SC-Bench: A Large-Scale Dataset for Smart Contract Auditing

Shihao Xia*, Mengting He*, Linhai Song*, Yiying Zhang†
*The Pennsylvania State University †University of California, San Diego
Email: {sxia, mvh6224}@psu.edu, songlh@ist.psu.edu, yiying@ucsd.edu

***Abstract*—There is a huge demand to ensure the compliance of smart contracts listed on blockchain platforms to safety and economic standards described in natural languages. Today, manual efforts in the form of auditing are commonly used to achieve this goal. ML-based automated techniques have the promise to alleviate human efforts and the resulting monetary costs. However, unlike other domains where ML techniques have had huge successes, no systematic ML techniques have been proposed or applied to smart contract auditing. We present SC-Bench, the first dataset for automated smart-contract auditing research. SC-Bench consists of 5,377 real-world smart contracts running on Ethereum, a widely used blockchain platform, and 15,975 violations of standards on Ehereum called ERCs. Out of these violations, 139 are real violations programmers made. The remaining are errors systematically injected by us to reflect the violations of different ERC rules. We evaluate SC-Bench using GPT-4 by prompting it with both the contracts and ERC rules. In addition, we manually identify each violated rule and the corresponding code site (*i.e.*, oracle) and prompt GPT-4 with the information asking for a True-or-False question. Our results show that without the oracle, GPT-4 can only detect 0.9% violations, and with the oracle, it detects 22.9% violations. These results show the potential room for improvement in ML-based techniques for smart-contract auditing.**

***Index Terms*—Smart Contract Auditing, Dataset**

I. Introduction

Ethereum [35], [13] is a decentralized, open-source blockchain platform that has become the de facto for running decentralized applications like smart contracts [14], [17]. To standardize smart contract implementation, Ethereum Request for Comments (ERCs) have been developed. Each ERC provides a set of formal standards and is typically written in natural languages [31]. For example, ERC20 [33] defines the rules for fungible tokes — digital assets that are interchangeable. ERCs are essential in the Ethereum ecosystem, offering a common framework that developers must follow when implementing smart contracts. Violations of ERC rules can result in interoperability issues, contract failures, and financial loss. Moreover, tokens that don't comply with ERCs may be delisted from exchanges, as many exchanges require ERC compliance for listing [15].

Despite the importance of adhering to ERC standards, developers often find it challenging to comply fully due to the complexity of the rules and their contract code. ERC standards consist of numerous rules, and this number continues to grow as new standards are introduced. For the three ERCs discussed in this work, there are 132 rules. These rules are presented in various formats, with some outlined as code comments and others explained in natural language paragraphs. Meanwhile, smart contract code is typically intricate, often spanning thousands of lines across multiple files. Some details may be hidden within complex caller-callee relationships, while others might involve objects and functions coded by different developers. This combination of ERC complexity and the intricacies of smart contracts makes it exceedingly difficult for programmers to ensure full compliance with ERC rules. Consequently, ERC violations are common in real-world smart contracts [8].

To detect ERC violations, today's common practice heavily relies on human efforts. Automated, program-analysis-based checkers for ERC rules do exist [9], [24], but they fail to detect complex violations because of many ERC rules' non-structured, natural-language-based definitions. As a result, smart contracts commonly undergo human auditing, provided by specialized services with security experts [6], [28], [25], [3], [20], [1], [8]. Auditing services are not only costly but also slow. For example, we examined the history of 30 smart contracts that were submitted for manual auditing on a platform called Ethereum Commonwealth Security Department [8]. We found that each contract only has an average of 260 lines of code but is audited for ten days with an estimated cost of $500. Clearly, manual auditing is not a scalable approach.

A promising, scalable approach for automated smart-contract auditing is to leverage large language models (LLMs), both because of LLMs' success in domains like program generation [30] and bug fixing [21], [38] and because of a fair amount of ERC rules' natural-language descriptions. To develop LLM-based techniques (*e.g.*, fine-tuning, few-shot learning) for smart contract auditing, an important step is to construct quality datasets. Unfortunately, no smart contract datasets exist for ERC rule checking and fixing. Traditional program bug datasets [29], [19] cannot be used for smart-contract auditing, as unlike ERC rules, compiler errors and program runtime errors have well-defined, structured definitions.

To drive research and practices in smart-contract auditing and to assist real users with their auditing tasks, we release *SC-Bench*, the first dataset of real-world smart contracts and their ERC-rule violations. SC-Bench consists of **15,975 ERC violations and 5,377 real-world smart contracts**, collected from etherscan.io [18], polygonscan.com [26], and Ethereum Commonwealth Security Department [8]. 139 violations from 30 contracts are real-world ERC violations we collect and inspect. As real violations are rarely published, we build

program analysis techniques and inject 15,836 violations into 5,347 contracts according to 88 ERC rules.

We evaluate SC-Bench using GPT-4 with two different approaches. First, we prompt GPT-4 with the contract to be inspected along with the full rule set of the corresponding ERC. Our results show that GPT-4 detects only 29% of real violations and 0.6% of injected violations. To assess how GPT-4 might perform with additional oracle information, we manually identify the violated ERC rules and the specific code sites causing the violations. We then prompt GPT-4 with this precise information and ask it a True-or-False question. In this case, GPT-4 successfully detects 42.8% of real violations and 22.8% of injected violations. The improved detection rates highlight the potential room for machine learning-based techniques in automating smart contract audits.

Overall, SC-Bench goes beyond being a key contribution to Ethereum smart contract auditing. Other software auditing and verification tasks, such as ensuring the compliance of API usage rules, can potentially use SC-Bench's contract dataset, its violation dataset, or its methodologies for evaluation.

We have released our dataset, results, and source code of our injecting scripts, all of which can be found at https://github.com/charlesxsh/scbench.

II. Background

This section provides background on Ethereum, smart contracts, ERCs, and ERC violations.

A. Ethereum and Smart Contracts

Ethereum is a blockchain platform where developers can create and deploy smart contracts to build decentralized applications (dApps) [35], [13]. Both Ethereum users and smart contracts have their own unique Ethereum addresses, which allow them to send and receive Ether (the native cryptocurrency of Ethereum) and interact with smart contracts to carry out complex transactions for a variety of purposes. Ethereum has grown into a thriving digital economy ecosystem, with a total market capitalization exceeding $200 billion at the time of writing and more than one million transactions processed daily, amounting to over $4 billion in volume [4], [2]. Smart contracts are central to Ethereum's success, driving the majority of transactions and powering key functionalities such as cryptocurrencies, NFTs, and decentralized finance (DeFi) [33], [12], [10].

Smart contracts are commonly written in the Solidity programming language [7], [23]. An example of a smart contract is presented in Figure 1. The contract has two contract fields in lines 2 and 3, `_balances` (line 2) and `_allowances` (line 3), tracking the number of tokens owned by each address and the tokens approved by the first dimension for manipulation by the second dimension, respectively. The function `transferFrom()` in lines 6–10 facilitates the transfer of `amount` tokens from one address to another. `transferFrom()` 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.

```
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  }
```

Fig. 1: An ERC20 rule violation that can be exploited to steal tokens. *(The code is simplified for illustration purpose.)*

B. Ethereum Request for Comment (ERC)

ERCs are technical specifications that define the requirements for implementing smart contracts. Those requirements aim to ensure compatibility across different contracts, applications, and platforms. By standardizing the contract implementations, ERCs help strengthen and promote the growth of the Ethereum ecosystem [16], [15], [32].

Typically, an ERC begins with a brief explanation of its motivation. For instance, ERC20 [33] aims to establish a standard token interface that can be used by applications such as wallets and decentralized exchanges. After the motivation, an ERC specifies all the necessary public functions and events by outlining their parameters, return values, and any optional attributes for the parameters. It also provides implementation requirements in the form of plain text or code comments for each function or event declaration. For example, besides the requirements for the function API and return value generation, ERC20 includes the following rules for the `transferFrom()` function (as shown in Figure 1), which mandate emitting a `Transfer` event, verifying that the message sender has been approved to manage the token owner's tokens (and throwing an exception if not), treating the transfer of zero tokens in the same way as any other amount, and requiring an event to be emitted even when transferring zero tokens.

C. ERC Rule Violations

An ERC rule violation occurs when a smart contract is expected to follow a specific rule, but certain aspects of the contract do not. Figure 1 illustrates an instance of an ERC20 rule violation in a real smart contract, where the `transferFrom()` function fails to check whether the caller has the necessary authorization to transfer the specified `amount` of tokens. This verification is required by ERC20 to

TABLE I: How ERC rules are distributed across different error-injection methods. *("Uncovered": rules whose violations cannot be injected by our designed error-injection methods.)*

ID	Check	API	Value	Call	Return	Logging	Uncovered	Total
ERC20	1	9	1	0	9	5	7	32
ERC721	12	10	0	2	4	10	22	60
ERC1155	7	6	0	2	0	7	18	40
Sum	20	25	1	4	13	22	47	132

TABLE II: How many errors are injected by different error-injection methods.

	# of Violations							# of Cont.
	Check	API	Value	Call	Return	Logging	Total	
ERC20	566	4605	736	0	5930	3612	15449	5211
ERC721	158	72	0	15	48	33	326	110
ERC1155	18	30	0	4	9	0	61	26
Total	742	4707	736	19	5987	3645	15836	5347

ensure financial security. As a result of this oversight, anyone could potentially steal tokens from any address by invoking `transferFrom()` to transfer tokens to their own address. The patch shown in line 7 offers a fix for this violation. It employs a two-dimensional map, `_allowances`, to track the number of tokens the "from" address allows `msg.sender` to manage. If the subtraction operation in this line results in an underflow, an exception is triggered, causing the transaction to terminate. This fix ensures that the message caller cannot transfer tokens unless they have enough privilege.

Violating ERC rules can lead to significant financial losses and unpredictable contract behavior. For example, ERC721 requires the `onERC721Received()` function to be called for each token transfer when the recipient is a contract. Additionally, it mandates that the caller must check if the return value of `onERC721Received()` matches a specific magic number. These two rules ensure that the recipient contract is capable of properly handling the transferred tokens. If tokens are sent to a contract that cannot handle them, they can become permanently locked within the contract. This issue was first reported in 2017 on Ethereum Reddit, resulting in the loss of $10,000 worth of tokens at the time, and it has since led to millions of dollars in losses [11]. In short, ensuring contracts comply with ERC rules is essential to protect financial assets and maintain proper contract functionality.

D. Today's Auditing Practices

The common practice for detecting ERC rule violations today relies on manual auditing, often provided by paid services [6], [28], [25], [3], [20], [1], [8]. One such service is the Ethereum Commonwealth Security Department [8], where users submit smart contracts for auditing by filing a GitHub issue. The service then manually audits the submitted contracts and provides feedback through the issue.

To reduce the manual workload and associated costs, some automated tools have been developed using static program analysis. For example, Slither offers specific checkers (*i.e.*, slither-check-erc [9]) that verify whether a given contract complies with the corresponding ERC standards for 11 ERCs. However, these tools have limited functionality. They primarily focus on ensuring the presence of required functions and events, confirming that these elements are correctly declared, and verifying that functions trigger the necessary events. Unfortunately, they are unable to check more advanced conditions, such as verifying whether the message caller has enough privilege to transfer tokens. To our knowledge, no machine learning-based automated auditing techniques have been proposed yet. We hope that the release of SC-Bench can foster a new line of research in this regard.

III. SC-Bench

This section outlines the process of building the dataset and provides some relevant statistics.

A. Construction

We collect real-world smart contracts from Ethereum Commonwealth Security Department [8], etherscan.io [18], and polygonscan.com [26]. As the purpose of SC-Bench is for evaluating automated ERC-rule checking, we include ERC rule violations of these collected contracts in SC-Bench using two approaches. First, we manually inspect a set of real smart contracts and identify all their ERC violations. This process is time-consuming, resulting in a limited number of violations. To address this, we perform automated error injection into a large number of real-world smart contracts, significantly increasing the number of violations. These two types of violations serve to validate each other and help ensure that the evaluated techniques demonstrate consistent performance.

Manual Inspection. We manually analyze 30 ERC20 contracts obtained from the Ethereum Commonwealth Security Department [8], selecting the most recent 30 audit requests that meet the following criteria: 1) they contain Solidity source code, 2) they have been approved by Solidity developers (as indicated by the "approved" tag), 3) they exhibit ERC rule violations, and 4) all contracts and the related Solidity code are within the same contract file. On average, each contract file contains 260.9 lines of Solidity code. Through detailed examination, we identify 139 ERC rule violations. Of these, 27 violations present a clear method for exploitation that can lead to financial losses, and we classify them as having a high-security impact. Another 48 violations result from incorrect implementation of required functionalities, but there is no apparent way to exploit these for financial gain. Therefore, we consider them to have a medium-security impact. The remaining 64 violations involve failures in generating necessary logs and are categorized as having a low-security impact.

Error Injection. To augment the limited manual-inspected real violations, we perform error injection to 5,347 real-world smart contract source code using the following methodology. We design six error injection methods corresponding to 85 ERC rules, resulting in a total of 15,836 ERC violations. Table I shows the number of rules each method covers.

```
1  contract ERC1155 {
2    function safeBatchTransferFrom(address from, address
       to, uint256[] memory ids, uint256[] memory amounts,
       bytes memory data) public  {
3      require(from==_msgSender()||isApprovedForAll(from,
       _msgSender()), "not token owner or approved");
4      _safeBatchTransferFrom(from,to,ids,amounts,data);
5    }
6
7    function _safeBatchTransferFrom(address from, address
       to, uint256[] memory ids, uint256[] memory amounts,
       bytes memory data) internal {
8      require(ids.length==amounts.length, "ids and amounts
        length mismatch");
9  -   require(to!=address(0), "transfer to address 0");
10     address operator = _msgSender();
11     for (uint256 i = 0; i < ids.length; ++i) {
12       uint256 id = ids[I];
13       uint256 amount = amounts[I];
14       _balances[id][from] -= amount;
15       _balances[id][to] += amount;
16     }
17     emit TransferBatch(operator,from,to,ids,amounts);
18   }
19 }
```

Fig. 2: Violation injection of a condition-check rule. *(Line 9 is deleted to perform the violation injection.)*

```
1 contract ERC20 {
2   mapping(address => uint256) _balances;
3
4   function balanceOf(address account) public view returns
        (uint256) {
5 -   return _balances[account];
6 +   return _balances[account]+827;
7   }
8 }
```

Fig. 3: Violation injection of a return rule. *(Line 5 is replaced with line 6 to perform the violation injection.)*

We focus on contracts in three ERC standards, ERC20 [33], ERC721 [12], and ERC1155 [27], for two reasons. First, these ERCs are significant and have numerous crucial financial applications. For instance, there are over 450,000 ERC20 tokens on the Ethereum platform [5], many of which have market capitalizations exceeding $1 billion (*e.g.*, USDT [37], SHIB [36], Binance USD [34]). Second, these three standards are among the most mature ERCs and inspire subsequent ERCs, making their rules representative of rules in other ERCs. As shown in Table I, these three ERCs specify 132 rules. Our error injection methods can cover 85 of them. We do not include the rest either because they are not clearly specified (*e.g.*, "throw if any other error" in ERC1155) or because they require more complex static analysis or injection methods.

We collect contracts in these three ERCs from etherscan.io [18] and polygonscan.com [26]. These two platforms are the most popular analytics platforms for Ethereum and its sidechain Polygon [22], respectively. As shown in Table II, we collected 5,211 contracts that are supposed to target ERC20, 110 for ERC721, and 26 for ERC1155[1]. On average, each of these contracts contains 477.5 lines of code. We collect more contracts targeting ERC20 than ERC721 and ERC1155 contracts due to the significantly higher prevalence of ERC20 in practice.

We randomly inject one to three errors in each contract based on the fit of rules to the contract to construct ERC violations. To inject an error, we first convert each input smart contract source-code file into its abstract syntax tree (AST). We then randomly select a rule and apply the corresponding error-injection method to the AST. After modifying the AST, we convert it back into source code. We perform error injections on ASTs because it is easier to conduct static program analysis and modify code on ASTs than to work directly with the source code. Table II shows the number of errors injected by each method. Finally, we verify that the modified source code compiles without errors using the Solidity compiler. Below, we discuss each error injection method in detail.

Violations of Condition-Check Rules. ERCs mandate certain `public` functions to perform condition checks on their input parameters or callers' addresses (*i.e.*, `msg.sender`). Sometimes, these checks validate the input parameters. For example, ERC721 disallows the input parameter of function `ownerOf()` to be zero. More crucially, these checks ensure the caller has the permission to perform an action. For instance, ERC20's `transfer(address _to, uint256 _value)` function sends tokens in the amount specified by its second parameter from the caller's account to the receiver, whose address is the first parameter. ERC20 requires verifying that the caller has enough tokens (an amount greater than or equal to the second parameter) and throwing an exception if this condition is not met.

Many of these rules are enforced using `require` statements. To inject an error of this type into a function, we remove all `require` statements that check the parameter required by the rule as part of their conditions. Figure 2 illustrates an example of how a rule violation is injected in this way. ERC1155 requires that function `safeBatchTransferFrom()` "MUST revert if '_to' is the zero address," where `_to` is the second parameter of the function. Violating this rule would result in a permanent loss of the transferred tokens. The violation injection method begins with the `safeBatchTransferFrom()` function in line 2, as the rule applies to this function. It ignores the `require` statement in line 3, since its conditions do not involve the `_to` parameter. The method then proceeds to analyze the callee function `_safeBatchTransferFrom()` defined in lines 7–18. Similar to line 3, the method retains the `require` statement in line 7. However, it removes the `require` statement in line 9, as this one checks the `_to` parameter. None of the remaining lines contain `require` statements, so they are left unchanged. As shown in Figure 2, the modified contract can still be compiled by the Solidity compiler, but it allows `safeBatchTransferFrom()` to transfer tokens to the zero address, thus violating the rule.

Violations of API Rules. Each ERC specifies a set of re-

[1]We mainly rely on a contract's name or the name of a base contract inherited by the contract to determine whether the contract implements a particular ERC.

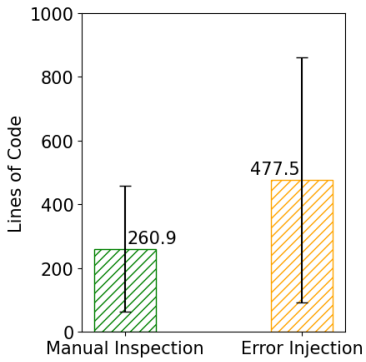

Fig. 4: Average contract size.

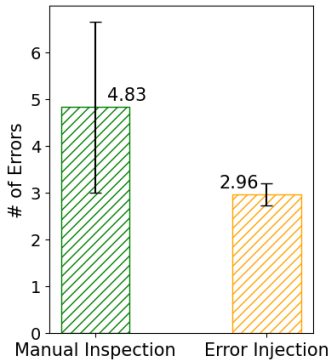

Fig. 5: Average error number in a contract.

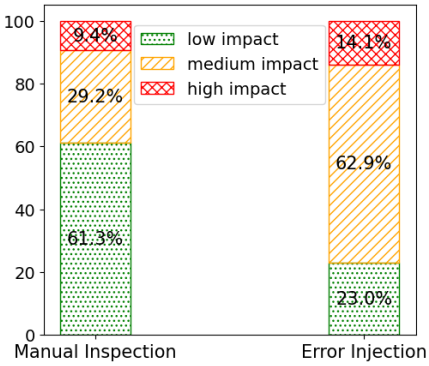

Fig. 6: How errors distribute across different security impacts.

quired function call APIs that all contracts following the ERC must include. To inject an error for a required API, we identify and remove the function's definition, including both its declaration and its function body. To ensure the contract can still be compiled, we also remove all call sites of the function.

Violations of Return-Value Rules. ERCs explicitly define how return values should be computed for certain function call APIs. For example, ERC20 requires that `allowance(address _owner, address _spender)` returns the token amount that `_owner` allows `_spender` to withdraw. As return-value rules apply to only four data types, integer, Boolean, address, and string, we design four distinct error-injection methods for each of them. We add a random integer to an integer return value, flip a Boolean return value, replace an address return with a random address, and change a string return value to an empty string if it is not already empty or replace an empty return string with a random string.

Figure 3 illustrates an example of injecting a violation of a rule that specifies how to generate a integer return value. According to ERC20, the function `balanceOf(address _owner)` must return "the account balance of the another account with address _owner." The contract in Figure 3 adheres to this rule, as shown in line 5. To inject a violation, we modify the return value by adding a random number to it, as demonstrated in line 6.

Violations of Value-Update Rules. Fields or state variables in a contract represent its state. Some ERC rules specify how a public function should update a state variable. For example. contracts following ERC20 rules use a two-dimensional `mapping` to record the number of tokens the first key allows the second key to withdraw. The public function `approve(address _spender, uint256 _value)` sets the number of tokens (specified by the second parameter `_value`) that the caller's address (*i.e.*, the message sender) permits the first address parameter `_spender` to withdraw. Therefore, ERC20 mandates that `approve(address _spender, uint256 _value)` updates the appropriate field of the two-dimensional `mapping` with `_value`. To inject an error of this type into a function, we remove all assignments to the corresponding state variable within the function. The challenge is that ERCs do not prescribe how to name state variables, as these variables are only accessible within a contract. Consequently, given a rule of this type, different contracts are likely to use different names for the corresponding state variable. Fortunately, ERCs require a getter function to return its value for most state variables, and this getter function has the same name across all contracts for each ERC. This requirement enables us to locate the correct state variable automatically and perform the error injection.

Violations of Function-Call Rules. Some ERC rules specify that a function must be called after a certain event. For instance, ERC721 requires `onERC721Received()` to be called if tokens are sent to a contract. To inject an error for a call rule about function `A()` inside function `B()`, we simply remove all call sites of `A()` in `B()`.

Violations of Logging Rules. ERCs require specific events to be emitted within certain functions or after particular code actions for logging purposes. To inject such an error that violates one of those rules within a function, we remove all code statements within the function that emit the corresponding event.

B. Dataset Summary

Our dataset contains 15,975 ERC rule violations. Among them, 139 are introduced by the real-world programmers, while the remaining 15,836 are injected by us. 39 errors made by real-world programmers cannot be replicated using the error-injection methods. These include 28 errors that violate the rule that transferring zero tokens must be treated the same as transferring non-zero tokens and 11 errors that violate the rule that the `transfer()` function must throw an exception if the sender does not have enough balance.

There are 5,377 contracts in our dataset, including 30 originally ERC-violating contracts and 5,347 contracts with injected errors. On average, each contract contains 476.29 lines of code and 2.97 errors. Among these contracts, 5,241 contracts implement ERC20, 110 contracts implement ERC721, and 26 contracts implement ERC1155. ERC20 has the most contracts since it is the most popular ERC.

```
The following code is the implementation of <ERC_type>.
  The <ERC_type> rules are
attached below the code. Does this implementation violate
   <ERC_type> rules?
Return a JSON array containing JSON objects with 'rule'
  and 'function' as keys,
indicating the specific rule content that is violated and
   the function where the
violation resides.
code:"""
<code>
"""
<ERC_type>:
<ERC_content>
```

Fig. 7: The full-rule prompting template. *(For a concrete prompt, <ERC_type> could be ERC20, ERC721, and ERC1155, <code> is replaced with the whole contract code, and <ERC_content> is replaced with the whole ERC document.)*

Figures 4 and Figure 5 compare errors from two sources on their contract sizes and the average number of errors per contract. On average, an originally violating contract contains 260.9 lines of source code, with a standard deviation of 198, whereas a contract with injected errors contains 477.5 lines of code, with a standard deviation of 385. Each originally violating contract contains 4.83 errors on average, and each modified contract contains 2.96 injected errors on average.

The security impact of the errors is tied to the methods used for error injection. A total of 3,645 errors arise from failures to emit events, resulting in contract activities going unlogged. These are considered to have a low-security impact. Among the 2,230 high-security impact errors, a significant number are caused by missing required checks to verify sufficient privileges for performing specific actions. The lack of these checks can be easily exploited, leading to financial losses (*e.g.*, stolen tokens, as illustrated in Figure 1). These errors are categorized as having a high-security impact. Additionally, failures to call required functions or incorrect updates to state variables can also have a high-security impact. The remaining 9,961 errors cause contracts to behave unpredictably for users, although they may not directly result in financial loss, and are classified as having a medium-security impact. All these errors are injected through API, Value, Call, and Return methods.

Figure 6 shows the distribution of errors across different security impacts for the two sources. For errors identified through manual inspection, the proportions of high-impact, medium-impact, and low-impact errors are 9.4%, 29.2%, and 61.3%, respectively. For injected errors, the proportions are 14.1%, 62.9%, and 23.0%, respectively.

IV. Evaluation

This section presents our evaluation results of SC-Bench using GPT-4. Our experiments are designed to answer the following research questions: 1) Coverage: How many errors can GPT-4 detect? and 2) Accuracy: How accurate are GPT-4's detection results?

```
Whether the following smart contract <contract_name>
  violate ERC rule <rule> for function <function_sig>?
  Answer YES or NO.
Code:"""
<code>
"""
```

Fig. 8: The oracle prompting template. *(For a concrete prompt, <contract_name>, <rule>, <function_sig>, and <code> are replaced with the name of the smart contract, the rule's natural language description, the declaration of the function, and the code of the function and all its callees.)*

```
[
    {
        "rule": "approve: Allows _spender to withdraw
  from your account multiple times, up to the _value
  amount. If this function is called again it overwrites
  the current allowance with _value.",
        "function": "approve function is missing in the
  contract"
    },
    {
        "rule": "Approval: MUST trigger on any successful
   call to approve(address _spender, uint256 _value).",
        "function": "Approval event is missing in the
  contract"
    }
]
```

Fig. 9: The GPT response for the contract in Figure 1 with full-rule prompting.

A. Methodology

We evaluate SC-Bench using GPT-4 via the OpenAI API access. We set its temperature value to zero to ensure that GPT-4's results are deterministic, enabling others to replicate our findings. Below, we detail our evaluation methodology.

For each auditing request, we ask GPT whether a contract violates any ERC rules, and if so, which function causes the violation. To instruct GPT on ERC rules, we adopt in-context learning by providing GPT ERC rules in addition to the contract under inspection. We use two methodologies to provide ERC rules. The first presents an ERC's official document with all rules to GPT, as shown in Figure 7. After GPT returns its output, we compare both the GPT-generated violating rules and violating functions to the ground truth. We report when both are correct, when only the violating rule is correct, and when neither is correct.

The second method assumes oracle knowledge of which specific ERC rule(s) a contract violates and the function where the violation happens. So, it directly provides both to GPT and asks GPT a simple True-or-False question, as shown by Figure 8. A True result is correct (True Positive), while a False is wrong (False Negative). For this method, we manually select rules and violation code sites based on either the original contract violations or our injected errors.

Presenting whole ERC rules with the contract serves as a baseline, while hand-picked rules can be viewed as an oracle. Note that we do not provide more advanced methodology such

TABLE III: Experimental results for full-rule prompting. *((x, y, z): x cases where both the reported rule and the reported violating function are correct, y cases where only the reported rule is correct, z cases where neither the rule nor the function is correct. "-": all numbers are zeros.)*

ERC	Manual Inspection				Error Injection				Total
	High	Medium	Low	Total	High	Medium	Low	Total	
ERC20	(16,0,40)	(21,0,36)	(5,2,6)	(42,2,82)	(3,0,3659)	(83,5,7381)	(16,0,533)	(102,5,11573)	(144, 7, 11655)
ERC721	-	-	-	-	(0,0,108)	(0,0,195)	(0,0,17)	(0,0,320)	(0,0,320)
ERC1155	-	-	-	-	(0,0,4)	(0,0,31)	(0,0,8)	(0,0,43)	(0,0,43)
Total	(16,0,40)	(21,0,36)	(5,2,6)	(42,2,82)	(3,0,3771)	(83,5,7607)	(16,0,558)	(102,5,11936)	(144, 7, 12018)

TABLE IV: Experimental results for oracle prompting. *((x, y): x true positives, and y false negatives. "-": all numbers are zeros.)*

ERC	Manual Inspection				Error Injection				Total
	High	Medium	Low	Total	High	Medium	Low	Total	
ERC20	(3,24)	(29,19)	(30,34)	(62,77)	(739,1273)	(2037,7788)	(823,2789)	(3599,11850)	(3661,11927)
ERC721	-	-	-	-	(2, 185)	(1,105)	(0,33)	(3,323)	(3,323)
ERC1155	-	-	-	-	(4,27)	(0,30)	-	(4,57)	(4,57)
Total	(3,24)	(29,19)	(30,34)	(62,77)	(745,1485)	(2038,7923)	(823, 2822)	(3606, 12230)	(3668,12307)

as Chain-of-Thoughts [39], as the focus of this work is to present our collected dataset and demonstrate its potential use. Future research can explore the room for accuracy improvements.

B. Experimental Results

Full-Rule Prompting. As shown in Table III, using full-rule prompting, GPT-4 successfully detects 144 errors (0.9% of the 15,975 errors), providing both the correct violated rules and the violating functions. For another 7 errors (0.04%), GPT-4 only reports the correct rule but fails to identify the correct location. For the remaining 15,824 errors (99%), GPT-4 does not provide any correct information.

For instance, Figure 9 illustrates GPT-4's response when analyzing the contract in Figure 1 using full-rule prompting. GPT-4 accurately identifies two rule violations and the functions responsible for them. Since the contract does not include the function `approve(address _spender, uint256 _value)`, it is considered violating both that the function should overwrite the current allowance with `_value` and that the function should emit the `Approval` event. However, GPT-4 fails to recognize that the contract fails to ensure the `transferFrom()` function properly validates its caller's privileges, missing a critical violation with a high security impact. In another example, when asked to audit the contract in Figure 3, GPT-4 fails to detect that the `balanceOf(address account)` function does not return the expected value as required by ERC20.

We further separate the results between errors identified through manual inspection and those that are injected. For manually inspected errors, GPT-4 correctly reports both the violated rule and the violating function in 30% of the 139 errors. For injected errors, this proportion is 0.6%. The difference is probably due to that the injected errors are more challenging to identify than those introduced by programmers.

For errors in different security impacts, we notice GPT-4 has a very high detection rate when pinpointing manually introduced errors with a high security impact, with a detection rate to be 59.2%. For errors in other security impacts, the detection rate ranges from 0.13% to 43.8%.

We then examine whether GPT-4's ability to detect violations varies across different ERCs. We found that the detection rate for errors violating an ERC20 rule (0.9%) is higher than for ERC721 (0%) and ERC1155 (0%). Notably, GPT-4 does not identify any errors in contracts implementing ERC721 and ERC1155. This discrepancy is likely because there are significantly more ERC20 contracts, meaning GPT-4 has probably been trained on a larger dataset of ERC20 contracts.

Oracle Prompting. Table IV shows the results with oracle prompting. Under this setting, GPT-4's performance significantly improves, with the detection rate increasing from 0.9% to 22.9%. The improvement varies depending on the source of the errors. For manually inspected errors, the detection rate rises from 30% to 44.6%, while for injected errors, it increases from 0.6% to 22.8%. This boost in detection is mainly due to breaking down a complex task into smaller, more manageable tasks, allowing GPT-4 to focus on each one individually. Those results show that simple prompting techniques even with oracle still have a large room for accuracy improvements.

For example, when presenting both the source code of the `transferFrom(address from, address to, uint256 amount)` function and its callee from lines 6 to 17 in Figure 1, along with the rule description stating that the function "should throw unless the `_from` account has deliberately authorized the sender of the message via some mechanism," and asking GPT-4 whether the rule is violated, GPT-4 correctly answers yes. In another example, when showing the `balanceOf()` function from lines 4 to 7 in Figure 3, along with the rule that the function "returns

the account balance of another account with address _owner," and asking GPT-4 to analyze if the rule is violated, it correctly responds yes.

V. Discussion and Conclusion

We present SC-Bench, the first dataset for smart contract auditing. SC-Bench contains 5,377 real-world smart contracts and 15,975 ERC violations, with two sources of violations: real-world errors and injected errors. Our evaluation of SC-Bench with GPT-4 shows that while ML-based techniques are promising, there is still huge room for improvement.

Notably, SC-Bench has certain limitations. For example, we only include the three most important ERC standards and only inject one to three errors per contract. Both adding more ERC standards and supporting more types of error injection are feasible — something we leave for future work. Another limitation is the imbalance of real-world violations and injected errors. As real-world violations are rarely reported, we believe the imbalance cannot be easily improved. One possible approach is to instruct ML models to create real-world-like errors.

References

[1] Antiersolutions. Smart contract auditing services, 2023. https://www.antiersolutions.com/smart-contract-audit/.

[2] BitInfoCharts. Ethereum (ETH) price stats and information, 2023. https://bitinfocharts.com/ethereum/.

[3] BLOCKHUNTERS. Smart contract audit, 2023. https://blockhunters.io/smart-contract-audit/.

[4] Blockworks. Today's Cryptocurrency Prices by Market Cap, 2023. https://blockworks.co/prices.

[5] 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/.

[6] CertiK. Securing the web3 world, 2023. https://www.certik.com/.

[7] Chainlink. Top 6 Smart Contract Languages in 2023, 2023. https://chain.link/education-hub/smart-contract-programming-languages.

[8] E. Commonwealth. Callisto smart-contract auditing department, 2023. https://github.com/EthereumCommonwealth/Auditing.

[9] Crytic. Erc conformance, 2023. https://github.com/crytic/slither/wiki/ERC-Conformance.

[10] defiprime. Ethereum DeFi Ecosystem, 2023. https://defiprime.com/ethereum.

[11] Dexaran. Erc-20 token standard, 2023. https://dexaran820.medium.com/erc-20-token-standard-7fa2316cdcac.

[12] W. Entriken, D. Shirley, J. Evans, and N. Sachs. Erc-721: Non-fungible token standard, 2018. https://eips.ethereum.org/EIPS/eip-20.

[13] Ethereum. Ethereum, 2023. https://ethereum.org/en/.

[14] Ethereum. Ethereum dapps, 2023. https://ethereum.org/en/dapps.

[15] Ethereum. Ethereum improvement proposals, 2023. https://eips.ethereum.org/erc.

[16] Ethereum. ethereum/ercs, 2023. https://github.com/ethereum/ERCs/blob/master/ERCS/eip-1.md.

[17] Ethereum. Smart contract anatomy, 2023. https://ethereum.org/en/developers/docs/smart-contracts/anatomy.

[18] Etherscan. The ethereum blockchain explorer. https://etherscan.io.

[19] J. Fan, Y. Li, S. Wang, and T. N. Nguyen. Ac/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 508–512, 2020.

[20] ImmuneBytes. Smart contract audit services, 2023. https://www.immunebytes.com/smart-contract-audit/.

[21] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1646–1656, 2023.

[22] P. Labs. Polygon. https://polygon.technology.

[23] Markus Waas. Top 7 Reasons To Learn Solidity Programming ASAP, 2022. https://zerotomastery.io/blog/top-7-reasons-to-learn-solidity-programming/.

[24] S. Palladino. Erc20 verifier, 2019. https://github.com/spalladino/erc20-verifier.

[25] PixelPlex. Smart contract audit, 2023. https://pixelplex.io/smart-contract-audit/.

[26] PolygonScan. Polygon pos chain explorer. https://polygonscan.com.

[27] 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.

[28] Revoluzion. Revoluzion smart contract audit report services, 2023. https://www.revoluzion.io/audit.

[29] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad. Bugs. jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th international conference on mining software repositories*, pages 10–13, 2018.

[30] I. Singh, V. Blukis, A. Mousavian, A. Goyal, D. Xu, J. Tremblay, D. Fox, J. Thomason, and A. Garg. Progprompt: program generation for situated robot task planning using large language models. *Autonomous Robots*, 47(8):999–1012, 2023.

[31] C. Smith. Token standards, 2023. https://ethereum.org/en/developers/docs/standards/tokens/.

[32] 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.

[33] F. Vogelsteller and V. Buterin. Erc-20: Token standard, 2015. https://eips.ethereum.org/EIPS/eip-20.

[34] Wikipedia. Binance, 2023. https://en.wikipedia.org/wiki/Binance.

[35] Wikipedia. Ethereum, 2023. https://en.wikipedia.org/wiki/Ethereum.

[36] Wikipedia. Shiba inu (cryptocurrency), 2023. https://en.wikipedia.org/wiki/Shiba_Inu_(cryptocurrency).

[37] Wikipedia. Tether (cryptocurrency), 2023. https://en.wikipedia.org/wiki/Tether_(cryptocurrency).

[38] 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.

[39] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.