCo on the 439 unique CCIs. Table VI reports the precision on different comment types as well as the accumulated results. Overall, SmartCoCo achieves a precision of 79.3%. For *role permission*, *parameter scope*, and *event emission*, the precision is 74.7%, 79.5%, and 87.9%, respectively.

Note that similar to prior analysis on smart contracts [47], [48], we did not evaluate the recall of SmartCoCo due to the lack of ground truth. More specifically, since constraint-related comments are sparsely distributed in code comments, it is very difficult to review all smart contract codes and comments for constructing the ground truth for evaluating the recall. For example, the source contract of Figure 1 contains nearly 1200 LOCs with 200 comments, but the CCI instance only relates to a few lines of code (i.e., line 9-23 in Figure 1).

To illustrate the cause of CCIs, we checked whether an inconsistency is caused by either the missing implementation

```
1  /// @dev only peers can call intendWithdraw
2  function intendWithdraw(...){
3    address receiver = msg.sender;
4    uint rid = c._getPeerId(receiver);
5    ...
6  }
7  function _getPeerId(Channel storage _c, address
       _peer) internal view returns(uint){
8    if (_peer==_c.peerProfiles[0].pAddr)return 0;
9    else if(_peer==_c.peerProfiles[1].pAddr)return 1;
10   else revert("Nonexisted");
11 }
```

Fig. 6: An example of false positive. The entity *_c.peerProfiles* in line 8 and 9 is not identified as *peer* in comments due to their low similarity.

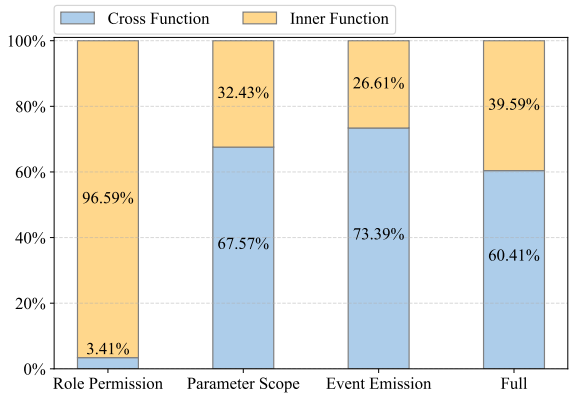

Fig. 7: The distribution of the position between the constraint and its matching code.

(MI) or confused implementation (CI). For *role permission*, there are 45 MIs and 100 CIs. For *parameter scope* and *event emission*, the distribution has some differences. The results are 94 MIs, 22 CIs and 68 MIs, 19 CIs respectively. This is because many comments are from template contracts and developers modified the function body without updating the comments. Such behaviors are disastrous, which may increase difficulty in code comprehension, or introduce security defects to the smart contract. To further pinpoint whether the errors of CCIs are in the code or comments, we tried our best to manually analyze and understand the functionality of these smart contracts. Among such CCIs, 139 of them are caused by errors in comments, while 209 of them are caused by errors in contract code.

False positive analysis. By manually inspecting the reported 439 CCIs in our evaluation dataset, we find there are a total number of 91 false positives. Looking at the distribution of such false positives in different constraint types, the FP rate in *role permission* constraint (i.e., 49/194, 25.3%) is a bit higher than the other two types. This is because comments related to *role permission* are more semantically vague. For example, it is rather difficult to correctly recover the actual roles which are described by words like "token", "participant", etc.

A large portion of false positives (54/91, 59.3%) are caused by ambiguous natural language comments. For example, in sentence *This function allows the owner to withdraw funds*, the "owner" in the sentence actually refers to the sender of the current transaction, not the owner of the contract. Unfortunately, the ambiguous statement misleads SmartCoCo and causes the false positive. A better comment here is to replace the word "owner" with "sender" or someone else instead. In addition, the Constraint Extractor may incorrectly extract some entities that are not related to *role permission*. For example, in sentence *proxy only allows delegate call for actions*, while the term "delegate call" satisfies the text pattern *Allows? <DT>? <JJ, VBN>? <NN>*, it is not a description for role permission. To eliminate such false positives, SmartCoCo can integrate more contextual information from the smart contract code when extracting comment constraints. For example, inferring the actual role by checking the variables related to `msg.sender`.

The other false positives (37/91, 40.7%) are related to the Inconsistency Detector which performs an inaccurate match between comments and code facts. Particularly, in many cases of such false positives, SmartCoCo failed to extract the actual code facts corresponding to the comments. As an example shown in Figure 6, SmartCoCo reports an inconsistency because the similarity between *peer* (in comment) and *_c.peerProfiles* (in code) is lower than our similarity threshold. In addition, some implementations of comment constraints are in smart contracts of another address, which can not be accessed by SmartCoCo. There are also some comment constraints implemented in the assembly code of the smart contract, therefore, these code facts are missed by SmartCoCo as it only analyzed code written in Solidity.

Effectiveness of constraint propagation and binding. The purpose of constraint propagation and binding is to expand code facts for one function and build a complete connection for a comment constraint and its corresponding code. Therefore, without this process, some implemented constraints will be identified as inconsistencies, which leads to an increase in false positive cases. To gain a better understanding of the cross-function implementations, we analyzed the consistent reports to statistic the number of cross-function implementations. To make sure that SmartCoCo preferentially checks code facts that are not from propagation, we move them to the top for inconsistency detection.

Figure 7 reports the distributions of cross-function and inner-function implementations. The result indicates that constraint propagation and binding are necessary. Specifically, there are more than 60% comment constraints implemented in another function. For *role permission*, they are extremely important for access control, so developers usually implement them along with the current function. For other constraint types, developers usually use an auxiliary internal function to implement the constraints. Note that modifier facts have been bound to the corresponding function, so we regard them as part of the function body rather than a cross-function invocation.

TABLE VII: Detection time (in seconds).

	Small 1/3	Medium 1/3	Large 1/3	Average
Code	1.3547	2.2020	4.3698	2.6411
Comment	1.9017	2.4915	3.6671	

In fact, the consistencies with modifiers account for 91.85% of the total consistencies in *role permission*. If we regard them as cross-function invocation, the ratio of *role permission* will almost overturn.

RQ2: SmartCoCo achieves a precision of 79.3% on 439 unique inconsistencies.

C. Performance of SmartCoCo

We evaluated the performance of SmartCoCo on the full dataset, and calculated the average analysis time on smart contracts with comment constraints. All the 101,780 contracts with different versions are successfully analyzed. To better illustrate the performance of SmartCoCo, we further studied the 39,372 contracts which have at least one comment constraint. Specifically, to illustrate performance on contracts with different sizes, we split these smart contracts into three subsets (i.e., small, medium, and large) based on the number of code facts and comment constraints. Table VII reports the average analysis time for each subset along with the full dataset. For these smart contracts, the average analysis time is as low as **2.64** seconds. Such a performance overhead is fully acceptable for performing a CCI inspection during code compilation. The result also shows that the running time does not increase significantly as the number of comments or code facts increases in quantity.

RQ3: SmartCoCo takes 2.64 seconds to check a smart contract with comment constraints on average.

D. Compared with Large Language Models

Since there is a lack of other tools for detecting CCIs in smart contracts, we compared SmartCoCo with large language models (LLMs). We randomly choose 10 different inconsistencies for *role permission* (RP), *parameter scope* (PS) and *event emission* (EE). Then we input them into the gpt-3.5 turbo model [49] and Bing AI [50], which are easy to use. Table VIII shows the results. They are not as good as expected. It is undeniable that the prompts and contract snippets we use may also influence the effectiveness. But an important reason is that LLMs prefer to explain the code rather than finding out inconsistencies by analyzing the answer.

Figure 8 shows the prompt and the obtained responses for checking inconsistencies of our motivating example (Figure 1). As can be seen, the GPT-3.5 omitted the role permission information. The Bing AI extracted the role constraint from the comments. However, it produced inaccurate statements about the effect of `whenApprovedProxy` modifier. Moreover, it omitted the `onlyOwner` in function `setAllowance`. To further explore the capability of GPT, we also provided the name of each constraint type as prompts for the question. Unfortunately, both models return the similar incorrect, or incomplete results as their previous ones.

TABLE VIII: Inconsistency detection results on LLMs (10 CCIs for each type which can be detected by SmartCoCo).

	# RP	# PS	# EE	# Total
Ground Truth	10	10	10	30
GPT-3.5	0	1	1	2
Bing AI	5	1	1	7

RP: *role permission*, **PS**: *parameter scope*, **EE**: *event emission*

Q: You are an experienced Solidity developer, please check whether there are inconsistencies between comments and code in the following smart contract:

```
contract Drago is Owned, SafeMath, ReentrancyGuard{
    /// @dev Allows owner to set an allowance
    function setAllowance(address _token, ...) external ...
        /// omitted contract code
    }
```

GPT-3.5: There does not appear to be any inconsistencies in the `Drago` smart contract.

Bing AI: The comment for the `setAllowance` function says that it allows owner to set an allowance to an approved token transfer proxy, but the code uses the `whenApprovedProxy` modifier, which checks if the proxy is approved by the authority, not the owner.

Fig. 8: A case for checking CCIs in Figure 1 by LLMs. Texts in red are inaccurate statements generated by these models.

RQ4: The current LLMs are less effective than SmartCoCo for checking smart contract comment-code inconsistencies.

V. Discussion

Threats to Validity. One threat to validity may come from the lack of ground truth for evaluating the effectiveness of SmartCoCo. Since SmartCoCo is the first of its kind to detect CCIs for smart contracts, there is a lack of an accurately labeled dataset of CCIs. To ensure the correctness of our manual-labeled dataset from 439 smart contracts (in Section IV-B), each CCI instance is double checked by two experts working on smart contract security. Besides, as mentioned earlier in Section IV-B, we did not evaluate the recall of SmartCoCo as it requires significant manual labor.

Another threat could be the incomplete constraint templates used for extracting comment constraints. Since the 20 proposed templates are summarized from a limited number of contracts, SmartCoCo may miss certain patterns and cause

false negatives. However, given that the Comment Constraint Extractor is fully independent from other modules, it is trivial to add/integrate new templates to improve SmartCoCo.

Admittedly, for some types of constraints such as *role permission*, SmartCoCo incurs a relatively high false positive rate (i.e., 25.3%) when checking comment-code inconsistencies. The false positives are mainly caused by incorrect parsing of the referred roles in comments. As future work, more fine-grained, contextual-sensitive NLP techniques (e.g., with domain-specific models [25], [51]) could be adopted to reduce such false positives.

Generality. Checking the comment-code inconsistency with a generic tool is non-trivial, as comments in natural language could be semantics-vague and contextual-dependent. The design of SmartCoCo assumes and expects that comments and their implementations follow specific development practices with relatively fixed patterns. The current design of SmartCoCo does not cover those less-frequent appeared constraints that do not fit our constraint templates and code facts. Lastly, as mentioned earlier in Section II-C, SmartCoCo can be easily extended to analyzing CCIs for smart contracts written in other languages such as Vyper [31].

VI. Related Work

Smart contract analysis. With the fast development of decentralized applications, many studies have been proposed to analyze and improve the security of smart contracts. In terms of dynamic analysis techniques, ContractFuzzer [52], sFuzz [53], and RLF [54] are based on fuzzing techniques. SODA [55] and TXSPECTOR [56] detect vulnerabilities based on the runtime information collected through a customized EVM. For static analysis techniques, Oyente [47] is the earliest symbolic execution tool for detecting vulnerabilities in Ethereum smart contracts. Securify [57] infers semantic facts of smart contracts and checks for compliance and violation patterns using datalog. SAILFISH [58] uses a graph to detect state-inconsistency vulnerabilities. VeriSmart [59] and SmartPulse [60] adopt formal verification to check the security of specific properties in smart contracts. In recent years, machine learning is utilized to detect smart contract vulnerabilities [61], [62]. Different from previous works that detect specific types of smart contract vulnerabilities, SmartCoCo focuses on detecting comment-code inconsistencies (CCI), which may further bring bugs or vulnerabilities to smart contracts. To the best of our knowledge, SmartCoCo is the first of its kind to automatically detect CCIs for smart contract security enhancement.

Comment-code inconsistency analysis. Comment-code inconsistencies can bring negative impacts on software development, such as lowering code readability and even security issues. Following this line of research, previous work [4], [63] performed empirical studies to investigate the evolution of code and comments in open-source projects, as well as the root cause of CCIs. Besides, several previous research proposed different mechanisms to detect CCIs in other program languages such as C/C++, Java, and Python. For example, Tan et al. [6], [16] inspected interrupt-related comments in C/C++ program. @tComment [5] checked whether comments related to null values are consistent with code written in Java. In recent years, machine learning techniques have also been used to detect CCIs in Java [21], [64] and Python [65] source code. Unfortunately, due to the differences in language-specific features and security assumptions, these approaches can not be trivially adopted to check CCIs for smart contracts. Our proposed framework, SmartCoCo is orthogonal to these frameworks by targeting different program languages and different security implications.

In addition to directly checking CCIs, tools such as CUP [66], HebCUP [67], and CBS [68] automatically generate high-quality comments when updating program code. Toradocu [69], Jdocter [19] and C2S [20] translate Javadoc comments into formal program specifications, which can be further used to check CCIs. Besides, TDCleaner [70] aims at the removal of obsolete `TODO` comments. Zhai et al. [71] and Panthaplackel et al. [72] associate comments with code entities to facilitate various tasks. There are also studies extracting useful information from comments for test case generation, such as temporal constraints [34] and metamorphic relations [73]. These approaches are generically applicable to eliminate code-comment inconsistencies across different program languages. As future work, SmartCoCo can integrate these techniques to improve its effectiveness in checking CCIs. For example, SmartCoCo can inspect more types of CCIs in smart contracts and repair some of them.

VII. Conclusion

This paper presents SmartCoCo, a static analysis framework for detecting three types of comment-code inconsistencies in smart contracts, namely *role permission*, *parameter scope* and *event emission*. SmartCoCo extracts constraints from code and comments, and builds connections between comments and functions via constraint propagation and binding. We evaluated SmartCoCo on 101,780 real-world smart contracts. Overall, SmartCoCo reports 4,732 inconsistencies from 1,745 smart contracts. SmartCoCo achieves a precision of 79% on our manual-labeled dataset with 439 unique inconsistencies. In terms of efficiency, SmartCoCo takes only 2.64 seconds to analyze a contract with constraints on average. Our research sheds light on a new direction to enhance the security of smart contracts.

Acknowledgment

We would like to thank the anonymous reviewers for their detailed and valuable comments. This work is supported in part by the National Natural Science Foundation of China (No.62032025), the Special Projects in Key Fields of Universities in Guangdong Province (No.2022ZDZX1001), the Technology Program of Guangzhou, China (No.202103050004), and the Fundamental Research Funds for the Central Universities, Sun Yat-sen University (No.22lgqb26).

REFERENCES

[1] "Ethereum," https://www.ethereum.org/, [Accessed 1-May-2023].

[2] A. Team, "Web3 development report (q4 2022)," https://www.alchemy.com/blog/web3-developer-report-q4-2022, [Accessed 1-May-2023].

[3] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance." in *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information, SIGDOC 2005, Coventry, UK, September 21-23, 2005*, 2005, pp. 68–75.

[4] F. Wen, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on code-comment inconsistencies," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. Montreal, QC, Canada: IEEE, May 2019, pp. 53–64.

[5] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. Montreal, QC, Canada: IEEE, Apr. 2012, pp. 260–269.

[6] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*icomment: Bugs or bad comments?*/." in *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, 2007, pp. 145–158.

[7] W. M. Ibrahim, N. Bettenburg, B. Adams, and A. E. Hassan, "On the relationship between comment update practices and software bugs." *J. Syst. Softw.*, vol. 85, no. 10, pp. 2293–2304, 2012.

[8] X. Chen, P. Liao, Y. Zhang, Y. Huang, and Z. Zheng, "Understanding code reuse in smart contracts." in *28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021*, 2021, pp. 470–479.

[9] "Erc-20: Token standard," https://eips.ethereum.org/EIPS/eip-20, [Accessed 1-May-2023].

[10] "Erc-721: Non-fungible token standard," https://eips.ethereum.org/EIPS/eip-721, [Accessed 1-May-2023].

[11] "What are erc-20 tokens: A complete overview," https://cleartax.in/s/erc-20-tokens, [Accessed 1-May-2023].

[12] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang, "Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. London United Kingdom: ACM, Nov. 2019, pp. 1503–1520.

[13] "Openzeppelin," https://docs.openzeppelin.com/openzeppelin/, [Accessed 1-May-2023].

[14] Y. Liu, Y. Li, S.-W. Lin, and C. Artho, "Finding permission bugs in smart contracts with role mining." in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, 2022, pp. 716–727.

[15] "CVE-2022-25335," https://nvd.nist.gov/vuln/detail/CVE-2022-25335, [Accessed 1-May-2023].

[16] L. Tan, Y. Zhou, and Y. Padioleau, "acomment: Mining annotations from comments and code to detect interrupt related concurrency bugs," in *Proceedings of the 33rd International Conference on Software Engineering*. Waikiki, Honolulu HI USA: ACM, May 2011, pp. 11–20.

[17] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic generation of oracles for exceptional behaviors." in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, 2016, pp. 213–224.

[18] I. K. Ratol and M. P. Robillard, "Detecting fragile comments." in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, 2017, pp. 112–122.

[19] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos, "Translating code comments to procedure specifications," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Amsterdam Netherlands: ACM, Jul. 2018, pp. 242–253.

[20] J. Zhai, Y. Shi, M. Pan, G. Zhou, Y. Liu, C. Fang, S. Ma, L. Tan, and X. Zhang, "C2s: Translating natural language comments to formal program specifications," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Virtual Event USA: ACM, Nov. 2020, pp. 25–37.

[21] S. Panthaplackel, J. J. Li, M. Gligoric, and R. J. Mooney, "Deep just-in-time inconsistency detection between comments and source code," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 1, pp. 427–435, May 2021.

[22] M. Motwani and Y. Brun, "Automatically generating precise oracles from structured natural language specifications," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, May 2019, pp. 188–199.

[23] Y. Zhou, C. Wang, X. Yan, T. Chen, S. Panichella, and H. C. Gall, "Automatic detection and repair recommendation of directive defects in java api documentation." *IEEE Trans. Software Eng.*, vol. 46, no. 9, pp. 1004–1023, 2020.

[24] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. W. Godfrey, "Docter: Documentation-guided fuzzing for testing deep learning api functions," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual South Korea: ACM, Jul. 2022, pp. 176–188.

[25] P. Hu, R. Liang, Y. Cao, K. Chen, and R. Zhang, "Aurc: Detecting errors in program code and documentation," *32nd USENIX Security Symposium, USENIX Security 2023, August 9-11, 2023*, 2023.

[26] C. Zhu, Y. Liu, X. Wu, and Y. Li, "Identifying solidity smart contract api documentation errors." in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, 2022, pp. 56:1–56:13.

[27] "SmartCoCo," https://github.com/SCCoCo/SmartCoCo, [Accessed 31-July-2023].

[28] "Solidity," http://solidity.readthedocs.io/, [Accessed 1-May-2023].

[29] "TRON," https://tron.network/, [Accessed 1-May-2023].

[30] "BNB Chain," https://www.bnbchain.org/, [Accessed 1-May-2023].

[31] "Vyper," http://vyper.readthedocs.io/en/latest/index.html, [Accessed 1-May-2023].

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

[33] L. Liu, L. Wei, W. Zhang, M. Wen, Y. Liu, and S.-C. Cheung, "Characterizing transaction-reverting statements in ethereum smart contracts," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Melbourne, Australia: IEEE, Nov. 2021, pp. 630–641.

[34] A. Blasi, A. Gorla, M. D. Ernst, and M. Pezzè, "Call me maybe: Using nlp to automatically generate unit test cases respecting temporal constraints," in *37th IEEE/ACM International Conference on Automated Software Engineering*. Rochester MI USA: ACM, Oct. 2022, pp. 1–11.

[35] S. Majumdar, A. Bansal, P. P. Das, P. D. Clough, K. Datta, and S. K. Ghosh, "Automated evaluation of comments to aid software maintenance." *J. Softw. Evol. Process.*, vol. 34, no. 7, 2022.

[36] "Summarized patterns of SmartCoCo," https://github.com/SCCoCo/SmartCoCo/tree/main/Pattern, [Accessed 31-July-2023].

[37] 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*, vol. 49, no. 4, pp. 2006–2019, Apr. 2023.

[38] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about datalog (and never dared to ask)." *IEEE Trans. Knowl. Data Eng.*, vol. 1, no. 1, pp. 146–166, 1989.

[39] M. Naik, C.-S. Park, K. Sen, and D. Gay, "Effective static deadlock detection." in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, 2009, pp. 386–396.

[40] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: Thorough, declarative decompilation of smart contracts," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, May 2019, pp. 1176–1186.

[41] G. Navarro, "A guided tour to approximate string matching." *ACM Comput. Surv.*, vol. 33, no. 1, pp. 31–88, 2001.

[42] Y. Zhao, L. Li, X. Sun, P. Liu, and J. Grundy, "Code implementation recommendation for android gui components." in *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*, 2022, pp. 31–35.

[43] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts." in *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019.*, 2019, pp. 8–15.

[44] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *Association for Computational Linguistics (ACL) System Demonstrations*, 2014, pp. 55–60. [Online]. Available: http://www.aclweb.org/anthology/P/P14/P14-5010

[45] Z. Zheng, N. Zhang, J. Su, Z. Zhong, M. Ye, and J. Chen, "Turn the rudder: A beacon of reentrancy detection for smart contracts on ethereum." in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, 2023, pp. 295–306.

[46] "Etherscan," https://etherscan.io/, [Accessed 1-May-2023].

[47] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Vienna Austria: ACM, Oct. 2016, pp. 254–269.

[48] Z. Liao, S. Hao, Y. Nan, and Z. Zheng, "Smartstate: Detecting state-reverting vulnerabilities in smart contracts via fine-grained state-dependency analysis." in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, 2023, pp. 980–991.

[49] "Gpt-3.5," https://platform.openai.com/docs/models/gpt-3-5, [Accessed 1-May-2023].

[50] "Bing," https://www.bing.com/new, [Accessed 1-May-2023].

[51] R. Tufano, L. Pascarella, and G. Bavota, "Automating code-related tasks through transformers: The impact of pre-training." in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, 2023, pp. 2425–2437.

[52] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection." in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, 2018, pp. 259–269.

[53] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Seoul South Korea: ACM, Jun. 2020, pp. 778–788.

[54] J. Su, H.-N. Dai, L. Zhao, Z. Zheng, and X. Luo, "Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing," in *37th IEEE/ACM International Conference on Automated Software Engineering*. Rochester MI USA: ACM, Oct. 2022, pp. 1–12.

[55] T. Chen, R. Cao, T. Li, X. Luo, G. Gu, Y. Zhang, Z. Liao, H. Zhu, G. Chen, Z. He, Y. Tang, X. Lin, and X. Zhang, "Soda: A generic online detection framework for smart contracts," in *Proceedings 2020 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2020.

[56] M. Zhang, X. Zhang, Y. Zhang, and Z. Lin, "Txspector: Uncovering attacks in ethereum from transactions." in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, 2020, pp. 2775–2792.

[57] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. T. Vechev, "Securify: Practical security analysis of smart contracts." in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, 2018, pp. 67–82.

[58] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, "Sailfish: Vetting smart contract state-inconsistency bugs in seconds." in *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, 2022, pp. 161–178.

[59] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "Verismart: A highly precise safety verifier for ethereum smart contracts." in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, 2020, pp. 1678–1694.

[60] J. Stephens, K. Ferles, B. Mariano, S. K. Lahiri, and I. Dillig, "Smartpulse: Automated checking of temporal properties in smart contracts." in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, 2021, pp. 555–571.

[61] Z. Liu, P. Qian, X. Wang, L. Zhu, Q. He, and S. Ji, "Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion," in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*. Montreal, Canada: International Joint Conferences on Artificial Intelligence Organization, Aug. 2021, pp. 2751–2759.

[62] Z. Zhang, Y. Lei, M. Yan, Y. Yu, J. Chen, S. Wang, and X. Mao, "Reentrancy vulnerability detection and localization: A deep learning based two-phase approach." in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, 2022, pp. 83:1–83:13.

[63] B. Fluri, M. Würsch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes." in *14th Working Conference on Reverse Engineering (WCRE 2007), 28-31 October 2007, Vancouver, BC, Canada*, 2007, pp. 70–79.

[64] F. Rabbi and M. S. Siddik, "Detecting code comment inconsistency using siamese recurrent network," in *Proceedings of the 28th International Conference on Program Comprehension*. Seoul Republic of Korea: ACM, Jul. 2020, pp. 371–375.

[65] L. Pascarella, A. Ram, A. Nadeem, D. Bisesser, N. Knyazev, and A. Bacchelli, "Investigating type declaration mismatches in python," in *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. Campobasso: IEEE, Mar. 2018, pp. 43–48.

[66] Z. Liu, X. Xia, M. Yan, and S. Li, "Automating just-in-time comment updating." in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, 2020, pp. 585–597.

[67] B. Lin, S. Wang, K. Liu, X. Mao, and T. F. Bissyandé, "Automated comment update: How far are we?" in *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*, 2021, pp. 36–46.

[68] Z. Yang, J. W. Keung, X. Yu, Y. Xiao, Z. Jin, and J. Zhang, "On the significance of category prediction for code-comment synchronization," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–41, Apr. 2023.

[69] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic generation of oracles for exceptional behaviors." in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, 2016, pp. 213–224.

[70] Z. Gao, X. Xia, D. Lo, J. C. Grundy, and T. Zimmermann, "Automating the removal of obsolete todo comments." in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, 2021, pp. 218–229.

[71] J. Zhai, X. Xu, Y. Shi, G. Tao, M. Pan, S. Ma, L. Xu, W. Zhang, L. Tan, and X. Zhang, "Cpc: Automatically classifying and propagating natural language comments via program analysis," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Seoul South Korea: ACM, Jun. 2020, pp. 1359–1371.

[72] S. Panthaplackel, M. Gligoric, R. J. Mooney, and J. J. Li, "Associating natural language comment and source code entities," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 05, pp. 8592–8599, Apr. 2020.

[73] A. Blasi, A. Gorla, M. D. Ernst, M. Pezzè, and A. Carzaniga, "Memo: Automatically identifying metamorphic relations in javadoc comments for test automation," *Journal of Systems and Software*, vol. 181, p. 111041, Nov. 2021.