SymGPT: Auditing Smart Contracts via Combining Symbolic Execution with Large Language Models

SHIHAO XIA, The Pennsylvania State University, USA
MENGTING HE, The Pennsylvania State University, USA
SHUAI SHAO, University of Connecticut, USA
TINGTING YU, University of Connecticut, USA
YIYING ZHANG, University of California San Diego, USA
NOBUKO YOSHIDA, University of Oxford, United Kingdom
LINHAI SONG*, SKLP, Institute of Computing Technology, Chinese Academy of Sciences, China

To govern smart contracts running on Ethereum, multiple Ethereum Request for Comment (ERC) standards have been developed, each defining a set of rules governing contract behavior. Violating these rules can cause serious security issues and financial losses, signifying the importance of verifying ERC compliance. Today's practices of such verification include manual audits, expert-developed program-analysis tools, and large language models (LLMs), all of which remain ineffective at detecting ERC rule violations.

This paper introduces *SymGPT*, a tool that combines LLMs with symbolic execution to automatically verify smart contracts' compliance with ERC rules. We begin by empirically analyzing 132 ERC rules from three major ERC standards, examining their content, security implications, and natural language descriptions. Based on this study, SymGPT instructs an LLM to translate ERC rules into a domain-specific language, synthesizes constraints from the translated rules to model potential rule violations, and performs symbolic execution for violation detection. Our evaluation shows that SymGPT identifies 5,783 ERC rule violations in 4,000 real-world contracts, including 1,375 violations with clear attack paths for financial theft. Furthermore, SymGPT outperforms six automated techniques and a security-expert auditing service, underscoring its superiority over current smart contract analysis methods.

CCS Concepts: • **Software and its engineering → Empirical software validation**.

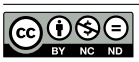
ACM Reference Format:
Shihao Xia, Mengting He, Shuai Shao, Tingting Yu, Yiying Zhang, Nobuko Yoshida, and Linhai Song. 2026. SymGPT: Auditing Smart Contracts via Combining Symbolic Execution with Large Language Models. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 109 (April 2026), 34 pages. https://doi.org/10.1145/3798217

1 Introduction

Ethereum and ERC. Since Bitcoin's creation, blockchain technology has significantly evolved. A key development is Ethereum [40, 117], a decentralized, open-source platform that enables the creation and execution of decentralized applications (DApps) and smart contracts—self-executing

*Corresponding author.

Authors' Contact Information: Shihao Xia, The Pennsylvania State University, State College, USA, szx5097@psu.edu; Mengting He, The Pennsylvania State University, State College, USA, mvh6224@psu.edu; Shuai Shao, University of Connecticut, Storrs, USA, shuai.shao@uconn.edu; Tingting Yu, University of Connecticut, Storrs, USA, tingting.yu@uconn.edu; Yiying Zhang, University of California San Diego, La Jolla, USA, yiying@ucsd.edu; Nobuko Yoshida, University of Oxford, Oxford, United Kingdom, nobuko.yoshida@cs.ox.ac.uk; Linhai Song, SKLP, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, songlinhai@ict.ac.cn.



ACM 2475-1421/2026/4-ART109
https://doi.org/10.1145/3798217

```
1  contract TicketOwnership is IERC721 {
2    mapping(uint=>address) ticketToOwner;
3    mapping(uint=>address) ticketApprovals;
4    mapping(address=>mapping(address=>bool)) _operatorApprovals;
5
6    function transferFrom(address _from, address _to, uint256 _tokenId) public {
7      require(ticketToOwner[_tokenId] == ticketApprovals[_tokenId] &&
       ticketToOwner[_tokenId] != msg.sender);
8      require(ticketToOwner[_tokenId] == _from);
9      require(ticketToOwner[_tokenId] != address(0));
10     require(_to != address(0));
11 +   require(_operatorApprovals[_from][msg.sender]);
12     _transfer(_from, _to, _tokenId);
13   }
14   function _transfer(address _from, address _to, uint256 _tokenId) internal {
15     ticketToOwner[_tokenId] = _to;
16     emit Transfer(_from, _to, _tokenId);
17   }
18   function approve(address _approved, uint256 _tokenId) public {
19     require(msg.sender == ticketToOwner[_tokenId]);
20     ticketApprovals[_tokenId] = _approved;
21     emit Approval(msg.sender, _approved, _tokenId);
22   }
23   function isApprovedForAll(address owner, address operator) public view
       returns (bool) {
24       return _operatorApprovals[owner][operator];
25   }
26   function ownerOf(uint256 _tokenId) external view returns (address) {
27       require(ticketToOwner[_tokenId] != address(0));
28       return ticketToOwner[_tokenId];
29   }
30   function getApproved(uint256 tokenId) external view returns (address) {
31       require(ticketToOwner[tokenId] != address(0));
32       return ticketApprovals[tokenId];
33   }}
```

Fig. 1. An ERC721 contract with a high-security impact ERC violation. (Code simplified for illustration.)

agreements written into code [41, 44]. To govern smart contracts on Ethereum, formal standards known as Ethereum Request for Comments (*ERCs*) have been established [102]. For example, the ERC20 standard defines the rules for fungible tokens [111], whereas ERC721 sets the requirements for non-fungible tokens (NFTs) [39]. ERCs are vital in the Ethereum ecosystem, providing a common set of specifications that ensure interoperability and compatibility among various Ethereum-based projects, wallets, and DApps [41].

ERC violations. Violating ERC rules could result in interoperability issues where the contracts may not work properly with wallets or DApps. It may also introduce security vulnerabilities, financial losses, or even token de-listing from exchanges that require compliance with ERC standards [42].

Figure 1 shows a real-world smart contract that violates a rule required by ERC721. The contract records NFT ownership (in this case, tickets) via `ticketToOwner` in line 2 and enables transfers through `transferFrom()` in lines 6–13. Under ERC721, an NFT may be transferred by its owner, an address approved for that specific NFT (`ticketApprovals` in line 3), or an address authorized to manage all of the owner's NFTs (`_operatorApprovals` in line 4). Consequently, `transferFrom()` must verify that `msg.sender` belongs to one of these categories. In this implementation, the developers intentionally prevent owners from transferring tokens, allowing only a

third-party platform to do so in line 7. They also use the condition "`ticketToOwner[_tokenId] == ticketApprovals[_tokenId]`" in line 7 as a proxy for the owner's intent to sell or transfer a ticket, after the owner calls `approve()` in lines 18–22 with his own address as the first parameter. However, they omit a check for whether the caller is authorized to manage all of the owner's tokens, enabling anyone to steal sellable tickets by transferring them to his own address. The patch in line 11 fixes this issue by using `_operatorApprovals` to verify whether the caller has the permission to manage all of the owner's tickets. The `require` throws an exception and blocks the transaction if the check fails, thereby preventing unauthorized transfers and restoring ERC721 compliance.

State of the art. Following ERC rules is crucial, but developers often struggle due to the complexity of understanding all ERC requirements and corresponding contract code. ERC standards encompass numerous rules, with 132 rules across the three ERCs we study. A single operation involves multiple rules. For example, ERC721 requires the public function `transferFrom()` of all ERC721 contracts complies with six rules, covering the API, caller privilege inspection, input validation, and logging. Failing to check caller privileges leads to the issue above. Moreover, ERC rules are often described in various formats, including both code declarations and natural language descriptions, which further complicates compliance. Contract implementations are often complex, spanning hundreds to thousands of lines of source code across multiple files. Some details are obscured by intricate caller-callee relationships, while others involve numerous objects and functions, potentially written by different developers. Additionally, custom business logic (*e.g.*, preventing NFT owners from transferring their own NFTs in Figure 1) further complicates the code. The complexities in both ERC rules and smart contracts make it difficult to ensure ERC compliance, leading ERC rule violations widely exist in real-world smart contracts [32].

Today's practices to avoid ERC violations are on three fronts. First, existing program-analysis tools can automatically verify certain smart contract criteria [27, 33, 63, 69, 71, 72, 88, 89, 110, 127]. However, their scope is limited, and they often struggle with complex ERC requirements. For example, none can detect the rule violation in Figure 1 without expert-crafted validation rules. This limitation arises because many ERC rules involve semantic information (*e.g.*, caller privileges) that is difficult to infer automatically, and some require contract-specific customization (*e.g.*, locating where caller privileges are stored), a process often time-consuming or even infeasible. Second, security experts offer auditing services [3, 11, 20, 32, 56, 90, 98], providing more thorough validation than program-analysis tools. However, these services are costly and time-consuming, requiring significant manual effort. Third, some approaches rely solely on large language models (LLMs) to verify ERC compliance [53, 84]. However, these methods are prone to LLMs' hallucinations, often resulting in many false positives and false negatives.

Our proposal. This research aims to develop an effective, automated, and cost-effective approach for verifying ERC rules. We begin with an empirical study of three widely used ERC standards, analyzing their 132 rules, focusing on the rules' nature, security risks of non-compliance, and their natural language descriptions. Our study reveals four key insights valuable for Solidity developers, security analysts, and ERC protocol designers. Notably, about one-sixth of ERC rules verify whether an operator, token owner, or recipient has appropriate authority to perform specific operations. Violations of such rules can open attack vectors and result in severe financial losses (*e.g.*, Figure 1). We also find that most ERC rules can be validated within a limited code scope (*e.g.*, a function). Furthermore, we observe that although ERC rules cover diverse contract semantics, they are expressed via a limited set of linguistic patterns by ERC documents.

Building on insights from our empirical study, we develop SymGPT, a fully automated tool for verifying smart contracts' compliance with ERC standards. SymGPT integrates the natural language understanding of LLMs with the formal guarantees of symbolic execution: it leverages an LLM

to extract compliance rules from ERC documents, and then applies symbolic execution to check contracts against these rules. While the pipeline is straightforward in concept, SymGPT tackles two key challenges in novel ways. ***Challenge I***: Symbolic execution leverages input constraints to determine when an ERC rule is violated. However, the space of possible constraints is enormous. How can we leverage LLMs to generate the constraints representing ERC standard violations with minimal hallucinations while preserving full automation? ***Challenge II***: Contracts may implement the same ERC differently, or even omit variables that encode essential ERC properties. How can we adapt symbolic execution to accommodate such variations effectively?

To address *Challenge I*, we design a domain-specific language (DSL) that formalizes ERC rules using code constructs (*e.g.*, contract fields) and execution events (*e.g.*, token burning). Rather than asking the LLM to generate symbolic execution constraints directly, we first have it translate natural language rules into the DSL for each ERC. We then apply deterministic program analysis to convert the translated rules into constraints for each contract. This two-step process ensures the generation of constraints in a correct and consistent format. Moreover, it restricts the LLM to a structured, narrow output space, thereby reducing variability and uncertainty in its responses. To our knowledge, we are the first to leverage an intermediate representation (*i.e.*, the DSL) to enhance LLMs' performance when generating formal specifications.

To tackle *Challenge II*, we introduce a set of constraint variables that capture the key semantic information required by ERC rules. For those that correspond to concrete contract program variables (*e.g.*, the contract field indicating whether an address allows another to manage its tokens in Figure 1), we develop program analysis routines to automatically identify these contract variables across diverse contract implementations. For variables that represent abstract contract execution states without direct contract program counterparts (*e.g.*, whether a function burns tokens), we define rules specifying how their values are updated during symbolic execution. These designs significantly reduce discrepancies among different contract implementations and eliminate the need for contract-specific customization. By integrating these mechanisms, SymGPT distinguishes itself from existing symbolic execution techniques [19, 31, 91]. As far as we know, no prior work validate ERC requirements as many as ours.

We evaluate SymGPT on three datasets: a large dataset of 4,000 contracts randomly selected from etherscan.io [45] and polygonscan.com [92], a ground-truth dataset of 40 contracts with violation labels (created by us), and a dataset of 20 contracts of two previously unstudied ERCs. On the large dataset, SymGPT detects 5,783 ERC rule violations, with 1,375 having a clear attack path leading to financial losses (one shown in Figure 1). The false discovery rate is 2.0%. SymGPT is effective in identifying ERC violations. We compare SymGPT against five static analysis techniques [34, 48, 69, 106, 127], a dynamic testing technique [110], and a human auditing service [32], using the ground-truth dataset. SymGPT detects over two times as many violations as all baselines while producing far fewer false positives, highlighting its superiority. Moreover, SymGPT reduces both time and expenses by a factor of a thousand compared to the human auditing service, emphasizing its cost efficiency. Finally, SymGPT identifies all violations in the third dataset, showing strong generalization to ERCs beyond those we have studied.

In sum, we make the following contributions.

- We conduct the first empirical study on ERC rules made for smart-contract implementations.
- We propose a novel method using a DSL to integrate LLMs with symbolic execution, enabling the generation of properly formatted verification constraints.
- We enhance ERC-rule verification by defining contract state variables and outlining procedures for updating them, addressing gaps in existing program variables.
- We design and implement SymGPT to pinpoint ERC violations, and confirm its effectiveness, advancement, and generality via thorough experiments.

```
1  /// @notice Transfer ownership of an NFT -- THE CALLER IS RESPONSIBLE
2  ///  TO CONFIRM THAT `_to` IS CAPABLE OF RECEIVING NFTS OR ELSE
3  ///  THEY MAY BE PERMANENTLY LOST
4  ///  @dev Throws unless `msg.sender` is the current owner, an authorized
5  ///  operator, or the approved address for this NFT. Throws if `_from` is
6  ///  not the current owner. Throws if `_to` is the zero address. Throws if
7  ///  `_tokenId` is not a valid NFT.
8  /// @param _from The current owner of the NFT
9  /// @param _to The new owner
10 /// @param _tokenId The NFT to transfer
11 function transferFrom(address _from, address _to, uint256 _tokenId) external payable;
```

Fig. 2. Natural language rules for ERC721's `transferFrom()` function. (The rule violated in Figure 1 is highlighted.)

2 Background

This section gives the background of our project, covering Solidity smart contracts, ERCs, techniques related to ours, and the threat model under consideration.

2.1 Ethereum and Solidity Smart Contracts

Ethereum is a blockchain platform that allows developers to write smart contracts for decentralized applications [40, 117]. Users and smart contracts are represented by distinct addresses to send and receive Ether (the native cryptocurrency) and perform complex transactions. Ethereum supports a vibrant digital economy, with Ether prices exceeding $3K and a total market value over $300B [12]. Daily transactions surpass one million on Ethereum, representing $4 billion in value [9]. Smart contracts play a central role in this ecosystem, as they govern most Ethereum's transactions and provide the foundation for advanced functionalities [35, 39, 111].

Solidity is the most widely used language for smart contracts [22, 80]. Its syntax is similar to ECMAScript [38], simplifying interactions with the Ethereum system. Writing a contract in Solidity is similar to defining a class in Java, featuring contract variables to store data and functions to implement logic. Functions can be public, internal, or private; public functions serve as the contract's interface for external access and can be called by any user or contract, while private or internal functions do not. Contracts can also define events, which are emitted during execution and are recorded on-chain for off-chain analysis.

Figure 1 illustrates an example contract. It contains three fields (lines 2–4) and two events defined in the `IERC721` interface, which are emitted in lines 16 and 21. Public function `transferFrom()` (lines 6–13) can be invoked by any Ethereum user or contract, whereas the internal function `_transfer()` (lines 14–17) is restricted to calls within the same contract.

2.2 Ethereum Request for Comment (ERC)

ERCs are technical documents that define how smart contracts should be implemented, ensuring they work consistently across different contracts, applications, and platforms, thus fostering the Ethereum ecosystem [42, 43, 103].

An ERC usually starts with a short motivation. For example, ERC721 introduces a standard interface that allows wallets and brokers to interact with NFTs on Ethereum [39]. It then provides a detailed specification, listing the required public functions and events along with their parameters, return values, and optional attributes. In addition, an ERC specifies requirements for each function or event through plain text or code comments before its declaration. For instance, Figure 2 shows a snippet of ERC721. It specifies that all ERC721-compliant contracts must implement the public function `transferFrom(address _from, address _to, uint256 _tokenId)` to transfer the NFT identified by `_tokenId` from `_from` to `_to`, along with four additional requirements on the

function's behavior: verifying the caller is authorized (either the owner, an address approved for the specific NFT, or an operator approved for all of the owner's NFTs), checking `_from` is the current owner, ensuring `_to` is not the zero address, and confirming `_tokenId` refers to a valid token.

Violating ERC rules can lead to serious financial losses and unexpected contract behavior. For instance, ERC721 requires `safeTransferFrom()` to call `onERC721Received()` when sending NFTs to a contract, and to check the return is a specific magic value. This ensures the receiver is able to handle the NFTs. Without this check, NFTs sent to an incompatible contract are permanently locked. Similarly, failing to verify that the caller is authorized to manage all of an owner's tokens in Figure 1 enables attackers to steal any tradable tokens. In short, strict compliance with ERC rules is essential to safeguard assets and guarantee correct contract behavior.

2.3 Related work

Several tools exist for detecting ERC rule violations, but their coverage is limited. Slither [34] provides dedicated checkers (*e.g.*, arbitrary-send-erc20, slither-check-erc) to assess ERC compliance. These mainly verify the presence of required functions and events, check event emissions, and analyze contracts interacting with ERC-compliant contracts. However, they cannot capture complex rules, such as the one violated in Figure 1. The ERC20 verifier [88] performs checks similar to Slither but focuses only on ERC20. AChecker [48] identifies instances where access-control guards are entirely absent or where contract fields used in such guards can be freely modified by contract users. ZepScope [69] derives rules from OpenZeppelin's code and validates their use in contracts built on OpenZeppelin. Zepcompare [70] detects vulnerabilities resulting from the use of buggy OpenZeppelin code. VerX [89] checks whether smart contracts satisfy project-specific properties. While they address some ERC rules (*e.g.*, access control), they miss many others (*e.g.*, required event emissions). NFTGuard [127] detects only five error types in ERC721, far fewer than the full ERC specification. Techniques for detecting inconsistencies or invariants also fall short: they either group message callers too coarsely [72], cover a subset of ERC-required functions [8, 27, 48, 57], assume partial correctness of code [7, 71], or depend on expert-crafted rules [1, 2, 15, 21, 24, 51, 60, 63, 82, 101]. Some of the techniques (*e.g.*, Certora [21], solc-verify [51], VERISOLID [82]) formalize only a limited subset of ERC rules, far fewer than those defined in the standards. To our knowledge, no prior work covers as many rules across ERC20, ERC721, and ERC1155 as we do. ERCx [110] checks compliance via unit tests, but the tests are incomplete and often fail to expose violations. Some customized ChatGPT-based auditing services [53, 84] have also emerged, but due to the large context of ERC documents, the size of contract code, and LLM hallucinations, they perform poorly when detecting ERC rule violations.

Researchers have also developed automated tools to identify other types of Solidity smart contract bugs, including reentrancy bugs [16, 30, 58, 59, 63, 68, 76, 94, 99, 109, 124, 135], nondeterministic payment bugs [64, 112], consensus bugs [28, 128], eclipse attacks [79, 122, 123], out-of-gas attacks [47, 49], errors in DApps [37], cryptographic errors [133], accounting errors [132], state-reverting errors [50, 65], documentation errors [137], centralization errors [67, 77], unprotected self-destruction [23, 62, 106, 129], integer bugs [6, 25, 81, 83, 107, 108, 115, 130], oracle manipulation attacks [36, 95], flash loan attacks [29], event-ordering bugs [61], fairness issues [73], and exploitable errors [134]. However, these techniques do not focus on ERC-specific semantics and cannot detect ERC rule violations.

Security experts audit smart contracts to detect vulnerabilities and logic flaws [3, 11, 20, 32, 56, 90, 98], with some also verifying ERC compliance. However, these services are costly and time-consuming, making them less appealing to developers than automated tools.

Prior work has applied LLMs (or machine learning) to analyze Solidity code for vulnerability detection [52, 66, 74, 75, 78, 100, 105, 136], bug fixing [55, 131], bug reproduction [114], and exploit

Table 1. ERC rules' content and security impacts.

content \ impact	High	Medium	Low	Total
Privilege Check	24	0	0	24
Functionality	12	30	0	42
API	0	33	0	33
Logging	0	0	33	33
Total	36	63	33	132

generation [104, 121]. Others have leveraged LLMs to generate verification specifications for Rust programs [26, 125]. In contrast, our goal is to validate contracts against a broad range of ERC implementation rules, tackling a problem distinct from these prior techniques.

In summary, existing program-analysis-based techniques either provide limited ERC compliance checks or target unrelated bugs. Manual audits are thorough but are costly and time-consuming. LLM-based approaches show promise for general error detection but struggle to identify specific ERC rule violations. Our research overcomes these limitations by developing a fully automated, end-to-end solution for verifying compliance with a substantial fraction of ERC rules.

2.4 Threat Model

In our study, attackers are not required to obtain elevated privileges (*e.g.*, compromising the Ethereum network, stealing private keys). Instead, as long as they have sufficient gas to invoke public functions of smart contracts deployed on Ethereum, they are capable of exploiting ERC rule violations to launch attacks.

3 Empirical Study on ERC Rules

From the 102 finalized ERCs, we select ERC20 [111], ERC721 [39], and ERC1155 [96] as our study targets. They are technical standards for fungible tokens (*e.g.*, cryptocurrencies), non-fungible tokens (NFTs), and contracts managing both. Our selection is based on their popularity, their complexity, and their significance in the Ethereum ecosystem [5, 10, 13, 87, 97, 116, 118–120]. For example, in the last 180 days, around 3.6 million contracts implementing ERCs were deployed, including 2.6 million for ERC20, 0.6 million for ERC721, and 0.3 million for ERC1155.

We carefully review the specification section of each ERC official document and manually identify a natural language sentence as a rule if it is related to contract implementations, specifies a clear restricting target, and provides actionable checking criteria. Certain rules explicitly use terms like "must" or "should" to convey obligations. We identify a total of 132 rules across the three ERCs: 32 from ERC20, 60 from ERC721, and 40 from ERC1155.

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 natural language? The goal is to garner insights for building techniques that can automatically detect rule violations. To ensure objectivity, two authors independently analyze each ERC and then discuss their findings to resolve any disagreements. In sum, the empirical study takes roughly four human-weeks of effort.

3.1 Rule Content (What)

An ERC rule generally requires a public function to include a specific piece of code. Based on the semantic nature of the code, we categorize the rules into four groups, as shown in Table 1.

Privilege Checks. Regarding the required code patterns, 20 rules involve checking a condition and throwing an exception if it fails, while others require calling a function for a subsequent check. For the object being checked in each rule, 10 rules pertain to the operator (*e.g.*, `msg.sender`) of a token operation. For example, the implementation in Figure 1 violates the highlighted rule in

Figure 2, which requires verifying `msg.sender` is authorized to transfer the NFT. Additionally, three rules address whether the token owner holds sufficient privileges. For example, ERC1155 mandates `safeTransferFrom()` only executes when the source address holds enough tokens. The remaining 11 rules focus on transfer recipients. For instance, ERC721 prohibits `transferFrom()` from sending NFTs to the zero address (line 6 in Figure 2). It also requires calling `onERC721Received()` if the recipient is a contract, checking whether the return value matches a magic number, and throwing an exception if it does not.

Functionality Requirements. Five types of code are required by rules in this category. First, 24 rules specify how functions should generate return values. For example, ERC1155 requires public function `balanceOf(address _owner, uint256 _id)` to return the amount of tokens of type `_id` owned by `_owner`. In particular, when a function returns a Boolean value, it implicitly requires returning `true` on success and `false` otherwise. Second, 12 rules address how to validate input parameters and when to throw exceptions. For instance, ERC20 dictates that `transferFrom()` treats zero-token transfers the same as non-zero transfers. Third, two rules explicitly mandate the associated function to throw an exception when any error occurs. Fourth, three rules specify how to update particular variables. For instance, one ERC20 rule requires `approve(address _spender, uint256 _value)` to overwrite the allowance value that the message caller allows `_spender` to manipulate with `_value`. The remaining rule is from ERC1155 which allows transferring multiple token types in one transaction but requires the balance update for each input token type to follow their order in the input array.

API Requirements. The three ERCs mandate 33 public functions for contract interaction. To ensure compatibility, developers must implement these APIs as specified.

Logging. ERCs enforce logging by requiring event emissions. For example, ERC721 mandates emitting a `Transfer` event whenever a transfer occurs (*e.g.*, line 16 in Figure 1). In total, 33 rules govern logging: 24 specify when to emit events, eight further define required parameters, and nine address event declarations.

Insight 1: *ERC rules encompass diverse contract semantics, making it challenging to develop program analysis techniques that cover these semantics and detect rule violations.*

We further study the valid scope for each rule. Among the 132 rules, 106 rules are confined to a single function. For instance, ERC721 mandates `transferFrom()` in Figure 1 scrutinizes whether the message caller is authorized to handle the token owner's tokens (the highlighted line in Figure 2). Moreover, nine rules pertain to event declarations. The valid scopes of the remaining 17 cases encompass the entire contract. For instance, for every token transfer, both ERC20 and ERC721 mandate emitting a `Transfer` event.

Insight 2: *Most ERC rules can be checked within a function or at a declaration site, and there is no need to analyze the entire contract for compliance with these rules.*

Insight 1 focuses on the requirements mandated by ERC rules, while Insight 2 highlights the code regions where these requirements should be inspected. They represent two orthogonal dimensions of ERC rule validation. Although many ERC rules can be checked within a limited scope, their validation remains challenging due to the wide variety of contract semantics involved.

3.2 Violation Impact (Why)

We analyze the security risks of rule violations to understand the rules' rationale, and categorize their impacts into three levels, as shown in Table 1.

High. A rule is considered high-impact if violating it allows attackers to craft malicious inputs for a public function to trigger unauthorized token transfers, incorrect token balances or allowances, or permanent token loss. Such violations present a direct attack path leading to financial losses.

Table 2. Linguistic Patterns. ([*]: an optional parameter. Subscript [root] marks the root of a sentence. Table 7 in the appendix lists all possible words for each symbol.)

ID	Patterns	Total
TP1	[SUB] [MUST] $\text{THROW}_{[root]}$ COND	28
TP2	ACTION MUST $\text{THROW}_{[root]}$	2
TP3	CALLER MUST $\text{APPROVE}_{[root]}$ ACTION	2
TP4	ACTION $\text{BE}_{[root]}$ INVALID	2
CP1	COND SUB MUST $\text{CALL}_{[root]}$ SUB [, VAR MUST ASSIGN VALUE]	4
EP1	[EVENT] [MUST] $\text{EMIT}_{[root]}$ [COND] [, VAR [MUST] ASSIGN [PREP] VALUE]	15
EP2	[ACTION] [MUST] $\text{EMIT}_{[root]}$ EVENT [PREP VAR ASSIGN VALUE COND]	9
RP1	SUB [MUST] $\text{RETURN}_{[root]}$ VALUE [COND]	24
AP1	[SUB] [MUST] $\text{ASSIGN}_{[root]}$ VALUE [PREP] VALUE	2
AP2	VALUE [MUST] $\text{ASSIGN}_{[root]}$ PREP VALUE	1
OP1	ACTION MUST $\text{FOLLOW}_{[root]}$ ORDER	1
Total		90

As shown in Table 1, 36 rules fall into this category, including all rules related to privilege checks. For example, failing to verify an operator's permissions allows NFT theft (*e.g.*, Figure 1), while not checking a recipient address is non-zero causes tokens to be lost forever.

Among the functionality requirement rules, violating 12 can also lead to financial losses. Of these, nine rules outline how to generate return values representing token ownership or privileges to operate tokens, such as `balanceOf()` returning an address's token balance. Errors that provide incorrect returns for these functions can trap tokens in an address or allow unauthorized token manipulation. The remaining three rules ensure proper token ownership updates. For instance, ERC20 requires function `approve(address _spender, uint256 _value)` to overwrite the amount of tokens `_spender` authorizes the message caller to manipulate with `_value`.

Medium. A rule has a medium impact if its violation causes unexpected contract or transaction behavior but does not create a direct path to financial losses. For example, if a public function's API fails to meet its ERC declaration, invoking the function with a message call following the requirement would trigger an exception. Another example is ERC20's requirement that `transferFrom()` treats zero-token transfers the same as non-zero transfers. If this rule is violated, the contract's 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 losses, emphasizing the urgency of detecting and addressing these violations.*

3.3 Linguistic Patterns (How)

Of the 132 rules, 42 specifically address function or event declarations using Solidity source code. The remaining 90 are described in natural language. As shown in Table 2, we identify 11 linguistic patterns used to express the 90 rules. These patterns correspond to six types of code implementations, as indicated by their ID prefixes (column ID in Table 2): TP indicates throwing or not throwing an exception under certain conditions; CP involves calling a function, possibly with specific argument requirements; EP denotes emitting an event under certain conditions, potentially with argument requirements; RP involves returning a required value under certain conditions; AP refers to updating a variable with a new value; and OP is for performing actions in a specific order.

Out of the 11 patterns, four (TP1, EP1, EP2, and RP1) account for over 80% of the rules. TP1 is primarily used for privilege checks. For example, the rule highlighted in Figure 2 falls under TP1.

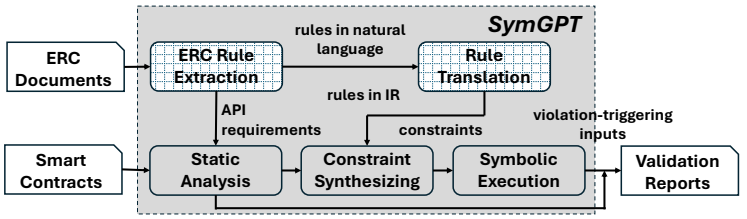


Fig. 3. The workflow of SymGPT. (Components with a pattern background are powered by an LLM. Example outputs for Rule Translation and Constraint Synthesizing are shown in Figures 6 and 8c.)

EP1 and EP2 are used to emit events, while RP1 specifies return values. In contrast, the remaining patterns cover many fewer rules. For instance, AP2 and OP1 are each associated with only one rule.

Insight 4: *Most ERC rules follow common linguistic patterns, while a few are uniquely specified.*

4 The Design of SymGPT

SymGPT takes as input ERC documents written in natural language and smart contract source code. It automatically determines whether and where the contracts violate any rules defined in the ERCs, *without any human intervention.* This makes it useful during in-house development for ensuring ERC compliance before deploying contracts on Ethereum.

Figure 3 illustrates SymGPT's workflow and architecture, which consists of five components. The two patterned components process ERC documents by *extracting* ERC rules from them (Section 4.1) and *converting* those rules into an intermediate representation (IR) (Section 4.2). Since ERC rules encompass diverse aspects of contract semantics (see Insight 1 in Section 3), manually extracting and translating them is both tedious and error-prone. We therefore employ an LLM to automate these two stages, leveraging its strong natural language understanding capabilities [126]. This fully automated pipeline also simplifies extending SymGPT to support new ERCs (see Section 5.4). The remaining three components analyze each individual contract by 1) checking that function and event declarations comply with ERC requirements, while gathering information for the next step (Section 4.4); 2) translating ERC rules from the IR into contract-specific, rule-violation constraints (Section 4.3); and 3) running symbolic execution to determine whether the contract satisfies these constraints, and reporting any violations (Section 4.4).

The rest of this section follows the workflow to explain SymGPT's technical details in depth.

4.1 ERC Rule Extraction

SymGPT focuses on rules outlined in ERC documents. While some correctness and performance rules can be identified and extracted from smart contract code [54, 69], validating compliance with them is beyond this paper's scope. Furthermore, the experimental results in Section 5.2 show that the rules extracted from contract code cover only a small subset of ERC requirements.

A naïve way to extract rules from an ERC document with an LLM is to provide the entire document and ask the LLM to identify all rules. However, this often leads to incomplete or inaccurate extraction. Instead, we break each ERC document into subsections and process them separately. Specifically, since each ERC's rules are detailed in the specification section and precede the relevant function or event declarations, we use regular expressions to isolate the specification section and split the section into smaller parts, ending at each function or event declaration. We then instruct the LLM to analyze each declaration along with its preceding text description.

We design prompts based on the linguistic patterns in Table 2. Each prompt begins with an introduction, followed by a set of linguistic patterns sharing the same ID prefix. It then presents

```
Given an ERC, which contains a list of functions, events, and rules. Each function has its own
 rules, usually above their Solidity declaration. Rules indicating throw or check usually have
 patterns like
"""
[sub] [must] throw condition
action must throw
caller must be approved action
action are considered invalid
"""
Other patterns that serve similar purposes also count.

Rules:"""
{{description}}
{{API}}
"""
List the conditions that need to be thrown or checked in the given rules for function
{{API}} in a JSON array with the following format:

if: <string, condition to throw or check>
throw: <boolean, whether it needs to throw or should not be thrown>

If there is no such rule, simply left empty.
```

Fig. 4. The template for prompts extracting TP rules. ("{{description}}": text descriptions precede a function declaration in an ERC document. "{{API}}": the declaration of the function.)

the text description and declaration of a function (or an event) and asks the LLM to extract rules from the description, including any relevant value requests (*e.g.*, emitting an event with a specific parameter), based on the patterns. Finally, the prompt explains the JSON schema for formatting the extracted rules. Including linguistic patterns helps the LLM better understand what ERC rules are, improving extraction accuracy. Figure 4 illustrates the prompt template used for rules in the TP linguistic-pattern group. The rule violated in Figure 1 can be accurately extracted using a prompt instantiated from this template, where the placeholders "{{API}}" and "{{description}}" are replaced with the declaration of `transferFrom()` and the textual description preceding it in the ERC document, both of which are shown in Figure 2. While summarizing linguistic patterns requires effort, most rules follow a limited number of patterns (see Insight 4 in Section 3.3), making the patterns likely to apply to ERCs that we have not studied.

4.2 ERC Rule Translation

Generating constraints directly from natural-language rules poses significant challenges for LLMs, primarily for two reasons. First, the vast space of possible constraints introduces substantial uncertainty and increases the likelihood of hallucinations. Second, different contracts implement the same ERC in diverse ways, leading some rules to depend on contract-specific details to be properly validated. Generating constraints for such rules would either require costly per-contract analysis by the LLM or risk reduced accuracy if relevant information is omitted.

To overcome these challenges, we define a domain-specific language (DSL) as an intermediate representation (IR) for ERC rules and prompt the LLM to translate each rule into this IR. We then

```
1 <check> ::= <throw> | <call> | <emit> | <assign> | <follow>
2 //Lines 3--7: five top-level non-terminals
3 <throw> ::= if <b_exp> then checkThrow(<fun>, <flag>)
4 <call> ::= if <b_exp> then checkCall(<fun>, <fun>) [with <b_exp>]
5 <emit> ::= if <b_exp> then checkEmit(<fun>, <event>) [with <b_exp>]
6 <assign> ::= checkEndValue(<fun>, <value>, <value>)
7 <follow> ::= checkOrder(<op>, <op>, <op>, <op>)
8
9 <b_exp> ::= not <b_exp> | <b_exp> (and | or) <b_exp> | <value> <b_op> <value> |
      <change> | <mint> | <burn>
10 <value> ::= msg.sender | <field_value> | <para> | ...
11 <field_value> ::= getFieldValue(<field> [,<key>]*)
12 <field> ::= getField(<anchor_fun>)
13 <para> ::= getPara(<fun> | <event>, <index>)
14 <change> ::= checkChange(<fun>, <field_name> [, <key>]*)
15 <mint> ::= checkMint(<fun>, <field_name>)
16 <burn> ::= checkBurn(<fun>, <field_name>)
```

Fig. 5. The EBNF grammar of the domain-specific language. (Terminals in blue are literal tokens and those in violet are utility functions. The grammar is simplified due to limited space.)

perform program analysis on individual contracts to instantiate contract-specific constraints from the IR (Section 4.3), thereby removing the need for the LLM to analyze each contract individually. We adopt extended Backus–Naur form (EBNF) to specify the grammar of the DSL, because ERC rules primarily describe interface-level behaviors and control-flow requirements, both of which can be naturally expressed using EBNF. Other alternatives (*e.g.*, regular expressions, Datalog) are either insufficiently expressive to capture the semantics involved in ERC rules or excessively powerful, introducing unnecessary complexity.

Domain-Specific Language. Figure 5 shows the language grammar, which iteratively expands the non-terminals (symbols enclosed by "<>") into terminals (those not enclosed by "<>"). The grammar contains five top-level non-terminals in line 1, corresponding to five linguistic-pattern groups in Table 2. All groups are covered except the one related to return value generation. This omission stems from the difficulty of using contract elements explicitly required by ERCs to define the required return values. For instance, ERC20 requires the `name()` function of every ERC20 contract to return the name of the token the contract manages, but does not mandate where the name must be stored, making it difficult to formalize this rule across diverse ERC20 implementations.

The terminals include contract elements (*e.g.*, public functions and their parameters) defined by ERCs, utility functions (shown in violet in Figure 5), and literal tokens (*e.g.*, `if`, and, shown in blue in Figure 5). Utility functions are analysis routines defined by us. Their names are carefully chosen to reflect their exact functionalities, aiding the LLM in understanding them and performing the formalization. For example, `checkThrow(<fun>, <flag>)` returns a Boolean value, indicating whether contract function `<fun>` throws an exception or not as indicated by `<flag>`.

Non-terminal <throw>. `<throw>` in line 3 formalizes how to check rules requiring a function to throw or not throw an exception under a specific condition. For one such rule, both non-terminal `<b_exp>`, representing the condition, and the parameter of the utility function `checkThrow(<fun>, <flag>)` need to be specified.

How to extend `<b_exp>` is shown in lines 9–16. In line 9, a `<b_exp>` can be a compound Boolean expression, a comparison between two `<value>`s, or the returns of three utility functions. Line 10 denotes a `<value>` can represent `msg.sender`, a contract field's value, or other similar elements. Utility function `getFieldValue()` in line 11 retrieves a contract field's value, taking the field name and optional keys (for array or mapping) as input. ERCs do not impose any requirements on field

```
1 //function transferFrom(address _from, address _to, uint256 _tokenId)
2 if msg.sender != getFieldValue(getField(`ownerOf()`), getPara(`transferFrom()`,
      2))
3    and msg.sender != getFieldValue(getField(`getApproved()`), getPara('
      transferFrom()', 2))
4    and not getFieldValue(getField(`isApprovedForAll()`), getPara(`transferFrom
      ()`, 0), msg.sender)
5 then
6   checkThrow(`transferFrom()`, true);
```

Fig. 6. The rule violated in Figure 1 in the DSL.

```
For {{API}}
rule: {{rule}}
{{args}}
By using the following JSON schema of the configuration for the rule verification:
{{DSL}}
{{anchors}}
Generate the JSON for the rule.
```

Fig. 7. The template for prompts translating a rule in natural language into the DSL. ("{{API}}": the source-code declaration of a function or an event. "{{rule}}": an extracted natural language rule. "{{args}}": possible arguments for the rule. "{{DSL}}": the JSON schema of the top-level non-terminal. "{{anchors}}": the list of all public, read-only functions required by the ERC.)

names but do require the names of certain public functions that return the values of contract fields. We consider these functions as anchor functions to identify field names. For example, function `isApprovedForAll()` in lines 23–25 of Figure 1 is explicitly required by ERC721, and it is the anchor function to identify the contract field tracking whether one address allows another to manage all its tokens. We treat all public, read-only functions required by ERCs as potential anchor functions, and request the LLM to determine any specific anchor function for each rule when it translates the rule into the DSL. Utility function `getField()` in line 12 takes an anchor function as input and outputs the name of the field returned by the anchor function.

For example, Figure 6 shows the rule violated in Figure 1 in the DSL. This rule applies to contract function `transferFrom()`, whose declaration appears in line 1. In lines 2, 3, and 4, the LLM identifies contract functions `ownerOf()`, `getApproved()`, and `isApprovedForAll()` as the anchor functions for the contract fields `ticketToOwner`, `ticketApprovals`, and `_operatorApprovals`, respectively. The utility function `getField()` analyzes these contract functions to extract the associated field names. Since these fields are mappings, utility function `getFieldValue()` requires one or two additional keys as parameters. The condition checks whether `msg.sender` is the token owner, the address approved for the token, or the address authorized to manage all of the owner's tokens. If `msg.sender` does not satisfy any of these cases, `transferFrom()` must throw an exception, which is verified by the utility function `checkThrow()` in line 6.

Non-terminals <call> & <emit>. These two check whether a contract function calls another function or emits an event under specific conditions. They also ensure that the parameters of the called function or emitted event meet the required criteria. For example, ERC1155 requires a function to emit a `TransferSingle` event with the event's second parameter set to zero when minting tokens. To enforce this rule, we use the utility function `checkMint(<fun>)` in line 15 of Figure 5 as the

Table 3. Translations from utility functions into constraints.

Utility Functions	Constraints
checkThrow(<fun>, <flag>)	$TH = flag$
checkCall(<fun>, <fun2>)	$CA_{fun2} = true$
checkEmit(<fun>, <e>)	$EM_e = true$
checkEndValue(<fun>,<v1>,<v2>)	$v_1 = v_2$
checkChange(<fun>, <f>)	$BC_f = true$
checkMint(<fun>, <f>)	$(BI_f \wedge \neg BD_f) = true$
checkBurn(<fun>, <f>)	$(\neg BI_f \wedge BD_f) = true$
checkOrder(<op1>,<op2>,<op3>,<op4>)	$(O_{op1} < O_{op2} \wedge O_{op3} < O_{op4}) \vee (O_{op1} > O_{op2} \wedge O_{op3} > O_{op4})$
getField(<anchor_function>)	contract field returned by <anchor_function>
getPara(<fun>, <i>)	the ith formal parameter of <fun>
getFieldValue(<f>(, <key>)*)	the value of <f> or the element of <f> indexed by <key>

first <b_exp> of <emit> to verify that the analyzed contract function only increases token balances without decreasing them, indicating a token minting action. Once confirmed, we then check if the contract function emits the required event using checkEmit(<fun>, <event>). If this is also confirmed, we proceed to verify that the second parameter of the emitted event is zero, specified by the second <b_exp>.

Non-terminal <assign>. <assign> leverages utility function checkEndValue(<fun>, <value>, <value>) in line 6 of Figure 5 to perform an unconditional check. The utility function inspects whether the first <value> matches the second at the end of contract function <fun>.

Non-terminal <follow>. <follow> checks whether two pairs of operations are in the same order using utility function checkOrder().

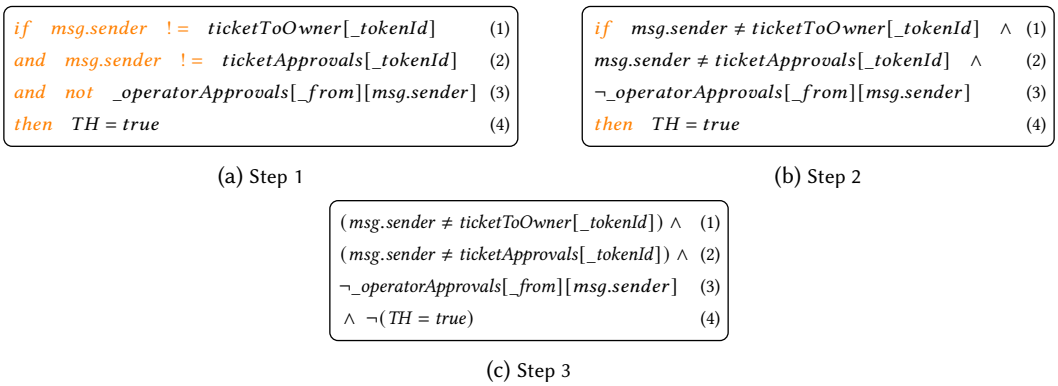
Prompting the LLM. Some ERC rules apply directly to function and event declarations and can be inspected without translation. For other rules, we use the LLM to translate each one individually. Figure 7 shows the prompt template. It begins with a function or event's source code declaration, followed by an extracted natural language rule for the function or event and any possible arguments. The template then specifies the JSON schema for the top-level non-terminal of the rule to help the LLM understand the grammar of the DSL. Next, the template introduces all public, read-only functions required by the ERC as possible anchor functions. Finally, it instructs the LLM to translate the rule into the DSL, according to the JSON schema.

4.3 Synthesizing Violation Constraints

This component analyzes individual contracts and translates rules in the DSL into constraints enriched with contract-specific information. These constraints specify the conditions under which the corresponding rules are violated, and are then examined by the symbolic execution engine.

The core innovation lies in the design of a suite of constraint variables that precisely capture the diverse semantics of ERC rules. Some variables correspond directly concrete contract variables, while others represent abstract contract states that have no explicit source code counterparts. We further develop program analysis mechanisms that identify the former when translating DSL rules into contract-specific constraints, and define how to update the latter during symbolic execution. Using state variables simplifies the solver's input compared to using constraint predicates or functions to represent contract states, as state variables' values are determined during symbolic execution. Together, these capabilities enable SymGPT to handle the variation across various contract implementations, distinguishing it from existing symbolic execution techniques [19, 31, 91].

Constraint Variables. For each public function *fun*, we define six types of *state variables* to track whether *fun* or any of its callees perform specific actions, thereby transitioning into the



(a) Step 1

(b) Step 2

(c) Step 3

Fig. 8. Results of the three Steps in constraint generation. (DSL elements are shown in orange, and constraint elements are shown in black.)

corresponding states. These variables play a crucial role in representing ERC violation constraints. The symbolic execution engine initializes all state variables to `false` and updates them to `true` once the actions are conducted by *fun*.

- *TH* denotes whether *fun* throws an exception. We consider exceptions raised by `require`, `revert`, `assert`, and `throw`. Exceptions triggered inside modifiers are naturally supported via inter-procedural analysis.
- EM_e denotes whether *fun* emits event *e*.
- CA_{fun2} denotes whether *fun* invokes function *fun2*.
- BI_f, BD_f and BC_f denote whether *fun* increases, decreases, or modifies field f, respectively.

Moreover, we define an *O* variable for each instruction, representing the instruction sequence along an execution path. We define value variables for contract fields, formal and actual function parameters, event parameters, local variables, and values defined by Ethereum (*e.g.*, `msg.sender`).

Generating Constraints. We generate constraints in three steps. In the first two steps, DSL elements are replaced by constraint elements, while in the final step, the composition structures are translated. Figure 8 illustrates the results of these three steps when translating Figure 6.

First, we perform static analysis on each contract and recursively translate all utility functions from the innermost to the outermost level, following the rules summarized in Table 3. Using Figure 6 as an example, we replace `getField('ownerOf()')` with `ticketToOwner`, since this field is the one returned by `ownerOf()` (see line 28 in Figure 1). Similarly, `getField('getApproved()')` and `getField('isApprovedForAll()')` are replaced with `ticketApprovals` and `_operatorApprovals`, respectively. We then replace all three `getPara()` invocations with the constraint variables representing the formal parameters of function `transferFrom()`. Next, we substitute each `getFieldValue()` call with the corresponding field lookup operation, using the formal parameters and `msg.sender` as keys. Figure 8a shows the result of this step.

Second, apart from `if`, `then`, and `with`, we replace all other DSL terminals with corresponding constraint variables, as well as Boolean and numerical operators for constraints. Figure 8b shows the result of this step when translating Figure 6.

Third, we group existing constraints into `if`'s condition (Φ_{if}) (*e.g.*, lines 1–3 in Figure 8b), the check part (Φ_{check}) (*e.g.*, line 4 in Figure 8b), and `with`'s condition (Φ_{with}), and eliminate `if`, `then`, and `with` using the rules in Table 4. The intuition is that if a rule requires an action under a condition, its violation happens when the condition is met but the action is not performed (*e.g.*, `<throw>`), or the action is performed, but with a wrong parameter (*e.g.*, `<emit>`). For example, the rule in Figure 6 follows the pattern "`if` Φ_{if} `then` Φ_{check}." According to the first row of Table 4, we eliminate

Table 4. Transformation rules used in the third step for constraint generation.

Non-Terminal	Step 2	Step 3
<throw>	if Φ_{if} then Φ_{check}	$\Phi_{if} \wedge \neg\Phi_{check}$
<call>	if Φ_{if} then Φ_{check} with Φ_{with}	$(\Phi_{if} \wedge \neg\Phi_{check}) \vee (\Phi_{if} \wedge \Phi_{check} \wedge \neg\Phi_{with})$
<emit>		
<assign>	Φ_{check}	$\neg\Phi_{check}$
<follow>		

the `if` and `then` by negating Φ_{check} and conjoining it with Φ_{if} using $\wedge$. The resulting constraints are shown in Figure 8c.

For rules requiring no exceptions under certain conditions, we design validation constraints as "$\neg\Phi_{if} \wedge \neg\Phi_{check}$." A violation is reported only when the violation constraints are met, but the validation constraints are not met, ensuring the exception is indeed due to the specified conditions. Additionally, for `<call>` and `<emit>`, we inspect whether the contracts contain the called functions or emitted events before synthesizing constraints.

4.4 Static Analysis and Symbolic Execution

Static Analysis. The static analysis routines are for two purposes: 1) verifying contracts contain the necessary functions and events and their declarations meet the requirements, and 2) implementing utility functions, such as `getField()` and `getOrder()`.

Symbolic Execution. We use symbolic execution to analyze each rule individually. For rules that apply to a specific public function, we perform symbolic execution directly on that function. For rules that concern the entire contract, we analyze each public function of the contract separately.

Our symbolic execution process is similar to existing techniques [19, 31, 91], but it differs in how we handle state variables. All state variables are initialized to `false` and set to `true` once the corresponding actions are executed. All other variables are treated as symbolic unless their values can be determined through static analysis. At the end of each public function (*e.g.*, upon `return` or an exception), we compute the conjunction of three types of constraints: 1) initial constraints imposed by Ethereum or variable types, 2) constraints enforced by the implementation (*e.g.*, path conditions), and 3) violation constraints synthesized for the rule under verification. If the solver (Z3) finds a satisfying solution for this conjunction, the rule is considered violated. To reduce potential false positives caused by the LLM, we ignore rules that are violated by every contract within an ERC. Additionally, we cap loop iterations and recursion depths at two to mitigate path explosion.

For example, when checking path 6-12-15-16-13 in Figure 1 against the rule in Figure 8c, *TH* is computed as false, since no exception is thrown along the path. The constraints for "`_from`" include "$\_from \geq 0$" due to type requirements and "$ticketToOwner[\_tokenId] = \_from$" from the path condition in line 8. Similarly, the constraints for "`_to`" include "$\_to \geq 0$" and "$\_to \neq 0$". Contract field `ticketToOwner` is updated in line 15, resulting its *BC* being set to true. Z3 finds multiple solutions for the conjunction of these initial constraints, computed constraints, and the violation constraint in Figure 8c. One solution is "$\_from = 11$", "$\_to = 3$", "$\_tokenID = 26285$", "$msg.sender = 3$", "$ticketToOwner[26285] = 11$", "$ticketApprovals[26285] = 11$", and "$\_operatorApprovals[11][3] = false$", indicating the rule is violated after given those values.

5 Evaluation

Implementation. We employ the GPT-5 model [85] as the LLM in SymGPT, interacting with it through OpenAI's official APIs. The model conducts extensive internal reasoning before producing responses [86]. The temperature parameter is fixed at 1.0 and cannot be adjusted. We set the reasoning-effort parameter to high to ensure the model's good performance.

Table 5. Evaluation results on the large dataset. ($x_{(y)}$: x true positives, and y false positives.)

	High	Medium	Low	Total
ERC20	1353_{122}	3686_{0}	172_{0}	5211_{122}
ERC721	18_{0}	22_{0}	502_{0}	542_{0}
ERC1155	4_{0}	12_{0}	14_{0}	30_{0}
Total	1375_{122}	3720_{0}	688_{0}	5783_{122}

The remaining functionalities are implemented in Python, including generating prompts (based on the templates) to extract rules from ERC documents and translate natural language rules into the DSL, synthesizing constraints, and performing symbolic execution. We implement the symbolic execution engine based on Slither [34], a static analysis framework for Solidity. Slither translates Solidity source code into an IR with control-flow and data-flow information to facilitate static analysis, but it does *not* provide the symbolic execution capability. Furthermore, we perform inter-procedural analysis for each call site by using Slither's API to identify the callee function, propagating necessary information (*e.g.*, constraints), and returning computed results back to the caller after analyzing the callee.

Research Questions. Our experiments are designed to answer the following research questions:

- *Effectiveness*: Can SymGPT accurately pinpoint ERC rule violations?
- *Advancement*: Does SymGPT outperform existing auditing solutions?
- *Rationality*: What are the benefits of each component of SymGPT?
- *Generality*: Can SymGPT detect violations for ERCs beyond those studied in Section 3?

Experimental Setting. All our experiments are performed on a server machine, with Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz, 256GB RAM, and Red Hat Enterprise Linux 9.

5.1 Effectiveness of SymGPT

5.1.1 Methodology. We create a *large* dataset of 4,000 *unique* contracts for evaluation, including 3,400 ERC20 contracts, 500 ERC721 contracts, and 100 ERC1155 contracts. The contracts are randomly sampled from etherscan.io [45] and polygonscan.com [92], with the former chosen because of its prominence and the latter selected due to its higher contract deployment compared to other platforms (*e.g.*, Arbitrum [4], BscScan [17]). We discard contracts that lack source code or cannot be compiled during the sampling. The collected contracts were deployed between September 2023 and July 2024. This dataset serves as a random sample of real-world contracts implementing the three ERCs, with enough contracts to support statistically confident conclusions. On average, each contract contains 469.3 lines of source code, with the largest one reaching 7,139 lines and 916 contracts exceeding 1,000 lines. The contracts are comparable in code size to other types of contract (*e.g.*, DeFi), and are suitable for evaluating SymGPT's scalability.

We run SymGPT on the dataset and count the violations and false positives reported by SymGPT to assess its effectiveness. For each flagged violation, we manually review the result of symbolic execution, the rule description, and the relevant contract code to determine its correctness. Each reported violation is reviewed by at least two paper authors, with an agreement rate over 94%. Due to the dataset's size, we do not manually analyze all the contracts to identify ERC rule violations in them. Instead, we sample the top 50 contracts with the highest number of detected violations, as they are more likely to be of lower quality and to contain more violations. Following the ERC standards, we manually identify all violations in these contracts and use them to count the false negatives of SymGPT.

5.1.2 Experimental Results. As shown in Table 5, SymGPT detects 5,783 ERC rule violations while reporting only 122 false positives, highlighting *SymGPT's effectiveness and accuracy in checking*

```
1   contract MyERC1155Token is ERC1155 {
2     mapping(uint256=>mapping(address=>uint256)) _balances;
3     mapping(address=>mapping(address=>bool)) _approvals;
4
5     function safeTransferFrom(address from, address to, uint256 id, uint256
       value, bytes memory data) public {
6 +     require(to != address(0), "invalid receiver");
7 +     require(_approvals[from][msg.sender], "the caller is not approved to
       manage the tokens");
8       _update(from, to, id, value);
9 +     if (to.code.length > 0) {
10 +      try IERC1155Receiver(to).onERC1155Received(msg.sender, from, id, value,
       data) returns (bytes4 response) {
11 +          ...
12 +      }}
13    }
14    function _update(address from, address to, uint256 id, uint256 v) internal {
15      if (from != address(0)) {_balances[id][from] -= v;}
16      if (to != address(0)) {_balances[id][to] += v;}
17    }}
```

Fig. 9. An ERC1155 contract contains four high-security impact violations. (Code simplified for illustration.)

ERC compliance. Furthermore, SymGPT captures all violations in the 50 contracts that are manually examined by the authors, showcasing *its good coverage of ERC violations in real-world scenarios.* In total, SymGPT issues 234 GPT-5 queries in this experiment to extract rules from the three ERCs and translate them into the IR, with a total cost of $5.49.

True Positives. Among the 5,783 detected ERC rule violations, 1,375 have a high-security impact, 3,720 have a medium-security impact, and 688 have a low-security impact. These violations span all three ERC standards, with 5,211 violations for ERC20, 542 for ERC721, and 30 for ERC1155. These results demonstrate *SymGPT's capability to handle rules with varying security impacts across different ERC standards.*

Among the 1,353 high-security violations in ERC20 contracts, 14 are due to not checking whether the message caller of `transferFrom(address _from, address _to, uint256 _value)` has sufficient privileges allowed by `_from` to transfer `_value` tokens. Another 1,266 cases violate the same rule in a different way. Instead of comparing the allowed amount with `_value`, they check if it equals `type(uint256).max`. Furthermore, they do not reduce the allowed amount after the transfer, which creates a backdoor that allows an address to drain any amount of tokens from another account. The remaining 73 violations breach the rule that `transfer(address _to, uint256 _value)` should revert if the caller lacks sufficient tokens. The flawed contracts either use `unchecked` to bypass the underflow check when reducing the caller's balance or include a path failing to decrease the caller's balance, potentially allowing an address to spend more tokens than it owns.

SymGPT identifies 18 high-security violations that do not comply with ERC721 requirements. Exploiting these vulnerabilities could lead to NFTs being stolen or permanently lost at the zero address. Figure 1 shows one of those violations.

SymGPT pinpoints four high-security violations in an ERC1155 contract. The simplified contract code is shown in Figure 9. Function `safeTransferFrom()` in lines 5–13 transfers `value` tokens of type `id` between addresses. Like ERC721, ERC1155 requires the message caller to be approved to manage the owner's tokens. Unfortunately, neither `safeTransferFrom()` nor its callee `_update()` performs this crucial check (via `_approvals`), allowing unauthorized transfers. The patch in line 7 adds a necessary check to ensure the message caller has the owner's approval, aligning with ERC1155. Moreover, ERC1155 mandates `safeTransferFrom()` checks whether the recipient is a

```
1  contract MintPassGELO is ERC1155 {
2   address private _owner;
3   mapping(uint256=>mapping(address=>uint256)) _balances;
4
5   function airDrop(address frm, address[] memory to, uint id, uint cnt) public{
6    require(_owner == msg.sender);
7    for(uint256 j = 0; j < to.length; j++) {
8     _mint(frm, to[j], id, cnt);
9    }
10  }
11  function _mint(address from, address to, uint256 id, uint256 amount) internal
       virtual {
12   _balances[id][to] += amount;
13 -  emit TransferSingle(msg.sender, from, to, id, amount);
14 +  emit TransferSingle(msg.sender, address(0), to, id, amount);
15  }}
```

Fig. 10. An ERC1155 contract failing to emit an event with correct parameters. (Code simplified for illustration.)

contract and, if so, ensures it can handle ERC1155 tokens by calling `onERC1155Received()` on it. Unfortunately, Figure 9 neglects these requirements, potentially trapping tokens in contracts without the token-handling capability. Lines 9–12 show the necessary checks. The remaining violation arises from failing to check if the recipient is address zero. Transferring tokens to address zero leads to irreversible token loss. The violation is fixed by line 6.

The 3,720 medium-security violations stem from several reasons. Of these, 3,611 involve ERC20 rules, where `transfer()` and `transferFrom()` incorrectly throw an exception when transferring zero tokens, instead of treating it as a normal transfer. Additionally, 55 violations result from missing functions required by the ERC. Another 32 violations occur due to missing return values or incorrect return-value types. Finally, 22 violations stem from failing to throw an exception for invalid input parameters.

Of the 688 event-related violations, 672 are due to missing event declarations, 15 fail to emit an event, and one emits an event but with a wrong parameter. Figure 10 shows the last case. According to ERC1155, a `TransferSingle(address indexed _operator, address indexed _from, address indexed _to, uint256 _id, uint256 _value)` event must be emitted whenever tokens are transferred. If the transfer is to mint new tokens, formal argument `_from` must be set to zero. In Figure 10, function `airdrop()` increases `_balances` without decreasing it, indicating tokens are minted. Although line 13 emits a `TransferSingle` event, argument `_from` is incorrectly set to variable `from` instead of zero, thereby violating the ERC1155 rule.

False Positives. *SymGPT demonstrates high accuracy*, reporting 122 false positives across the 4,000 contracts. The false discovery rate is 2.0%. The false positives are due to three reasons. First, 116 occur because the actions required by ERCs are in external contracts, and their code is unavailable during evaluation. Second, five false positives are due to SymGPT's inability to handle assembly code. Third, the remaining false positive arises from an implementation of `transferFrom()` of an ERC20 contract, where the source address is compared with the message caller and an authorization check is only performed if they differ. We consider this a false positive, as the authorization check is skipped only when the caller is transferring her own tokens, making exploitation impossible. Notably, the ERC20 standard does not require implementations to check the caller's authorization only when it differs from the token owner. This flexibility allows scenarios in which a token owner restricts how many of her own tokens she can spend (*e.g.*, to prevent overspending).

False Negatives. The 50 contracts with the highest number of detected violations consist of 42 ERC20 contracts, five ERC721 contracts (including Figure 1), and three ERC1155 contracts (including

Figure 9). Two authors independently examine these contracts against the corresponding ERC standards and identify a total of 254 violations. Among these, 48 have a high-security impact, 158 have a medium-security impact, and 48 have a low-security impact. In total, the contracts violate 38 distinct rules, including 15 ERC20 rules, 9 ERC721 rules, and 14 ERC1155 rules.

Comparing the manually identified violations with those detected by SymGPT, we find that SymGPT successfully detects all violations present in these contracts. While SymGPT is unable to detect certain classes of violations (*e.g.*, violations related to return-value generation) due to design and implementation limitations, such cases are rare in practice. Overall, *SymGPT provides strong coverage of real-world violations in ERC contracts.*

Results of the LLM. We manually inspect the LLM's outputs to assess their impact on SymGPT's results, which takes less than one day. Note that SymGPT itself is fully automated and does not require any manual review or correction of LLM outputs during normal use.

For ERC rule extraction, the LLM successfully identifies all 132 rules across the three ERC documents. Additionally, it mistakenly extracts six non-existent rules, five incorrectly require functions not to throw exceptions under certain conditions, and one pertains to ERC20's `approve()` function, requiring the function to set allowance to zero before reassignment. The latter requirement, however, applies to Ethereum client code that interacts with `approve()`, rather than to the implementation of the function itself. Since these rules are violated by all contracts of the ERCs, SymGPT automatically ignores them (see Section 4.4).

Out of the 138 extracted rules (including the six incorrect ones), 42 are function or event interfaces that do not require translation. Among the remaining 96 rules, the LLM successfully translates 69 but fails on 27 for two reasons. First, 24 rules govern how to generate return values, but we do not define any DSL for return-value rules (see Section 4.2). As a result, SymGPT skips their translation. Second, three rules lack sufficient detail in their textual descriptions, making the LLM not generate anything. For instance, ERC1155 requires `safeTransferFrom()` to throw on any error, but it does not specify all possible errors.

Answer to effectiveness: *SymGPT accurately detects various types of ERC rule violations, many clearly linked to potential financial losses.*

5.2 Comparison with Baselines

5.2.1 Methodology. Since the large dataset lacks "*ground-truth*" labels, we create a small, labeled dataset to compare SymGPT with existing auditing solutions. We randomly select ERC20 contracts audited by the Ethereum Commonwealth Security Department (ECSD), an expert group that reviews GitHub-submitted audit requests and publishes their audit results on GitHub [32]. The selection criteria are: 1) Solidity source code provided, 2) approval by Solidity programmers with the "approved" tag, 3) ERC rule violations identified, and 4) all code in a single contract file. These contracts and the audit results allow us to compare automated tools with expert reviews. As ECSD has limited ERC721 and ERC1155 contracts, we supplement our dataset with samples from ERCx [110]. In total, the ground-truth dataset consists of 40 contracts: 30 ERC20, five ERC721, and five ERC1155, with an average of 553 lines of code per contract. These contracts were either audited by ECSD between January 2019 and October 2023, or analyzed by ERCx between May 2024 and December 2024. After careful inspection, we identify 159 violations: 28 high-impact, 55 medium-impact, and 76 low-impact.

We compare SymGPT against five static analysis tools, including Slither's ERC-related checkers[1], AChecker [48], ZepScope [69], NFTGuard [127], and Mythril [106], and a dynamic analysis tool,

[1] arbitrary-send-erc20, erc20- interface, erc721-interface, arbitrary-send-eth, arbitrary-send-erc20-permit, and slither-check-erc

Table 6. Evaluation results on the ground-truth dataset. ($x_{(y,z)}$: x true positives, y false positives, and z false negatives.)

	Slither	AChecker	ZepScope	NFTGuard	Mythril	ERCx	SymGPT
High	$0_{(0,28)}$	$0_{(0,28)}$	$0_{(0,28)}$	$0_{(0,28)}$	$0_{(0,28)}$	$0_{(2,28)}$	$28_{(1,0)}$
Medium	$26_{(0,29)}$	$0_{(0,55)}$	$2_{(19,53)}$	$0_{(0,55)}$	$0_{(0,55)}$	$12_{(21,43)}$	$53_{(0,2)}$
Low	$13_{(0,63)}$	$0_{(0,76)}$	$0_{(0,76)}$	$0_{(0,76)}$	$0_{(0,76)}$	$0_{(0,76)}$	$76_{(0,0)}$
Total	$39_{(0,120)}$	$0_{(0,159)}$	$2_{(47,157)}$	$0_{(0,159)}$	$0_{(0,159)}$	$12_{(23,147)}$	$157_{(1,2)}$

ERCx [110]. Other recent techniques either rely on manually written, contract-specific rules [21, 63, 81], require access tokens [89], lack publicly available source code [70], or depend on outdated Etherscan APIs [71, 72], making direct comparison infeasible. These six tools represent the state of the art in ERC compliance validation. We also compare SymGPT with ECSD, as ECSD is the only human auditing service whose audit results are available to us.

5.2.2 Experimental Results. As shown in Table 6, SymGPT detects 157 out of the 159 violations, *far outperforming the six baseline techniques,* with only one false positive. Most detected bugs and the false positive share code patterns observed in the large dataset. Since SymGPT does not check rules on return value generation, it misses two violations. Nonetheless, it detects all others, *demonstrating strong coverage of ERC rule violations.*

Slither detects the highest number of violations among all baselines, including 26 cases where an ERC-mandated function is either missing or does not match the required API, and 13 event-related cases. Some of its checkers aim to detect errors in verifying the message caller's privileges. However, these checkers mainly target contracts that interact with ERC-compliant contracts rather than the ERC-compliant contracts themselves. Consequently, the buggy code patterns they cover are absent from the ground-truth dataset, and they fail to capture any of the high-impact violations in the ground-truth dataset.

AChecker targets access control vulnerabilities characterized by code patterns where critical instructions lack protection or modifications to contract fields containing access control information are unguarded. However, all access-control-related violations in the ground-truth dataset are cases where access control checks are present but they fail to comply with ERC requirements. Consequently, AChecker fails to identify any of these violations.

ZepScope reports 49 missing-check cases based on OpenZeppelin's code. Only two are real bugs: in an ERC721 contract, `ownerOf()` and `balanceOf()` fail to check whether their inputs are zero. These two are also detected by SymGPT. The remaining are false positives: 37 where the check exists but ZepScope misidentifies it as missing, and ten where the check is absent but poses no security or functionality risk. For example, ZepScope flags an ERC721 contract's `approve()` for not verifying the input address differs from the message sender. Yet allowing a caller to approve herself is harmless, as she already has that privilege. Meanwhile, ZepScope misses 157 violations, showing that rules inferred solely from OpenZeppelin's code are inadequate for detecting ERC violations.

NFTGuard reports the contracts do not have five types of issues (*e.g.*, reentrancy, unlimited minting), but fails to detect any ERC rule violations. Relying on NFTGuard's bug patterns alone is insufficient for identifying ERC compliance issues.

Mythril's repository includes predefined constraints for detecting 11 categories of smart contract vulnerabilities. However, none of these constraints are related to ERC rules. Consequently, Mythril does not report any ERC rule violations when analyzing the ground-truth dataset.

ERCx detects 12 violations: five involving ERC API non-compliance, six where implementations forbid zero input values despite ERC20 explicitly allowing them, and one where `getApproved(uint256 _tokenID)` in an ERC721 contract fails to validate its input. However, many violations

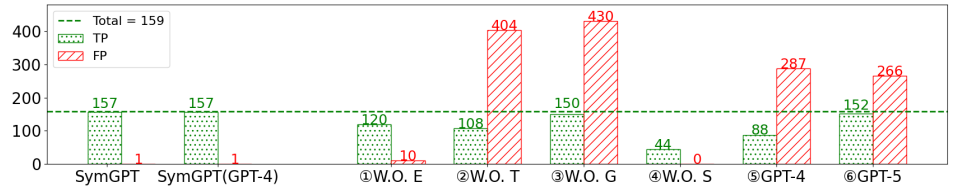


Fig. 11. Contributions of SymGPT's components. (There are 159 violations in the ground-truth dataset. W.O.: without, E: rule extraction, T: rule translation, G: constraint generation, and S: symbolic execution.)

are missed because ERCx's unit tests are incomprehensive to dynamically trigger them. ERCx reports 23 false positives. Six stem from misinterpreting ERC rules, such as flagging four APIs as non-compliant but they are not required by the ERCs. The remaining 17 result from flawed unit tests. For instance, ERCx reports two violations where `safeBatchTransferFrom()` does not invoke a function required by ERC1155. In reality, the call is only mandatory when transferring tokens to a contract, a case not triggered by the tests.

Comparison with the ECSD Auditing Service. We compare SymGPT with ECSD using the 32 contracts from ECSD's GitHub. ECSD detects 68 violations, all of which SymGPT successfully identifies, along with 70 more violations missed by ECSD. ECSD produces 12 false positives—two due to auditors not knowing Solidity auto-generates getter functions for contract fields, and ten from incorrectly assuming `transferFrom()` must verify the owner's balance, similar to `transfer()`, despite no such ERC20 requirement.

ECSD experts spent 128 days auditing the contracts, measured from the GitHub request submission to the final expert review, while SymGPT completes the analysis in just 1,037 seconds, measured as the sum of the ChatGPT analysis time for the ERCs and the code analysis time for all contracts. Additionally, ECSD charges $160,000 for auditing (estimated by the average hourly rate and time spent), whereas SymGPT costs only $2.94 for the use of ChatGPT[2]. Although this is a best-effort comparison, it clearly highlights SymGPT's vastly lower time and monetary costs.

Answer to advancement: *SymGPT detects significantly more violations than the baselines while minimizing false positives, monetary cost, and execution time, demonstrating its clear advantage over existing auditing solutions.*

5.3 Rationality of SymGPT's Components

5.3.1 Methodology. To evaluate the contribution of each component in SymGPT, we conduct an ablation study by disabling them individually and constructing multiple variants of SymGPT. Specifically: ① *Without rule extraction.* We directly prompt the LLM to translate ERC documents' specification sections into individual rules in the DSL. ② *Without rule translation.* We ask the LLM to generate constraints directly from the natural language rules. ③ *Without constraint generation.* We provide the LLM with each rule in the DSL, along with the corresponding public function, its direct and indirect callees, and all contract fields accessed by the public function and all its callees (collectively referred to as the related contract code). The LLM is then asked to determine whether the rule is violated. ④ *Without symbolic execution.* We supply the LLM with the violation constraints and the related contract code, and request it to assess whether the constraints can be satisfied. We compare these ablated variants with the full-featured SymGPT on the ground-truth dataset to assess the necessity and rationale of each component. Figures 12, 13, 14, and 15 in the appendix show the prompt templates designed for these variants, respectively.

[2]ERC1155 is not involved, so the number differs from Section 5.1.2.

In addition, we evaluate a setting where the LLM analyzes each extracted natural language rule directly against the related contract code, a scenario where ERC compliance validation relies *solely on the LLM*. Besides GPT-5 (⑥ in Figure 11), we further test GPT-4 (⑤ in Figure 11) to examine how ERC compliance validation performance evolves across successive generations of LLMs. Figure 16 in the appendix shows the prompt template.

Since GPT-5 was released on August 7, 2025, and many contracts may have been used in its training, we additionally evaluate the performance of SymGPT using GPT-4 as the underlying LLM. The corresponding results are labeled "SymGPT (GPT-4)" in Figure 11.

5.3.2 Experimental Results. As shown in Figure 11, the full-featured version of SymGPT (labeled "SymGPT") detects more violations and reports fewer false positives than any of its ablated variants, highlighting *the necessity of each component*. Replacing the underlying LLM with GPT-4 (labeled "SymGPT (GPT-4)") yields the same results, demonstrating that *SymGPT is robust across different LLMs and does not rely on the evaluated contracts having appeared in its LLM's training data.*

① Without rule extraction, SymGPT generates fewer rules in the DSL because it directly processes the entire specification section, leading to 39 missed violations. ② Disabling rule translation causes the LLM to struggle with generating constraints that depend on contract-specific details (*e.g.*, contract fields) and to introduce errors when synthesizing constraints. Consequently, it misses 51 violations and produces 404 false positives. ③ When constraint generation is disabled and the LLM is asked to identify violations based on rules in the DSL, it captures almost all violations in the ground-truth dataset (150/159). However, hallucinations of the LLM lead to 430 false positives, the highest among all configurations. ④ Comparing the results of disabling constraint generation with those of disabling symbolic execution, we observe that providing the LLM with rules in constraints, rather than rules in the DSL, reduces both detected violations and false positives.

⑥ Relying solely on GPT-5 to check ERC rules in natural language against related contract code pinpoints most ERC rule violations in the dataset (152/159). The missed violations can be categorized into three types. First, ERC721 requires events `Transfer`, `Approval`, and `ApprovalForAll` to be declared with `indexed` to attribute their parameters. The LLM fails to detect five declarations that violate these requirements. Second, the LLM overlooks one instance in which the `transferFrom()` function does not verify the privilege of `msg.sender`. Third, in one case, the `totalSupply()` function of an ERC20 contract does not return the actual total number of supplied tokens, but the LLM fails to identify this violation. SymGPT also misses this issue because it lacks the capability to validate rules that involve generating specific return values. On the other hand, using only GPT-5 results in 266 false positives, far higher than SymGPT, which reports only a single false positive. This result highlights the advantage of combining symbolic execution with GPT-5. ⑤ Compared with GPT-5, using GPT-4 alone identifies fewer true violations while producing more false positives, highlighting the necessity of leveraging more recent LLMs when validating ERC compliance.

Answer to rationality: *Each component of SymGPT contributes to its effectiveness, and combining symbolic execution with the LLM significantly enhances the accuracy of ERC rule validation.*

5.4 Generality of SymGPT

5.4.1 Methodology. We use ERC3525 and ERC4907 to evaluate whether SymGPT can validate contracts beyond the ERCs we have studied against potentially more complex ERC rules. ERC3525 is designed for semi-fungible tokens, where each token is unique like an NFT but also possesses a quantitative nature similar to fungible tokens [113]. ERC4907 extends ERC721 by introducing time-limited usage rights: for each NFT, it allows a designated user to use the token within a specified time window while prohibiting transfer during that period [93]. Both ERC3525 and ERC4907 have already been adopted in financial instruments [14, 18, 46]. Introduced at least two years after the

ERCs analyzed in Section 3, both ERC3525 and ERC4907 represent more recent developments in the ERC ecosystem. Following the same methodology, we manually identify 58 rules in ERC3525, and 12 rules in ERC4907.

We do not find any ERC3525 or ERC4907 contracts on etherscan.io or polygonscan.com. Therefore, we search GitHub for contracts of these ERCs. We find only one ERC3525 contract on GitHub. In contrast, many ERC4907 contracts are available, from which we randomly select ten. After manual inspection, we confirm that only one ERC4907 contract contains a single violation, where function `setUser()` fails to inspect whether the input `tokenId` is valid, as required by ERC4907, while all other contracts fully comply with ERC3525 or ERC4907. To evaluate SymGPT, we inject violations into the compliant contracts by randomly removing code related to ERC compliance. For the ERC3525 contract, we perform random error injection ten times, injecting three errors each time. For the remaining nine ERC4907 contracts, we inject three errors into each contract. In total, we construct a dataset consisting of 20 contracts: 19 contracts each contain three injected violations, and one contract contains a single violation originally introduced by its developer.

5.4.2 Empirical Study Results. We conduct the same empirical study as Section 3 on the 70 rules in ERC3525 and ERC4907. Regarding rule content, nine rules pertain to privilege checks, 24 relate to functionality requirements, 18 define public functions' APIs, and 19 specify logging requirements. In terms of security impact, 16 rules are classified as having a high-security impact, 35 as medium, and the 19 logging-related rules as low. Of the 70 rules, 23 concern function or event declarations, which are expressed in Solidity source code. Among the remaining 47 rules, 42 can be matched using the linguistic patterns listed in Table 2. Consistent with the three previously studied ERCs, most of these rules fall under four primary linguistic patterns: TP1, EP1, EP2, and RP1.

5.4.3 Experimental Results. After running SymGPT *without any modifications* on the 20 contracts, it successfully detects all 58 violations with zero false positives. In the intermediate steps, the LLM correctly identifies all required rules during rule extraction, but it erroneously extracts three additional rules. One rule talks about setting the return value of function `transferFrom()`, and the other two require function `userOf()` and function `userExpires()` not to throw when returning zero. During rule translation, SymGPT fails to translate these extra rules, along with 18 other rules related to return value generation. Thus, those erroneously extracted rules do not introduce any false positives. SymGPT does not make any errors in all other steps.

Answer to generality: *The findings from the empirical study and SymGPT's violation detection capabilities are not limited to the three studied ERCs but generalize to a broader range of ERC standards.*

6 Limitations and Discussion

Threats to Validity. The validity of our study may be affected by several factors, including the limited number of ERCs and smart contracts analyzed, the use of the LLM, the absence of dynamic validation for the detected violations, and the unsoundness of SymGPT's static analysis.

In Section 3, we analyze three widely adopted ERCs that illustrate common development practices and typical issues in smart contracts. Although many ERCs remain unexplored, we regard these three as representative examples. As demonstrated in Section 5.4, the findings and linguistic patterns identified in our study also generalize to ERC3525 and ERC4907, two previously unexamined standards. Moreover, SymGPT, which was developed based on these insights, successfully detects all injected and original violations of the two ERCs. To evaluate SymGPT, we constructed three datasets comprising over 4,000 contracts collected from five different sources. While additional repositories such as Arbitrum [4] and BscScan [17] are available, we are not aware of substantial

differences between contracts from those sources and the ones included in our datasets. Therefore, we consider our datasets to be a best-effort representation of real-world ERC implementations.

SymGPT employs GPT-5 as its underlying LLM. Although GPT-5 represents the state of the art at the time of writing, the field of LLMs is advancing rapidly, and it may soon be surpassed by newer models. Furthermore, our evaluation adopts a single configuration of GPT-5 without exploring alternative parameter settings. These limitations, however, are mitigated by SymGPT's design: the DSL constrains the LLM to a structured translation task, while critical bug detection is carried out deterministically through symbolic execution. Together, these features reduce SymGPT's dependence on any specific LLM or configuration.

We evaluate SymGPT on over 4,000 smart contracts and identify more than 5,000 ERC rule violations. All detected results are carefully reviewed by the authors to ensure their accuracy. Section 5 presents a detailed categorization of the detected violations and reported false positives, demonstrating our thorough understanding and analysis of the results. We do not perform dynamic validation because it would require deploying each contract locally (including downloading all external dependencies), configuring the execution context (*e.g.*, assigning appropriate values to state variables), and creating inputs to trigger the relevant functions. Conducting these steps at this scale is impractical. This practice aligns with many prior research papers on static bug detection.

SymGPT contains three static-analysis components: API compliance checkers, utility functions when synthesizing constraints, and symbolic execution engine. The first two are sound, as their functionality is straightforward. However, the symbolic execution engine has several limitations that may lead to both false positives and false negatives: it bounds loop iterations and recursive call depth to two, cannot handle assembly code, and cannot analyze external calls when the code of external contracts is unavailable. We did not observe issues caused by the first limitation; however, the latter two result in the false positives reported in Section 5.1.

Discussion. Some tools support custom rules, assertions, or contract behaviors [21, 81, 89, 106, 115]. SymGPT could integrate with them by translating ERC rules into the required input format and extending their functionalities to capture ERC-specific semantics. For instance, for rule requiring functions to throw an exception when invoked with certain inputs, SymGPT can generate specifications in Certora's input format, by designating target functions, assuming input values, and instructing Certora to verify whether the functions throw an exception under the inputs. Likewise, extending the DSL and the symbolic execution engine would allow SymGPT to generate constraints from natural language descriptions beyond ERCs and detect a broader range of smart contract issues. For example, we can extend the engine by assigning a financial type to each variable and define a DSL to specify which financial types can be summed. The LLM could then translate natural language descriptions of financial-type rules into the DSL, allowing SymGPT to detect accounting errors [132]. We leave these directions for future work.

7 Conclusion

This paper presents an empirical study on the implementation rules in ERC documents, focusing on their content, security implications, and detailed specifications in natural language. Based on our findings, we developed SymGPT, an automated tool that integrates an LLM with symbolic execution to assess smart contracts' compliance with ERC rules. SymGPT effectively detects numerous rule violations and outperforms six automated techniques and a manual auditing service. This project enhances the understanding of ERC rules and their violations, promoting further exploration in the field. Moreover, our work demonstrates the advantages of combining LLMs with formal methods and encourages continued research in this direction.

Data-Availability Statement

This work provides an open-source artifact that supports the claims made in our paper. The artifact includes the empirical study results of ERC rules, the source code of SymGPT, evaluation datasets, experimental outputs, and scripts to fully automate all experiments. The scripts interact with public APIs offered by OpenAI, and they requires users to have a valid OpenAI API key set up in advance. Apart from standard open-source licensing terms, the artifact imposes no additional restrictions. It is available at https://github.com/symbolic-gpt/symgpt and submitted for artifact evaluation.

Acknowledgments

We are grateful to the anonymous reviewers for their insightful comments and suggestions. This work was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) under grants EP/T006544/2, EP/T014709/2, and EP/Z533749/1; by the Horizon Europe project TaRDIS (grant agreement 101093006; UKRI number 10066667); by the U.S. National Science Foundation (NSF) under awards CNS-1955965 and CCF-2145394; and by the National Natural Science Foundation of China under grant 92582108.

References

[1] M. Abraham and K. P. Jevitha. Runtime verification and vulnerability testing of smart contracts. In *Advances in Computing and Data Sciences (ICACDS 2019)*, volume 1046 of *Communications in Computer and Information Science*, pages 333–342, Singapore, 2019. Springer Singapore.

[2] D. Annenkov, M. Milo, J. B. Nielsen, and B. Spitters. Extracting smart contracts tested and verified in coq. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '21)*, Virtual, Denmark, 2021.

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

[4] Arbitrum. The future of ethereum, 2024. https://arbitrum.io/.

[5] L. Arena. 9lives arena - the ultimate pvp gaming experience, 2023. https://www.9livesarena.com/.

[6] G. Ayoade, E. Bauman, L. Khan, and K. W. Hamlen. Smart contract defense through bytecode rewriting. In *Proceedings of the 2019 IEEE International Conference on Blockchain (Blockchain '19)*, Atlanta, GA, USA, 2019.

[7] S. M. Beillahi, G. F. Ciocarlie, M. Emmi, and C. Enea. Behavioral simulation for smart contracts. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*, London, UK, 2020.

[8] R. Ben Fekih, M. Lahami, M. Jmaiel, and S. Bradai. Formal modeling and verification of erc smart contracts: Application to nft. In *Proceedings of the 2023 IEEE Symposium on Computers and Communications (ISCC '23)*, Gammarth, Tunisia, 2023.

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

[10] T. Blackstone. Reewardio loyalty platform putting enj nft items to work – real use case, 2019. https://www.castlecrypto.gg/reewardio-loyalty-platform-demo-review/.

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

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

[13] 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/.

[14] O. Blog. Erc-4907 the standard for nft rentals and ownership separation, 2025. https://onekey.so/blog/ecosystem/erc-4907-the-standard-for-nft-rentals-and-ownership-separation/.

[15] C. Bräm, M. Eilers, P. Müller, R. Sierra, and A. J. Summers. Rich specifications for ethereum smart contract verification. 2021.

[16] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981*, 2018.

[17] BscScan. Bnb smart chain (bnb) blockchain explorer, 2025. https://bscscan.com/.

[18] Buffer. Buffer finance — options trading simplified, 2023. https://buffer.finance/.

[19] C. Cadar, D. Dunbar, and D. Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, 2008.

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

[21] Certora Inc. Certora, 2025. https://www.certora.com.

[22] Chainlink. Top 6 Smart Contract Languages in 2023, 2023. https://chain.link/education-hub/smart-contract-programming-languages.
[23] J. Chang, B. Gao, H. Xiao, J. Sun, Y. Cai, and Z. Yang. scompile: Critical path identification and analysis for smart contracts. In *International Conference on Formal Engineering Methods (ICFEM '19)*, Shenzhen, China, 2019.
[24] H. Chen, G. Whitters, M. J. Amiri, Y. Wang, and B. T. Loo. Declarative smart contracts. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22)*, Singapore, 2022.
[25] J. Chen, Z. Shao, S. Yang, Y. Shen, Y. Wang, T. Chen, Z. Shan, and Z. Zheng. Numscout: Unveiling numerical defects in smart contracts using llm-pruning symbolic execution. *IEEE Transactions on Software Engineering*, pages 1538–1553, 2025.
[26] T. Chen, S. Lu, S. Lu, Y. Gong, C. Yang, X. Li, M. R. H. Misu, H. Yu, N. Duan, P. Cheng, F. Yang, S. Lahiri, T. Xie, and L. Zhou. Automated proof generation for rust code via self-evolution. In *The International Conference on Learning Representations (ICLR '24)*, Vienna, Austria, 2024.
[27] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang. Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security (CCS '19)*, London, UK, 2019.
[28] 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 *Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP '23)*, San Francisco, California, 2023.
[29] Z. Chen, S. M. Beillahi, and F. Long. Flashsyn: Flash loan attack synthesis via counter example driven approximation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*, Lisbon, Portugal, 2024.
[30] Z. Chen, Y. Liu, S. M. Beillahi, Y. Li, and F. Long. Demystifying invariant effectiveness for securing smart contracts. 2024.
[31] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, Newport Beach, California, USA, 2011.
[32] E. Commonwealth. Callisto smart-contract auditing department, 2023. https://github.com/EthereumCommonwealth/Auditing.
[33] Crytic. Erc conformance, 2023. https://github.com/crytic/slither/wiki/ERC-Conformance.
[34] Crytic. Slither, the smart contract static analyzer, 2023. https://github.com/crytic/slither.
[35] defiprime. Ethereum DeFi Ecosystem, 2023. https://defiprime.com/ethereum.
[36] X. Deng, S. M. Beillahi, C. Minwalla, H. Du, A. Veneris, and F. Long. Safeguarding defi smart contracts against oracle deviations. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*, Lisbon, Portugal, 2024.
[37] Y. Duan, X. Zhao, Y. Pan, S. Li, M. Li, F. Xu, and M. Zhang. Towards automated safety vetting of smart contracts in decentralized applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*, Los Angeles, CA, USA, 2022.
[38] ECMA. ECMA-262, 2023. https://www.ecma-international.org/publications-and-standards/standards/ecma-262/.
[39] W. Entriken, D. Shirley, J. Evans, and N. Sachs. Erc-721: Non-fungible token standard, 2018. https://eips.ethereum.org/EIPS/eip-20.
[40] Ethereum. Ethereum, 2023. https://ethereum.org/en/.
[41] Ethereum. Ethereum dapps, 2023. https://ethereum.org/en/dapps.
[42] Ethereum. Ethereum improvement proposals, 2023. https://eips.ethereum.org/erc.
[43] Ethereum. ethereum/ercs, 2023. https://github.com/ethereum/ERCs/blob/master/ERCS/eip-1.md.
[44] Ethereum. Smart contract anatomy, 2023. https://ethereum.org/en/developers/docs/smart-contracts/anatomy.
[45] Etherscan. The ethereum blockchain explorer. https://etherscan.io.
[46] F. Finance. Fujidao — the auto-refinancing borrow protocol, 2023. https://v1.fuji.finance/.
[47] 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.
[48] A. Ghaleb, J. Rubin, and K. Pattabiraman. Achecker: Statically detecting smart contract access control vulnerabilities. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '23)*, Melbourne, Victoria, Australia, 2023.
[49] 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.

[50] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20)*, Virtual Event, 2020.

[51] Á. Hajdu and D. Jovanović. solc-verify: A modular verifier for solidity smart contracts. In *Verified Software. Theories, Tools, and Experiments (VSTTE 2019)*, volume 12301 of *Lecture Notes in Computer Science*, pages 161–179, Cham, 2020. Springer.

[52] S. Hao, Y. Nan, Z. Zheng, and X. Liu. Smartcoco: Checking comment-code inconsistency in smart contracts via constraint propagation and binding. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE '23)*, Kirchberg, Luxembourg, 2023.

[53] R. Harvey. Smart contract auditor, 2024. https://chatgpt.com/g/g-VRtUR3Jpv-smart-contract-auditor.

[54] M. He, S. Xia, B. Qin, N. Yoshida, T. Yu, Y. Zhang, and L. Song. How to save my gas fees: Understanding and detecting real-world gas issues in solidity programs. *IEEE Transactions on Software Engineering*, 2025.

[55] G. Ibba, M. Ortu, R. Tonelli, and G. Destefanis. Leveraging chatgpt for automated smart contract repair: A preliminary exploration of gpt-3-based approaches.

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

[57] R. Ji, N. He, L. Wu, H. Wang, G. Bai, and Y. Guo. Deposafe: Demystifying the fake deposit vulnerability in ethereum smart contracts. In *Proceedings of the 2020 25th International Conference on Engineering of Complex Computer Systems (ICECCS '20)*, Singapore, 2020.

[58] B. Jiang, Y. Liu, and W. K. Chan. Contractfuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*, Montpellier, France, 2018.

[59] S. Kalra, S. Goel, M. Dhawan, and S. Sharma. Zeus: Analyzing safety of smart contracts. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS '18)*, San Diego, California, USA, 2018.

[60] E. Keilty, K. Nelaturu, B. Wu, and A. Veneris. A model-checking framework for the verification of move smart contracts. In *Proceedings of the 2022 IEEE 13th International Conference on Software Engineering and Service Science (ICSESS '22)*, Beijing, China, 2022.

[61] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena. Exploiting the laws of order in smart contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*, Beijing, China, 2019.

[62] J. Krupp and C. Rossow. {teEther}: Gnawing at ethereum to automatically exploit smart contracts. In *Proceedings of the 27rd USENIX Security Symposium (USENIX '18)*, Baltimore, MD, USA, 2018.

[63] A. Li, J. A. Choi, and F. Long. Securing smart contract with runtime validation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*, Virtual Event, 2020.

[64] 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 *Proceedings of the 40th IEEE International Conference on Distributed Computing Systems (ICDCS '20)*, Singapore, 2020.

[65] Z. Liao, S. Hao, Y. Nan, and Z. Zheng. Smartstate: Detecting state-reverting vulnerabilities in smart contracts via fine-grained state-dependency analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*, ISSTA 2023, Seattle, WA, USA, 2023.

[66] X. Lin, Q. Xie, B. Zhao, Y. Tian, S. Zonouz, N. Ruan, J. Li, R. Beyah, and S. Ji. Promfuzz: Leveraging llm-driven and bug-oriented composite analysis for detecting functional bugs in smart contracts. 2025.

[67] Z. Lin, J. Chen, J. Wu, W. Zhang, and Z. Zheng. Definition and detection of centralization defects in smart contracts. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE '25)*, Ottawa, Ontario, Canada, 2025.

[68] 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.

[69] H. Liu, D. Wu, Y. Sun, H. Wang, K. Li, Y. Liu, and Y. Chen. Using my functions should follow my checks: Understanding and detecting insecure OpenZeppelin code in smart contracts. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security '24)*, Philadelphia, PA, 2024.

[70] H. Liu, D. Wu, Y. Sun, S. Wang, Y. Liu, and Y. Chen. Demystifying openzeppelin's own vulnerabilities and analyzing their propagation in smart contracts. 2025.

[71] Y. Liu and Y. Li. Invcon: A dynamic invariant detector for ethereum smart contracts. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*, MI, USA, 2022.

[72] Y. Liu, Y. Li, S.-W. Lin, and C. Artho. Finding permission bugs in smart contracts with role mining. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22)*, Virtual Event, 2022.

[73] Y. Liu, Y. Li, S.-W. Lin, and R. Zhao. Towards automated verification of smart contract fairness. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20)*, Virtual, USA, 2020.

[74] Y. Liu, Y. Xue, D. Wu, Y. Sun, Y. Li, M. Shi, and Y. Liu. Propertygpt: Llm-driven formal verification of smart contracts through retrieval-augmented property generation. In *Proceedings of the 32nd Annual Network and Distributed System Security Symposium (NDSS '25)*, San Diego, California, USA, 2025.
[75] F. Luo, R. Luo, T. Chen, A. Qiao, Z. He, S. Song, Y. Jiang, and S. Li. Scvhunter: Smart contract vulnerability detection based on heterogeneous graph attention network. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*, Lisbon, Portugal, 2024.
[76] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, Vienna, Austria, 2016.
[77] F. Ma, M. Ren, L. Ouyang, Y. Chen, J. Zhu, T. Chen, Y. Zheng, X. Dai, Y. Jiang, and J. Sun. Pied-piper: Revealing the backdoor threats in ethereum erc token contracts. *ACM Transactions on Software Engineering and Methodology*, 32(3), Apr. 2023.
[78] W. Ma, D. Wu, Y. Sun, T. Wang, S. Liu, J. Zhang, Y. Xue, and Y. Liu. Combining fine-tuning and llm-based agents for intuitive smart contract auditing with justifications, 2025.
[79] Y. Marcus, E. Heilman, and S. Goldberg. Low-resource eclipse attacks on ethereum's peer-to-peer network. *Cryptology ePrint Archive*, 2018.
[80] Markus Waas. Top 7 Reasons To Learn Solidity Programming ASAP, 2022. https://zerotomastery.io/blog/top-7-reasons-to-learn-solidity-programming/.
[81] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE '19)*, San Diego, CA, USA, 2019.
[82] K. Nelaturu, A. Mavridou, E. Stachtiari, A. Veneris, and A. Laszka. Correct-by-design interacting smart contracts and a systematic approach for verifying erc20 and erc721 contracts with verisolid. *IEEE Transactions on Dependable and Secure Computing*, 20(4):3110–3127, 2023.
[83] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering (ICSE '20)*, Seoul, South Korea, 2020.
[84] M. D. Norman. Smart contract auditor, 2024. https://chatgpt.com/g/g-XFyKAnTpO-smart-contract-auditor.
[85] OpenAI. Openai gpt5. https://platform.openai.com/docs/models/gpt-5.
[86] OpenAI. Reasoning model. https://platform.openai.com/docs/guides/reasoning.
[87] OpenSea. Opensea, the largest nft marketplace, 2023. https://opensea.io/.
[88] S. Palladino. Erc20 verifier, 2019. https://github.com/spalladino/erc20-verifier.
[89] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. Verx: Safety verification of smart contracts. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (SP '20)*, Virtual Event, 2020.
[90] PixelPlex. Smart contract audit, 2023. https://pixelplex.io/smart-contract-audit/.
[91] S. Poeplau and A. Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20)*, Virtual Event, 2020.
[92] PolygonScan. Polygon pos chain explorer. https://polygonscan.com.
[93] E. I. Proposals. Erc-4907: Rental nft, an extension of eip-721, 2022. https://eips.ethereum.org/EIPS/eip-4907.
[94] 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.
[95] K. Qin, Z. Ye, Z. Wang, W. Li, L. Zhou, C. Zhang, D. Song, and A. Gervais. Enhancing smart contract security analysis with execution property graphs. 2025.
[96] 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.
[97] Rarible. Rarible — aggregated nft marketplace with rewards, 2023. https://rarible.com/.
[98] Revoluzion. Revoluzion smart contract audit report services, 2023. https://www.revoluzion.io/audit.
[99] M. Rodler, W. Li, G. O. Karame, and L. Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv preprint arXiv:1812.05934*, 2018.
[100] C. Sendner, H. Chen, H. Fereidooni, L. Petzi, J. König, J. Stang, A. Dmitrienko, A.-R. Sadeghi, and F. Koushanfar. Smarter contracts: Detecting vulnerabilities in smart contracts with deep transfer learning. In *Proceedings of the 30th Annual Network and Distributed System Security Symposium (NDSS '23)*, San Diego, California, USA, 2023.
[101] R. Shyamasundar. A framework of runtime monitoring for correct execution of smart contracts. In *International Conference on Blockchain (ICBC '22)*, Honolulu, HI, USA, 2022.
[102] C. Smith. Token standards, 2023. https://ethereum.org/en/developers/docs/standards/tokens/.
[103] 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.

[104] K. Sun, Z. Xu, K. Li, L. Zhang, Y. Feng, D. Wu, and Y. Liu. Learning from the past: Real-world exploit migration for smart contract poc generation. 2025.
[105] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*, Lisbon, Portugal, 2024.
[106] M. Team. Mythril, 2018. https://github.com/ConsenSysDiligence/mythril.
[107] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB '18)*, Gothenburg, Sweden, 2018.
[108] C. F. Torres, J. Schütte, and R. State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC' 18)*, San Juan, PR, USA, 2018.
[109] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security (CCS '18)*, Toronto, Canada, 2018.
[110] R. Verification. Secure your assets with ercx, 2024. https://ercx.runtimeverification.com/.
[111] F. Vogelsteller and V. Buterin. Erc-20: Token standard, 2015. https://eips.ethereum.org/EIPS/eip-20.
[112] S. Wang, C. Zhang, and Z. Su. Detecting nondeterministic payment bugs in ethereum smart contracts. 2019.
[113] 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.
[114] C. Wei, S. Cai, Y. Charalambous, T. Wu, S. Godboley, and L. C. Cordeiro. Veriexploit: Automatic bug reproduction in smart contracts via llms and formal methods. 2025.
[115] G. Wei, D. Xie, W. Zhang, Y. Yuan, and Z. Zhang. Consolidating smart contracts with behavioral contracts. In *Proceedings of the 45st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '24)*, Copenhagen, Denmark, 2024.
[116] Wikipedia. Binance, 2023. https://en.wikipedia.org/wiki/Binance.
[117] Wikipedia. Ethereum, 2023. https://en.wikipedia.org/wiki/Ethereum.
[118] Wikipedia. Horizon (video game series), 2023. https://en.wikipedia.org/wiki/Horizon_(video_game_series).
[119] Wikipedia. Shiba inu (cryptocurrency), 2023. https://en.wikipedia.org/wiki/Shiba_Inu_(cryptocurrency).
[120] Wikipedia. Tether (cryptocurrency), 2023. https://en.wikipedia.org/wiki/Tether_(cryptocurrency).
[121] Y. Wu, X. Xie, C. Peng, D. Liu, H. Wu, M. Fan, T. Liu, and H. Wang. Advscanner: Generating adversarial smart contracts to exploit reentrancy vulnerabilities using llm and static analysis. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*, Sacramento, CA, USA, 2024.
[122] K. Wüst and A. Gervais. Ethereum eclipse attacks. Technical report, ETH Zurich, 2016.
[123] 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.
[124] 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)*, Melbourne, Australia, 2020.
[125] C. Yang, X. Li, M. R. H. Misu, J. Yao, W. Cui, Y. Gong, C. Hawblitzel, S. K. Lahiri, J. R. Lorch, S. Lu, F. Yang, Z. Zhou, and S. Lu. Autoverus: Automated proof generation for rust code. 2025.
[126] J. Yang, H. Jin, R. Tang, X. Han, Q. Feng, H. Jiang, S. Zhong, B. Yin, and X. Hu. Harnessing the power of llms in practice: A survey on chatgpt and beyond. *ACM Transactions on Knowledge Discovery from Data*, 18(6):1–32, 2024.
[127] S. Yang, J. Chen, and Z. Zheng. Definition and detection of defects in nft smart contracts. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*, Seattle, Washington, USA, 2023.
[128] Y. Yang, T. Kim, and B.-G. Chun. Finding consensus bugs in ethereum via multi-transaction differential fuzzing. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)*, Virtual Event, 2021.
[129] J. Ye, M. Ma, Y. Lin, L. Ma, Y. Xue, and J. Zhao. Vulpedia: Detecting vulnerable ethereum smart contracts via abstracted vulnerability signatures. *Journal of Systems and Software*, 192:111410, 2022.
[130] L. Yu, Z. Huang, H. Yuan, S. Cheng, L. Yang, F. Zhang, C. Shen, J. Ma, J. Zhang, J. Lu, and C. Zuo. Smart-llama-dpo: Reinforced large language model for explainable smart contract vulnerability detection. 2025.
[131] H. Yuan, X. Hou, L. Yu, L. Yang, J. Tang, J. Xu, Y. Liu, F. Zhang, and C. Zuo. Leveraging mixture-of-experts framework for smart contract vulnerability repair with large language model. 2025.
[132] B. Zhang. Towards finding accounting errors in smart contracts. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*, Lisbon, Portugal, 2024.
[133] J. Zhang, Y. Shen, J. Chen, J. Su, Y. Wang, T. Chen, J. Gao, and Z. Chen. Demystifying and detecting cryptographic defects in ethereum smart contracts. In *Proceedings of the IEEE/ACM 47th International Conference on Software*

Engineering (ICSE '25), Ottawa, Ontario, Canada, 2025.

[134] Z. Zhang, B. Zhang, W. Xu, and Z. Lin. Demystifying exploitable bugs in smart contracts. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '23)*, Melbourne, Victoria, Australia, 2023.

[135] Z. Zheng, N. Zhang, J. Su, Z. Zhong, M. Ye, and J. Chen. Turn the rudder: A beacon of reentrancy detection for smart contracts on ethereum. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '23)*, Melbourne, Victoria, Australia, 2023.

[136] J. Zhong, D. Wu, Y. Liu, M. Xie, Y. Liu, Y. Li, and N. Liu. Detecting various defi price manipulations with llm reasoning. 2025.

[137] C. Zhu, Y. Liu, X. Wu, and Y. Li. Identifying solidity smart contract api documentation errors. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*, MI, USA, 2022.

Appendix

Table 7. Words for Abbreviation

Abbr	Words	POS
SUB	a function, a contract	NSUBJ, DOBJ
MUST	“must”, “must not”, “cannot”, “should”	AUX
THROW	‘throw”, “revert”, “be treated as normal”	ROOT
COND	an if statement, an unless statement	ADVCL
ACTION	an operation performed by a function	NSUBJ, DOBJ
CALLER	“caller”	NSUBJ
APPROVE	“be approved to”	ROOT
BE	“are considered”	ROOT
INVALID	“invalid”	ADJ
CALL	“call”	ROOT
VAR	a variable	NSUBJ
ASSIGN	“set to”, “overwrite”, “be reset”, “allow”, “be set”, “be”, “==”, “create”	ROOT
PREP	“with”, “to”	ADP
VALUE	a specific value	DOBJ
EVENT	an event	NSUBJ
EMIT	“fire” , “trigger”, “emit”	ROOT
RETURN	“return”, “@return”	ROOT
FOLLOW	“follow”	ROOT
ORDER	a specific order	DOBJ

```
Given the ERC, return all the rules in the ERC with the given constraints.
ERC:"""
{{specification}}
"""
Constraint JSON Schema:"""
{{schema}}
"""
Return all constraints in single JSON array: [{
    "target": "string, optional, can either be function interface or event name which rule
should apply, or empty if the rule should be applied to every function",
    "constraint": "constraint object, can be one of CallVerify, EmitVerify, OrderVerify,
ReturnVerify, StateAssignVerify, or ThrowVerify which follows the constraint JSON schema"
}]
```

Fig. 12. The prompt template for translating the rules in an ERC’s specification section into the DSL. (“{{specification}}” refers to the specification section of an ERC, and “{{schema}}” denotes the JSON schema defining the DSL.)

Received 2025-10-10; accepted 2026-02-17

```
Given the rule and the constraint syntax, generate the buggy constraints for the given rule
and code.
Constraints can be concatenated with the ==, !=, <, <=, >, >=, &, ||,!, and ().
Interface:"""
{{API}}
"""
Rule: """
{{rule}}
"""
Code:"""
{{code}}
"""
Constraint Syntax:"""
{{syntax}}
"""
Return JSON format: {"constraints": "string, the buggy constraints"}
```

Fig. 13. The prompt template for translating a rule written in natural language into constraints. ("{{API}}" denotes the declaration of the function to which the rule applies, "{{rule}}" represents the rule described in natural language, "{{code}}" refers to the related contract code, and "{{syntax}}" specifies the constraint format.)

```
Audit the following code against the provided rule:"""
{{rule}}
"""
Rule Schema:"""
{{schema}}
"""
Code:"""
{{code}}
"""
Return YES if violated. NO otherwise. DO NOT EXPLAIN ANYTHING ELSE.
```

Fig. 14. The prompt template for checking whether the given code violates a rule in the DSL. ("{{rule}}" represents a rule in the DSL, "{{schema}}" denotes the JSON schema of the DSL, and "{{code}}" refers to the related contract code of the rule.)

```
Audit the following code with the initial z3 constraints and buggy constraints.
Verify whether the code satisfied the buggy constraints if the entrypoint is the function
{{entryfunction}}
Initial Constraints:"""
{{initialconstraint}}
"""
Buggy Constraints:"""
{{buggyconstraint}}
"""
Code:"""
{{code}}
"""
Return in YES if satisfied, NO otherwise. DO NOT EXPLAIN ANYTHING ELSE.
```

Fig. 15. The prompt template for checking whether the given code satisfies the specified constraints. ("{{initialconstraint}}" represents the initial constraints derived from contract code, "{{buggyconstraint}}" denotes the buggy constraints translated from rules in the DSL, and "{{code}}" refers to the relevant smart contract code.)

```
Given the related code for function {{API}} and the rule "{{rule}}", return YES if the code
 violates the rule, NO otherwise.
Code:"""
{{code}}
"""
```

Fig. 16. The prompt template for checking whether the given code violates a natural language ERC rule. ("{{rule}}" denotes the rule written in natural language, and "{{code}}" represents related smart contract code.)