# AuditGPT: Auditing Smart Contracts with ChatGPT

Shihao Xia
Pennsylvania State University
USA

Shuai Shao
University of Connecticut
USA

Mengting He
Pennsylvania State University
USA

Tingting Yu
University of Connecticut
USA

Linhai Song
Pennsylvania State University
USA

Yiying Zhang
University of California, San Diego
USA

## ABSTRACT

To govern smart contracts running on Ethereum, multiple Ethereum Request for Comment (ERC) standards have been developed, each containing a set of rules to guide the behaviors of smart contracts. Violating the ERC rules could cause serious security issues and financial loss, signifying the importance of verifying smart contracts follow ERCs. Today's practices of such verification are to either manually audit each single contract or use expert-developed, limited-scope program-analysis tools, both of which are far from being effective in identifying ERC rule violations.

This paper presents a tool named *AuditGPT* that leverages large language models (LLMs) to automatically and comprehensively verify ERC rules against smart contracts. To build AuditGPT, we first conduct an empirical study on 222 ERC rules specified in four popular ERCs to understand their content, their security impacts, their specification in natural language, and their implementation in Solidity. Guided by the study, we construct AuditGPT by separating the large, complex auditing process into small, manageable tasks and design prompts specialized for each ERC rule type to enhance LLMs' auditing performance. In the evaluation, AuditGPT successfully pinpoints 418 ERC rule violations and only reports 18 false positives, showcasing its effectiveness and accuracy. Moreover, AuditGPT beats an auditing service provided by security experts in effectiveness, accuracy, and cost, demonstrating its advancement over state-of-the-art smart-contract auditing practices.

## 1 INTRODUCTION

**Ethereum and ERC.** Since the creation of Bitcoin, blockchain technology has evolved significantly. One of the most important developments is Ethereum [26, 61], a decentralized, open-source blockchain platform. Ethereum enables the creation and execution of decentralized applications (DApps) like financial services and smart contracts, which are self-executing agreements with the terms of the contract directly written into code [27, 30].

To govern smart contracts running on Ethereum, a set of formal standards called Request for Comments (*ERCs*) have been developed [54]. For example, the ERC20 standard defines a common set of rules for fungible tokens—tokens (digital assets) that are interchangeable with one another [57]. ERCs play a crucial role in the Ethereum ecosystem by providing a common set of rules and specifications that developers can follow when implementing smart contracts. ERCs ensure interoperability and compatibility between different Ethereum-based projects, wallets, and DApps [27].

**ERC violation.** The violation of an ERC could result in interoperability issues where a violating contract may not work properly with wallets or DApps. ERC violations could also lead to security vulnerabilities and financial loss. Additionally, ERC violations could result in de-listing of tokens from exchanges, as many exchanges have listing requirements of following ERC standards [28]. Figure 1 shows a violation of an ERC20 rule in a real smart contract. The `_balances` field in line 2 monitors the number of tokens held by each address. Function `transferFrom()` in lines 6–10 facilitates the transfer of `amount` tokens from one address to another. ERC20 imposes multiple rules on `transferFrom()`, such as the necessity to fire an `Transfer` event for logging purposes and to treat the transfer of zero tokens the same as transferring other amounts, both of which the contract follows. However, the function `transferFrom()` violates a crucial ERC20 rule that mandates the function to verify whether the caller has the privilege to execute the transfer of `amount` tokens, which ensures financial security. Due to this violation, anyone can steal tokens from any address by invoking `transferFrom()` to transfer tokens to his address. The patch in line 7 illustrates how to fix the violation. The patch uses a two-dimensional map, `_allowances`, to track how many tokens "from" allows "msg.sender" to manipulate. The subtraction operation in this line triggers an exception and the termination of the transaction in case of underflow, thus preventing a caller of the function from transferring tokens if they do not have enough privilege.

```
1  contract ERC20 {
2    mapping(address => uint256) _balances;
3    mapping(address => mapping(address => uint256)) _allowances;
4    event Transfer(address indexed _from, address indexed _to,
       uint256 _value);
5
6    function transferFrom(address from, address to, uint256
       amount) public returns (bool) {
7  +   _allowances[from][msg.sender] -= amount;
8      _transfer(from, to, amount);
9      return true;
10   }
11   function _transfer(address from, address to, uint256 amount)
       internal {
12     require(from != address(0), "transfer from address zero");
13     require(to != address(0), "transfer to address zero");
14     _balances[from] -= amount;
15     _balances[to] += amount;
16     emit Transfer(from, to, amount);
17   }
18 }
```

**Figure 1: An ERC20 rule violation that can be exploited to steal tokens.** ***(Code simplified for illustration purpose.)***

**State of the art.** Despite the importance of following ERC rules, it is hard for developers to do so, as they need a comprehensive understanding of all ERC requirements and their contract code. ERC standards include a huge number of rules. For just the four ERC standards that we study, there are 222 rules involved. A simple operation could involve multiple rules. For example, the `transferFrom()` function in Figure 1 involves six rules from ERC20, one of which was overlooked by the programmers causing the issue described

above. To make matters worse, these rules are described in different manners, with some being code declarations and others being natural languages.

Meanwhile, contract implementations are often complex as well. One contract and its dependent code usually contain hundreds to thousands of lines of source code in multiple files. Some code details may be obscured within intricate caller-callee relationships, while others may involve numerous objects and functionalities possibly written by different programmers. All these complexities in ERC rules and smart contracts make it extremely hard for programmers to manually check for ERC violations. As a result, ERC rule violations widely exist in real-world smart contracts [18].

Today's practices to avoid ERC violations are on two fronts. First, to avoid programmers' manual checking efforts, there have been efforts to develop program-analysis tools to automatically verify certain criteria of smart contracts [19, 46]. These tools are limited by the types of verification they support. They only verify whether interfaces in smart contracts meet basic requirements, such as the declaration of all functions required by the corresponding ERC. These tools cannot detect complex requirements like the violated rule in Figure 1. A fundamental reason why these and potentially future program-analysis tools cannot cover a wide range of verifications is that many ERC rules involve semantic information and require customization for individual contracts—a process that is time-consuming if not unfeasible. Second, there are several auditing services provided by security experts [2, 6, 12, 18, 36, 47, 52]. Although auditing services are more thorough and comprehensive than program-analysis tools, they usually involve vast amounts of manual auditing efforts behind the scenes. Thus, these services are often costly and involve lengthy auditing periods. Overall, auditing services are only suitable for limited users and use cases.

*Are there any cheaper, automated, and thorough ERC rule verification methods?*

**Our proposal.** This paper answers this research question in the affirmative by building *AuditGPT*, a tool that leverages large language models (LLMs) to automatically audit smart contracts for ERC compliance. LLMs like ChatGPT have seen tremendous success in recent years and have been used in programming-language-related tasks like bug finding [56], malware detection [53], and program repairing [35, 66]. However, as far as we know, there has been no exploration of LLMs in rule auditing, such as ERC compliance auditing. The use of LLMs in ERC auditing introduces several new challenges. First, there are many ERC rules with different natures and specified in different natural languages. It is unclear how to represent them effectively to an LLM. Second, most ERC rules are specific to program semantics. Thus, we need a good representation of smart-contract code to an LLM. Third, LLMs largely work as black boxes during the auditing process, and their effectiveness largely depends on the prompts employed.

To confront these challenges, we first conduct an empirical study of ERC standards to gain insights into ERC rules. Specifically, we examine four popular ERC standards and the 222 rules they encompass. Our study aims to understand what the rules are about, the security impacts of violating them, and how they are articulated in natural language within the ERCs and implemented in smart contracts. Our study yields five insights that can benefit Solidity programmers, security experts, and ERC protocol designers. For example, we observe that approximately one-fifth of ERC rules focus on verifying whether an operator, token owner, or token recipient possesses adequate privileges to execute a specific operation. Violating these rules can result in a clear attack path, leading to financial loss (*e.g.*, Figure 1). Furthermore, we discover that most ERC rules can be validated within a single function. Additionally, we identify linguistic patterns used to express rules in ERCs and notice a correlation between the natural language specification of a rule and its implementation.

Second, we design and implement AuditGPT based on three principles. *Divide and conquer*: We split the auditing process into a startup phase for automatically extracting rules from ERCs and a working phase for inspecting individual contracts. Additionally, we divide large contracts into small code segments and instruct LLMs to inspect each segment individually against a specific rule to focus their attention. *Guided by our study*: We separate each individual contract based on its public functions. We leverage identified linguistic patterns as one-shot examples during ERC rule extraction. *Specialization*: We create specialized questions for each type of rules and automate the process from ERC rule extraction to generating specialized prompts using our designed informative YAML format. Furthermore, we craft one-shot Solidity code examples for specific rules to enhance LLMs' auditing effectiveness. The mechanisms employed in AuditGPT can serve as inspiration for future researchers exploring the application of LLMs in other programming-language-related tasks.

We construct two datasets to assess AuditGPT: a large dataset comprising 200 contracts randomly selected from etherscan.io [31] and polygonscan.com [48], and another dataset consisting of 30 contracts with ground-truth violation labeling (performed by us) and human-auditing reports crafted by security experts at the Ethereum Commonwealth Security Department (ECSD) [18]. AuditGPT identifies 279 ERC rule violations from contracts in the large dataset, with four violations exhibiting a clear attack path leading to potential financial losses (one is shown in Figrue 1). Additionally, AuditGPT reports only 15 false positives. Overall, AuditGPT is effective and accurate in auditing smart contracts and identifying ERC rule violations. We compare AuditGPT with an automated program analysis technique [19] and the human auditing service [18] with the small dataset. AuditGPT detects 50% more violations than the two baseline solutions. Furthermore, AuditGPT reduces time and monetary costs by a thousand fold compared with the human auditing service.

In sum, we make the following contributions.

- We conduct the first empirical study on Ethereum ERC rules made for smart-contract implementations.
- We design and implement AuditGPT to audit smart contracts and pinpoint ERC rule violations.
- We conduct thorough experiments to assess AuditGPT and confirm its effectiveness, accuracy, and advancement.

## 2 BACKGROUND

This section gives the background of the project, including Solidity smart contracts, ERCs, and existing techniques related to ours.

## 2.1 Ethereum and Solidity Smart Contracts

Ethereum is a blockchain system that enables programmers to create and deploy smart contracts for the development of decentralized applications [26, 61]. Both Ethereum users and smart contracts are represented by distinct Ethereum addresses, which can be used to send and receive Ethers (the native Ethereum cryptocurrency) and interact with smart contracts, thereby leveraging their functionalities for conducting complex transactions. Ethereum cultivates a thriving digital economy ecosystem. At the time of writing, the price of one Ether is over $2K, with the total market value of all Ethers exceeding $200B [7]. Daily transactions on Ethereum surpass one million, with a volume exceeding $4B [4]. Smart contracts play a pivotal role in Ethereum's success, as they guide the majority of transactions and enable crucial functionalities [21, 25, 57]

Solidity stands out as the most widely used programming language for writing smart contracts [13, 42]. With a syntax resembling ECMAScript [24], Solidity effectively conceals the intricacies of the Ethereum blockchain system. Implementing a contract in Solidity is similar to implementing a class in Java. A contract contains contract fields (state variables) to store the contract's states and functions to realize its functionalities. A function in Solidity can be public, internal, or private. Public functions serve as the contract's interface, providing external access to its functionalities. These functions can be invoked by a different contract or an Ethereum user through a message call, while private or internal functions cannot. Additionally, contracts can define events emitted during execution, serving as logs on-chain that can be analyzed by off-chain applications.

An example of a contract is presented in Figure 1. The contract has two contract fields in lines 2 and 3, and defines event `Transfer()` in line 4 and emits it in line 16. The public function `transferFrom()` (lines 6–10) can be called by any Ethereum user or contract after the contract is deployed, while the internal function `_transfer()` (lines 11–17) is restricted to calls from the same address.

## 2.2 Ethereum Request for Comment (ERC)

ERCs serve as technical documents that outline the requirements for implementing smart contracts, ensuring interoperability and compatibility across various contracts, applications, and platforms and helping foster the Ethereum ecosystem [28, 29, 55]. Typically, an ERC begins with a concise motivation. For example, ERC20 emphasizes its role in defining a standard token interface for tokens to be manipulated by applications like wallets and decentralized exchanges [57]. Subsequently, an ERC details all necessary public functions and events, specifying their parameters, return values, and optional attributes for the parameters. Additionally, an ERC typically outlines requirements through plain text or code comments for each function or event around its declaration. For instance, beyond the requirements for the function API and return value generation, ERC20 incorporates four additional rules for `transferFrom()` (*e.g.*, Figure 1): mandating the emission of a `Transfer` event, verifying whether the message sender is approved to manipulate the token owner's tokens and throwing an exception if not, treating the transfer of zero tokens similarly to other amounts, and emitting an event when transferring zero tokens.

Violating ERC rules can result in substantial financial losses and unexpected contract behaviors. For instance, ERC721 mandates `onERC721Received()` to be called for each token transfer, when the recipient is a contract. Additionally, it further requires that the caller must verify that the return is a specific magic number. These two rules ensure that the recipient contract has the capability to handle the transferred tokens. Transferring tokens to a contract lacking this capability can result in the tokens being permanently trapped in the recipient contract. This issue was initially reported in 2017 on Ethereum Reddit, leading to a loss of $10,000 worth of tokens at that time, and has since caused millions of dollars in losses [22]. As another example, the rule violation in Figure 1 opens the door for a hacker to pilfer tokens from any account. In summary, it is crucial to ensure that contracts adhere to ERC rules to safeguard financial assets and ensure the proper functionality of the contracts.

## 2.3 Related work

There are tools designed to automatically pinpoint ERC rule violations. Slither contains specific checkers (*i.e.*, slither-check-erc [19]) that scrutinize whether a given contract adheres to the corresponding ERC requirements for 11 ERCs. However, these checkers have limited functionalities; they primarily focus on verifying the presence of required functions and events, ensuring that the declarations of these functions and events align with the specified requirements, and confirming that functions emit the necessary events. Unfortunately, these checkers lack the capability to inspect more advanced requirements, such as determining if the message sender has the necessary privilege to transfer a token, and thus, they miss complex violations, that can be detected by AuditGPT (see Section 5.2). ERC20 verifier is a tool dedicated to ERC20 contracts [46]. Its assessments are similar to the checks performed by Slither.

Researchers have developed automated tools for identifying various Solidity bugs, including reentrancy bugs [40, 49, 68], nondeterministic payment bugs [39, 58], consensus bugs [17, 69], eclipse attacks [41, 65, 67], out-of-gas attacks [33, 34], and code snippets consuming unnecessary gas [9, 10, 14–16, 20, 37, 43]. Unfortunately, these techniques focus on bugs that are unrelated to program semantics and may manifest in any Solidity or Ethereum contract. Consequently, they are unable to identify ERC rule violations like AuditGPT does, as rule violations stem from the failure to meet specific semantic requirements outlined by individual ERCs.

Security experts offer auditing services to identify security vulnerabilities or logic flaws in Solidity contracts [2, 6, 12, 18, 36, 47, 52]. Some of these services also assess ERC compliance. However, these auditing services come with considerably higher financial cost compared to automated tools, and Solidity programmers may be hesitant to utilize them. Furthermore, manual contract auditing is a time-consuming process and it is challenging for it to handle the amount of contracts deployed on Ethereum daily.

Researchers have already utilized LLMs provided by ChatGPT for analyzing Solidity code, such as detecting vulnerabilities [56], patching bugs [35], and testing Solidity programs [1]. However, the problem we address differs from existing techniques. Given that ERCs specify a multitude of implementation rules and most of them

are related to program semantics, we essentially resolve bugs in a much wider array of types than existing techniques.

## 3 EMPIRICAL STUDY ON ERC RULES

This section presents our empirical study on implementation rules specified in the ERC documents, including the methodology employed for the study and the categories established for the rules.

### 3.1 Methodology

From the 84 ERCs in the final status, we choose ERC20, ERC721, ERC1155, and ERC3525 as our study targets. We base our selection on several criteria, including their popularity with numerous corresponding contracts, their complexity with various requirements, and their significance in the Ethereum ecosystem.

*ERC20* is a technical standard for fungible tokens (*e.g.*, cryptocurrencies) and is probably the most famous ERC standard. It outlines operational requirements for minting, burning, and transferring tokens [57]. Presently, there are over 450,000 ERC20 tokens on the Ethereum platform [8], with many boasting a market capitalization surpassing $1 billion (*e.g.*, USDT [64], SHIB [63], Binance USD [60]).

*ERC721* is designed for non-fungible tokens (NFTs), where each token is distinct and indivisible [25]. ERC721 specifies how ownership of NFTs is managed and stands as the most popular NFT standard. It is adhered to by major NFT marketplaces [45, 51].

*ERC1155* aims to enable a single contract to oversee both fungible and non-fungible tokens [50]. Additionally, it facilitates batch operations. For example, multiple types of tokens can be transferred from one address to another together. ERC1155 has found adoption in various gaming and charity donation projects [3, 5, 62].

*ERC3525* is tailored for semi-fungible tokens, where each token is distinct like NFTs but incorporates an additional qualitative nature like fungible tokens [59]. ERC3525 has already been leveraged for financial instruments [11, 32].

We carefully review the official documents (including text descriptions and associated code) and manually identify rules by evaluating their relevance to contract implementations, whether they have clear restricting targets, and whether they offer actionable checking criteria. To maintain objectivity, all extracted rules are examined by at least two paper authors. Certain rules are extracted from the textual descriptions, with some explicitly using terms like "must" or "should" to convey obligations. The remaining rules are gained from code sections pertaining to function and event declarations. In total, we identified 222 rules from the four ERCs.

Our study primarily answers three key questions regarding the identified rules: 1) what rules are specified? 2) why are they specified? and 3) how are they specified in the ERCs and implemented in contracts? The objective is to garner insights for building techniques to automatically detect rule violations. All study results are carefully reviewed by at least two paper authors.

### 3.2 Rule Content (What)

We first inspect the content of the 222 ERC rules to understand what checks we need to detect their violations. As shown by the rows in Table 1, we separate the ERC rules into four categories.

**Table 1: ERC rules' content and security impacts.**

| content \ impact | High | Medium | Low | Total |
|---|---|---|---|---|
| **Privilege Check** | 46 | 0 | 0 | 46 |
| **Functionality** | 22 | 40 | 0 | 62 |
| **Usage** | 0 | 56 | 0 | 56 |
| **Logging** | 0 | 0 | 58 | 58 |
| **Total** | 68 | 96 | 58 | 222 |

*Privilege Checks.* 46 rules delineate the necessary privileges for executing specific token operations. Among them, 14 pertain to verifying if the operator (*e.g.*, a message caller) possesses the required privilege. For example, the implementation in Figure 1 violates an ERC20 rule about `transferFrom()`, which stipulates that the implementation must verify the message caller is authorized by the token owner to transfer the `amount` tokens. Additionally, 13 rules address whether the token owner holds sufficient privilege. For example, function `transfer(address to, uint256 value)()` in ERC20 sends `value` tokens from the message caller (the token owner) to recipient `to`, and ERC20 mandates the message caller has enough tokens. The remaining 19 rules focus on the token recipient when the operation is a transfer. For instance, ERC721 specifies the recipient cannot be address zero for `safeTransferFrom()`. Moreover, it also mandates calling `onERC721Received()` on the recipient when transferring tokens to a contract. It further requires the caller to check whether the return value of `onERC721Received()` is a magic number and mandates the caller to throw an exception if not. Both ERC1155 and ERC3525 have similar rules.

*Functionality Requirements.* Among the 62 rules governing code implementation, 43 rules specify how to generate the return value for a function. For example, ERC1155 mandates that `balanceOf(address _owner, uint256 _id)` should return the amount of tokens of type `_id` owned by `_owner`. In particular, when an ERC defines a function with a Boolean return, we interpret it as an implicit requirement for the function to return `true` upon successful completion and `false` otherwise. Ten rules focus on managing input parameters. For instance, ERC20 dictates that `transferFrom()` should treat scenarios where zero tokens are transferred the same way as cases when non-zero tokens are transferred. The implementation shown in Figure 1 adheres to this rule. Four rules explicitly mandate the associated function to throw an exception when any error occurs or the recipient rejects a transfer. Four rules specify how to update particular variables. For example, one ERC1155 rule for transferring multiple tokens together requires the balance update for each input token type to follow their order in the input array. The last rule is mandated by ERC3525, which allows a contract to manage multiple types of fungible tokens, where tokens within the same slot belong to the same type. This rule requires that when transferring tokens, the slot of the recipient that receives the tokens must be the same as the slot the sender uses.

*Code Usage.* 56 rules are about how a piece of code is used. For example, the four ERCs mandate the declaration of 52 functions, as users may interact with contracts following the interfaces outlines in the ERCs after the contracts are deployed. ERC20 specifies that three functions are optional and necessitates checks for their existence before invoking those functions. Additionally, ERC20 requires that

if a function returns a Boolean value, the caller of the function must inspect the return and cannot assume the call always succeeds.

*Logging.* ERCs impose logging requirements by emitting events. For example, ERC20 mandates the emission of a `Transfer` event when a transfer operation occurs (*e.g.*, line 14 in Figure 1). In total, there are 58 rules related to logging. Out of these, 32 specify when an event should be emitted, while the remaining 26 pertain to event declarations, including 15 rules specifically mandating attributes of event parameters.

**Insight 1:** *Given that ERC rules primarily involve contract semantics, constructing program analysis techniques for identifying ERC rule violations across diverse contracts poses a significant challenge.*

We further study the valid scope for each rule. The valid scopes of 200 rules are confined to a single function. For instance, ERC20 mandates that `transferFrom()` in Figure 1 scrutinizes whether the message caller possesses the privilege to handle the token owner's tokens. As another example, 43 rules concern how a function generates its return value. Moreover, 11 rules pertain to event declarations. In the remaining 11 cases, their valid scopes encompass the entire contract. For instance, ERC20 necessitates emitting a `Transfer` event whenever a token transfer occurs. Similarly, ERC721 requires calling `onERC721Received()` on the recipient contract for each token transfer, along with checking the return value of the function.

**Insight 2:** *Most ERC rules can be checked within a limited scope (e.g., a function, an event declaration site), and there is no need to analyze the entire contract for compliance with these rules.*

## 3.3 Violation Impact (Why)

We analyze the security implications of rule violations to understand the reasons behind the rule specifications. As shown by the columns Table 1, we categorize the rules' impacts into three levels.

*High.* A rule is deemed to have a high-security impact if there exists a clear attack path exploiting its violation, resulting in financial loss. As shown in Table 1, 68 rules fall into this category, encompassing all rules related to privilege checks. For instance, the failure to verify an operator's privilege can enable a hacker to pilfer tokens (*e.g.*, Figure 1) and neglect to inspect whether a recipient address is non-zero can lead to tokens being lost permanently. Concerning rules associated with the implementation of specific functions, non-compliance with 22 of them can also result in financial loss. Among these, 18 rules outline how to generate return values representing token ownership, such as `balanceOf()` returning the number of tokens belonging to an address and `ownerOf()` designating the owner address of the input token. Errors that fail to provide accurate returns for these functions can lead to scenarios where a valid token is trapped in an address, or an address sends out tokens not belonging to it. The remaining four are related to properly updating tokens' ownership. For example, ERC3525 mandates that the receiver's slot of transferred tokens must match the sender's. As each slot in ERC3525 represents a type of tokens, violating this rule can cause tokens with a high value to transform into tokens with a low value after a transfer. ERC721, ERC1155, and ERC3525 all require inspection of whether an address has the capability to handle received tokens during a token transfer to prevent tokens from becoming trapped in the recipient address. These rules pertain to code usage and also have a high impact.

**Table 2: Linguistic Patterns.** ***([*]: a parameter in a linguistic pattern, and {*}: an optional parameter. P: privilege checks, F: functionality requirements, U: code usage, and L: logging. [subject] could be a function or an event. [must] could be "must", "must not", and "should". [action] could be "handle", "fire", and "throw". [assign] could be "be the" and "be set to". [role] could be "an authorized operator". )***

| ID | Patterns | Content | | | | Total |
|---|---|---|---|---|---|---|
| | | P | F | U | L | |
| CP1 | [subject] [must] [action] {condition} | 17 | 7 | 4 | 0 | 28 |
| CP2 | [action] [must] result in revert | 3 | 0 | 0 | 0 | 3 |
| CP3 | Caller must be approved to [action] | 2 | 0 | 0 | 0 | 2 |
| CP4 | [must] revert [condition] | 14 | 4 | 0 | 0 | 18 |
| CP5 | Caller [must] be [role] | 2 | 0 | 0 | 0 | 2 |
| CP6 | [action] is considered invalid | 0 | 1 | 0 | 0 | 1 |
| CP7 | [condition] [subject] [must] call [function] | 8 | 0 | 0 | 0 | 8 |
| EP1 | [must] [action] [event] {condition} | 0 | 0 | 0 | 6 | 6 |
| EP2 | [event] emits {condition} | 0 | 0 | 0 | 25 | 25 |
| EP3 | {condition} without emitting [event] | 0 | 0 | 0 | 1 | 1 |
| RP1 | return | 0 | 15 | 0 | 0 | 15 |
| RP2 | @return/@notice | 0 | 28 | 0 | 0 | 28 |
| AP1 | [subject] [must] [assign] | 0 | 7 | 0 | 15 | 22 |
| **Total** | | 46 | 62 | 4 | 47 | 159 |

*Medium.* A rule is deemed to have a medium impact if its violation can lead to unexpected contract or transaction behavior, while not having a clear attack path to causing financial loss. For example, if a public function's API fails to adhere to its ERC declaration requirement, invoking the function with a message call following the requirement would trigger an exception. Another example is ERC20's requirement that the function `transferFrom()` (*e.g.*, Figure 1) treats the transfer of zero tokens the same way as transferring non-zero values. If a contract does not adhere to this rule, its behavior would be unexpected for the message caller.

*Low.* All event-related rules are about logging. We consider their security impact as low.

**Insight 3:** *For numerous rules, their violations present a clear attack path for potential financial loss, emphasizing the urgency of detecting and addressing these violations.*

## 3.4 Specification and Implementation (How)

Among the 222 rules, 63 pertain to function or event declarations, and they are precisely outlined by providing the correct declaration using Solidity code. The remaining 159 rules are articulated in natural language texts. We have identified 12 linguistic patterns to categorize how these rules are presented. As shown in Table 2, four patterns encompass more than 20 rules each. CP1 and RP2 are the most widely employed patterns, both covering 28 rules. CP1 is primarily utilized for conducting privilege checks. For instance, ERC20 specifies the violated rule in Figure 1 as "the function SHOULD throw unless the _from account has deliberately authorized the sender of the message via some mechanism." RP2 is employed to delineate return values. Conversely, there are patterns covering a very limited number of rules; CP6 and EP3 are each associated with only one rule.

**Insight 4:** *The majority of rules can be specified using common linguistic patterns, while there are a few rules that are specifically outlined in natural language in a distinct manner.*

We further categorize the 12 patterns into six groups based on how these rules are implemented in Solidity. Patterns with IDs sharing the same prefix are grouped together in Table 2. Group CP encompasses seven linguistic patterns where rule implementations involve a condition check followed by the execution (or non-execution) of an action if the check passes. This condition check may be explicitly implemented using an `if` or a `require` statement, or it could be implemented implicitly. For instance, the check required by the violated rule in Figure 1 can be performed using a subtraction operation, as illustrated by line 7. Group EP pertains to rules related to emitting or not emitting events, with implementations or violations involving the keyword `emit`. Similarly, group RP involves rules where implementations revolve around `return`, and group AP deals with rules where implementations involve updating field values.

**Insight 5:** *How a rule should be implemented usually correlates with how the rule is specified in the ERC.*

# 4 AUDITGPT DESIGN

AuditGPT utilizes an LLM to verify whether a contract implementation adheres to ERC requirements. Consequently, the design of AuditGPT primarily centers around creating appropriate LLM prompts, including how to specify ERC requirements in prompts, how to supply contract code to the LLM, and strategies to enhance the effectiveness and accuracy of violation detection. This section commences with an overview of AuditGPT, followed by a detailed presentation of the three design aspects of prompt construction.

## 4.1 Overview

AuditGPT takes a startup phase to analyze ERCs and a subsequent working phase dedicated to inspecting individual contracts.

We observe that an LLM is more effective when assessing the satisfaction of individual rules rather than concurrently checking multiple rules. Consequently, we adopt a rule sequentialization policy, wherein we iteratively guide the LLM through all rules in multiple prompts, checking only one rule in each prompt. To facilitate this process, during the startup phase, AuditGPT employs the LLM to extract rules from an ERC and express them in a specific YAML format to streamline prompt construction during the working phase. AuditGPT achieves this by first instructing the LLM to enumerate all public APIs of an ERC and subsequently requesting the LLM to list rules for each API. For example, AuditGPT identifies nine functions and two events for ERC20 and extracts five rules for the function `transferFrom()` in Figure 1.

Similarly, we notice AuditGPT exhibits better performance when scrutinizing smaller code segments separately compared to assessing the entire contract implementation. Consequently, during the working phase, AuditGPT individually inspects each public function. For every public function, AuditGPT apply code slicing to compute its associated code and then employs individual prompts to inquire whether the public function, along with its related code, adheres to each rule extracted for the function during the startup phase. For instance, AuditGPT generates five prompts to ascertain whether function `transferFrom()` in Figure 1 satisfies the five rules. Moreover, AuditGPT employs mechanisms like prompt specialization and one shot [23] to enhance its effectiveness. For example, after employing these mechanisms, an additional 4 prompts are generated for `transferFrom()` in Figure 1.

## 4.2 ERC Rule Extraction

AuditGPT follows a three-step process to extract rules from an ERC and stores them in a YAML file.

First, AuditGPT supplies the ERC to its LLM and instructs it to enumerate all functions in the ERC, presenting the information in a YAML array. Each element of the array possesses three properties: the function name, a list containing all the function's parameter names and their corresponding types, and the return type. After that, AuditGPT instructs the ERC to extract all event declarations from the ERC using a similar approach.

Second, AuditGPT iterates through each extracted function to extract rules for them individually. To prevent scenarios where a function contains numerous rules, and the LLM overlooks some when using only one prompt, AuditGPT employs multiple prompts for each function to extract different types of rules. As AuditGPT utilizes these extracted rules to analyze contract implementations, it selects implementation categories in Table 2. Specifically, AuditGPT employs four prompts for each function, corresponding to the four pattern groups. In each prompt, AuditGPT provides the entire ERC document, the function's declaration, and a concise explanation of the linguistic group. Additionally, AuditGPT presents all identified patterns for the group as one-shot examples in the prompt. Moreover, AuditGPT instructs the LLM to present the extracted rule in a YAML format tailored to each group. For the CP group, AuditGPT instructs the LLM to extract the condition, the condition type, and the action. For the EP group, AuditGPT directs the LLM to extract the condition and the event name. For the RP group, AuditGPT specifies that the LLM should extract the method for generating the return. Lastly, for the AP group, AuditGPT instructs the LLM to extract the action.

Take the rule violated in Figure 1 as an example, the extracted condition is " the _from account has deliberately authorized the sender of the message via some mechanism", the condition type is "unless", and the action is "throw".

Third, AuditGPT examines each extracted event and extracts rules regarding when it should or should not be emitted, using a prompt similar to those designed for extracting rules in a linguistic pattern within the EP group from a function.

Following this process, AuditGPT successfully extracts 212 rules out of the 222 rules, misses 10 rules, and mistakenly extracts four extra rules. We manually correct the errors that occurred during rule extraction to set up AuditGPT for subsequent violation detection. We emphasize the irreplaceable value of automated rule extraction due to two main reasons: 1) The LLM efficiently processes and condenses ERC documents into a concise format, and manual efforts are inherently constrained. 2) Extracted rules are reusable across all smart contracts implementing the ERC, reducing manual efforts to a one-time occurrence.

### 4.3 Code Slicing

A contract file might be excessively large, surpassing the input token limit of the LLM, making it impractical to input the entire file in a single prompt. Even if a contract file can fit within a prompt, supplying an extensive amount of code to the LLM could result in violations being buried in the input code, complicating the identification of the violations. Additionally, there is a risk that even for detected violations, the LLM may not report them due to the word limit of output. Consequently, AuditGPT divides each input contract file into smaller pieces and analyzes them individually.

We separate each contract based on its `public` functions as defined in the corresponding ERC for two reasons. First, after deployment, interactions with the contract occur through its `public` functions, and rule violations within the contract are exploited through those functions. Second, our study finds that, for the majority of rules, their scopes are confined to a `public` function (Insight 2), and even for those rules whose scopes are the entire contract, they can be verified by examining all `public` functions.

For each `public` function, AuditGPT calculates its associated code and mandates the LLM to analyze the related code together with the function. This approach is necessary because understanding the semantics of a function may be impractical solely by reading its code. We define a function's direct or indirect callees as its related code. Additionally, we consider contract fields accessed by the function or any of its callees as part of the related code. To distinguish functions defined in different contracts, we include the contract declaration of the `public` function or any of its callees. Lastly, given the LLM's proficiency in understanding natural languages, we incorporate comments associated with the function and its callees, both preceding and inside the function and its callees.

Taking the contract in Figure 1 as an example, the entire contract contains 127 lines of code in total. We conduct a separate analysis of its nine `public` functions. When analyzing `transferFrom()`, we consider `_transfer()` as its related code since it is called by `transferFrom()`. We also include line 2 as related code, as this field is accessed by `_transfer()` in lines 14 and 15. Conversely, we do not consider line 3 as related code since the field is not accessed by `transferFrom()` and its callee. Furthermore, we incorporate the contract declarations in lines 1 and 18. Ultimately, the code analyzed by AuditGPT for `transferFrom()` contains 26 lines.

### 4.4 Optimization Mechanisms

We have developed three mechanisms to enhance the LLM's understanding of ERC rules and rule violations.

*Prompt Specialization* Rather than issuing general inquiries to the LLM to determine if the input code violates a given rule, we craft specific questions tailored to the information stored in YAML format for each rule. This approach enhances the LLM's understanding of the rule, thereby improving effectiveness. For example, utilizing the information extracted for the violated rule in Figure 1, we instruct the LLM to examine whether each transferFrom() function throws unless "the _from account has deliberately authorized the sender of the message via some mechanism."

*One shot.* To enhance the LLM's understanding of certain rule descriptions and Solidity program semantics, we employ one-shot learning. Particularly we incorporate a single example in the prompts for 36 rules. These examples are specifically designed for the rules and remain the same across various contracts.

The rules with a one-shot example can be categorized into three scenarios. First, we employ an example to elucidate a specific Solidity language feature for 30 rules. For instance, post version 0.5, Solidity introduced keyword `emit` for event emission. Prior to version 0.5, events were emitted akin to function calls. Consequently, we demonstrate how to emit events before version 0.5 for rules necessitating event emission. Second, we utilize examples to guide the LLM's understanding of implementing specific semantics for five rules. For example, ERC20 mandates the emission of a `Transfer` event for "initial token distribution." We include an example of an ERC20 contractor constructor to aid the LLM's comprehension. Additionally, we devise examples for rules requiring a function to return `false` when it fails to complete its functionality, preventing the LLM from erroneously equating throwing an exception with returning `false`. Third, the remaining rule contains obscure descriptions. ERC20 stipulates that `transferFrom()` should verify whether the message sender has been authorized "via some mechanism." We construct an example to illustrate that this mechanism should utilize the `_allowances` field, like line 10 of Figure 1.

*Breaking down compound rules.* There are 24 rules structured in a way that dictates an action should be taken when a condition is met. One such rule is mandated by ERC20, specifying that a caller must verify the return value is equal to `false`, when a function returns a Boolean value. The remaining rules dictate that when a certain condition is met, a specific event must be emitted.

When directly querying the LLM about whether a function violates one of those compound rules, the LLM often interprets the absence of the prerequisite as a rule violation, resulting in a false positive. To address this, we break down each compound rule into two prompts. The former prompt prompts the LLM to assess the existence of the condition. If the LLM confirms its presence, then AuditGPT sends to latter prompt to instruct the LLM to examine whether the action exists. AuditGPT reports a rule violation when the LLM considers the action is absent.

## 5 EVALUATION

***Implementation.*** We employ GPT-4 Turbo [44] as the LLM in AuditGPT and interact with it using OpenAI's APIs. All other functionalities are implemented in Python, covering tasks such as automatically creating prompts, sending prompts to the LLM, and parsing the LLM's responses. Notably, when constructing prompts, we require the LLM to inspect functions individually, rather than analyzing the entire Solidity contract. To achieve this, for each function, we perform static program analysis to compute both direct and indirect callees, contract fields referenced by the function or any of its callees, and contracts declaring the function and its callees. The results of the static analysis are then mapped back to the Solidity source code with the line-number information. Relevant source code lines and comments are extracted to form the prompts. The static analysis is conducted using the slither framework [20].

***Benchmarks.*** We construct two datasets for the evaluation. The first one is a *large* dataset with 200 contracts in total, including 100 ERC20 contracts, 50 ERC721 contracts, and 50 ERC1155 contracts. We randomly collect ERC20 contracts from etherscan.io [31]

**Table 3: Evaluation results on the large dataset.** ***$((x, y)$: $x$ true positives, and $y$ false positives.)***

| | High | Medium | Low | Total |
|---|---|---|---|---|
| **ERC20** | (1, 0) | (97, 1) | (29, 8) | (127, 9) |
| **ERC721** | (0, 1) | (3, 3) | (109, 1) | (112, 5) |
| **ERC1155** | (3, 0) | (12, 0) | (25, 1) | (40, 1) |
| **Total** | (4, 1) | (112, 4) | (163, 10) | (279, 15) |

and ERC721 and ERC1155 contracts from polygonscan.com [48]. These two platforms are the most popular analytics platforms for Ethereum and its sidechain Polygon [38], respectively. In the contract sampling process, we specifically target contracts where both the contract itself and its associated Solidity code (*e.g.*, libraries, inherited contracts) are contained in a single file to simplify the following experiments. On average, each contract file contains 847.7 lines of Solidity source code. Since the number of contract files is large, we do not manually analyze these contracts. We only inspect the results after applying AuditGPT on them.

The second dataset is a *ground-truth* dataset with 30 ERC20 contracts. All these contracts undergo manual auditing by the Ethereum Commonwealth Security Department, a process where Solidity programmers submit auditing requests through filing an issue on GitHub, and the department subsequently provides auditing results by responding to the issue [18]. To form this dataset, we select the most recent 30 auditing requests meeting the following criteria: 1) providing the Solidity source code, 2) approved by the Solidity programmers indicated by the "approved" tag, 3) containing ERC rule violations, and 4) having the contract and the Solidity code used by the contract in the same contract file. On average, each contract file contains 260.9 lines of Solidity source code. We carefully inspect these contracts and identify 142 ERC rule violations. Among them, 21 violations have a high-security impact, 60 have a medium-security impact, and 61 have a low-security impact.

***Baseline Techniques.*** We choose two baselines. The first one is the auditing service provided by Ethereum Commonwealth Security Department (ECSD). This is the only service whose auditing results are available to us. We compare AuditGPT's results with the auditing results provided by their Solidity security experts. The second baseline is slither-check-erc (SCE). As discussed in Section 2.3, it is the most powerful open-source technique to validate ERC conformance, and it covers all ERCs involved in our benchmarks.

***Research Questions.*** Our experiments are designed to answer the following research questions: 1) *Effectiveness*: can AuditGPT accurately pinpoint ERC rule violations? 2) *Advancement*: does AuditGPT perform better than existing auditing solutions? 3) *Necessity*: how does each component of AuditGPT contribute to its auditing capability?

***Experimental Setting.*** We configure the temperature parameter to zero to make our experimental results stable when interacting with the LLM. All our experiments are performed on a desktop machine, with Intel(R) Core(TM) i7-10700 CPU, 64GB RAM, and Ubuntu 22.04 OS version.

## 5.1 Effectiveness of AuditGPT

***Methodology.*** We execute AuditGPT on the large dataset and count violations and false positives reported by AuditGPT to gauge its effectiveness. For each violation flagged by AuditGPT, we manually examine the description generated by the LLM and the corresponding smart-contract code to determine whether AuditGPT accurately identifies a true violation or if it reports a false alarm. Each reported violation is analyzed by at least two paper authors. Any disagreements are resolved through multiple rounds of discussion. As the total number of violations in the large dataset is unknown to us, we do not assess false negatives in this experiment.

```
1   contract MyERC1155Token {
2     mapping(uint256 => mapping(address => uint256)) _balances;
3     mapping(address => mapping(address => bool)) _opApprovals;
4
5     function safeTransferFrom(address from, address to, uint256
        id, uint256 value, bytes memory data) public {
6   +   require(_opApprovals[from][msg.sender], "the caller is not
        approved to manage the tokens");
7       _update(from, to, id, value);
8   +   if (to.isContract()) {
9   +       try IERC1155Receiver(to).onERC1155Received(msg.sender,
        from, id, amount, data) returns (bytes4 response) { ... }
10  +   }
11    }
12    function _update(address from, address to, uint256 id,
        uint256 value) internal {
13      if (from != address(0)) {
14        _balances[id][from] -= value;
15      }
16      if (to != address(0)) {
17        _balances[id][to] += value;
18      }
19    }
20  }
```

**Figure 2: An ERC1155 rule violation with a high-security impact.** ***(Code simplified for illustration purpose.)***

***Effectiveness Results.*** As shown in Table 3, AuditGPT detects 279 ERC rule violations, including four with a high-security impact, 112 with a medium-security impact, and 163 with a low-security impact. Additionally, AuditGPT reports 15 false positives.

AuditGPT identifies four violations with a high-security impact. One of the violations is pinpointed from an ERC20 contract as shown in Figure 1, while the other three are associated with two ERC1155 contracts. Figure 2 illustrates one of the ERC1155 contracts. The function `safeTransferFrom()` in lines 5–8 is designed to transfer `value` amounts of tokens of a specific type (indicated by parameter `id`) from one address to another. This process involves decreasing the balance of `from` in line 11 and increasing the balance of `to` in line 14. ERC1155 mandates that the caller of `safeTransferFrom()` must be approved to manage the tokens for the token owner. Unfortunately, neither `safeTransferFrom()` nor its callee `_update()` performs this crucial check (via `_opApprovals`), allowing anyone to transfer tokens from any address to their own. The patch in line 6 introduces a verification step to ensure that the token owner `from` has approved the message sender to handle his tokens, thus aligning with the ERC1155 requirement. Moreover, ERC1155 mandates that `safeTransferFrom()` checks whether the recipient is a contract and if so, whether the contract is capable of handling ERC1155 tokens. Unfortunately, the implementation in Figure 2 neglects these checks. Without the required capability, transferring tokens to a recipient contract results in the tokens being indefinitely trapped within the contract. Lines 8-10 show the necessary checks to fulfill ERC1155 requirements. The final high-security impact violation arises from the failure to check whether the recipient of a token transfer operation is address zero. Transferring tokens to address zero leads to the irreversible loss of the tokens. Overall, *AuditGPT*

**Table 4: Evaluation results on the ground-true dataset. (($x, y, z$): *x true positives, y false positives, and z false negatives, and Columns Time and Money represent the total time and monetary cost.)***

| | High | Medium | Low | Total | Time(s) | Money |
|---|---|---|---|---|---|---|
| **ECSD** | (10, 10, 11) | (26, 2, 34) | (37, 0, 24) | (73, 12, 69) | $2.6 * 10^7$ | $\$1.5 * 10^5$ |
| **SCE** | (0, 0, 21) | (27, 0, 33) | (12, 0, 49) | (39, 0, 103) | 0.02298 | - |
| **AuditGPT** | (21, 1, 0) | (57, 1, 3) | (61, 1, 0) | (139, 3, 3) | 1780.1 | $11.92 |

*is capable of detecting violations of ERC rules that can result in the loss of digital assets.*

The 112 violations with a medium-security impact stem from various reasons. Among them, 82 violate an ERC20 rule that dictates the function `transfer()` and `transferFrom()` should treat the transfer of zero values as a normal transfer. However, in 83 cases, `transfer()` or `transferFrom()` of an ERC20 contract implementation throws an exception when the message caller attempts to send zero tokens. 14 violations occur due to the absence of implementation for a function mandated by the corresponding ERC. Nine violations involve a function lacking the required return-value type specified by the ERC or having no return at all, in contrast to ERC requirements. Three violations result from neglecting to check the return of a function that returns a Boolean value, as mandated by ERC20. An additional three violations arise from the failure to throw an exception while implementing the `ownerOf()` function of ERC721. The remaining violation is also depicted in Figure 2, as ERC1155 requires `safeTransferFrom()` to utilize its input parameter `data` to call `onERC1155Received()`. These findings underscore *AuditGPT's capability to identify various types of violations.*

Of the 155 event-related violations, 146 fail to emit an event, and 9 are due to the omission of an event declaration.

In general, *AuditGPT exhibits accuracy*, with only 15 reported false positives for four reasons. First, in three cases, the input code is too long (exceeding 200 lines) or too complex (using a variable in multiple function calls and condition checks). Consequently, AuditGPT mistakenly concludes that a required action is not taken. Second, three false positives arise from AuditGPT failing to understand the `require` statement triggers an exception when its input condition is unsatisfied. Third, AuditGPT makes errors when inferring program semantics, resulting in eight false positives. For example, for seven false positives, the inspected code invokes an external function named `transfer()`, not necessarily implying the transfer of tokens. Despite this, AuditGPT incorrectly insists on the need for the code snippet to emit a transfer event. Fourth, in the remaining case, the input function unconditionally triggers an exception, which, in our assessment, satisfies the rule since an exception is indeed triggered when the condition is met. However, AuditGPT reaches a different conclusion.

## 5.2 Comparison with Baselines

***Methodology.*** We perform a head-to-head comparison between AuditGPT and the two baseline solutions using the ground-truth dataset. We execute AuditGPT and SCE on the ground-truth dataset. We manually examine their results and the auditing reports from ECSD security experts to count true positives, false positives, and false negatives for each tool. We assess the solutions' effectiveness and accuracy based on these statistical measures.

Moreover, we assess the costs associated with each solution in terms of time and money. For AuditGPT and SCE, we execute each tool on every contract file three times and report the average execution time. The time spent by ECSD is measured from the moment a Solidity programmer submits an auditing request by filing an issue report on GitHub until a security expert provides the auditing result by responding to the issue, considering that the Solidity programmer must wait for this duration to receive the result. The monetary cost of AuditGPT is represented by the fees charged by OpenAI. We only have expense information for one auditing request in the dataset, which amounts to $1000. We calculate the average expense per hour by dividing 1000 by the number of hours security experts spent on the contract. For each contract file without expense information, we multiply the average expense per hour with the time spent on the auditing request to estimate the monetary cost. Since SCE is an open-source software, no monetary expenditure is associated with its use.

***Effectiveness and Accuracy.*** As shown in Table 4, AuditGPT detects nearly all violations with just three false negatives and reports only three false positives. Compared with the two baseline solutions, *AuditGPT demonstrates greater effectiveness by identifying more true violations and increased accuracy with fewer false positives.*

AuditGPT accurately identifies all 21 violations with a high-security impact. Among them, nine stem from a failure to verify whether the message sender possesses the required privilege, as mandated by ERC20, mirroring the violation in Figure 1. One ERC20 rule requires function `approve(address _spender, uint256 _value)` to overwrite the allowance value of `_spender` with the input `_value`. AuditGPT pinpoints a case where `approve(address _spender, uint256 _value)` increases the allowance of `_spender` by `_value`, thus violating the rule. The remaining 11 violations result from neglecting to check if the sender holds a sufficient balance when transferring tokens. Notably, Solidity introduced the underflow check for subtraction operations in version 0.8.0. Consequently, Solidity programmers must explicitly compare the balance with the transferred amount for versions predating 0.8.0, and failing to do so leads to the 11 identified violations. AuditGPT also uncovers 13 instances with a medium-security impact where a function fails to generate its return as specified in ERC20. For instance, ERC20 dictates that `totalSupply()` should return the quantity of supplied tokens. However, in one violation, the implementation of `totalSupply()` returns the result of subtracting the balance of address zero from the supplied tokens. For all other violations with a medium- or low-security impact, AuditGPT identifies similar issues within the large dataset.

AuditGPT misses three violations. Two are in the same the implementation of `transfer()`, which conducts an overflow check. This check uses a `require` statement to mandate the sum of the receipt's balance and the token amount must be greater than the receipt's balance, resulting in the inability to transfer zero tokens, thus violating an ERC20 rule. Unfortunately, AuditGPT fails to comprehend such program semantics. The remaining false negative is attributed to the excessive length of the input code, consisting of 180 lines. This length causes AuditGPT to miss checking the returned Boolean value of an external function call.

AuditGPT reports three false positives with reasons distinct from those in the large dataset. Two of the false positives emerge in a contract's `transferFrom()` function. This function should subtract the token amount from the current value of an element in the allowance map and utilize the result to update the element (like line 5 in Figure 1). However, the implementation contains a bug wherein it subtracts the token amount from the value of an element in the balance map and uses the result to update the element in the allowance map. We postulate that these two false positives result from this bug, as AuditGPT does not recognize that the `transferFrom()` implementation checks whether the message sender has the privilege using the allowance map. Additionally, it considers the function's return does not match whether it completes the transfer. Regarding the remaining false positive, we cannot ascertain the reason. The code is relatively straightforward, and the `approve()` function emits the required `Approval` event. Nonetheless, AuditGPT reports that the necessary event is not emitted.

ECSD. The manual auditing service identifies 73 violations. Only two violations, where an overflow check causes the inability to handle transferring zero tokens, are missed by AuditGPT. Additionally, AuditGPT identifies 68 violations that the auditing service cannot detect. Furthermore, the service has 12 false positives, exceeding the number in AuditGPT. Two false positives occur because auditors are unaware that Solidity automatically generates getter functions for contract fields, leading them to mistakenly believe that two required APIs are not implemented. ERC20 mandates `transfer()` to check if the token owner has enough balance for the transfer, but no such rule applies to `transferFrom()`. However, security auditors mistakenly assume a similar rule exists for `transferFrom()`, resulting in the remaining ten false positives. In conclusion, *AuditGPT demonstrates superior violation detection capability and accuracy compared to the manual auditing service.*

SCE. As explained in Section 4, the functionality of SCE is relatively straightforward, leading it to overlook the most violations among the three solutions. All 39 violations detected by SCE are also identified by AuditGPT. Among these violations, 27 involve instances where a function mandated by ERC20 is either not implemented or does not match the API specified in ERC20. The remaining 12 violations are event-related. Given that the rules covered by SCE are uncomplicated, SCE does not generate any false positives.

***Time and Monetary Cost.*** Among the three solutions, SCE incurs the smallest cost. It analyzes the 30 contract files in the ground-truth dataset in less than 0.1 seconds and does not involve any charges. However, due to its limitation in detection capability, SCE is inadequate in identifying ERC rule violations.

AuditGPT requires interaction with the LLM provided by OpenAI, resulting in a longer processing time compared to SCE, which solely conducts static program analysis. Conversely, AuditGPT incurs significantly lower costs compared to the manual service provided by ECSD. The time usage of the manual service is 14535 times higher, and the monetary usage is 12562 times greater. *AuditGPT outperforms the manual service with significantly lower costs.*

## 5.3 Necessity of AuditGPT design points

***Methodology.*** To comprehend the significance of individual design points, namely ERC rule sequentialization, prompt specialization, code slicing, one-shot, and breaking down compound rules, we utilize AuditGPT on the ground-truth dataset with each design point deactivated. We then count the number of true positives and false positives in each experimental setting.

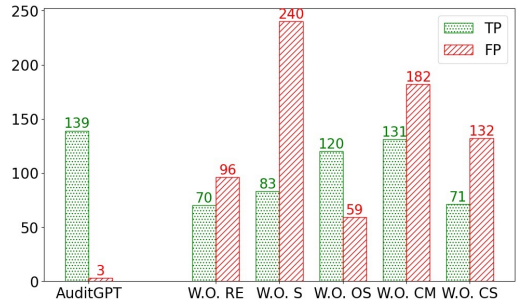


**Figure 3: Evaluation results on the ground-truth dataset with each design point deactivated. *(W.O.: without, RE: rule sequentialization, S: specialization, OS: one shot, CM: breaking down, CS: code slicing)***

***Experimental Results.*** As shown in Figure 3, the full-featured AuditGPT identifies the highest number of violations and reports the fewest false positives, thereby showcasing *the rationale behind each design point of AuditGPT.* Additionally, the full-featured version detects all violations identified by other versions, except for one. This particular violation stems from the failure to check the returned Boolean value of an external call. While the full-featured version misses it, the version with disabled code slicing captures it, indicating that a more advanced setting does not necessarily yield better results in all cases.

Both ERC rule sequentialization transforms the process of examining the entire ERC20 document (with 35 rules in total) into scrutinizing individual rules. Similarly, code slicing breaks down the entire contract into individual functions. Both approaches simplify the auditing problem within a single prompt. Consequently, disabling either of them would lead to AuditGPT missing the most violations among all settings. Prompt specialization reduces the difficulty for the LLM to understand a ERC rule and aids the LLM in comprehending when the rule is violated. Without it, AuditGPT reports the largest number of false positives among all settings. Five ERC20 rules require a specific action when a condition is met. Without breaking down compound rules, AuditGPT considers many cases where a condition does not exist as rule violations, leading to 182 false positives. One shot assists the LLM in understanding what constitutes a violation for a particular rule with a concrete example. It helps reduce 19 false negatives and 56 false positives.

## 5.4 Threats to Validity

Threats to validity arise from two primary sources. First, we randomly sample smart contracts for our experiments, and we don't particularly include contracts with intricate control flow or data dependence relations. We anticipate LLMs might exhibit worse performance when auditing code with such complexities. Second, our empirical study is confined to four ERCs, and our experiments only encompass three of them. Consequently, the findings and observations from our study and experiments may not be generalized to other ERCs. Evaluating AuditGPT with more intricate code and additional ERCs remains a topic for future investigation.

## 6 CONCLUSION

In this paper, we conduct an empirical study on the implementation rules outlined in ERCs, considering four key aspects: their content, security impacts, specifications in ERCs, and potential implementations in Solidity programs. Using the study insights, we develop an automated tool called AuditGPT that employs LLMs to compare smart contracts with their corresponding ERCs and identify rule violations. AuditGPT successfully pinpointed numerous rule violations, outperforming the two baseline solutions. We anticipate that this research will enhance the understanding of ERC rules and their violations, inspiring further exploration in this field. Future work may investigate leveraging LLMs to determine whether a code implementation aligns with a natural language description in other scenarios, as well as detecting other types of Solidity bugs.

## REFERENCES


[1] U. I. Alıcı, A. Oksuztepe, O. Kilincceker, and E. Karaarslan. Openai chatgpt for smart contract security testing: Discussion and future directions. *Available at SSRN 4412215*, 2023.
[2] Antiersolutions. Smart contract auditing services, 2023. https://www.antiersolutions.com/smart-contract-audit/.
[3] L. Arena. 9lives arena - the ultimate pvp gaming experience, 2023. https://www.9livesarena.com/.
[4] BitInfoCharts. Ethereum (ETH) price stats and information, 2023. https://bitinfocharts.com/ethereum/.
[5] T. Blackstone. Reewardio loyalty platform putting enj nft items to work – real use case, 2019. https://www.castlecrypto.gg/reewardio-loyalty-platform-demo-review/.
[6] BLOCKHUNTERS. Smart contract audit, 2023. https://blockhunters.io/smart-contract-audit/.
[7] Blockworks. Today's Cryptocurrency Prices by Market Cap, 2023. https://blockworks.co/prices.
[8] B. Blog. Erc-20 tokens: What they are and how they are used, 2023. https://bitpay.com/blog/erc-20-tokens-what-they-are-and-how-they-are-used/.
[9] T. Brandstätter, S. Schulte, J. Cito, and M. Borkowski. Characterizing efficiency optimizations in solidity smart contracts. In *2020 IEEE International Conference on Blockchain (Blockchain '20)*, pages 281–290. IEEE, 2020.
[10] T. Brandstätter, S. Schulte, J. Cito, and M. Borkowski. Characterizing efficiency optimizations in solidity smart contracts. In *Proceedings of the 2020 IEEE International Conference on Blockchain (Blockchain '20)*, pages 281–290. IEEE, 2020.
[11] Buffer. Buffer finance — options trading simplified, 2023. https://buffer.finance/.
[12] CertiK. Securing the web3 world, 2023. https://www.certik.com/.
[13] Chainlink. Top 6 Smart Contract Languages in 2023, 2023. https://chain.link/education-hub/smart-contract-programming-languages.
[14] T. Chen, Y. Feng, Z. Li, H. Zhou, X. Luo, X. Li, X. Xiao, J. Chen, and X. Zhang. Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts. *IEEE Transactions on Emerging Topics in Computing*, 9(3):1433–1448, 2020.
[15] T. Chen, X. Li, X. Luo, and X. Zhang. Under-optimized smart contracts devour your money. In *Proceedings of the 24th IEEE international conference on software analysis, evolution and reengineering (SANER '17)*, pages 442–446. IEEE, 2017.
[16] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, and X. Zhang. Towards saving money in using smart contracts. In *Proceedings of the 2018 IEEE/ACM 40th international conference on software engineering: New ideas and emerging technologies results (ICSE-NIER '18)*, pages 81–84. IEEE, 2018.
[17] Y. Chen, F. Ma, Y. Zhou, Y. Jiang, T. Chen, and J. Sun. Tyr: Finding consensus failure bugs in blockchain system with behaviour divergent model. In *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.
[18] E. Commonwealth. Callisto smart-contract auditing department, 2023. https://github.com/EthereumCommonwealth/Auditing.
[19] Crytic. Erc conformance, 2023. https://github.com/crytic/slither/wiki/ERC-Conformance.
[20] Crytic. Slither, the smart contract static analyzer, 2023. https://github.com/crytic/slither.
[21] defiprime. Ethereum DeFi Ecosystem, 2023. https://defiprime.com/ethereum.
[22] Dexaran. Erc-20 token standard, 2023. https://dexaran820.medium.com/erc-20-token-standard-7fa2316cdcac.
[23] Y. Duan, M. Andrychowicz, B. Stadie, O. Jonathan Ho, J. Schneider, I. Sutskever, P. Abbeel, and W. Zaremba. One-shot imitation learning. *Advances in neural information processing systems*, 2017.
[24] ECMA. ECMA-262, 2023. https://www.ecma-international.org/publications-and-standards/standards/ecma-262/.
[25] W. Entriken, D. Shirley, J. Evans, and N. Sachs. Erc-721: Non-fungible token standard, 2018. https://eips.ethereum.org/EIPS/eip-20.
[26] Ethereum. Ethereum, 2023. https://ethereum.org/en/.
[27] Ethereum. Ethereum dapps, 2023. https://ethereum.org/en/dapps.
[28] Ethereum. Ethereum improvement proposals, 2023. https://eips.ethereum.org/erc.
[29] Ethereum. ethereum/ercs, 2023. https://github.com/ethereum/ERCs/blob/master/ERCS/eip-1.md.
[30] Ethereum. Smart contract anatomy, 2023. https://ethereum.org/en/developers/docs/smart-contracts/anatomy.
[31] Etherscan. The ethereum blockchain explorer. https://etherscan.io.
[32] F. Finance. Fujidao — the auto-refinancing borrow protocol, 2023. https://v1.fuji.finance/.
[33] A. Ghaleb, J. Rubin, and K. Pattabiraman. etainter: detecting gas-related vulnerabilities in smart contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22)*, pages 728–739, 2022.
[34] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, 2018.
[35] G. Ibba, M. Ortu, R. Tonelli, and G. Destefanis. Leveraging chatgpt for automated smart contract repair: A preliminary exploration of gpt-3-based approaches.
[36] ImmuneBytes. Smart contract audit services, 2023. https://www.immunebytes.com/smart-contract-audit/.
[37] Q.-P. Kong, Z.-Y. Wang, Y. Huang, X.-P. Chen, X.-C. Zhou, Z.-B. Zheng, and G. Huang. Characterizing and detecting gas-inefficient patterns in smart contracts. *Journal of Computer Science and Technology*, 37(1):67–82, 2022.
[38] P. Labs. Polygon. https://polygon.technology.
[39] Y. Li, H. Liu, Z. Yang, Q. Ren, L. Wang, and B. Chen. Safepay on ethereum: A framework for detecting unfair payments in smart contracts. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 1219–1222, 2020.
[40] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe. Reguard: Finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)*, Gothenburg, Sweden, 2018.
[41] Y. Marcus, E. Heilman, and S. Goldberg. Low-resource eclipse attacks on ethereum's peer-to-peer network. *Cryptology ePrint Archive*, 2018.
[42] Markus Waas. Top 7 Reasons To Learn Solidity Programming ASAP, 2022. https://zerotomastery.io/blog/top-7-reasons-to-learn-solidity-programming/.
[43] K. Nelaturu, S. M. Beillahit, F. Long, and A. Veneris. Smart contracts refinement for gas optimization. In *Proceedings of the 2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS '21)*, pages 229–236. IEEE, 2021.
[44] OpenAI. Openai gpt4 turbo. https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo.
[45] OpenSea. Opensea, the largest nft marketplace, 2023. https://opensea.io/.
[46] S. Palladino. Erc20 verifier, 2019. https://github.com/spalladino/erc20-verifier.
[47] PixelPlex. Smart contract audit, 2023. https://pixelplex.io/smart-contract-audit/.
[48] PolygonScan. Polygon pos chain explorer. https://polygonscan.com.
[49] P. Qian, Z. Liu, Q. He, R. Zimmermann, and X. Wang. Towards automated reentrancy detection for smart contracts based on sequential models. *IEEE Access*, 8:19685–19695, 2020.
[50] W. Radomski, A. Cooke, P. Castonguay, J. Therien, E. Binet, and R. Sandford. Erc-1155: Multi token standard, 2018. https://eips.ethereum.org/EIPS/eip-1155.
[51] Rarible. Rarible — aggregated nft marketplace with rewards, 2023. https://rarible.com/.
[52] Revoluzion. Revoluzion smart contract audit report services, 2023. https://www.revoluzion.io/audit.
[53] N. Şahin. Malware detection using transformers-based model gpt-2. Master's thesis, Middle East Technical University, 2021.
[54] C. Smith. Token standards, 2023. https://ethereum.org/en/developers/docs/standards/tokens/.
[55] M. Stefanović, Đ. Pržulj, D. Stefanović, S. Ristić, and D. Čapko. The proposal of new ethereum request for comments for supporting fractional ownership of non-fungible tokens. *Computer Science and Information Systems*, (00):38–38, 2023.
[56] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu. When gpt meets program analysis: Towards intelligent detection of smart contract logic vulnerabilities in gptscan. *arXiv preprint arXiv:2308.03314*, 2023.
[57] F. Vogelsteller and V. Buterin. Erc-20: Token standard, 2015. https://eips.ethereum.org/EIPS/eip-20.
[58] S. Wang, C. Zhang, and Z. Su. Detecting nondeterministic payment bugs in ethereum smart contracts. 2019.
[59] W. Wang, M. Meng, Y. Cai, R. Chow, Z. Wu, and A. Du. Erc-3525: Semi-fungible token, 2020. https://eips.ethereum.org/EIPS/eip-3525.
[60] Wikipedia. Binance, 2023. https://en.wikipedia.org/wiki/Binance.
[61] Wikipedia. Ethereum, 2023. https://en.wikipedia.org/wiki/Ethereum.
[62] Wikipedia. Horizon (video game series), 2023. https://en.wikipedia.org/wiki/Horizon_(video_game_series).

[63] Wikipedia. Shiba inu (cryptocurrency), 2023. https://en.wikipedia.org/wiki/Shiba_Inu_(cryptocurrency).
[64] Wikipedia. Tether (cryptocurrency), 2023. https://en.wikipedia.org/wiki/Tether_(cryptocurrency).
[65] K. Wüst and A. Gervais. Ethereum eclipse attacks. Technical report, ETH Zurich, 2016.
[66] C. S. Xia and L. Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. *arXiv preprint arXiv:2304.00385*, 2023.
[67] G. Xu, B. Guo, C. Su, X. Zheng, K. Liang, D. S. Wong, and H. Wang. Am i eclipsed? a smart detector of eclipse attacks for ethereum. *Computers & Security*, 88:101604, 2020.
[68] Y. Xue, M. Ma, Y. Lin, Y. Sui, J. Ye, and T. Peng. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*, 2020.
[69] Y. Yang, T. Kim, and B.-G. Chun. Finding consensus bugs in ethereum via multi-transaction differential fuzzing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)*, pages 349–365, 2021.