Identifying Solidity Smart Contract API Documentation Errors

Chenguang Zhu*
cgzhu@utexas.edu
The University of Texas at Austin
USA

Ye Liu
li0003ye@e.ntu.edu.sg
Nanyang Technological University
Singapore

Xiuheng Wu
xiuheng001@e.ntu.edu.sg
Nanyang Technological University
Singapore

Yi Li
yi_li@ntu.edu.sg
Nanyang Technological University
Singapore

ABSTRACT

Smart contracts are gaining popularity as a means to support transparent, traceable, and self-executing decentralized applications, which enable the exchange of value in a trustless environment. Developers of smart contracts rely on various libraries, such as OpenZeppelin for Solidity contracts, to improve application quality and reduce development costs. The API documentations of these libraries are important sources of information for developers who are unfamiliar with the APIs. Yet, maintaining high-quality documentations is non-trivial, and errors in documentations may place barriers for developers to learn the correct usages of APIs. In this paper, we propose a technique, DocCon, to detect inconsistencies between documentations and the corresponding code for Solidity smart contract libraries. Our fact-based approach allows inconsistencies of different severity levels to be queried, from a database containing precomputed facts about the API code and documentations. DocCon successfully detected high-priority API documentation errors in popular smart contract libraries, including mismatching parameters, missing requirements, outdated descriptions, etc. Our experiment result shows that DocCon achieves good precision and is applicable to different libraries: 29 and 22 out of our reported 40 errors have been confirmed and fixed by library developers so far.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; **Documentation**.

KEYWORDS

Smart contract, API documentation, program facts.

*This work was done in part while the first author was a research assistant at Nanyang Technological University.

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556963

ACM Reference Format:
Chenguang Zhu, Ye Liu, Xiuheng Wu, and Yi Li. 2022. Identifying Solidity Smart Contract API Documentation Errors. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3556963

1 INTRODUCTION

Blockchain is a distributed ledger shared between nodes of a peer-to-peer network following a certain consensus protocol, and one of the fastest-growing technologies of the modern time. The blockchain technology owes its initial popularity to Bitcoin [73], which first appeared in 2008. Later, Ethereum [91] started the era of Blockchain 2.0 and expanded the capabilities and applications of blockchain by introducing Turing-complete *smart contracts* [82], which are self-executing computer programs managing large sums of money, carrying out transactions of valuable assets, and governing the transfer of digital rights between multiple parties. Decentralized applications (DApps) allow users to interact with smart contracts, and can be used to govern the automatic executions of transactions on top of blockchain platforms. DApps enable autonomous, transparent, and fully-traceable exchange of value in a trustless environment. Significant progress has been made in applying them to support a wide range of activities. Many traditional industries, such as supply chain, energy, finance, legal and medical services, are expected to be revolutionized by this new technology. As of May 2022, there are nearly 50 million Solidity [80] smart contracts deployed on Ethereum, which is a 1.96x increase from just two years ago [5]. These smart contracts have enabled 4,056 DApps serving about 113.86K daily active users [6]. As the complexity of the DApps grows, smart contract developers are increasingly relying on well-established libraries to reduce development costs and avoid security vulnerabilities. For example, a recent study [45] revealed that code reuse is prevalent in smart contracts, and a large portion of the reuse is attributed to the most popular smart contract library—OpenZeppelin [29]: at least eight out of the top 20 most reused subcontracts are from OpenZeppelin. Furthermore, according to the Etherscan [5] search data, over 100K contracts have imported OpenZeppelin as a part of their source code. The most used OpenZeppelin APIs include: (1) the standard implementations of the ERC interfaces [15], such as ERC-20 (*fungible tokens*) and ERC-721 (*non-fungible tokens*), (2) proven implementations of the core utilities, such as the SafeMath APIs, providing math

utilities including the standard arithmetic operations and typecast functions that are free from the overflow/underflow vulnerabilities, and (3) common smart contract design patterns, such as the *role-based access control* pattern, and the *pull-payment* pattern that is immune to the reentrancy attacks [88].

Still in their early stages, both the Solidity language standard and the smart contract libraries are still quickly evolving and undergoing major upgrades. Therefore, the API documentations are important sources for developers who are unfamiliar with the libraries to learn the correct usages of the APIs. Yet, maintaining high-quality documentations is non-trivial and documentation errors are common even in widely-used and well-maintained libraries. For example, during the past six months since November 2021, 46% of the commits from OpenZeppelin have modified the documentation, many of which are fixes. Similar patterns can also be seen in Java libraries [46, 79]. The major causes of Solidity documentation errors include typos as well as grammatical errors, out-of-date descriptions of the API usages, failures in indicating important prerequisites as well as side effects of the APIs, etc.

In general, API documentation errors may pose a burden on developers and may mislead them, resulting in API misuses [37] and reduced productivity. Due to the unique structure of DApps, some specific features of Solidity library APIs are of particular importance to DApp developers. For example, a typical DApp consists of smart contracts deployed on a blockchain, and some off-chain code supporting the front-end user interface, creating, and sending transactions to the smart contracts (see Sect. 2.1 for more details). Because Ethereum DApps are event-driven, one needs to be aware of the *events* emitted by each API function to be able to properly handle them in the off-chain code. Mishandling events may lead to synchronization bugs [96]. Moreover, transactions sent through an API function may be *reverted*, if their *requirements* (e.g., having sufficient balance) are not met. Such information is usually described in API documentations, but can become out-of-date when libraries evolve (see Fig. 2b and the corresponding fix [30]).

To detect documentation errors in Solidity smart contract APIs, we propose an automated technique, called DocCon. DocCon identifies inconsistencies between the API documentations and the corresponding library code through a *fact-based* approach. We first obtain code facts by parsing the library code and extract relevant relations between important code elements, such as the event emission, contract inheritance, function override, and function call relations. We then extract similar relations from the corresponding API documentations as document facts, based on custom natural language templates. Both types of facts are represented with Datalog [35] following a uniform *fact schema* and stored in a *factbase*, which can be queried to detect documentation inconsistency errors at various severity levels. In particular, we deem *incorrect* documentations of highest severity (*level-1*), where a fact only appears in the documentation but not in the code. There are also *incomplete* documentations, where a fact from the code does not have corresponding descriptions in the documentations. These are further classified into the external (*level-2*) and internal (*level-3*) incompleteness errors based on the importance of the facts. We define the severity levels to make the management of documentation errors more effective: the *level-1* errors can be reported to the library developers with high confidence, while the *level-2* and *level-3* errors might be ignored if the library is not following a strict documentation guideline.

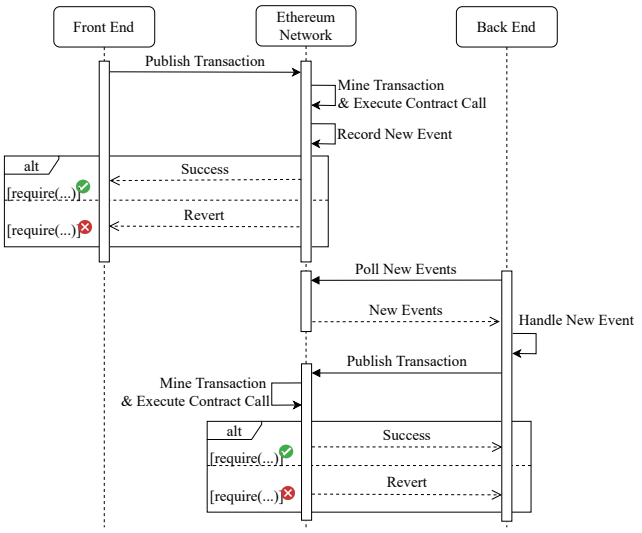

Figure 1: Typical workflow of Ethereum DApps.

Contributions. To summarize, in this paper, we made the following contributions.

- We proposed the first automated documentation error detection technique DocCon, for Solidity smart contract library APIs. Existing techniques in the same area focus on Java code and documentations, which demonstrate distinct structures and pose very different concerns.
- We designed a comprehensive schema and developed custom fact extractors for relevant Solidity code and documentation facts. We defined documentation inconsistency errors of various severity levels and implemented the corresponding factbase queries.
- We evaluated DocCon on three popular real-world smart contract libraries, namely, OpenZeppelin [29], Dappsys [9], and ERC721 Contract Extensions [16]. DocCon discovered high-priority API documentation errors, such as incorrect descriptions, missing events, and missing transaction requirements. We reported 40 errors to smart contract library developers, who have confirmed 29 and fixed 22 errors so far.
- Our dataset and tool implementation is made available online: https://sites.google.com/view/doccon-tool.

2 BACKGROUND

In this section, we review necessary terminology and background needed by the rest of the paper.

2.1 Solidity and DApp Primer

Solidity is the programming language used to write smart contracts on Ethereum and other EVM-compatible blockchain platforms, such as Binance Smart Chain [8] and TRON Network [34]. Solidity [80] supports code reuse through both contract inheritance and external library import. Different from traditional programming languages, Solidity itself is equipped with logging functionalities via two keywords, *event* and *emit*. Then the off-chain code can subscribe to these events and react based on DApp business logic.

```
/** ...  Emits a {TokensReleased} event. */
function release(address token) public virtual {
    uint256 releasable = vestedAmount(token,
    ↪ uint64(block.timestamp)) - released(token);
    _erc20Released[token] += releasable;
    emit ERC20Released(token, releasable);
    SafeERC20.safeTransfer(IERC20(token), beneficiary(),
    ↪ releasable);
}
```

(a) An incorrectly documented event emission in `VestingWallet`.

```
/** ...  Requirements:
* - `tokenId` must be already minted.
* - `receiver` cannot be the zero address.
* - `feeNumerator` cannot be greater than the fee denominator. */
function _setTokenRoyalty(uint256 tokenId, address receiver,
    uint96 feeNumerator) internal virtual {
    require(feeNumerator<=_feeDenominator(), "ERC2981: ...");
    require(receiver!=address(0),"ERC2981: Invalid parameters");
    ... }
```

(b) A spurious transaction requirement documented in `ERC2981`.

```
/** @dev Stores the sent amount as credit to be withdrawn.
*   @param payee The destination address of the funds. */
function deposit(address payee) public payable virtual onlyOwner {
    uint256 amount = msg.value;
    _deposits[payee] += amount;
    emit Deposited(payee, amount); }

/** @dev Called by the payer to store the sent amount as credit
    ↪ to be pulled ...
*   @param dest The destination address of the funds.
*   @param amount The amount to transfer. */
function _asyncTransfer(address dest, uint256 amount) internal
    ↪ virtual {
    _escrow.deposit{value: amount}(dest); }
```

(c) Undocumented transitive event emission in `PullPayment`.

```
/**  @dev Returns the item at the beginning of the queue. */
function front(Bytes32Deque storage deque) internal view returns
    ↪ (bytes32 value) {
    if (empty(deque)) revert Empty();
    int128 frontIndex = deque._begin;
    return deque._data[frontIndex]; }

/** @dev Returns the item at the end of the queue. */
function back(Bytes32Deque storage deque) internal view returns
    ↪ (bytes32 value) {
    if (empty(deque)) revert Empty();
    int128 backIndex;
    unchecked { backIndex = deque._end - 1;}
    return deque._data[backIndex]; }
```

(d) Undocumented transaction reversions in `DoubleEndedQueue`.

Figure 2: Examples of Solidity smart contract API documentation errors.

Figure 1 shows the typical workflow of Ethereum DApps. A DApp delegates its core functionality to the smart contracts deployed on the Ethereum network. A user can publish a transaction through the front-end to execute a certain smart contract function. When all the transaction requirements are satisfied, the contract execution is successful and events may be emitted as well; otherwise, the transaction is reverted. The back-end of a DApp is usually deployed on a off-chain server, which is subscribed to blockchain events and may decide to publish further transactions.

Suppose a user would like to give approval to a delegate account (e.g., a marketplace) to withdraw some ERC-20 [13] tokens from her account and to transfer them to other accounts. The user would first send a transaction from the front end calling the `approve` function, which emits an `Approval` event upon successful completion. The marketplace back end listening to the `Approval` event, would trigger a new transaction calling the `transferFrom` function to redeem the approved allowance and perform relevant transfers. The results of the transactions can also be reflected on the front end with proper handling code of the transaction status events.

2.2 Software Fact Extraction and Representation

Software artifacts can be reverse engineered to extract useful information as *facts*. Fact extractors are custom, human-defined analyzers that automatically scan structured software artifacts to retrieve pertinent details to be included in a resultant *factbase*. Fact extractors for different programming languages and platforms have been previously built, including *Javax* [2] for Java, *Cppx* [1] and *ClangEx* [3] for C/C++, and *ASX* [47] for assembler, objects, dynamic libraries and executables.

Datalog is a declarative language using a syntax similar to the logic programming language, Prolog. We use Datalog to represent facts and leverage its inference capabilities to capture the differences between document and code facts. Atoms are building blocks of Datalog. An atom $P(x_1, \ldots, x_n)$ represents an n-ary predicate P, and $x_1, \ldots, x_n$ are its arguments. For example, we can use a binary predicate `Call(`x`, `y`)` to describe function y being invoked in function x. An atom with all its arguments being constant represents a fact, e.g., `Call("transferFrom", "_move")` asserts that function `transferFrom` calls another function named `_move`. Datalog engines deduce new facts from existing ones according to some inference rules, which are horn clauses of atoms in the form,

$$P_1(x_1, \ldots, x_n) \leftarrow P_2(y_1, \ldots, y_s), \ldots, P_n(z_1, \ldots, z_t).$$

The predicate on the left hand side (P_1) is the *head* of a rule and the right hand side is the *body*. The head predicate is true if all the predicates in the body are true. For example, `IndirectCall(`x`, `z`) ← Call(`x`, `y`), Call(`y`, `z`)` states that function x indirectly calls z, if x calls y and y calls z. A Datalog program can contain multiple rules like this to implement a complex query. Modern Datalog implementations such as Soufflé [57] can efficiently deduce facts on a large set of facts and have been used in tasks such as pointer analysis [43].

3 EXAMPLES

We illustrate Solidity API documentation errors with the examples in Fig. 2. All of these examples are from the OpenZeppelin library, the de facto standard library of the Solidity community [45]. OpenZeppelin has more than 17.7K stars on GitHub and is actively maintained. Some parts of the library, such as *SafeMath*, have even been integrated into the Solidity language v0.8.+ [80]. We show these errors not only since they represent the typical errors in Solidity smart contract API documentations, but also because they were all successfully detected by DocCon.

Figure 2a is an example where an event emission is *incorrectly* documented. The `release` function belongs to a `VestingWallet` contract which is used to handle the vesting of Ether (the native

cryptocurrency on Ethereum) and ERC-20 tokens [13] for beneficiaries. An `ERC20Released` event is emitted to notify the users before the transfer starts. In Fig. 2a, the event is incorrectly documented as "Emits a `TokensReleased` event" (Line 1), and there is no such event defined in the library. As such, one may be misled into handling a non-existent event in the off-chain code.

Figure 2b shows an example where a transaction requirement is *incorrectly* documented. The `ERC2981` contract is the reference implementation of the EIP-2981 standard [12], which is used to retrieve royalty payment information for a given non-fungible token (NFT) or a set of NFTs. The `_setTokenRoyalty` function is for setting the royalty information for a specific token id. In this case, the requirement documented on Line 2, "`tokenId` must be already minted", is not enforced in the code. The OpenZeppelin developers also confirmed that this requirement is spurious and they have recently removed this requirement [30].

Figure 2c shows an example where the API documentation is *incomplete*, missing an event emission. This case involves both the `PullPayment` and `Escrow` contracts, which implement a pull-payment design pattern [90]. The function `Escrow.deposit` emits a `Deposited` event to notify the payee (Line 6). Yet, the event is documented neither in the `Escrow.deposit` function, nor its caller, the `PullPayment._asyncTransfer` function. This incomplete documentation affects not only the users of `Escrow.deposit`, but also users of functions further along the call chain.

Figure 2d is an example of an undocumented transaction reversion. The `DoubleEndedQueue` contract from OpenZeppelin implements a double-ended queue that supports pushing and popping operations from the both ends. Both the `front` and `back` functions have a transaction reversion condition such that the operations performed on empty queues will be reverted (Lines 3 and 9). However, this reversion is not documented for neither of the functions. Missing a transaction reversion in the documentation can mislead DApp developers, for example, resulting in unexpected reversion.

DocCon successfully detected all these issues from the corresponding versions of the library. Our detection results are either directly confirmed by the library developers in their responses to our submitted bug reports, or validated by developers' independent patches [7, 26, 30, 32].

Apart from the errors discussed above, we manually inspected the most recent commits of OpenZeppelin for the past six months and discovered that 46% of the commits involve fixes of the documentations. This shows that maintaining high-quality API documentations is indeed challenging—this also motivates us in developing an automated technique to detect API documentation errors for Solidity smart contracts.

4 METHODOLOGY

We now define the problem of identifying smart contract *API documentation errors* and describe our approach in details. An API documentation error in Solidity is an inconsistency between the library source code and its API documentations. Following the literature [87, 99], such inconsistencies can be further classified into the *incorrectness* and *incompleteness* issues. The former means that certain information is stated in the documentation but never implemented in the code, while the latter means that certain information in the code is never documented. In general, incorrectness is more severe than incompleteness. We further break incompleteness errors down to the *external* and *internal* incompleteness errors, based on their affected program elements. Specifically, external incompleteness concerns events and operations on transactions, while internal incompleteness affects other code entities. External incompleteness is more severe than internal incompleteness as it has direct impact on the users of the library. DocCon targets all the three types of errors.

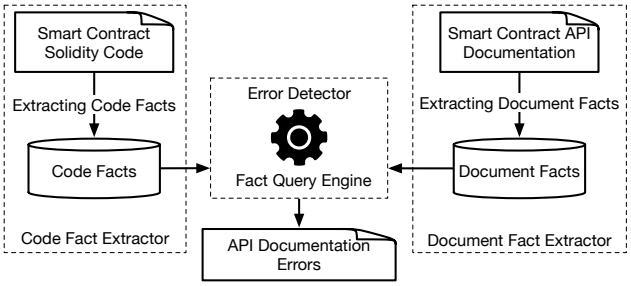

Figure 3: Overview of DocCon.

Figure 3 shows an overview of DocCon: it consists of three main components, namely, the code fact extractor, document fact extractor, and error detector. The code fact extractor accepts as input the Solidity code of the smart contract library under analysis, extracts, and outputs the extracted code facts. The document fact extractor takes as input the API documentation text of the same library and outputs the extracted document facts. Finally, the error detector performs queries on the extracted code facts and document facts with its query engine, and outputs the detected errors, if any.

4.1 Fact Schema

Table 1 shows the schema used for document and code facts. The left column lists the declarations of the predicates, showing their names and argument types. The right column gives a short description for each predicate. The first two rows represent contract inheritance and function overriding. The next five predicates are used to capture syntactic characteristics of the contract code, such as the modifiers and function parameters. The `Call` relation captures all function invocations in the program; `Require`, `Revert`, and `Emit` describe important semantics of functions, corresponding to the `require`, `revert`, and `emit` statements in Solidity programs. These three types of facts can be propagated through the call chain, i.e., if the callee has a requirement, event emission, or reversion under certain conditions, its callers also have those characteristics. The last one (`SeeFn`) is specific to document facts. As a convention, library developers sometimes write "see …" in the document of a function, referring users to the document of another function which shares the same descriptions as the current one. Upon seeing `SeeFn(a, b)`, we propagate all document facts of function `b` to `a`.

Most argument types in the predicates, such as `Ct`, `Fn`, are simply used to indicate which corresponding syntactic elements of the program the arguments refer to. The actual aguments of `Call` as well as the conditional expressions of `Require`, `Revert`, and `Emit` are of `Expr` type. `Expr` is defined in Table 1 as an algebraic data type

Table 1: Schema of documentation and code facts.

Predicates and Types	Descriptions
`Inherit(ca:Ct, cb:Ct)`	Contract cb **inherits** contract ca
`Override(ca:Ct, fa:Fn, cb:Ct, fb:Fn)`	Function cb.fb **overrides** ca.fa
`HasStateVar(c:Ct, v:SVar)`	Contract c **has** a state variable v
`HasFn(c:Ct, f:Fn)`	Contract c **has** a function f
`CtHasMod(c:Ct, m:Mod)`	Contract c **has** a modifer m
`HasParam(c:Ct, f:Fn, p:Var)`	Function c.f **has** a parameter p
`FnHasMod(c:Ct, f:Fn, m:Mod)`	Function c.f **has** a modifier m
`Call(ca:Ct, fa:Fn, e:ExprList, cb:Ct, fb:Fn, p:VarList)`	Function ca.fa **calls** cb.fb, e are arguments, p are parameters
`Require(c:Ct, f:Fn, e:Expr)`	c.f **requires** condition e to be true
`Revert(c:Ct, f:Fn, e:Expr)`	c.f **reverts** under condition e
`Emit(c:Ct, f:Fn, ev:Event, e:Expr)`	c.f **emits** event ev under condition e
`SeeFn(fa:Fn, fb:Fn)` [*doc only*]	fa **refers** fb with a "see fb" note

```
.type Expr = Cmp{a:Expr, b:Expr, op:BinOp} | Bool{b:symbol}
 | Logic{a:Expr, b:Expr, op:LogicOp} | Neg{e:Expr}
 | Math{a:Expr, b:Expr, op:MathOp} | Lit{x:symbol}
```

representing various forms of expressions, including binary comparisons, basic logic formulas (negation, conjunction, and disjunction), arithmetic expressions, Boolean values, and literals (e.g., strings, numbers, and dictionary lookups). For example, we may represent the expression "`owner != address(0)`" as `Cmp(Lit(("owner"), Lit("address(0)"),NEQ)`,[1] where "NEQ" stands for "not equal to". Being able to represent complicated expressions structurally enables us to better determine the equivalence of two conditional expressions, and to map variable names accordingly when propagating facts through the call chain.

4.2 Extracting Code Facts

DocCon aims to generate code facts defined in Table 1. DocCon first obtains a set of compilation units of the Solidity library code, and then leverages the Slither [31] parser to build their Abstract Syntax Trees (ASTs). Then the AST nodes are visited to collect relevant code names and their relations. The code names include types (contracts, interfaces, and events), constructors and functions, function modifiers, and variables (parameters and state variables). Their relations include the modifier uses, function calls, *require* statements, conditional *revert* statements, and (conditional) event *emit* statements.

```
(a) HasFn("VestingWallet", "release")
(b) HasParam("VestingWallet", "release", "token")
(c) Emit("VestingWallet", "release", "ERC20Released", "true")
(d) Call("VestingWallet", "release", ["IERC20(token)",
    ↪ "beneficiary()", "releasable"], "SafeERC20",
    ↪ "safeTransfer", ["token", "to", "value"])
```

Figure 4: Extracted code facts for `release` function in Fig. 2a

Figure 4 shows the extracted code facts for the `release` function in Fig. 2a. In Fig. 4, Line (a) is a fact that the `VestingWallet` contract has a `release` function, while Line (b) shows the function has a parameter named `token`. The event emission behavior and function call relation are captured by Lines (c) and (d), respectively. In (c), the "true" condition indicates that the function *always* emit the event. The call fact on Line (d) also records the correspondence between the callee parameter list and the argument list.[2]

Table 2: A partial list of DocCon's document templates.

Document Templates	Facts
<In cb: "Implement 'ca' Contract">	`Inherit(ca, cb)`
<In cb.fb: "Overridden from 'ca.fa'">	`Override(ca, fa, cb, fb)`
<In c.f: "@param 'p'">	`HasParam(c, f, p)`
<In c.f: "Guaranteed by the 'm' modifier">	`FnHasMod(c, f, m)`
<In c.f: "Requirements: - 'va' must be strictly less than 'vb'">	`Require(c, f, va < vb)`
<In c.f: "Reverts with ... if 'va' is at least 'vb'">	`Revert(c, f, va >= vb)`
<In c.f: "Emits an {e} event">	`Emit(c, f, e, "true")`
<In c.f: "Might emit an {e} event">	`Emit(c, f, e, "")`
<In ca.fa: "@dev See {cb.fb}">	`SeeFn(ca, fa, cb, fb)`

4.3 Extracting Document Facts

To figure out what facts to extract from the smart contract API documentations, we manually inspected the documents of 140 core functions (in the "Core" sections of the documentation) of the OpenZeppelin library. During the inspection, we focused on finding out: (1) whether the documents cover key features of Solidity contracts, including event emissions, transaction requirements, transaction reversions, which are pertinent to DApp construction (see Sect. 2.1); (2) whether the information is documented in a structured or semi-structured way such that we can build an automated fact extractor to extract document facts from them. We inspected OpenZeppelin because it is the most widely-used Solidity smart contract library, and we expected its documentation to be well maintained.

Observations. We found that the API documentation of OpenZeppelin indeed has an emphasize on event emissions, transaction requirements, and transaction reversions. Specifically, of the 140 functions we inspected, 77 explicitly document at least one event emission, transaction requirement, or transaction reversion. Besides, we discovered that the developers also documented many other aspects which can be captured by our fact schema defined in Sect. 4.1, including contract inheritance, function overriding, function with modifiers, function parameter descriptions, etc. We also observed that these key information is usually documented in a semi-structured way, enabling us to automatically extract the corresponding document facts. For example, in OpenZeppelin, the event emissions are documented in a uniform format on a separate line, "Emits/Might emit an {e} event", where "e" is the event name defined in code.

Based on these observations, we summarized 37 document templates covering all of our fact schema. Table 2 shows a partial list of them, while the complete list is available online [33]. Each document template encodes a rule for extracting a document fact from a sentence in the documentation. For example, the document template, <In `c.f`: "Emits an {e} event"> → `Emit(c,f,e,"true")`, translates the sentence "Emits an `e` event" from function `f`'s document, to a fact `Emit(c,f,e,"true")`. In the example of Fig. 2a, the

[1] In later sections, for brevity, we may write complex expressions in their original forms instead of using the actual Datalog representations.

[2] For brevity and easier understanding, lists in fact (d) are written in a form simpler than their actual Datalog representation.

fact `Emit("VestingWallet", "release", "TokensReleased", "true")`is generated from Line 1. Notice that each template only extracts one document fact, but the same document fact can be extracted from multiple document templates depending on the actual sentences. We used these document templates to extract document facts from the libraries evaluated in our experiments.

4.4 Detecting Inconsistencies

With the extracted code facts and document facts, DocCon's error detector executes queries on the combined facts, which checks if any fact does not exist in either side, respecting necessary propagation of facts along certain relations. If DocCon detects such a fact, then it reports either an *incorrectness* error, i.e., a fact is in the documentation but is not implemented in the code, or an *incompleteness* error, i.e., a fact exists in the code but is not documented.

Severity Levels. We set three severity levels for the detected API documentation errors. (1) An error is *level-1* if the document fact does not have corresponding code facts. (2) An error is *level-2* if the code fact does not have corresponding document facts, and the code fact is `Emit`, `Require`, or `Revert`. (3) An error is *level-3* if the code fact does not have corresponding document facts, and the code fact falls in other categories in the schema.

In general, level-1 errors are the most severe, as incorrect API documentations may mislead the library users. Level-2 and level-3 errors are generally less severe, and their key difference is that level-2 errors focus on the *external* behaviors of smart contracts. These include event emissions, transaction requirements, and transaction reversions. Failing to document such external behaviors may mislead DApp developers—unhanded events may lead to bugs in DApps. Finally, level-3 errors target *internal* behaviors, which include code structural information. For example, a parameter of a function may not be documented, which may be against the documentation guidelines.

Inconsistency Queries. Let D and C be the sets of document facts and code facts respectively. We first infer additional facts D' and C' according to the following rules, respectively.

(1) If a function reverts, its callers also revert, under the same condition. This propagation can be achieved with rules such as: `Revert(ca, caller, e)` ← `Revert(cb, callee, e)`, `Call(ca, caller, _, cb, callee, _)`. Here we leave out the handling of function parameters for brevity.

(2) If a function emits an event, its callers also emit this event. A rule similar to (1) follows: `Emit(ca, caller, ev, e)` ← `Emit(cb, callee, ev, e)`, `Call(ca, caller, _, cb, callee, _)`.

(3) If a function has a requirement, its callers should also have the corresponding requirement. Here we show a more complex rule to demonstrate how we substitue variable names according to the `Call` facts to achieve a better accuracy. The following rule deduces `require(caller, v ⋈ n)` for caller when the callee requires `p ⋈ n`, where `p` is a parameter of the `callee`, `v` is an actual argument and ⋈ is a binary comparison operator such as "≠". `Require(caller, v ⋈ n)` ← `Call(caller, [v], callee, [p])`, `Require(callee, p ⋈ n)`.

(4) Some relations are transitive, including inheritance, function overriding and function call . For example, contract `cb` inherits `ca`, and `cc` inherits `cb`, lead to a new fact: contract `cc` inherits `ca`. `Inherit(ca, cc)` ← `Inherit(ca, cb)`, `Inherit(cb, cc)`.

(5) If the document of function `ca.fa` has a "see `cb.fb`" note, all facts about `cb.fb` also apply for `ca.fa`. The following rule propagates `Revert` according to the `SeeFn` facts. `Revert(ca, fa, e)` ← `SeeFn(ca, fa, cb, fb)`, `Revert(cb, fb, e)`.

The first four categoreis of the rules expand code facts C to C' and the last one expands document facts D to D'. In the following paragraphs, we append suffixes *-D* or *-C* after predicate names to distinguish code facts from document facts, and use prefixes *L1-*, *L2-*, and *L3-* before predicate names to indicate inconsistencies of the correponding levels. We also use subscripts to indicate subset of D' or C': D'_{revert} represents the `Revert` facts in D'.

To discover level-1 inconsistencies, for each predicate p, we compute the set difference $D'_p \setminus C'_p$. For instance, $D'_{\text{revert}} \setminus C'_{\text{revert}}$ can be computed by: `L1Revert(c, f, e)` ← `RevertD(c, f, e)`, `!RevertC(c, f, e)`. We use Datalog inference instead of simple set algebra for more flexible comparison, e.g.,document facts usually do not have details about function parameters, so we can leave out the parameters when detecting inconsistencies: `L1Call(ca, fa, cb, fb)` ← `CallD(ca, fa, _, cb, fb, _)`, `!CallC(ca, fa, _, cb, fb, _)`. Also, documentation tends to not explicitly state the condition of `emit`, but only indicate "might emit". Thus, we only distinguish "might emit" and "(always) emit", but do not treat emit under concrete conditions in code facts and "might emit" with unknown condition in document facts as inconsistencies, leading to the following two rules.

(6) "(always) emit" in document facts, but not in code facts: `L1Emit(c, f, ev, "true")` ← `EmitD(c, f, ev, "true")`, `!EmitC(c, f, ev, "true")`.

(7) "might emit" in document facts, but not in code facts: `L1Emit(c, f, ev, "")` ← `EmitD(c, f, ev, e1)`, `!EmitC(c, f, ev, e2)`, `e1 ≠ "true"`, `e2 ≠ "true"`.

For level-2 inconsistencies, we calculate $C'_p \setminus D'_p$ for $p \in$ {revert, emit, require}; and for level-3 inconsistencies, similarly, the inconsistencies are $C'_p \setminus D'_p$ for $p \notin$ {revert, emit, require}.

5 EVALUATION

In this section, we first propose research questions for evaluating DocCon, then describe our experimental setup and the subjects for evaluation. Finally, we present our experiment results, discuss the findings, and answer the research questions.

We evaluated DocCon with respect to the following research questions. **RQ-1**: How precise is DocCon in detecting errors in Solidity smart contract API documentations? **RQ-2**: How relevant are the smart contract API documentation errors detected by DocCon? **RQ-3**: What are the categories of the smart contract API documentation errors detected by DocCon?

5.1 Experimental Setup

DocCon was implemented in Python. It consists of three main components: (1) code fact extractor, (2) document fact extractor, and (3) error detector. We used the Slither [31] static analysis framework to analyze the ASTs of Solidity programs and extract code facts. We used the document templates defined in Sect. 4.3 to extract

Table 3: Solidity libraries used in the experiments.

Library	URL (https://github.com/)	Version	LOC	#Stars
OpenZeppelin	OpenZeppelin/openzeppelin-contracts	v4.5.0	17.3k	17.7K
Dappsys	dapphub/dappsys	HEAD	3.5k	1094
ERC721 Cont. Ext.	1001-digital/erc721-extensions	v0.0.18	0.9k	114

document facts. We performed fact querying using the Datalog engine Soufflé [57].

Table 3 lists three popular libraries that DocCon was evaluated on. The selected libraries are of high quality and widely used. OpenZeppelin [29] is the most popular smart contract libraries [45]. Dappsys [9], launched from August 2016, is another collection of building blocks for DApps, including ds-token [11], ds-proxy [10], etc. ERC721 Contract Extensions [16] provides a set of composable extensions for the OpenZeppelin ERC-721 base contracts. ERC721 Contract Extensions is used by many popular NFT DApps such as OpenSea having 32.83K daily users on Ethereum [6]. For OpenZeppelin and ERC721 Contract Extensions, We used their latest release versions at the time of our experiments, which are v4.5.0 and v0.0.18 respectively. As Dappsys does not publish a recent release version, we used the latest SHA of its each sub-library at the time of our experiments. The complete list of SHAs can be found on DocCon's website [33].

We ran DocCon on all the libraries with all the three severity levels. For each library, we ran both the code fact and document fact extractors on the source code and API documentations, respectively. Notice that different libraries may put their API documentations in separate locations. OpenZeppelin and ERC721 Contract Extensions keep their API documentations in the source files, and the documentation of each function is next to the corresponding code block. Dappsys keeps its documentation in separate text files. Therefore, we created separate parsers for each library to associate each function's code with its documentation. When processing code facts from a library, we excluded the facts of its dependent libraries to avoid double counting. Specifically, ERC721 Contract Extensions depends on OpenZeppelin, while Dappsys has dependencies among its component libraries. During the run, DocCon skipped a dependent code fact if it had already been processed previously. We extracted document facts in a uniform way for all the library functions using our document templates. Finally, with the extracted code facts and document facts, we executed the error detector for all the three severity levels, obtaining a set of errors on each level.

All the experiments were performed on a 4-core Intel(R) Core(TM) i7-8650 CPU @ 1.90 GHz machine with 16GB of RAM, running Ubuntu 16.04, with Python 3.8.13, Slither 0.8.3, and Conda 4.7.10.

5.2 Results

Table 4 shows the results of DocCon in detecting API documentation errors on all the libraries. In total, DocCon detected 56, 787, and 4,566 errors when the severity level is set to level-1, level-2, and level-3, respectively. DocCon successfully detected errors on all the severity levels on all the libraries, demonstrating its effectiveness. The observation that level-1 errors (i.e., incorrectness) appear in all the libraries indicates that many errors exist in smart contract library API documentations, which further validates the value of DocCon. The results also confirm the generalizability of DocCon. Although our fact schema was initially designed based on observations from OpenZeppelin, we were able to effectively extract facts from other libraries as well. It again attests that smart contract developers generally focus on similar aspects of APIs, as captured by DocCon's fact schema.

5.2.1 DocCon's Precision in Detecting Smart Contract API Documentation Errors. To measure the precision of DocCon in detecting smart contract API documentation errors, we manually inspected the level-1 and level-2 errors detected by DocCon to determine whether an error is true positive or false positive. For level-1 errors, we inspected all the cases. For level-2 errors (567, 141, and 79 in OpenZeppelin, Dappsys, and ERC721 Contract Extensions, respectively), we inspected all errors from Dappsys and ERC721 Contract Extensions. We sampled 229 errors from OpenZeppelin, in order to reach a confidence level of 95 % with a margin error within ±5 % on whether the sample is representative of all reports. For these 505 (49+4+3+229+141+79) errors, two of the authors spent five minutes per error to confirm the true positives, respectively. In case the verdict by the two authors was not unanimous, a third author broke the tie. Via this confirmation process, we got the precision of DocCon, which is shown in Table 4. The overall precision of DocCon on level-1 and level-2 errors is 76% and 66%, respectively. DocCon showed highest precision on ERC721 Contract Extensions and lowest precision on Dappsys.

Answer to RQ1: DocCon successfully detected 56 level-1 and 787 level-2 API documentation errors in all the three smart contract libraries, with the level-1 and level-2 precision of 76% and 66%, respectively.

False Positives. There are two main sources of false positives for DocCon: (1) imprecise static analysis in code fact extraction, and (2) missing domain knowledge in document fact extraction.

Figure 5 shows an example false positive case caused by imprecise static analysis. It is a level-1 error reported on function `_spendAllowance` from the OpenZeppelin library. In this case, the document fact is `Emit("ERC20._spendAllowance", "Approval", "")` (extracted from Line 1, where `""` indicates that the function might emit an event under some unstated condition). The document fact is correct, because the `Approval` event will only be emitted by the call to the `_approve` function (Line 7), when the `if` condition is evaluated to `true`. However, there is no corresponding code fact for it, thus DocCon reported a level-1 error. We inspected the code facts and found a related one: `Emit("ERC20._spendAllowance", "Approval", "true")`. The key difference between the two is the emit condition, i.e., `"true"` (always emit) versus `""` (might emit). This is because DocCon over-approximates call relations: a fact, `Call(`f_1`, _, `f_2`, _)`, is extracted regardless of the calling condition. With a more sophisticated context-sensitive analysis, this false positive can be eliminated with some timing overhead.

The second reason for false positives is missing domain knowledge. From domain experts' view, certain code fact and document fact have equivalent semantics, but DocCon reported errors due to their mismatched representations. For example, DocCon does not

```
1 /** ...  Might emit an Approval event. */
2 function _spendAllowance(address owner, address spender,
3                           uint256 amount) internal virtual {
4   uint256 currentAllowance = allowance(owner, spender);
5   if (currentAllowance!=type(uint256).max) {
6     require(currentAllowance>=amount,"...insufficient
  ↪ allowance");
7     unchecked{_approve(owner, spender, currentAllowance-amount);}
8   }
9 }
```

Figure 5: A false positive example caused by imprecise static analysis.

Table 4: Results of DocCon on detecting Solidity API documentation errors.

Library	#Detected			Precision	
	Lv-1	Lv-2	Lv-3	Lv-1	Lv-2
OpenZeppelin	49	567	3,741	78%	72%
Dappsys	4	141	448	50%	53%
ERC721 Cont. Ext.	3	79	377	100%	73%
Overall	56	787	4,566	76%	66%

understand that the concept "the caller having X number of tokens" has the same meaning as "the message sender has X balance" in the scenarios of using ERC-777 non-fungible tokens [14]. Such domain knowledge are way too complex to be captured by DocCon's rule-based document templates. With more advanced NLP techniques, such false positives can potentially be reduced, but the technical complexity will be significantly increased.

5.2.2 Practical Relevance of DocCon. To evaluate the practical relevance of the errors detected by DocCon, we reported the detected true positive errors to library developers and collected their feedback. We reported the errors in a hierarchical way based on their severity levels. We reported all the distinct level-1 incorrectness errors (we excluded 18 errors due to their repetitive causes) and five randomly sampled distinct level-2 incompleteness errors for each library. Since level-3 errors focus on internal incompleteness issues, such as undocumented function parameters, we did not report them due to their low importance.

In total, we reported 40 errors detected by DocCon to the library developers, of which 25 are level-1 errors and 15 are level-2 errors. Our reported errors have received extensive welcomes from the developers. By the time of submission, the library developers have already confirmed 29 errors and fixed 22 errors, with a significant confirmation rate of 72.5%. The links to the corresponding GitHub issues, together with developers' responses and their pull requests fixing the errors, are publicly available on DocCon's website [33].

Discussions. DocCon successfully detected all the four errors in Fig. 2, and the errors are either confirmed and fixed by the library developers based on our bug reports [26, 32], or validated by their independent fixes [7, 30]. From the `release` function of the Fig. 2a example, DocCon managed to extract a document fact, `Emit("VestingWallet", "release", "TokensReleased", "true")`, and a related code fact, `Emit("VestingWallet", "release", "ERC20Released", "true")`. This mismatch is captured by the error detection rules. We have submitted a bug report for it, which was later confirmed and fixed [26]. For the example in Fig. 2b, DocCon extracted the document fact `Require ("ERC2981", "_setTokenRoyalty", "_minted(tokenId)", "true")`, while no corresponding code fact exists. This case was also validated by developers' independent fixes that happened after our experimented release version [30].

For the example in Fig. 2c, DocCon extracted the code facts `Call ("PullPayment", "_asyncTransfer", "Escrow", "deposit")` and `Emit ("Escrow", "deposit", "Deposited", "true")`. The emission of event `Deposited` was not documented for the `Escrow.deposit` function. Based on the propagation of events through function call chains, DocCon's fact propagation rules inferred that function `PullPayment._asyncTransfer` should also emit `Deposited`, which was not documented either. Therefore, DocCon reported two errors as a result. We submitted a bug report on this case, which was also confirmed and fixed by the library developers [32]. The developers also responded us in details, providing additional domain knowledge and mentioning that they would improve the wording of the documentation as well:

> *"I'll add the details..._asyncTransfer will indeed trigger an event, but at another address (the escrow and the PullPayment are two different contract) so the wording should reflect that."* [20]

For the example in Fig. 2d, DocCon extracted the code facts `Revert ("DoubleEndedQueue", "front", "Empty()")` and `Revert ("Double-EndedQueue", "back", "Empty()")`. Neither fact has corresponding document facts in the results. Therefore, when running on the library version with this error, DocCon detected the mismatches and reported two errors. This case was also validated by developers' independent fix that happened after our experimented release version [7].

With the fact propagation rules (Sect. 4.4), DocCon was capable of detecting errors that need to be exposed via complex function call chains and documentation references. Figure 6 shows such an example. In this case, function **`ERC721.`**`safeTransferFrom` implements the interface function **`IERC721.`**`safeTransferFrom`, and its documentation only contains a reference to the corresponding interface documentation (Line 7: `@dev See {IERC721-safeTransferFrom}`). The interface documentation explicitly requires that the `from` parameter value cannot be the zero address (Line 3), but the implementation function `ERC721.safeTransferFrom` itself, together with all its (transitive) callee functions, did not enforce this transaction requirement. DocCon extracted code fact `Require("IERC721", "safeTransferFrom", "from"!="address(0)")`. With the fact propagation rule (3) in 4.4, it inferred the code fact `Require("ERC721", "safeTransferFrom", "from"!="address(0)")` which has no corresponding document fact. Therefore, DocCon reported an error. Our bug report on this case was also confirmed [28]. The developers did not fix it though, as in their implementation this requirement is implied by another `require` statement (Line 14), according to their explanations:

> *"In our implementation, the address(0) is an invalid owner. You cannot transfer any token to 0 (unless you are burning the token)..."* [18]

```
/** @dev Safely transfers `tokenId` token from `from` to `to` ...
* Requirements:
* - `from` cannot be the zero address. ... */
function safeTransferFrom(
    address from, address to, uint256 tokenId) external;

/**  @dev See {IERC721-safeTransferFrom}. */
function safeTransferFrom(address from, address to,
    uint256 tokenId) public virtual override {
    safeTransferFrom(from, to, tokenId, "");
}
function _transfer(address from, address to,
                   uint256 tokenId) internal virtual {
  require(ERC721.ownerOf(tokenId) == from, "... incorrect owner");
  require(to!=address(0), "ERC721: transfer to the zero address");
  _beforeTokenTransfer(from, to, tokenId); ...}
```

Figure 6: A level-1 error in OpenZeppelin exposed through complex reference chains.

```
/* ... Requirements: `index` must be strictly less than length. */
function at(Bytes32Set storage set, uint256 index) internal view
  ↪  returns (bytes32) { return _at(set._inner, index); }
```

Figure 7: Another level-1 error in OpenZeppelin.

This response is interesting as it indicates that in the API documentations, certain facts can be implied by others and can potentially be extracted with more advanced documentation analysis. Currently, DocCon is not able to figure out such implications as it requires complex domain knowledge that cannot be captured by our current document templates.

There are seven cases confirmed by developers, but they were not fixed. One such example is Fig. 7. In this case, the `EnumerableSet.at` function requires that `index` must be strictly less than the length of the array (Line 1). However, in the function code, there is no corresponding `require` statement enforcing this rule. DocCon also detected that the document fact `Require("EnumerableMap", "at", "index"<"length")` does not have any corresponding code fact and thus reported an error. We submitted a bug report on this case, which was confirmed by the library developers [24]. But they did not fix it, because the newer Solidity compiler performs the index-out-of-bound check itself.

> *"set._values is a solidity array. The solidity compiler checks that the index is lower than the length. It's true that we do not check the index ourselves because solidity does it..."* [19]

To capture this information, one needs to incorporate deeper language semantics into the analysis, which is beyond the scope of DocCon. Encouragingly, even for those cases that were not fixed (seven in total), the developers still positively confirmed our findings and provided detailed explanations.

The fact that developers confirmed and fixed most of our reported errors firmly validates that DocCon is valuable in practice. The developers also admitted that they needed more consistency and clearer documentation guidelines after confirming several errors reported by us:

> *"Thank you for pointing that out. We definitely need more consistency or at least clearer guidelines on how we approach that matter..."* [21]

They encouraged us to submit further issues and pull requests: *"You're welcome to submit pull requests as well next time"* [22]. We would also like to take our experiment findings and their feedback into account to further enhance DocCon in the future.

```
/* @dev Moves `amount` of tokens from `sender` to `recipient` */
function _transfer(address from, address to, uint256 amount
) internal virtual {
  require(from != address(0), "... from the zero address"); ... }
```

Figure 8: A level-1 error in the element containment category.

Answer to RQ2: We reported 40 errors detected by DocCon in total to smart contract library developers, who have already confirmed 29 errors and fixed 22 by the time of submission (confirmation rate = 72.5%). The results validate that DocCon's detection results are valuable in practice.

5.2.3 Categorization of Smart Contract API Documentation Errors. To study the errors in smart contract API documentations in depth, we categorized all the manually validated true-positive level-1 and level-2 errors (the manual inspection and sampling process is described in Sect. 5.2.1). For each error, we determined its category based on the facts involved in detecting it.

Event Emission. The errors in this category were detected by the `Emit` facts. An event emission error indicates either i) an event is documented but does not exist in the corresponding code (level-1), or ii) an event is emitted in the code but not documented in the corresponding documentation (level-2). Figure 2a and Fig. 2c demonstrate these two scenarios, respectively.

Transaction Requirement and Reversion. The errors in this category were detected by the facts `Require` and `Revert`. Similar to the previous category, an error indicates either a transaction requirement or reversion is documented but not enforced (level-1), or it is enforced but not documented (level-2). Figure 2b and Fig. 2d are level-1 and level-2 errors in this category, respectively.

Element Containment. The errors in this category were detected by the facts `HasFn`, `CtHasMod`, `HasStateVar`, and `HasParam`. Similarly, an error indicates either a containment relation is documented but not in the code (level-1), or it exists in the code but is not documented (level-3). Currently, DocCon supports the following containment relations: a contract containing functions/modifiers/state variables, and a function containing parameters. Figure 8 demonstrates an example of a level-1 element containment error. In this case, the documentation incorrectly references parameter names, `sender` and `recipient`, since the actual parameter names in the code are `from` and `to`. This error was detected by DocCon using the `HasParam` fact. It was confirmed and fixed by the library developers [23, 25].

Element Reference. The errors in this category were detected by the facts `Inherit`, `Override`, and `FnHasMod`. An error indicates either a reference relation is documented but not in the code (level-1), or it exists in the code but is not documented (level-3). Currently, DocCon supports the following reference relations: a contract inherits another contract, a function overrides another function, and a

```
1 /// Implement the `HasSecondarySalesFees` Contract ...
2 function getFeeRecipients(uint256) public view override returns (
3   address payable[] memory ) {
4   address payable[] memory recipients = new address payable[](1);
5 ... }
```

Figure 9: A level-1 error in the element reference category.

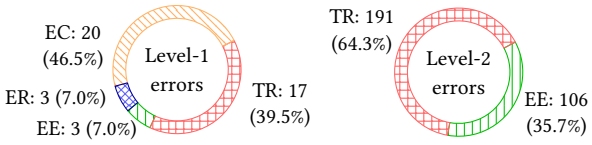

Figure 10: Distribution of categories of errors. (EE: Event Emission, TR: Transaction Requirement or Reversion, EC: Element Containment, ER: Element Reference)

function uses a modifier. Figure 9 shows an example of a level-1 element reference error. In this case, the documentation (Line 1) incorrectly refers to a non-existing contract `HasSecondarySalesFees`. It was detected by DocCon with the `Override` fact. It was confirmed and fixed by the library developers [17, 27].

Figure 10 shows the distribution of error categories for level-1 and level-2 errors. For level-1 errors, the largest category is element containment, which is mainly due to outdated information, such as certain code entities not being consistently renamed. The event- and transaction-related cases take about 47% of the level-1 errors. These key features, although critical to smart contracts, are often incorrectly documented. They also constitute all of level-2 errors, which indicates that these features are often incompletely documented as well.

We observed that in Solidity smart contracts, the API documentation errors are different from those of the general-purpose programming languages, e.g., errors in Javadoc. To the best of our knowledge, all the existing works on detecting Javadoc inconsistencies focus on aspects including code names [59, 98] and parameter constraints (nullness, types, value ranges, and exceptions) [99, 100]. In Solidity, many of the API documentation errors are related to events and transactions, which do not have their counterparts in Java. Therefore, these works are not directly applicable in our domain. Even if one re-implemented a technique for Javadoc to check Solidity libraries, the technique could detect no more than two categories of the errors (i.e., element containment and element reference), which only take 6.76 % of the errors detected by DocCon.

Answer to RQ3: There are four categories of the smart contract API documentation errors detected by DocCon: event emission (32.06 %), transaction requirement or reversion (61.18 %), element containment (5.88 %), and element reference (0.88 %). Half of the categories have no counterparts in general-purpose programming languages such as Java.

5.3 Threats to Validity

Internal. There is no ground truth for the Solidity smart contract API documentation errors. To mitigate this issue, we manually verified the errors detected by DocCon. Two of the authors spent five minutes per error to confirm whether it is a true positive. Besides, many reported true positive errors have been confirmed or fixed by the library developers, which also attests the accuracy of our manual labeling.

External. Our experimental findings may not generalize to other smart contract libraries. To mitigate this threat, we selected high-quality and widely used smart contract libraries as the representatives where all of their source code and API documentations are available. DocCon can only support Solidity-written smart contract libraries, and in the future our approach can also be extended to the libraries of other smart contract programming languages.

6 RELATED WORK

API Documentations. There is a large body of empirical studies on API documentations [36, 38, 46, 55, 67, 71, 75, 78, 79, 87, 97]. Their findings indicate that maintaining high-quality documentations is non-trivial and documentation errors are common even in widely-used and well-maintained libraries [46, 79]. Moreover, developers have higher chances of introducing bugs and ask more questions when using APIs with linguistic anti-patterns in the documents [36]. Saied et al. [78] also conducted an observational study on the API usage constraints and their documentation.

Zhong and Su [98] proposed an approach combining NLP and code analysis to detect API documentation errors. Their work focuses on grammatical errors (e.g., erroneous spellings) and incorrect code names (document-referred names that do not exist in source code). Lee et al. [59] proposed a technique extracting change rules from code revisions and applying the rules to detect outdated API names in Java API documents. Their work targets API names, including names of Java classes, methods, and fields. Compared with these works, DocCon targets not only name errors but also semantic errors in the API documentation, such as event emissions, transaction requirements, and transaction reversions. Zhou et al. [99] proposed a technique that combines NLP and static analysis to detect defects on parameter usage constraints in Java APIs. Their work focuses on Java parameter constraints, including parameter nullness, type restriction, and range limitation. Their subsequent work [100] automatically repairs these defects in Java API documents based on the detection results. DocCon differs from [99] in two main aspects. First, most of the parameter constraints in [99] are not applicable to Solidity, as there is no notion of nullness in Solidity and the type of an address parameter cannot be statically determined. Second, DocCon targets many semantic constraints that are specific to the smart contract domain, such as event emission and transaction requirements, which do not have their counterparts in Java programs.

Code Comments. There is also a large body of works on analyzing code comments [42, 53, 60, 62, 68, 69, 74, 76, 77, 81, 83–85, 89, 94, 95]. One direction is to detect inconsistencies between comments and code, sometimes called fragile comments [77], through program analysis [95] and heuristic rules [83, 89]. Similar idea can also be applied to suggest updates to comments. For example, techniques including CUP [69], CUP^2 [68], and HebCup [60] were proposed to automatically perform just-in-time comment updates when the corresponding code is changed. Habib and Pradel [53] proposed

an approach to automatically infer thread safety documentations for Java classes that can assist test generation. Another direction focuses on retrieving information with NLP-based techniques from various software artifacts, including documents, discussions, issues, reviews, and manuals. Compared with these works, DocCon targets very different research questions. Typical Solidity programs are very succinct for efficiency considerations (e.g., gas usage). As a result, they have few inline comments. Besides, the tags used in Solidity API documentations are significantly different from Javadoc (e.g., there is no `@return` tags), as Solidity API developers mainly focus on event and transaction related properties.

Fact-Based Approaches. The idea of reverse engineering programs and representing relevant information as facts is not new. The existing work on fact extraction can be broadly categorized into the *intra-version* and *inter-version* ones. Intra-version fact extraction focuses on a single version of the program artifacts. There are many downstream analyses performed on the intra-version facts for architecture understanding [40], visualization, and redocumentation [58]. Inter-version fact extraction relies on sophisticated *structural differencing* [41, 44, 48] and *code change classification* [50, 51, 54] algorithms. The former is used to compute an optimal sequence of atomic edit operations that can transform one AST into another, and the latter is used to classify atomic changes according to their change types. Wu et al. [92] proposed a uniform exchangeable representation of differential facts, which establishes links between intra-version facts and inter-version facts. Their fact representation supports efficient querying and manipulation, and thus can be used in various program analysis tasks. This work combines program facts and documentation facts for inconsistency checking.

Smart Contract Analysis. There have been a large number of static [4, 49, 52, 61, 70, 72] and dynamic [56, 63, 86, 88, 93] analysis techniques focusing on the security aspects of smart contracts. For example, Oyente [70] is one of the earliest static analysis tool to detect security vulnerabilities such as *reentrancy* and Slither [49] performs taint analysis to find information flow vulnerabilities. Model-based testing [64, 65] and fuzzing [56, 86, 88, 93] have also be explored to discover common security issues, such as permission bugs [64]. Another interesting property concerning the quality of smart contracts is their fairness. For instance, Bartoletti et al. [39] found through a survey that a large number of the transactions on Ethereum could be owing to Ponzi schemes. Such fairness issues may endanger the trust of users towards the blockchain platforms. Liu et al. [66] developed an automated tool to verify fairness properties through a game-theoretic approach. To the best of our knowledge, DocCon is the first work for detecting smart contract API documentation errors. It can be integrated into the current tool chains to enhance the overall quality of DApp development.

7 CONCLUSION

In this paper, we presented an automated technique DocCon to detect API documentation errors for Solidity smart contract libraries. Our approach is based on the extraction and querying of pertinent facts from both the library code and documentation. We designed a unified documentation-code fact schema and outline the queries for various severity levels. Through experimental evaluation on popular Solidity libraries, DocCon is shown to be effective in identifying documentation errors and its practical value is further proven by the positive feedback from the library developers.

ACKNOWLEDGMENTS

This research is supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (MOE2019-T2-1-040) and the Singapore National Research Foundation under the National Satellite of Excellence in Mobile Systems Security and Cloud Security (NRF2018NCR-NSOE004-0001).

REFERENCES

[1] 2001. Cppx - Open Source C++ Fact Extractor. http://www.swag.uwaterloo.ca/cppx.
[2] 2010. Javex - Java Fact Extractor. http://www.swag.uwaterloo.ca/javex.
[3] 2018. ClangEx – A Fast C/C++ Fact Extractor. https://github.com/bmuscede/ClangEx.
[4] 2019. Mythril: A Security Analysis Tool for EVM Bytecode. https://github.com/ConsenSys/mythril.
[5] 2020. Etherscan. https://etherscan.io.
[6] 2021. State of The DApps. https://www.stateofthedapps.com/zh/platforms/ethereum.
[7] 2022. Add missing docs about reverts in DoubleEndedQueue. https://github.com/OpenZeppelin/openzeppelin-contracts/commit/525a6728625d40c738dc7d1ec9e518e066f22054.
[8] 2022. BNB Smart Chain. https://www.bnbchain.world/en/smartChain.
[9] 2022. Dappsys. https://github.com/dapphub/dappsys.
[10] 2022. DSProxy. https://github.com/dapphub/ds-proxy.
[11] 2022. DSToken. https://github.com/dapphub/ds-token.
[12] 2022. EIP-2981: NFT Royalty Standard. https://eips.ethereum.org/EIPS/eip-2981.
[13] 2022. ERC-20 Token Standard. https://ethereum.org/en/developers/docs/standards/tokens/erc-20.
[14] 2022. ERC-777 Token Standard. https://ethereum.org/en/developers/docs/standards/tokens/erc-777.
[15] 2022. ERC: Ethereum Improvement Proposals. https://eips.ethereum.org/erc.
[16] 2022. ERC721 Contract Extensions. https://github.com/1001-digital/erc721-extensions.
[17] 2022. Fix comments. Fixes #12. https://github.com/1001-digital/erc721-extensions/commit/c4b3ec8ebbf3534f3a6f1105e13f7d674029aa3d.
[18] 2022. GitHub Comment on Issue #3362. https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3362#issuecomment-1110708897.
[19] 2022. GitHub Comment on Issue #3366. https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3366#issuecomment-1111937624.
[20] 2022. GitHub Comment on Issue #3369. https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3369#issuecomment-1111948931.
[21] 2022. GitHub Comment on Issue #3374. https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3374#issuecomment-1113172780.
[22] 2022. GitHub Comment on Issue #3374. https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3374#issuecomment-1117950893.
[23] 2022. Improve wording consistency in code/doc. https://github.com/OpenZeppelin/openzeppelin-contracts/pull/3365.
[24] 2022. Inconsistency between the code and doc of EnumerableSet._at. https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3366.
[25] 2022. Inconsistency between the code and doc of ERC20._transfer. https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3359.
[26] 2022. Inconsistency between the code and the doc of VestingWallet.release. https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3368.
[27] 2022. Inconsistency between the code and the doc of WithFees.getFeeRecipients and WithFees.getFeeBps. https://github.com/1001-digital/erc721-extensions/issues/12.
[28] 2022. Inconsistency between the code of ERC721.safeTransferFrom and the doc of IERC721.safeTransferFrom. https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3362.
[29] 2022. OpenZeppelin. https://openzeppelin.com.
[30] 2022. Remove outdated documentation in ERC2981._setTokenRoyalty. https://github.com/OpenZeppelin/openzeppelin-contracts/commit/dd018894345a99e5058578cd0dc15bbb31b631b3.
[31] 2022. Slither: The Solidity Source Analyzer. https://github.com/crytic/slither.
[32] 2022. Suggesting updates on the doc of Escrow.deposit and Escrow.withdraw. https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3369.
[33] 2022. The DocCon Website. https://sites.google.com/view/doccon-tool.
[34] 2022. TRON Network. https://tron.network/.

[35] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley Reading.

[36] Emad Aghajani, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2018. A Large-Scale Empirical Study on Linguistic Antipatterns Affecting APIs. In *International Conference on Software Maintenance and Evolution*. 25–35.

[37] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2018. A Systematic Evaluation of Static API-Misuse Detectors. *IEEE Transactions on Software Engineering* 45, 12 (2018), 1170–1188.

[38] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2016. Linguistic Antipatterns: What They Are and How Developers Perceive Them. *Empirical Software Engineering* 21, 1 (2016), 104–158.

[39] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. 2020. Dissecting Ponzi Schemes on Ethereum: Identification, Analysis, and Impact. *Future Generation Computer Systems* 102 (2020), 259–277.

[40] Dirk Beyer, Andreas Noack, and Claus Lewerentz. 2003. Simple and Efficient Relational Querying of Software Structures. In *Working Conference on Reverse Engineering*. 216–225.

[41] Philip Bille. 2005. A Survey on Tree Edit Distance and Related Problems. *Theoretical Computer Science* 337, 1-3 (2005), 217–239.

[42] Arianna Blasi, Nataliia Stulova, Alessandra Gorla, and Oscar Nierstrasz. 2021. RepliComment: Identifying Clones in Code Comments. *Journal of Systems and Software* 182 (2021), 111069.

[43] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 243–262.

[44] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. 1996. Change Detection in Hierarchically Structured Information. In *ACM SIGMOD International Conference on Management of Data*. 493–504.

[45] Xiangping Chen, Peiyong Liao, Yixin Zhang, Yuan Huang, and Zibin Zheng. 2021. Understanding Code Reuse in Smart Contracts. In *International Conference on Software Analysis, Evolution and Reengineering*. 470–479.

[46] Barthélémy Dagenais and Martin P Robillard. 2010. Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 127–136.

[47] I. J. Davis and M. W. Godfrey. 2010. From Whence It Came: Detecting Source Code Clones by Analyzing Assembler. In *Working Conference on Reverse Engineering*. 242–246.

[48] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *International Conference on Automated Software Engineering*. 313–324.

[49] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *International Workshop on Emerging Trends in Software Engineering for Blockchain*. 8–15.

[50] Beat Fluri and Harald C. Gall. 2006. Classifying Change Types for Qualifying Change Couplings. In *International Conference on Program Comprehension*. 35–45.

[51] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. 2007. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering* 33, 11 (2007), 725–743.

[52] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts. *Proceedings of the ACM on Programming Languages* (2018), 116.

[53] Andrew Habib and Michael Pradel. 2018. Is This Class Thread-Safe? Inferring Documentation using Graph-Based Learning. In *International Conference on Automated Software Engineering*. 41–52.

[54] Masatomo Hashimoto and Akira Mori. 2008. Diff/TS: A Tool for Fine-Grained Structural Change Analysis. In *Working Conference on Reverse Engineering*. 279–288.

[55] Andrew Head, Caitlin Sadowski, Emerson Murphy-Hill, and Andrea Knight. 2018. When Not to Comment: Questions and Tradeoffs with API Documentation for C++ Projects. In *International Conference on Software Engineering*. 643–653.

[56] Bo Jiang, Ye Liu, and WK Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *International Conference on Automated Software Engineering*. 259–269.

[57] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *International Conference on Computer Aided Verification*. 422–430.

[58] Holger M. Kienle and Hausi A. Müller. 2010. Rigi – An Environment for Software Reverse Engineering, Exploration, Visualization, and Redocumentation. *Science of Computer Programming* 75, 4 (2010), 247–263.

[59] Seonah Lee, Rongxin Wu, Shing-Chi Cheung, and Sungwon Kang. 2019. Automatic Detection and Update Suggestion for Outdated API Names in Documentation. *IEEE Transactions on Software Engineering* 47, 4 (2019), 653–675.

[60] Bo Lin, Shangwen Wang, Kui Liu, Xiaoguang Mao, and Tegawendé F Bissyandé. 2021. Automated Comment Update: How Far are We?. In *International Conference on Program Comprehension*. 36–46.

[61] Shang-Wei Lin, Palina Tolmach, Ye Liu, and Yi Li. 2022. SolSEE: A Source-Level Symbolic Execution Engine for Solidity. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[62] Shuang Liu, Jun Sun, Yang Liu, Yue Zhang, Bimlesh Wadhwa, Jin Song Dong, and Xinyu Wang. 2014. Automatic Early Defects Detection in Use Case Documents. In *International Conference on Automated Software Engineering*. 785–790.

[63] Ye Liu and Yi Li. 2022. InvCon: A Dynamic Invariant Detector for Ethereum Smart Contracts. In *International Conference on Automated Software Engineering*.

[64] Ye Liu, Yi Li, Shang-Wei Lin, and Cyrille Artho. 2022. Finding Permission Bugs in Smart Contracts with Role Mining. In *International Symposium on Software Testing and Analysis*. 716–727.

[65] Ye Liu, Yi Li, Shang-Wei Lin, and Qiang Yan. 2020. ModCon: A Model-Based Testing Platform for Smart Contracts. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1601–1605.

[66] Ye Liu, Yi Li, Shang-Wei Lin, and Rong Zhao. 2020. Towards Automated Verification of Smart Contract Fairness. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 666–677.

[67] Yang Liu, Mingwei Liu, Xin Peng, Christoph Treude, Zhenchang Xing, and Xiaoxin Zhang. 2020. Generating Concept Based API Element Comparison Using a Knowledge Graph. In *International Conference on Automated Software Engineering*. 834–845.

[68] Zhongxin Liu, Xin Xia, David Lo, Meng Yan, and Shanping Li. 2021. Just-In-Time Obsolete Comment Detection and Update. *IEEE Transactions on Software Engineering* (2021).

[69] Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. 2020. Automating Just-In-Time Comment Updating. In *International Conference on Automated Software Engineering*. 585–597.

[70] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *ACM SIGSAC Conference on Computer and Communications Security*. 254–269.

[71] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. 2012. What Should Developers Be Aware of? An Empirical Study on the Directives of API Documentation. *Empirical Software Engineering* 17, 6 (2012), 703–737.

[72] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *International Conference on Automated Software Engineering*. 1186–1189.

[73] Satoshi Nakamoto et al. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. (2008).

[74] Pengyu Nie, Rishabh Rai, Junyi Jessy Li, Sarfraz Khurshid, Raymond J Mooney, and Milos Gligoric. 2019. A Framework for Writing Trigger-Action Todo Comments in Executable Format. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 385–396.

[75] Sebastiano Panichella and Nik Zaugg. 2020. An Empirical Investigation of Relevant Changes and Automation Needs in Modern Code Review. *Empirical Software Engineering* 25, 6 (2020), 4833–4872.

[76] Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J. Mooney. 2021. Deep Just-In-Time Inconsistency Detection Between Comments and Source Code. In *Thirty-Fifth AAAI Conference on Artificial Intelligence*. 427–435.

[77] Inderjot Kaur Ratol and Martin P Robillard. 2017. Detecting Fragile Comments. In *International Conference on Automated Software Engineering*. 112–122.

[78] Mohamed Aymen Saied, Houari Sahraoui, and Bruno Dufour. 2015. An Observational Study on API Usage Constraints and Their Documentation. In *International Conference on Software Analysis, Evolution and Reengineering*. 33–42.

[79] Lin Shi, Hao Zhong, Tao Xie, and Mingshu Li. 2011. An Empirical Study on Evolution of API Documentation. In *International Conference on Fundamental Approaches to Software Engineering*. 416–431.

[80] Solidity 2022. Solidity. https://solidity.readthedocs.io/en/v0.5.1/.

[81] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. 2013. Quality Analysis of Source Code Comments. In *International Conference on Program Comprehension*. 83–92.

[82] Nick Szabo. 1997. Formalizing and Securing Relationships on Public Networks. *First Monday* (1997).

[83] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* iComment: Bugs or Bad Comments? */. In *ACM SIGOPS Symposium on Operating Systems Principles*. 145–158.

[84] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. aComment: Mining Annotations from Comments and Code to Detect Interrupt Related Concurrency Bugs. In *International Conference on Software Engineering*. 11–20.

[85] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. 2012. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *International Conference on Software Testing, Verification and Validation*. 260–269.

[86] Trail of Bits 2019. *Echidna*. Trail of Bits. https://github.com/trailofbits/echidna

[87] Gias Uddin and Martin P Robillard. 2015. How API Documentation Fails. *IEEE Software* 32, 4 (2015), 68–75.

[88] Haijun Wang, Ye Liu, Yi Li, Shang-Wei Lin, Cyrille Artho, Lei Ma, and Yang Liu. 2022. Oracle-Supported Dynamic Exploit Generation for Smart Contracts. *IEEE Transactions on Dependable and Secure Computing* 19, 3 (May 2022), 1795–1809.

[89] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A Large-Scale Empirical Study on Code-Comment Inconsistencies. In *International Conference on Program Comprehension*. 53–64.

[90] Maximilian Wohrer and Uwe Zdun. 2020. From Domain-Specific Language to Code: Smart Contracts and the Application of Design Patterns. *IEEE Software* 37, 5 (2020), 37–42.

[91] Gavin Wood. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum Project Yellow Paper* 151 (2014), 1–32.

[92] Xiuheng Wu, Chenguang Zhu, and Yi Li. 2021. Diffbase: A Differential Factbase for Effective Software Evolution Management. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 503–515.

[93] Valentin Wüstholz and Maria Christakis. 2020. Harvey: A Greybox Fuzzer for Smart Contracts. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1398–1409.

[94] Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. 2020. C2S: Translating Natural Language Comments to Formal Program Specifications. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 25–37.

[95] Juan Zhai, Xiangzhe Xu, Yu Shi, Guanhong Tao, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2020. CPC: Automatically Classifying and Propagating Natural Language Comments via Program Analysis. In *International Conference on Software Engineering*. 1359–1371.

[96] Wuqi Zhang, Lili Wei, Shuqing Li, Yepang Liu, and Shing-Chi Cheung. 2021. ÐArcher: Detecting On-Chain-Off-Chain Synchronization Bugs in Decentralized Applications. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 553–565.

[97] Hao Zhong, Na Meng, Zexuan Li, and Li Jia. 2020. An Empirical Study on API Parameter Rules. In *International Conference on Software Engineering*. 899–911.

[98] Hao Zhong and Zhendong Su. 2013. Detecting API Documentation Errors. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 803–816.

[99] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. 2017. Analyzing APIs Documentation and Code to Detect Directive Defects. In *International Conference on Software Engineering*. 27–37.

[100] Yu Zhou, Changzhi Wang, Xin Yan, Taolue Chen, Sebastiano Panichella, and Harald Gall. 2018. Automatic Detection and Repair Recommendation of Directive Defects in Java API Documentation. *IEEE Transactions on Software Engineering* 46, 9 (2018), 1004–1023.