PropertyGPT: LLM-driven Formal Verification of Smart Contracts through Retrieval-Augmented Property Generation

Ye Liu[1], Yue Xue[2†], Daoyuan Wu[3*], Yuqiang Sun[4], Yi Li[4], Miaolei Shi[2], and Yang Liu[4,5]
[1]Singapore Management University
[2]MetaTrust Labs
[3]The Hong Kong University of Science and Technology
[4]Nanyang Technological University
[5]China-Singapore International Joint Research Institute (CSIJRI)

***Abstract*—Formal verification is a technique that can prove the correctness of a system with respect to a certain specification or property. It is especially valuable for security-sensitive smart contracts that manage billions in cryptocurrency assets. Although existing research has developed various static verification tools (or provers) for smart contracts, a key missing component is the *automated* generation of *comprehensive* properties, including invariants, pre-/post-conditions, and rules. Hence, industry-leading players like Certora have to rely on their own or crowdsourced experts to manually write properties case by case.**

With recent advances in large language models (LLMs), this paper explores the potential of leveraging state-of-the-art LLMs, such as GPT-4, to transfer existing human-written properties (e.g., those from Certora auditing reports) and automatically generate customized properties for unknown code. To this end, we embed existing properties into a vector database and retrieve a reference property for LLM-based in-context learning to generate a new property for a given code. While this basic process is relatively straightforward, ensuring that the generated properties are (i) *compilable*, (ii) *appropriate*, and (iii) *verifiable* presents challenges. To address (i), we use the compilation and static analysis feedback as an external oracle to guide LLMs in iteratively revising the generated properties. For (ii), we consider multiple dimensions of similarity to rank the properties and employ a weighted algorithm to identify the top-K properties as the final result. For (iii), we design a dedicated prover to formally verify the correctness of the generated properties. We have implemented these strategies into a novel LLM-based property generation tool called PropertyGPT. Our experiments show that PropertyGPT can generate comprehensive and high-quality properties, achieving an 80% recall compared to the ground truth. It successfully detected 26 CVEs/attack incidents out of 37 tested and also uncovered 12 zero-day vulnerabilities, leading to $8,256 in bug bounty rewards.

This work was done while Ye Liu was a student at Nanyang Technological University.

†Yue Xue and Ye Liu are the co-first authors.

*Daoyuan Wu is the corresponding author.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.241357
www.ndss-symposium.org

I. Introduction

Smart contracts are transaction-driven programs deployed and executed on blockchain platforms, automating the execution of digital agreements among users. Most smart contracts are written in Turing-complete programming languages, such as Solidity [47], and have been widely adopted on popular blockchain platforms like Ethereum [61] and BSC [1]. Smart contracts are extensively used in decentralized applications such as DeFi [25] and NFTs [5]. However, they are susceptible to various types of attacks, including integer overflow [51], re-entrancy [43], front-running [50], and access control vulnerabilities [15], [34]. These vulnerabilities primarily arise from loopholes in smart contracts due to programming errors, incorrect implementations, and logical bugs [66].

Formal verification is one of the most advanced approaches to identify contract loopholes by performing a comprehensive examination with different kinds of specifications. To perform formal verification, it is necessary to generate customized formal specifications for different smart contracts. Formal specifications for smart contracts usually include temporal logic properties and Hoare logic properties, as surveyed in [54]. *Invariants* are the most common contract specification, stating a property that holds for any contract execution, followed by *function pre-/post-conditions* for particular functional usage, as well as *rules* that cover cross-function properties. In most cases, temporal logic properties can be transformed into Hoare properties that could be instrumented into smart contract code [42]. Hence, existing works typically use Hoare-style specifications for vulnerability detection [56], [58], inconsistency detection [11], and correctness validation [42], [60].

Despite the promise of formal verification in enhancing the security and reliability of smart contracts, one notable challenge remains: the community still lacks the automated generation of comprehensive properties for effective formal verification of smart contracts. While several works have attempted this, they have not yet achieved the ultimate goal of automatically generating necessary properties, including invariants, pre-/post-conditions, and rules, for an unknown contract code. For example, InvCon [33], [36] can dynamically infer *likely* contract invariants and function pre-/post-conditions, but it requires historical transaction information. Likewise, Cider [32] and SmartInv [59] employed a machine-learning-

based approach to generate specifications through the training-and-inference paradigm, but only for the *invariant* properties. As a result, industry-leading players like Certora [10] have to rely on their own or crowdsourced experts [9] to manually write properties case by case, which hinders the effective formal verification of smart contracts on a large scale.

In this paper, we explore how recent advances in large language models (LLMs) could enable automated generation of comprehensive smart contract properties. Given LLMs' strong capability for in-context learning (see background in §II), we try to achieve effective transfer learning from existing human-written properties to customized properties for unknown code. More specifically, we embed existing properties into a vector database and retrieve a reference property for LLM-based in-context learning to generate a new property for a given code. In this way, we can generate diverse types of properties as long as there are existing samples for each type in the collected vector database. Moreover, compared to the training-and-inference paradigm mentioned above [32], [59], our approach does not require the error-prone labeling process (we can directly use existing raw property results, such as those from Certora auditing reports), nor re-training when there is updated data.

While the basic property generation process is relatively straightforward, it is challenging to ensure that the retrieval-augmented properties are (i) *compilable*, (ii) *appropriate*, and (iii) *verifiable*. To address these challenges, we employ three novel designs and implement them in an LLM-driven system called PropertyGPT. *First*, we use compilation and static analysis feedback as an external oracle to guide LLMs in iteratively revising the generated properties. *Second*, we consider multiple dimensions of similarity to rank the properties and find a balanced metric for all these dimensions. The resulting weighted algorithm thus identifies the top-K properties as the final result. *Third*, we design a dedicated prover to formally verify the correctness of the generated top-K properties.

To evaluate PropertyGPT, we collected 623 human-written properties from 23 Certora projects. We first split 90 of them as a ground-truth testing set and used the rest as reference properties. We found that PropertyGPT can cover 80% equivalent properties in the ground truth as judged by human experts, with a reasonable precision of 64%. Note that the additional properties (FPs) produced by PropertyGPT generally also hold, complementing the human-written ones. We further used all 623 properties as a knowledge base to supply PropertyGPT for detecting real-world CVEs and past attack incidents. Our results showed that PropertyGPT successfully detected 9 out of 13 CVEs and 17 out of 24 attack incidents. Moreover, during this process, PropertyGPT demonstrated sufficient generalizability in analyzing an entirely different dataset. Furthermore, we ran PropertyGPT on four real-world bounty projects to demonstrate its ability to find zero-day bugs. PropertyGPT successfully generated 22 bug findings, out of which 12 have been both confirmed and fixed, earning us a total of $8,256 in bounty rewards.

Contributions. We summarize the contributions as follows:

- We proposed a novel LLM-based property generation tool, PropertyGPT, to drive comprehensive formal verification for smart contracts, with the major step of retrieval-augmented property generation described in §V.
- To facilitate PropertyGPT, we also designed a property specification language (PSL) for smart contracts (§IV) and a dedicated prover for property verification (§VI).
- We conducted extensive experiments and ablation studies to evaluate PropertyGPT in various real-world settings; see §VII and §VIII.

Availability. The property dataset and raw experimental data are available at https://github.com/Pr0pertyGPT/PropertyGPT, while the prototype is being commercialized by our industry partner, MetaTrust Labs. A partially open-source version will be updated on the above GitHub link.

II. Preliminary

Large language models (LLMs), such as GPT-3.5 [40] and CodeLLama [44], have been widely used in many natural language processing tasks, such as text generation, translation, and summarization. GPT series models are trained on a large corpus of text data and have the potential to generate human-like text, while CodeLLama is a fine-tuned version of LLama 2 [55] on open-source code. The LLMs are pre-trained on a large corpus of text data and then fine-tuned on specific tasks and these datasets usually contain code from different programming languages. Additionally, the pre-trained LLMs have exercised its potential to revolutionize the traditional software tasks, e.g., code generation [8], repairing [41], [63], vulnerability detection [50].

In-context learning (ICL). Based on the pre-trained knowledge, LLMs could leverage existing human-written properties written with various specification languages. Yet, due to the limitations of the pre-training data and the efforts needed for training, LLMs may not be able to include the real-time information. To address this problem, in-context learning (ICL) mechanism have been proposed by offering LLMs with the ability to learn from the latest conversation or task context [46], [67]. In essence, in-context learning is a specialized kind of few-shot learning [7], basing itself on a few examples or a small amount of data to learn a new task.

Instead of using fine-tuning [59], in this work, we employ the in-context learning ability from the state-of-the-art GPT-4 model [4] for retrieval-augmented property generation.

III. PropertyGPT Overview

In this section, we present the overall design of PropertyGPT, which leverages LLMs' ICL capability to transfer existing human-written properties and generate customized properties for formally verifying unknown code. At a high level, PropertyGPT takes a piece of subject smart contract code as input and ultimately produces its corresponding properties along with the verification results.

As illustrated in Fig. 1, PropertyGPT consists of eight major steps: ① PropertyGPT first creates a vector database for reference properties by embedding their corresponding critical code. Note that the reference properties themselves will not be embedded because they are not the search key. ② Given a piece of subject code under testing (typically one function), PropertyGPT queries the vector database to ③ retrieve all similar code within the threshold and map each code to their

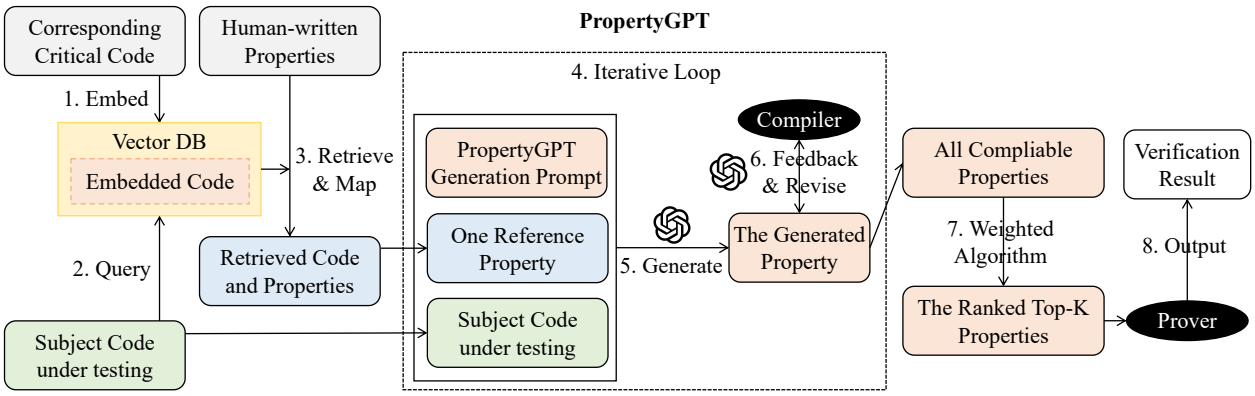

Fig. 1: A high-level workflow of PropertyGPT.

original reference properties. ④ All the reference properties are then tested with the subject code one by one in an iterative loop. ⑤ For each reference property, PropertyGPT employs a generation prompt to generate a candidate property for the subject code. ⑥ This candidate property is then checked by the compiler for grammar, and if it is not grammatically correct, it will be further revised according to the compiler's feedback using a revising prompt. ⑦ Eventually, we obtain a list of compilable properties and rank them according to our weighted algorithm for the top-K appropriate properties. ⑧ These properties are finally formally verified by our prover, aiming to discover smart contract vulnerabilities.

To explore and understand the details of PropertyGPT, we first introduce its property specification language in §IV. Following this, we describe the main process of LLM-based property generation and refinement in §V. Finally, we connect the generated properties with our dedicated formal prover in §VI for property verification.

IV. Property Specification Language

To bridge the gap between property generation in §V and formal verification in §VI, we propose an intermediate language in this section to specify the properties of smart contracts.

Fig. 2 illustrates our property specification language (PSL), which extends the popular smart contract programming language Solidity. In Solidity, a smart contract (SC) consists of a group of state variables recording persistent program state and a list of public functions allowing user interactions. The symbol $\bowtie$ represents a set of arithmetic, comparison, or logical operators, namely $\{+, -, /, >, <, ==, !\ =, >=, <=, \&, ||\}$. $bool\_expr\ \lfloor_{(v*,C*)}$ indicates boolean expressions involving state variables and constant values. PSL includes three kinds of properties with respect to different purposes. Invariants are properties that always hold true during contract execution and are defined over state variables; function pre-/post-conditions are properties that can be expressed in Hoare triples $\{p*\}func\{q*\}$, checking whether parameters and the modification of state variables satisfy functionality requirements. Scenario-based properties, defined on restricted environments, can be implemented as different rules by enforcing varied assumptions and customized assertions. It is worth noting that

$$
\begin{aligned}
v \in StateVar \quad tmp \in TemporalVar \quad \mathcal{C} \in Constant \\
SC &= v*; func* \\
func \in Function &= param*; stmt* \\
expr \in Expression &= tmp \mid v \mid \textbf{old}(v) \mid param \mid \mathcal{C} \mid expr \bowtie expr \\
\textbf{\textit{Spec}}(SC) &= inv*; \{p*\} func \{q*\}; rule* \\
inv \in Invariant &= bool\_expr \lfloor_{(v*,C*)} \\
p \in Precondition &= bool\_expr \lfloor_{(param, \textbf{old}(v), C)} \\
q \in Postcondition &= bool\_expr \lfloor_{(param, v, C)} \\
rule \in Rule &= \textbf{assume}(expr) * \mid func(expr*) \mid \textbf{assert}(expr)*
\end{aligned}
$$

Fig. 2: Property Specification Language (PSL).

the current PSL prototype supports safety properties since they are more security-related compared with liveness properties.

PSL brings several benefits to automate formal verification over existing works [10], [60]. VeriSol [60] demands that assertion-based properties must be inserted into smart contract code. Certora's Verification Language (CVL)[1] is powered by a closed-source commercial verification tool, which restricts our ability to use CVL to build a self-contained pipeline for PropertyGPT. Additionally, the learning curve of CVL is quite steep since it requires not only knowledge about smart contracts but also several non-trivial techniques, such as using a *hook* for data reading or writing, to handle the low-level execution model of smart contracts (an example of this limitation of CVL is provided in Appendix B). In contrast, writing or maintaining PSL specifications is easier because they share similar structures with the Solidity language [47]. Note that the semantics of PSL will be illustrated in §VI-B.

V. Property Generation and Refinement

With the targeted PSL introduced in §IV, our objective is to automatically generate properties written in PSL for the given code. The generated properties are the result of LLM-based transfer learning from existing human-written properties, which can be written in any specification language, not limited to PSL, such as CVL.

[1] https://docs.certora.com/en/latest/docs/cvl/index.html

Generation Prompt for Rule Properties

Based on the rule code ([rule code]) and the code example ([code example]), generate corresponding rule code for [contract code to be tested].
1. Using the syntax style demonstrated in the provided code example, generate rule code. Focus on structural and syntactic aspects rather than replicating specific variable or function names from the example.
2. $ is for a symbolic variable, such as $varA for symbolic varA.
3. MUST NOT replicate specific variable or function names from the [code example].
4. MUST focus on the structural and syntactic aspects from the [code example].
5. When writing the rule code, closely follow the syntax and style from the provided example, focusing on its structural and syntactic essence rather than copying specific names.
6. The output MUST NOT contain any elements not predefined in the contract or function.

[function code to be tested]: {func_code}
[contract code to be tested]: {contract_code}
[rule code]: {rule_property}
[code example]: {spec_grammar}

The Output MUST be in the form of:
rule [name of rule]() {{logic of rule}}
REMEMBER, ASSERT should not include an error message; just use the comparison operator directly.
REMEMBER, the rule must aim to test the function, not for another function.

Fig. 3: The prompt for generating rule properties.

PropertyGPT can achieve such powerful transfer learning fundamentally rooted in LLMs' capability for in-context learning (see §II). Nevertheless, we need to design a novel pipeline to facilitate this. Our idea is to mimic the RAG (retrieval-augmented generation) process in the NLP [29] or code [49] domain, using the reference properties retrieved to augment the generation of new properties. As previously illustrated in Fig. 1, we first detail how such retrieval-augmented property generation is conducted in PropertyGPT in §V-A. After that, PropertyGPT iteratively revises the LLM-generated properties to fix their compilation errors in §V-B. Furthermore, we design a weighted algorithm to help PropertyGPT rank all compilable properties and obtain only the top-K appropriate properties for the prover's verification in §VI.

A. *Retrieval-Augmented Property Generation*

Here we focus on the basic retrieval-augmented property generation that occurs from step ② to step ⑤ in Fig. 1. But the entire property generation process also includes property ⑥ revising and ⑦ ranking, which will be introduced in §V-B and §V-C, respectively.

Generation Prompt for Function Invariants/Conditions

Based on the following code ([condition code]), generate the corresponding precondition and postcondition code for [function code to be tested].
1. The basic syntax of preconditions and postconditions is in Solidity code format.
2. You can use the '__old__(xxx)' keyword if you need to reference the initial value of a variable.
3. You can directly use 'xxxx==/!=/¿/¡' without 'assert' or 'require' to compare the value of the variable.
4. MUST NOT use 'require' or 'assert' for assertions; just use operator comparison directly.
5. MUST NOT use the ternary operator in the precondition and postcondition, but USE 'if/else' expressions.
6. Exclude the event and implementation of the function itself, only output the precondition and postcondition of the function.
7. MUST NOT use any variables that I or the function have not defined, such as __result__, __return__, only follow the syntax I provide.
8. MUST NOT use 'if/else' expressions in the precondition and postcondition, but USE the ternary operator.
9. MUST NOT INVOKE other functions or other undefined variables or non-state variables in the contract, only use the state variables in the func_name itself.
10. Ignore and delete all conditions related to the return value.

[function code to be tested]: {func_code}
[condition code]: {condition_property}

The Output MUST be in the form of:

```
function {func_name}{{
    precondition{{
        Insert generated code here, ensuring it follows the syntax style of the example.
    }}
    postcondition{{
        Insert generated code here, ensuring it follows the syntax style of the example.
    }}
}}
```

Fig. 4: The prompt for generating invariants/conditions.

① **Knowledge Preprocessing.** One critical step in RAG-based systems is to first build a knowledge base, typically a vector database [49]. In PropertyGPT's scenario, we do not aim to extract "knowledge" from the existing human-written properties; instead, we use them as reference properties for LLMs' in-context learning. Therefore, we directly use the raw information from human-written reference properties to construct our vector database. As shown in Fig. 1, we embed the corresponding critical code of existing properties to build the search key used by RAG. Note that the reference properties themselves will not be embedded because they cannot be

queried against by the subject test code.

②-③ **Similar Example Retrieval.** With the vector database, we can retrieve similar reference properties given the subject code to enable one-shot LLM learning in subsequent steps. To do this, the subject code is also embedded in step ② and the dot product is calculated [24] with all the vectors in the database. The top similar code with the highest dot products are then retrieved, and their corresponding properties are returned as the result in step ③. Here, we use a conservative code similarity threshold (e.g., 0.8) to limit the number of retrieved reference properties (typically 10 to 20 properties), which is acceptable because PropertyGPT eventually uses a weighted algorithm in §V-C to rank only the top-K generated properties as the final result.

④-⑤ **In-context Learning.** All the reference properties are then tested with the subject code one by one in an iterative loop. For each reference property, PropertyGPT employs a generation prompt to generate a candidate property for the subject code, using the reference property as a one-shot example. Specifically, there are two types of generation prompts. One is used to generate global, cross-function rule properties, as shown in Fig. 3, and the other is used to generate function-level pre-/post-conditions, as illustrated in Fig. 4. Note that here we omit the generation prompt template for contract-level invariants because they are usually in a simpler form and equivalent to the one-for-all pre-/post-conditions for every public contract function. Both prompt templates consist of three parts: the first part details the generation instructions, the second part lists the code and reference property, and the third part defines the output property format. In particular, we supply a rule example in Fig. 3 to help LLMs understand the grammar of our rule properties, while the grammar of invariants/conditions is directly specified using natural language instructions in Fig. 4 as it is relatively simple. Additionally, since rule properties are cross-function, we provide not only the function code but also the entire contract code in the prompt template shown in Fig. 3.

To determine which generation prompt should be used, PropertyGPT leverages the type of the retrieved reference property. If the reference property is a rule, PropertyGPT uses the first type of prompt template to generate the property. Otherwise, if the reference property is classified as pre-/post-conditions, PropertyGPT uses the second type of prompt template.

B. Revising Property to Fix Compilation Errors

While the basic property generation process is relatively straightforward, one particular challenge is how to guarantee that the generated property is *compilable*. To address this challenge, we are inspired from [28], [52] and leverage the feedback from the compiler and static checking to iteratively revise the property until it is compilable or until it reaches the maximum number of attempts, as shown in step ⑥.

Leveraging Compiler Feedback. Our PSL compiler (see its compilation details in Appendix C) provides compilation error information, including detailed error locations and reasons, if the property cannot be successfully compiled. PropertyGPT thus leverages this feedback to instruct LLMs to iteratively revise the property. Specifically, we design and employ a

Common Prompt for Revising (Rule) Properties

Here is the rule I provided: {spec_res}.
When this code is compiled with a solc-like program, an error occurs: {error_info}.

Your task is to understand the rule I provided, fix the rule code, and correct the error within the rule. Refer to the contract code provided above.
Note, only modify the rule code; do not add other code. If the error is due to a non-existent variable, find feasible methods to reimplement it, or if it is not implementable, delete this line.
Here is the function code to be tested: {func_code}
Here is the contract code to be tested: {contract_code}
Provide me with the repaired rule code. The revised rule code must not be the same as the old rule code.
1. Using the syntax style demonstrated in the provided code example, generate a rule code. Focus on the structural and syntactic aspects rather than replicating specific variable or function names from the example.
2. $ is for a symbolic variable, such as $varA which symbolizes varA.

Rule Code Output MUST be in the form of:
`rule [name of rule](){{logic of rule}}`
REMEMBER, ASSERT does not include an error message, just use the operator comparison directly.
REMEMBER, the rule must aim to test the function [{function_name}], not for other functions.

Fig. 5: The common prompt for revising (rule) properties.

common prompt template as shown in Fig. 5 for revising rule properties; note that the prompt for revising function pre-/post-conditions is similar and is therefore not shown here. In this prompt, we ask LLMs to first understand the generated property code, identify and fix errors, maintain stylistic consistency throughout the process, and finally ensure that the revised rule code meets specific formatting requirements. We set a threshold for the maximum number of attempts to avoid an endless loop. In our experiment, as shown in §VIII-D, we found that 74% of properties could be successfully compiled with no revisions (63%) or with only one attempt, and 84% of all properties could be succesfully revised within five attempts. This makes the iterative process manageable.

Employing Static Checking. However, we found that even when the compiler does not report any errors, it does not mean that the generated property is fully correct. One notable issue is that LLM-generated properties could fail to include the target subject function, which renders the property meaningless. To address this, we perform additional static checking for all compilable properties passed in the above step. If PropertyGPT identifies that the property is missing the target subject function, it employs a special prompt template as shown in Fig. 6 (listing only the scenario for rule properties, similar

Special Prompt for Revising (Rule) Properties

Here is the knowledge rule you should learn from: {knowledge_rule}.
Here is the rule I provided: {spec_res}.
This rule lacks core function execution for: {function_name}.
The contract code is: {contract_code}.

Your task is to understand the rule I provided, absorb the knowledge provided, and fix the rule code by adding the core function execution for: {function_name}.
Here is the function code that needs to be tested: {func_code}.
Provide me with the revised rule code; the new rule code must not be the same as the old rule code.
1. Using the syntax style demonstrated in the provided code example, generate a rule code. Focus on the structural and syntactic aspects rather than replicating specific variable or function names from the example.
2. $ is for a symbolic variable, such as $varA for symbolic varA.

Rule Code Output MUST be in the form of:
`rule [name of rule](){{logic of rule}}`
REMEMBER, ASSERT does not include an error message, just use the operator comparison directly.
REMEMBER, the rule must aim to test the function [{function_name}], not for other functions.

Fig. 6: The special prompt for revising (rule) properties.

to Fig. 5). It is generally similar to the common revising prompt but explicitly addresses the rule's lack of testing for the core function execution. In this way, we not only guarantee that the generated property is grammatically correct but also functionally meaningful.

C. Ranking the Top-K Appropriate Properties

Another challenge is how to select the appropriate properties from all compilable properties as the final generation result. To do so, we propose a weighted algorithm to rank all the resulting properties, as shown in step ⑦. Specifically, we rank the properties based on the following four embedding-based metrics:

$X_{raw}(f, g)$: Similarity between contract code f and g.

$X_{summary}(f, g)$: Similarity between high-level functionality summaries of code f and g.

$Y_{raw}(\phi_1, \phi_2)$: Similarity between raw properties ϕ_1 and ϕ_2.

$Y_{summary}(\phi_1, \phi_2)$: Similarity between high-level property summaries for ϕ_1 and ϕ_2.

Note that we introduce $X_{summary}$ and $Y_{summary}$ to cope with that variety could exist for same-functionality code or same-semantic properties, where high-level natural language summaries are made by large language model for given code or properties.

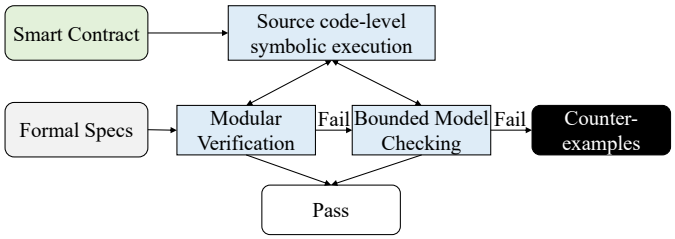

Fig. 7: Workflow of Property Verification.

Given an unknown code f, let ϕ_1 be its property generated corresponding to reference code g having property ϕ_2, we score ϕ_1 using a weighted algorithm as below.

$$Score(f, \phi_1) = \alpha * X_{raw}(f, g) + \beta * X_{summary}(f, g) + \gamma * Y_{raw}(\phi_1, \phi_2) + \eta * Y_{summary}(\phi_1, \phi_2) \quad (1)$$

where α, β, γ, η are coefficients and $\alpha + \beta + \gamma + \eta = 1$.

To tune these coefficients, we train a linear regression model to approximate the actual property score $\hat{Score}(f, \phi_1)$, with details available in Appendix D. In this work, for simplicity, we consider $\hat{Score}(f, \phi_1) = Y_{summary}(\phi_1, \hat{\phi_1})$, where $\hat{\phi_1}$ is the corresponding ground truth property of ϕ_1. We have conducted a primitive experiment on 3,622 properties generated by PropertyGPT, and the results show that α: 0.134, β: 0.556, γ: 0.141, and η: 0.168 are the optimal weights.

Consequently, properties with different scores are ranked in descending order, where we believe that properties with a higher rank are more likely to be important for the prover to verify.

VI. Property Verification

The properties generated by PropertyGPT are not only *compilable* and *appropriate* but also *verifiable*. Fig. 7 illustrates the workflow of our property verification process. Our prover accepts smart contracts written in Solidity, along with their corresponding PSL specifications. We employ forward symbolic execution to conduct a strongest postcondition analysis for each contract statement. Subsequently, we perform modular verification to determine whether these formal specifications have been accurately implemented in the smart contract and can produce a proof if the properties hold. In cases where the properties are violated, we use bounded model checking to verify whether the violated properties genuinely remain unfulfilled during contract execution. Upon encountering counterexamples, we can confidently conclude that the properties indeed fail to hold, suggesting the presence of vulnerabilities in the smart contracts, which necessitate further manual verification.

A. Modeling Smart Contract Execution

The runtime behaviors of smart contract execution rely on persistent contract states stored in the blockchain, transaction environment information, and specific contract statements to

execute. The persistent states are maintained in a group of contract state variables. The transaction message includes the current block timestamp, the caller, the callee contract, and its called method. Given a set of contract statements $S = [s]$, each statement execution can be modeled as a Hoare triple $\{\delta\}\ s\ \{\delta'\}$, where δ, δ' indicate program states before and after executing s. Unlike traditional programs, there is no crash in smart contract execution. Any unexpected behavior will cause a reversion of the smart contract transaction, leaving the contract state unchanged. Such reversion behaviors may affect availability, e.g., denial of service, but generally do not pose a threat to smart contract safety and therefore we exclude them from our analysis.

B. Verification Technique

We employ the small-step operational semantics of smart contracts to formally verify invariants, function pre/post-conditions, and rules. Readers can refer to KSolidity [23] for detailed operational semantics.

Function Pre/post-condition Verification. Given a function f, let p and q be its precondition and postcondition to verify, respectively. Specification $\{p\}\ f\ \{q\}$ is provable if and only if the below predicate holds

$$\forall \delta \in \Delta, sp(f, p \cap \delta) \implies q \tag{2}$$

where Δ encompass all feasible contract states.

We elide verification details for contract invariants because invariants can be deemed as a variant of function-level specifications that hold for every contract function, with the same precondition and postcondition standing there.

Rule Verification. Given a rule-based property *rule*, with user-given assumptions δ declared in `assume` statements and assertions q declared in `assert` statements, *rule* holds if and only if the below predicate holds

$$sp(rule, \delta) \implies q \tag{3}$$

Note that δ may not always be feasible contract state.

We utilize source code-level symbolic execution to conduct strongest postcondition analysis for Solidity smart contracts. Distinguishing ourselves from existing research [31], our novel symbolic execution approach implements comprehensive small-step semantics, enabling automated analysis of real-world, complex smart contracts. Although SolSee [31] has made strides in symbolic execution for Solidity smart contracts, it lacks support for certain critical features commonly used in smart contracts, such as the aggregated effect of intricate expressions and polymorphic handling during complex inheritance relationships. We address these limitations by meticulously adhering to the practices of the Solidity compiler, ensuring precise semantics of complex expressions. For instance, expressions are evaluated from left to right as specified by the compiler. Furthermore, our approach accurately resolves polymorphism during both the compilation and execution stages of smart contracts. To mimic actual contract execution, our symbolic execution approach maintains a comprehensive list of function signatures and revisits contract inheritance chains to determine the exact function implementation for ambiguous calls, such as `super().call()`, where `super` refers to an unknown parent contract.

To deal with the complexity of smart contracts, we implemented several over-approximation techniques to handle unknown or non-linear operation semantics. First, the behaviors of function calls to on-chain smart contracts remain unknown at the verification stage, so we assume all on-chain calls succeed but make their return data symbolic to accommodate any possible outcome. This approach is necessary because on-chain contracts may not be open source, and their inter-contract interactions can be overly complex, falling beyond the scope of our current research. Second, non-linear native functions, such as `sha3`, which computes the hash value of a string, are challenging to model precisely. To address this, we utilize uninterpreted functions [19] to capture their primary features, such as treating `sha3` as an injective function.

We employed modular verification and bounded model checking. During modular verification, we lift all state constraints by making all state variables symbolic. Correctness can be safely ensured when the specification properties hold accordingly. Otherwise, we perform bounded model checking to systematically explore all feasible states to find counterexamples that violate the property being verified. Any violated property and its counterexamples will be manually investigated to confirm the existence of vulnerabilities, similar to the approach used in SmartInv [59]. The depth of bounded model checking is capped at three by default, and we allow up to five loop iterations in the case of non-terminated execution.

VII. Implementation and Setup

We implemented PropertyGPT in around 3K lines of Python code for LLM-based property generation, and around 38K lines of C++ code for grammar support and verification of PSL property specifications. Additionally, for applying symbolic execution to smart contracts written in various versions of Solidity, we developed a converter to map smart contracts written in Solidity versions 0.6.x and 0.7.x into abstract syntax trees compatible with the latest Solidity version 0.8.x, where we have systematically investigated their syntax and semantic differences. We use the Z3 solver, version 4.11.2, to discharge symbolic constraints for path feasibility checking and property satisfiability checking.

A. Property Knowledge Collection

To obtain high-quality human-written properties as the knowledge base for in-context learning, we systematically analyzed 61 audit reports from the Certora platform, for which experts have written property specifications to facilitate formal analysis of smart contracts. These audit reports were published from 2019 to 2023. Through further investigation, we removed 38 projects whose contract code and raw properties were not available, and eventually, we collected 23 Certora projects, including 623 human-written properties, which will be detailed in Table VII in Appendix A. It is worth noting that for the selected projects, all human-written properties were collected, whether they represent violated or non-violated ones in their particular audited projects.

To study the characteristics of these properties, we employed the affinity propagation clustering algorithm [18] from the `sklearn` [2] library to discern property categories,

[2]https://scikit-learn.org/

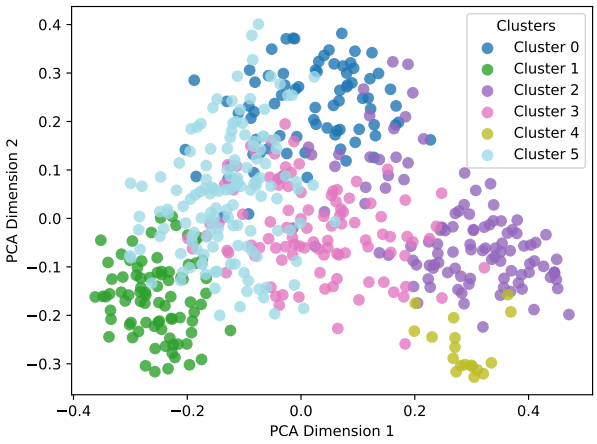

Fig. 8: The property cluster distribution after two-dimension PCA [20] reduction.

based on pairwise embedding similarity across the properties. Specifically, we performed some preliminary experiments and found that this setting *AffinityPropagation(damping=0.5, preference=-75, random_state=5)* could establish a good result for property clustering. Fig. 8 illustrates the distribution where six clusters are labeled with different colors. However, it is clear that overlapping exists among clusters, especially for clusters #3 and #5.

Furthermore, we investigated all the clusters and have the following classifications of human-written properties as shown in Table I. There are six property categories as follows:

- *DeFi*, involving the management of its essential protocol components including reserves, collateral, and liquidity pools;
- *Token*, which is the cornerstone of entire DeFi ecosystems, specifying standard behaviors such as token balance and critical operations like transfer and minting;
- *Arithmetic*, focusing on the correctness of numerical conversions and the consistency of asset splitting;
- *Usability*, examining the validity of operations containing timestamp-based constraints (Temporal Use) and operations with contract state-based constraints (State-dependent Use);
- *Governance*, which plays an important role in the management of decentralized applications, usually through a voting mechanism, concerning issues such as the transfer of voting power to a delegator and the double-voting problem;
- *Security*, checking for the presence of common vulnerability types including front running and overflow.

B. Experimental Setup

We use the large language model GPT-4-turbo provided by OpenAI through its API `gpt-4-0125-preview`. Regarding the model configuration, we adhere to the default settings where the temperature is 0.8, top-p is 1, frequency penalty and presence penalty are both 0, and the maximum response length

TABLE I: Characteristics of the collected Certora properties.

Category	Classification	Property Examples
DeFi	Reserve Collateral Liquidity	setReserveFactorIntegrity integrityOfisUsingAsCollateralAny checkBurnAllLiquidity
Token	Balance Transfer/TransferFrom Mint/Burn Approve	total_supply_is_sum_of_balances transferBalanceIncreaseEffect integrityMint, additiveBurn approvedTokensAreTemporary
Arithmetic	Numerical Conversion Asset Splitting	toElasticAndToBaseAreInverse1down moneyNotLostOrCreatedDuringSplit
Usability	Temporal Use State-dependent Use	timestamp_constrains_fromBlock unsetPendingTransitionMethods
Governance	Delegation Voting	integrityDelegationWithSig totalNonVotingGEAccountNonVoting
Security	Front run Overflow	cannotFrontRunSplitTwoSameUsers integrityOfMulDivNoOverflow

is 2000. Moreover, we calculate all embedding similarities using the pre-trained model `text-embedding-ada-002` from OpenAI. For property generation, we cap revising attempts at nine to limit LLM usage for better economics. All experiments were conducted on a Docker with Ubuntu 20.04 OS, an Intel Core Xeon 2.2 GHz processor, and 2GB RAM.

VIII. Evaluation

In this work, we aim to answer the following research questions (RQs):

- RQ1: (**Property Generation**) How accurately does PropertyGPT generate properties for smart contracts?
- RQ2: (**Vulnerability Detection**) How effectively does PropertyGPT discover smart contract vulnerabilities? Can PropertyGPT achieve state-of-the-art results?
- RQ3: (**Generalizability**) Does PropertyGPT have sufficient generalizability to enable powerful transfer learning?
- RQ4: (**Influencing Factors**) What factors influence the performance of PropertyGPT?
- RQ5: (**Impact**) How well does PropertyGPT find zero-day vulnerabilities in real-world smart contract projects?

Methodology. To answer RQ1, we divide Certora properties into a testing dataset and a "training" dataset as the knowledge base. We instruct PropertyGPT to generate properties for smart contracts in the testing dataset using smart contracts and their properties from the knowledge base. We compare the resulting properties by PropertyGPT with ground-truth human-written ones to investigate its effectiveness. Specifically, we randomly selected nine (40%) Certora projects as our testing dataset and then picked 10 properties for each project. Consequently, our testing dataset includes 90 ground truth properties from nine projects. During this experiment, PropertyGPT first extracts the subject function code where the ground truth properties are specified, and then queries the knowledge base to enable ICL to automate property generation.

To answer RQ2, we compare PropertyGPT with SmartInv [59], a concurrent work with ours, published recently in May 2024. We contacted the authors to obtain a copy

TABLE II: The evaluation benchmarks.

Benchmark	RQs
23 Certora projects (623 properties; 90 for testing)	RQ1, RQ4
13 CVEs + 24 projects from the SmartInv benchmark	RQ2, RQ3

of their source code and benchmark[3], which includes 60 attack incident projects that have suffered significant losses. Upon reviewing their benchmark, we identified several issues. Among the listed cases, 2 are repeated, 9 lack public exploit transactions (e.g., sherlockYields), 2 are not open-sourced, and 2 have incomplete code. Of the remaining cases, 11 are reentrancy attacks that could be easily remedied by adding the widely-used `nonReentrancy` modifier. Furthermore, 8 cases involve price manipulation attacks, which may be impractical to identify using the simple invariant properties that SmartInv generated. For example, Tolmach*et al.* [53] proposes a semi-automated formal composite analysis for DeFi protocols that detects such problems with fairness properties, while others use either runtime monitoring [62] to identify attack behavior or static analysis [26] to flag vulnerable code with predefined patterns. Through this deep analysis of their benchmark, we curated 24 attack incidents from the SmartInv benchmark for our evaluation. Additionally, we compare PropertyGPT with state-of-the-art tools [2], [16], [38], [50] on well-known smart contract CVEs. As of April, 2024, there are 577 smart contract CVEs, predominantly 477 integer overflows. To avoid bias, we randomly selected 13 CVEs of different types: three integer overflow cases, three access control vulnerabilities, four other logic bugs, etc., details of which are provided in Table IV.

Benchmarks. As shown in Table II, we evaluate the property generation process using Certora audited projects and test the applicability of PropertyGPT in vulnerability detection using well-known CVEs and attack incident projects studied by SmartInv. Additionally, RQ3 and RQ4 also use the corresponding benchmarks, the details of which will be elaborated later.

A. RQ1: Property Generation

We evaluated PropertyGPT on 90 ground-truth properties from nine Certora projects to investigate the effectiveness of property generation. Table III shows PropertyGPT's property generation results using the rest of Certora properties as the knowledge base. Note that this RQ only measures whether the generated properties match the ground-truth properties, where an FP indicates a property unmatched with the ground truth.

The first two columns show the project name and the number of properties written by Certora experts, i.e., #Property (Certora). The middle five columns list the number of properties generated by PropertyGPT, i.e., #Property (ours), true positives that are equivalent to the ground-truth properties (TP), the number of ground-truth properties hit by the properties generated (#Hit), the number of missed ground-truth properties (FN), as well as false positives (FP). The last three columns are recall, precision, and F1-score metrics where recall $= \frac{\#Hit}{\#Hit+FN}$, precision $= \frac{TP}{TP+FP}$, and F1-score $= \frac{2\times\text{recall}\times\text{precision}}{\text{recall}+\text{precision}}$. Because Certora properties are written in the proprietary Certora Verification Language (CVL) that supports formal verification of smart contracts at the EVM bytecode level, while PropertyGPT uses PSL to facilitate property formulation and verification at the source code level, there is currently no automated analysis tool available for equivalence checking between properties from these two distinct specification systems. Therefore, two authors with 5 years of research and auditing experience independently examined the equivalence between the ground-truth properties by Certora and the properties generated by PropertyGPT, with a third author breaking ties in case of disagreement. We welcome other researchers to conduct replication and verification using our released data, available at https://github.com/Pr0pertyGPT/PropertyGPT.

Table III shows that PropertyGPT can generate comprehensive properties with relatively high recall and reasonable precision. Most properties generated (26/42) are true positives, and most ground truth properties (8/10) can be successfully reproduced, achieving a satisfactory recall (0.80), reasonable precision (0.64), and fairly good F1-score (0.71). Delving into project-specific results, PropertyGPT was able to reproduce all the ground-truth properties for four projects including `aave_proof_of_reserve`, `celo_governance`, `ousd`, and `sushi_benttobox`. In contrast, PropertyGPT reproduced only two ground-truth properties for `openzepplin`, suffering the lowest recall and F1-score, although with the highest precision. We investigated the results and found this is largely because OpenZeppelin [3] is a foundational contract library that has been directly imported by nearly all real-world applications, and client code is unlikely to re-implement similar functionality, thus leading to the scarcity of reliable reference properties. In terms of precision, `opyn_gamma_protocol` achieves the lowest, reaching only 0.47. We investigated all 16 false positives about it and later recognized that 11 of these false positives are properties that hold for smart contracts but are not documented in the ground truth by Certora.

Answer to RQ1: PropertyGPT can generate comprehensive and high-quality properties, covering 80% equivalent properties in the ground truth as judged by human experts. Moreover, the additional properties (FPs) produced by PropertyGPT generally also hold, complementing the human-written ones.

B. RQ2: Vulnerability Detection

We investigate the applicability of PropertyGPT in the vulnerability detection task on well-known smart contract CVEs and the attack incident projects studied by SmartInv. Note that to mimic the situation of real-world deployment, we set the top-K to top-2, as measured by §VIII-D, as the best configuration starting from this section.

CVEs. Table IV demonstrates PropertyGPT's effectiveness in detecting 13 smart contract CVEs. We compared PropertyGPT with GPTScan [50], which employs the variable recognition ability of LLMs to instantiate high-level detection patterns for logic bugs, and Slither [16], a popular static analysis tool used to detect a wide range of common vulnerability types. In particular, since the original GPTScan covers only ten types of logic bugs, we have enhanced it with the recent unsupervised paradigm [49] and refer to the enhanced version as GPTScan+.

[3]https://github.com/sallywang147/attackDB

TABLE III: The property generation results for 90 ground-truth properties from nine Certora projects.

Project	#Property (Certora)	#Property (ours)	TP	#Hit	FN	FP	Recall	Precision	F1-score
aave_proof_of_reserve	3*	38	25	3	0	13	**1.00**	0.66	0.79
aave_v3	17	61	32	15	2	29	0.88	0.52	0.66
celo_governance	10	39	29	10	0	10	**1.00**	0.74	**0.85**
furucombo	10	23	11	7	3	12	0.70	0.48	0.57
openzepplin	10	2	2	1	9	0	**0.10**	**1.00**	**0.18**
opyn_gamma_protocol	10	30	14	8	2	16	0.80	**0.47**	0.59
ousd	10	100	67	10	0	33	**1.00**	0.67	0.80
radicle_drips	10	17	9	7	3	8	0.70	0.53	0.60
sushi_benttobox	10	70	49	10	0	21	**1.00**	0.70	0.82
Average	10	42	26	8	2	16	0.80	0.64	0.71

* This project contains only three human-written properties, so we picked seven more from `aave_v3`. Both are from the same institution.

TABLE IV: Vulnerability detection results for 13 CVEs.

CVE ID	*Description*	*Avg. Code Sim.*	*PropertyGPT*	*GPTScan+*	*Slither*	*Manticore*	*Mythril*
2021-34273	access control	0.693	✓	✓	×	×	×
2021-33403	overflow	0.666	✓	×	×	×	✓
2018-18425	logic error	0.704	✓	×	×	×	×
2021-3004	logic error	0.691	×	×	×	×	×
2018-14085	delegatecall	0.662	×	×	✓	×	×
2018-14089	logic error	0.661	✓	✓	×	×	×
2018-17111	access control	0.636	×	×	×	×	×
2018-17987	bad randomness	0.660	×	✓	×	×	×
2019-15079	access control	0.701	✓	×	×	×	×
2023-26488	logic error	0.682	✓	×	×	×	×
2021-34272	access control	0.693	✓	✓	×	×	×
2021-34270	overflow	0.671	✓	✓	×	×	✓
2018-14087	overflow	0.661	✓	×	×	×	✓

Additionally, Manticore [38] and Mythril [2] are two bytecode-level symbolic execution tools that automate comprehensive program state exploration and exploit generation of smart contract vulnerabilities. In Table IV, the first two columns list CVE names and their vulnerability types, while the remaining columns show the detection results by each tool.

The detection results presented in Table IV illustrate that PropertyGPT outperforms all the comparison tools by detecting 9 out of 13 CVEs, followed by GPTScan detecting five CVEs, Slither detecting only one delegatecall-related CVE, Mythril detecting three overflow-related CVEs, and Manticore detecting zero CVEs. We also investigated the remaining four CVEs that PropertyGPT failed to detect. It is unknown what valid properties can express the expectation of proper randomness and delegatecall use. CVE-2018-17111 is caused by the misuse of access control rather than the lack of access control, which is quite challenging for PropertyGPT to recognize this subtle difference during property generation.

The ability of PropertyGPT can be enhanced by the introduction of newly confirmed vulnerable code and properties into our knowledge database. As shown in Table I, the studied properties written by Certora experts seem to lack support for access control, which could limit the effectiveness of PropertyGPT in detecting other wild access control vulnerabilities, even though we realized that PropertyGPT has demonstrated a certain level of generalization capability in the aforementioned CVE detection results.

TABLE V: Evaluation results for 24 attack incident projects from the curated SmartInv benchmark.

Contracts	Detection	#Property	Avg. Code Similarity	Generation (seconds)	Verification (seconds)
dfxFinance	✓	8	0.675	235	7
AnySwap	×	11	0.692	518	7
Dodo	✓	17	0.703	1,182	19
Bancor	✓	19	0.699	1,948	9
BeautyChain	✓	5	0.676	104	9
Melo	✓	9	0.732	252	8
BGLD	×	9	0.654	229	39
GYMNetwork	✓	21	0.681	274	71
elasticSwap	✓	37	0.681	1,136	120
EulerFinance	×	23	0.669	376	43
monoSwap	✓	5	0.722	69	12
nimBus	✓	32	0.678	4,288	30
VTF	×	8	0.672	358	21
Nomad	✓	14	0.673	590	70
Umbrella	✓	14	0.688	404	25
Fortress Loan	✓	2	0.668	71	5
ShadowFinance	✓	25	0.683	551	80
Revest	✓	4	0.646	75	10
Cartel	✓	11	0.683	401	20
sushiSwap	×	10	0.686	419	20
ChainSwap	×	9	0.690	307	25
Ragnarok	✓	42	0.684	1,890	88
templeDao	✓	13	0.677	302	30
BabySwap	×	33	0.679	1,842	50
Overall		17	16	0.683	743	34

Attack Incidents. Table V shows the evaluation results on 24 attack incident projects from the curated SmartInv benchmark. Because the authors of SmartInv did not share their instrumented buggy contract code, and the ground truth and raw experimental results about their generated invariant properties are also missing, we perform a qualitative rather than a quantitative comparison with SmartInv and will discuss this in §IX. In Table V, the first two columns list project names and the amount of attack loss. The remaining columns show the detection results, the number of properties generated, the time used for property generation, and formal verification, respectively.

PropertyGPT successfully identified vulnerabilities in 17 out of 24 real-world attack incidents, on average generating 16 properties per project, spending around 12 minutes for property generation and only 34 seconds for formal verification. For the

remaining seven projects that PropertyGPT failed to detect, we studied the root causes of their reported vulnerabilities. We recognized that PropertyGPT does not support the runtime context of smart contracts, which may be essential for generating properties in particular use scenarios, for example, deflationary token abuse for the BGLD project, which we leave as future work.

Answer to RQ2: PropertyGPT can effectively detect vulnerabilities in both simple and complex smart contracts. Specifically, PropertyGPT has distinguished itself from the current state-of-the-art by detecting 9 out of 13 CVEs. Additionally, PropertyGPT achieved relatively good results in identifying logic bugs in 17 out of 24 attack incidents.

C. RQ3: Generalizability Measurement

Following the effective vulnerability detection demonstrated in §VIII-B, we are still interested in whether PropertyGPT has sufficient generalizability to enable powerful transfer learning. Given that PropertyGPT's vector database was constructed from Certora audit projects, we use the dataset of CVEs and attack incidents used in RQ2 to measure the generalizability of PropertyGPT's transfer learning during retrieval-augmented property generation for an entirely different dataset.

We use the similarity between the tested code and the retrieved reference code to *indirectly* measure the generalizability of PropertyGPT's property generation for the vulnerability detection results shown in Table IV and V. That is, if the similarity is lower, that indicates PropertyGPT's generalizability is stronger. For the metric of similarity, to avoid bias, we use the average of two popular similarity metrics, namely cosine similarity and Word Mover's Distance [64]. The results are then presented as standalone columns in Table IV and V, respectively.

Overall, we find that the mean average code similarity for all 26 successful cases (i.e., PropertyGPT successfully generated the correct property) is 0.68, with a range between 0.64 and 0.73, while for all 11 failed cases (i.e., PropertyGPT failed to generate the appropriate property or the verification was unsuccessful) it is 0.67, with a range between 0.63 and 0.69. This result has two implications. First, the absolute similarity values are within a reasonable range—not too high (meaning the test code is very similar to the reference code) nor too low (meaning the vector database fails to provide an effective reference case), indicating that PropertyGPT demonstrates sufficient generalizability in analyzing an entirely different dataset. Second, the average similarity for the failed cases is only slightly lower than for the successful cases, 0.67 vs. 0.68, which suggests that the failures were not due to PropertyGPT lacking the generalizability to test a new piece of code.

Answer to RQ3: PropertyGPT demonstrates sufficient generalizability in analyzing an entirely different dataset through our indirect measurement of the similarity between the tested code and the retrieved reference code.

D. RQ4: Influencing Factors

In our ablation study, we first systematically explored the impact of varying Top-K settings on the property selection process. Conducting different trials on the same Certora projects in RQ1, in Fig. 9, we plotted recall, precision, and F1-score for the resulting properties accordingly. It is clear that all the metrics are above 0.5. More importantly, when moving from top-1 to top-2, all the metrics increase, although precision has only a very slight increase from top-1 (0.64) to top-2 (0.65), and afterward, precision goes down and finally fluctuates around 0.60. Therefore, for the sake of efficiency, we suggest selecting top-2 properties generated for use by external community experts or other compatible program analysis tools.

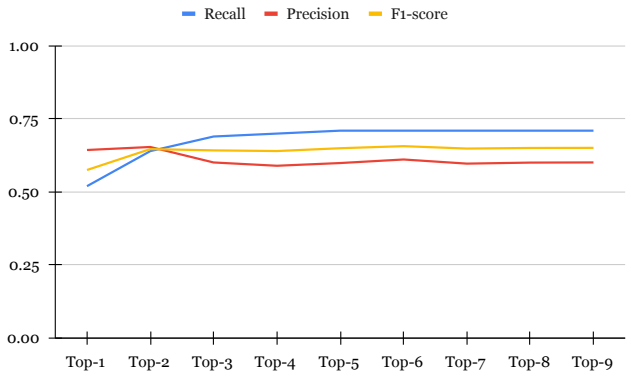

Fig. 9: The impact of Top-K settings on property accuracy.

TABLE VI: The success rate of property generation.

Method	#Compilable	#Failed	Success Rate
GPT-4.0-turbo w/o fix	234	136	0.63
PropertyGPT w/ §V-B	321	49	0.87

In addition, we delved into our property generation process with a focus on the success rate of property generation and property repair times using compiler feedback information. Table VI shows that GPT-4.0-turbo without revising or repair achieves a 63% success rate, which is quite lower compared to PropertyGPT (87%). Fig. 10 visualizes the distribution of property repair times, where we capped repair time at nine. We can see that most compilable properties (84%) can be generated with no more than five fix attempts. We also investigated the remaining 49 properties that could not be fixed by PropertyGPT and discovered the main compiler error message to be the use of undeclared variables, which may be addressed with a pattern-based approach [27].

Answer to RQ4: PropertyGPT can effectively generate compilable properties, with 84% of all properties being successfully revised within five attempts, and the highest success rate reaching 87%. Among them, the top-2 properties achieve the best balance between precision and recall.

E. RQ5: Real-world Impact

To demonstrate PropertyGPT's ability to identify zero-day vulnerabilities, we ran PropertyGPT on real-world bounty projects hosted by popular platforms such as Secure3 [45] and

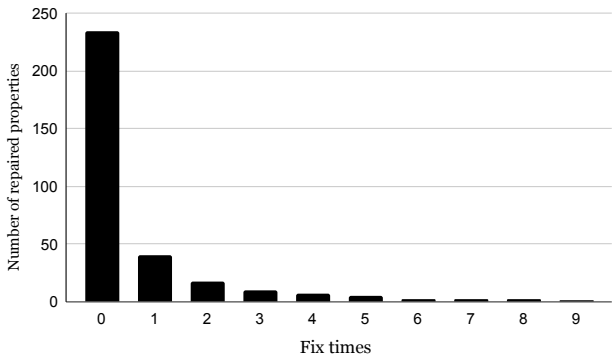

Fig. 10: The distribution of property fix times.

```
pragma solidity ^0.8.0;
contract SimplifiedStandaloneZkLink {
 address private _owner;
 mapping(address => bool) private _validators;
 uint256 public totalValidatorForwardFee;
 uint256 public
  ↪ totalValidatorForwardFeeWithdrawn;

 function withdrawForwardFee(uint256 _amount)
  ↪ external nonReentrant onlyValidator {
   require(_amount > 0, "Invalid amount");
   uint256 newWithdrawnFee =
    ↪ totalValidatorForwardFeeWithdrawn +
    ↪ _amount;
   require(totalValidatorForwardFee >=
    ↪ newWithdrawnFee, "Withdraw exceed");

   totalValidatorForwardFeeWithdrawn =
    ↪ newWithdrawnFee;
   (bool success, ) = msg.sender.call{value:
    ↪ _amount}("");
    require(success, "Withdraw failed");
    emit WithdrawForwardFee(_amount);
  }
}
```

Fig. 11: The vulnerable withdrawForwardFee function.

```
1  function withdrawForwardFee(uint256 _amount)
2  precondition {
3   _validators[msg.sender] == true;
4   _amount > 0;
5   old(totalValidatorForwardFee) >=
     ↪ old(totalValidatorForwardFeeWithdrawn) +
     ↪ _amount;
6  }
7  postcondition {
8   totalValidatorForwardFeeWithdrawn ==
     ↪ old(totalValidatorForwardFeeWithdrawn) +
     ↪ _amount;
9   totalValidatorForwardFee -
     ↪ totalValidatorForwardFeeWithdrawn ==
     ↪ old(totalValidatorForwardFee) -
     ↪ old(totalValidatorForwardFeeWithdrawn) -
     ↪ amount;
10 }
```

Fig. 12: The property generated for the case in Fig. 11.

```
function addEnvelope(
string calldata envelopeID, bytes32
 ↪ hashedMerkleRoot,
uint32 bitarraySize, address
 ↪ erc721ContractAddress,
uint256[] calldata tokenIDs
) public {
 require(tokenIDs.length > 0, "Trying to
  ↪ create an empty envelope!");
 MerkleEnvelopeERC721 storage envelope =
  ↪ idToEnvelopes[envelopeID];  // bug:
  ↪ overwrite storage.
 envelope.creator = msg.sender;
 envelope.unclaimedPasswords =
  ↪ hashedMerkleRoot;
 envelope.isPasswordClaimed = new
  ↪ uint8[](bitarraySize / 8 + 1);
 envelope.tokenAddress =
  ↪ erc721ContractAddress;
 envelope.tokenIDs = tokenIDs;
        ...
}
```

Fig. 13: The vulnerable addEnvelope function.

Code4Rena [12]. PropertyGPT successfully generated 22 bug findings for 4 projects, 12 of which have been both confirmed and fixed. In return, we received $8,256 in bug bounties from vendors. In this section, for case studies, we list two zero-day bugs that have been fixed for responsible disclosure, and we do not mention their project source to respect the anonymity policy.

Fig. 11 shows that the `withdrawForwardFee` function contains a critical vulnerability allowing validators to withdraw more than their allocated share of forwarding fees, potentially leading to unfair distributions and loss of funds. The vulnerability arises because the function fails to track and limit individual validators' withdrawals according to their proportionate share. It calculates the new total withdrawn fee by simply adding the requested withdrawal amount `_amount` to `totalValidatorForwardFeeWithdrawn` (Lines 10-13), without considering the requesting validator's entitled share. The only check performed is against the total collected forwarding fees, ensuring that the new total withdrawn does not exceed this amount (Line 11). However, this does not prevent individual validators from withdrawing more than their share.

PropertyGPT detected this vulnerability through the generation and verification of function pre-/post-conditions listed in Fig. 12. The pre-conditions(Lines 3-5) hold for this contract as they precisely capture the constraints `onlyModifier` and the other two `require` statements. The post-conditions describe the expected functionality. However, one of the post-conditions (Line 9), `totalValidatorForwardFee - totalValidatorForwardFeeWithdrawn == old(totalValidatorForwardFee) - old(totalValidatorForwardFeeWithdrawn) - amount`, does not hold, thus identifying the contract vulnerability in Fig. 11.

```
rule
 ↪  checkAddEnvelopeCorrectSenderAndCreator()
 ↪  {
 assume(msg.sender == 0x0000000000000000000
  ↪  0000000000000001);
 string memory envelopeID = "uniqueID";
 bytes32 hashedMerkleRoot =
  ↪  0x1234567890abcdef1234567890abcdef123456
  ↪  7890abcdef1234567890abcdef;
 uint32 bitarraySize = 128;
 address erc721ContractAddress = 0x0000000000
  ↪  0000000000000000000000000002;
 uint256[] memory tokenIDs = new uint256[](1);
 tokenIDs[0] = 12345;

 MerkleEnvelopeERC721 storage $envelopeBefore
  ↪  = idToEnvelopes[envelopeID];
 bool $existsBefore =
  ↪  ($envelopeBefore.creator != address(0));

 addEnvelope(envelopeID, hashedMerkleRoot,
  ↪  bitarraySize, erc721ContractAddress,
  ↪  tokenIDs);

 MerkleEnvelopeERC721 storage $envelopeAfter
  ↪  = idToEnvelopes[envelopeID];
 bool $correctlyAdded =
  ↪  ($envelopeAfter.creator == msg.sender);
 bool $notExistsBefore = ! $existsBefore;

 assert($correctlyAdded && $notExistsBefore);
}
```

Fig. 14: The property generated for the case in Fig. 13.

Fig. 13 shows a vulnerable `addEnvelope` function where it does not enforce uniqueness of envelope (Line 7), where existing storage can be overwritten arbitrarily. Fig. 14 presents the property generated by PropertyGPT to detect such issue. Interestingly, PropertyGPT can skillfully construct varied input data (Line 3-8). When the function call (Line 13) succeeds, we check the condition (Line 19) that same-id envelope does not exist before and the envelope creator equals to the current caller. In other words, this condition disallows calling `addEnvelope` function with same envelope id and ensures the effect of `addEnvelope` will set envelope creator to be the function caller. Due to the overwrite bug in Fig. 13, this assertion does not hold for this function.

F. Threats to Validity

Internal Validity. We evaluated the effectiveness of PropertyGPT on an established Certora property dataset. Nevertheless, there is lack of equivalence checking tool between Certora-style properties and our proposed PSL-style properties generated by PropertyGPT. To mitigate this issue, three authors independently reviewed these properties to determine equivalence and we release all the properties generated and the according labeling results for public use.

External Validity. Our findings in vulnerability detection may not apply to other kinds of smart contracts and other types of contract vulnerabilities. In this work, we evaluated PropertyGPT on 13 representative smart contract CVEs covering many kinds of vulnerabilities and 24 real-world victim projects of different application domains. Moreover, we generated 24 bug findings for high-profile projects to audit and 12 have been confirmed and fixed, with $8,256 bounty reward. Therefore, PropertyGPT offers a practical formal verification technique for detecting a broader range of smart contract vulnerabilities.

IX. Related Work

Vulnerability Detection. Numerous automated and semi-automated analysis tools have been proposed to detect smart contract vulnerabilities. On the one hand, static analysis tools analyze code sequences along the abstract syntax tree of contracts [16] or use a fact-based transformation and query system [6], [57] to flag weaknesses and vulnerabilities against expert-written patterns. In contrast, with test oracles, fuzzers examine runtime behaviors including operation traces [22], [39] and execution effects [35], [58] for exploit generation, usually leading to higher precision but lower recall compared with static analyses. On the other hand, formal verification has been widely employed in techniques to ensure smart contract correctness. Automated tools like Manticore [38] and Mythril [2] use symbolic execution to explore as many program states as possible to identify vulnerable behavior with a set of predefined detection rules. Semi-automated tools require users to provide specification properties, including invariants [56], function pre-/post-conditions [60], temporal properties [42], [48], and other customized rules [10], [21].

PropertyGPT distinguishes itself by automating property generation using a large language model and proposing a powerful prover based on source code-level symbolic execution of smart contracts, supporting the detection of a wide range of contract vulnerabilities.

Property Generation. Static inference [51], [60] and dynamic inference [33], [36] have been applied in property generation for smart contracts, and recently, machine-learning-based models have also been used for invariant property generation [32], [59]. VeriSol [60] applies the Houdini algorithm [17] to reason about correct invariant properties from a set of hypothesized candidates. SolType [51] discovers type invariants for Solidity smart contracts, requiring developers to add refinement type annotations to the contracts. However, their properties are limited to arithmetic operations to secure smart contracts from integer overflow and underflow. InvCon [33] and its subsequent work [36] apply dynamic invariant detection and static inference to produce contract invariants and function pre-/post-conditions, but they necessitate contracts having sufficient transaction histories.

Our work aligns with previous efforts in machine-learning-based approaches, i.e., Cider [32] and SmartInv [59]. Cider uses a deep reinforcement learning approach but only generates *likely* invariant properties, while PropertyGPT can verify all the properties generated with a prover. Both SmartInv and PropertyGPT are powered by large language models. PropertyGPT differs from SmartInv in that we use in-context learning rather than fine-tuning, and our properties generated extend beyond function pre-/post-conditions.

LLM-based Security Systems. By combining LLMs, various security tasks have been addressed more effectively. Sun *et al.* [50] proposed GPTScan, and Li *et al.* [30] introduced

methods that combine LLMs with static program analysis for vulnerability detection, covering more types of vulnerabilities than traditional tools. Beyond vulnerability detection, LLMs have been used for other security tasks. Deng *et al.* [13] proposed TitanFuzz, which utilizes LLMs to guide the fuzzing of deep learning libraries such as PyTorch and TensorFlow. They also introduced FuzzGPT [14] to synthesize unusual programs for fuzzing vulnerabilities. ChatAFL [37] utilizes LLMs to guide the fuzzing of protocols by interpreting the protocol documents. Additionally, LLMs have been applied to program repairing tasks, such as ACFix [65] and ChatRepair [63].

X. Conclusion

In this paper, we proposed retrieval-augmented property generation for smart contracts by utilizing LLMs' in-context learning capabilities. We implemented this approach in a tool called PropertyGPT and addressed challenges to ensure the generated properties are compilable, appropriate, and runtime-verifiable. Our evaluation results indicate that PropertyGPT can detect many real-world contract vulnerabilities, especially in high-profile projects, collectively receiving $8,256 in bounty rewards from vendors. For future work, we plan to include more comprehensive contract context information, such as documentation, in our approach and enhance PropertyGPT with richer property knowledge from various sources.

Acknowledgement

We thank all the reviewers for their constructive feedback on this paper. This research/project is supported by the Singapore Ministry of Education Academic Research Fund Tier 1 (RG12/23), the Nanyang Technological University Centre for Computational Technologies in Finance (NTU-CCTF), the National Research Foundation Singapore and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-RP-2020019), and the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of NTU-CCTF, National Research Foundation, Singapore and Cyber Security Agency of Singapore.

References

[1] "Binance Smart Chain," https://docs.binance.org/smart-chain/guides/bsc-intro.html, 2020, introduction of Binance Smart Chain.

[2] "Mythril," https://github.com/Consensys/mythril, 2024.

[3] "Openzeppelin–a library for secure smart contract development," https://github.com/OpenZeppelin/openzeppelin-contracts, 2024.

[4] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[5] A. Arora, Kanisk, and S. Kumar, "Smart contracts and nfts: non-fungible tokens as a core component of blockchain to be used as collectibles," in *Cyber Security and Digital Forensics: Proceedings of ICCSDF 2021*. Springer, 2022, pp. 401–422.

[6] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: a smart contract security analyzer for composite vulnerabilities," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 454–469.

[7] T. B. Brown, B. Mann, N. Ryder, and Others, "Language models are few-shot learners," no. arXiv:2005.14165, Jul. 2020, arXiv:2005.14165 [cs].

[8] 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*, ser. OSDI'08. USA: USENIX Association, 2008, pp. 209–224.

[9] Certora, "A community of hackers putting their formal verification skills to the test and earning rewards from leading protocols," https://www.certora.com/contests.

[10] ——, "Securing web3 with decentralized intelligence," https://www.certora.com/.

[11] 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*, 2019, pp. 1503–1520.

[12] Code4rena, "Keeping high severity bugs out of production," https://code4rena.com/.

[13] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Seattle WA USA: ACM, Jul. 2023, p. 423–435.

[14] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries." IEEE Computer Society, Nov. 2023, p. 830–842.

[15] Y. Fang, D. Wu, X. Yi, S. Wang, Y. Chen, M. Chen, Y. Liu, and L. Jiang, "Beyond "protected" and "private": An empirical security analysis of custom function modifiers in smart contracts," in *Proc. ACM ISSTA*, 2023.

[16] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.

[17] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for esc/java," in *International Symposium of Formal Methods Europe*. Springer, 2001, pp. 500–517.

[18] B. J. Frey and D. Dueck, "Clustering by passing messages between data points," *science*, vol. 315, no. 5814, pp. 972–976, 2007.

[19] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey, "An abstract domain of uninterpreted functions," in *Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17*. Springer, 2016, pp. 85–103.

[20] B. M. S. Hasan and A. M. Abdulazeez, "A review of principal component analysis algorithm for dimensionality reduction," *Journal of Soft Computing and Data Mining*, vol. 2, no. 1, pp. 20–30, 2021.

[21] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu *et al.*, "KEVM: A complete formal semantics of the Ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018, pp. 204–217.

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

[23] J. Jiao, S. Kan, S.-W. Lin, D. Sanan, Y. Liu, and J. Sun, "Semantic understanding of smart contracts: Executable operational semantics of solidity," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1695–1712.

[24] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019.

[25] J. Kim and S. Kim, "A survey of decentralized finance (defi) based on blockchain," *Journal of the Korea Society of Computer and Information*, vol. 26, no. 3, pp. 59–67, 2021.

[26] Q. Kong, J. Chen, Y. Wang, Z. Jiang, and Z. Zheng, "Defitainter: Detecting price manipulation vulnerabilities in defi protocols," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1144–1156.

[27] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, vol. 25, pp. 1980–2024, 2020.

[28] U. Kulsum, H. Zhu, B. Xu, and M. d'Amorim, "A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback," in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, 2024, pp. 103–111.

[29] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," no. arXiv:2005.11401, Apr. 2021, arXiv:2005.11401 [cs].

[30] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," in *Proc. ACM Program. Lang.*, Nov. 2023.

[31] S.-W. Lin, P. Tolmach, Y. Liu, and Y. Li, "Solsee: a source-level symbolic execution engine for solidity," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1687–1691.

[32] J. Liu, Y. Chen, B. Tan, I. Dillig, and Y. Feng, "Learning contract invariants using reinforcement learning," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–11.

[33] 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*, 2022, pp. 1–4.

[34] Y. Liu, Y. Li, S. Lin, and C. Artho, "Finding permission bugs in smart contracts with role mining," in *Proc. ACM ISSTA*, 2022.

[35] 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)*. New York, NY, USA: ACM, Jul. 2022, pp. 716–727.

[36] Y. Liu, C. Zhang, and Y. Li, "Automated invariant generation for solidity smart contracts," *arXiv preprint arXiv:2401.00650*, 2024.

[37] R. Meng, M. Mirchev, M. Bohme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *Proceedings of the Symposium on Network and Distributed System Security 2024*.

[38] 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)*. IEEE, 2019, pp. 1186–1189.

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

[40] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," 2022, arXiv:2203.02155.

[41] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2023, pp. 2339–2356. [Online]. Available: https://ieeexplore.ieee.org/document/10179324/

[42] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," in *2020 IEEE symposium on security and privacy (SP)*. IEEE, 2020, pp. 1661–1677.

[43] 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.

[44] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," 2023, arXiv:2308.12950.

[45] Secure3, "Securing web3 with decentralized intelligence," https://www.secure3.io/.

[46] S. Shin, S.-W. Lee, H. Ahn, S. Kim, H. Kim, B. Kim, K. Cho, G. Lee, W. Park, J.-W. Ha, and N. Sung, "On the effect of pretraining corpora on in-context learning by a large-scale language model," in *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Seattle, United States: Association for Computational Linguistics, Jul. 2022, pp. 5168–5186. [Online]. Available: https://aclanthology.org/2022.naacl-main.380

[47] "Solidity," https://solidity.readthedocs.io/en/v0.5.1/, 2022.

[48] J. Stephens, K. Ferles, B. Mariano, S. Lahiri, and I. Dillig, "Smartpulse: Automated checking of temporal properties in smart contracts," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 555–571.

[49] Y. Sun, D. Wu, Y. Xue, H. Liu, W. Ma, L. Zhang, M. Shi, and Y. Liu, "Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning," no. arXiv:2401.16185, Jan. 2024, arXiv:2401.16185 [cs].

[50] 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*, 2024, pp. 1–13.

[51] B. Tan, B. Mariano, S. K. Lahiri, I. Dillig, and Y. Feng, "Soltype: refinement types for arithmetic overflow in solidity," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–29, 2022.

[52] S. Thakur, J. Blocklove, H. Pearce, B. Tan, S. Garg, and R. Karri, "Autochip: Automating hdl generation using llm feedback," *arXiv preprint arXiv:2311.04887*, 2023.

[53] P. Tolmach, Y. Li, S.-W. Lin, and Y. Liu, "Formal analysis of composable defi protocols," in *Financial Cryptography and Data Security. FC 2021 International Workshops: CoDecFin, DeFi, VOTING, and WTSC, Virtual Event, March 5, 2021, Revised Selected Papers 25*. Springer, 2021, pp. 149–161.

[54] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, "A survey of smart contract formal specification and verification," *ACM Computing Surveys (CSUR)*, vol. 54, no. 7, pp. 1–38, 2021.

[55] H. Touvron, L. Martin, K. Stone, and et al., "Llama 2: Open foundation and fine-tuned chat models," 2023, arXiv:2307.09288.

[56] *Echidna*, Trail of Bits, 2019. [Online]. Available: https://github.com/trailofbits/echidna

[57] 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*, 2018, pp. 67–82.

[58] H. Wang, Y. Liu, Y. Li, S.-W. Lin, C. Artho, L. Ma, and Y. Liu, "Oracle-supported dynamic exploit generation for smart contracts," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[59] S. J. Wang, K. Pei, and J. Yang, "Smartinv: Multimodal learning for smart contract invariant inference," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 126–126.

[60] Y. Wang, S. K. Lahiri, S. Chen, R. Pan, I. Dillig, C. Born, and I. Naseer, "Formal specification and verification of smart contracts for azure blockchain," *arXiv preprint arXiv:1812.08829*, 2018.

[61] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[62] S. Wu, D. Wang, J. He, Y. Zhou, L. Wu, X. Yuan, Q. He, and K. Ren, "Defiranger: Detecting price manipulation attacks on defi applications," *arXiv preprint arXiv:2104.15068*, 2021.

[63] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," no. arXiv:2304.00385, Apr. 2023, arXiv:2304.00385 [cs].

[64] X. Yi, D. Wu, L. Jiang, Y. Fang, K. Zhang, and W. Zhang, "An empirical study of blockchain system vulnerabilities: modules, types, and patterns," ser. ESEC/FSE 2022. New York, NY, USA: Association

for Computing Machinery, 2022, p. 709–721. [Online]. Available: https://doi.org/10.1145/3540250.3549105

[65] L. Zhang, K. Li, K. Sun, D. Wu, Y. Liu, H. Tian, and Y. Liu, "Acfix: Guiding llms with mined common rbac practices for context-aware repair of access control vulnerabilities in smart contracts," 2024.

[66] Z. Zhang, B. Zhang, W. Xu, and Z. Lin, "Demystifying exploitable bugs in smart contracts," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 615–627.

[67] Y. Zhu, J. R. A. Moniz, S. Bhargava, J. Lu, D. Piraviperumal, S. Li, Y. Zhang, H. Yu, and B.-H. Tseng, "Can large language models understand context?" no. arXiv:2402.00858, Feb. 2024, arXiv:2402.00858 [cs].

APPENDIX

A. Supplementary Material

Table VII lists the raw information for all the 61 Certora projects, of which we collected 23 projects with available code and properties.

B. An Example of CVL

```
hook Sstore _checkpoints[KEY address account][INDEX uint32 index].votes uint224 newVotes (uint224 oldVotes) STORAGE {
 havoc userVotes assuming
   userVotes@new(account) == newVotes;

 havoc totalVotes assuming
   totalVotes@new() == totalVotes@old() + newVotes - userVotes(account);

  havoc lastIndex assuming
    lastIndex@new(account) == index;
}
```

Fig. 15: Part of the CVL specification for ERC20Votes.

Following the description in §IV, we provide a CVL example here to illustrate its reliance on a low-level rather than a high-level execution model for smart contracts.

Fig. 15 shows part of the CVL specification for ERC20Votes[4]. It specifies the semantic behavior for updating the field item `votes` of `mapping(address => Checkpoint[]) private _checkpoints`, a data structure used to record user votes. Essentially, the specification states that after users cast new votes, the recorded user votes should be updated accordingly, the total votes should be recalculated, and the last index, i.e., the size of the user checkpoint array, should also be updated accordingly.

In our experience, writing such specifications is non-trivial for smart contract developers, as Fig. 15 requires explicit semantic definitions for each field item of complex data structures. Additionally, explicitly declaring low-level opcodes such as `sstore` and specifying the variable type of involved field items (e.g., STORAGE indicating a storage variable) burdens users with subtle and exhaustive details of smart contract data storage.

[4]https://github.com/Pr0pertyGPT/PropertyGPT/blob/main/certora_projects/openzepplin/specs/ERC20Votes.spec

In comparison, PSL extends Solidity, and its execution follows the well-studied Solidity semantics, allowing users to write specifications without needing to know the intricate details of the verification process.

C. Compilation Phase of PSL Specifications

PSL is developed as a variant of Solidity, where we modify the Solidity compiler by adding new syntactic structures and imposing certain constraints to support the compilation of specifications written in PSL. Specifically, we add the following keywords by modifying Solidity compiler v0.8.17: `invariant` for declaring invariant specification code blocks, `rule` for declaring customized rule code blocks, and `precondition` and `postcondition` for function preconditions and postconditions. We also add the keyword `assume` for verification purposes (c.f. Fig. 2). Moreover, we implement checks to permit only expression statements in invariant, precondition, and postcondition specifications. Through these measures, our customized PSL compiler can accept specifications written in PSL and validate their syntactic correctness.

D. Learning Optimal Coefficients via Training a Linear Regression Model

Following the introduction in §V-C, this section details how we train a linear regression model to learn optimal coefficients, which is vital for ranking the top-k appropriate properties as the final property generation result.

Data Preparation. We used PropertyGPT to generate a total of 3,622 property generation records. Specifically, for each property generation, we randomly selected a known subject code f from the Cetora dataset, and then PropertyGPT yielded a rule property ϕ_1 for it. We measured $X_{raw}(f,g)$, $X_{summary}(f,g)$, $Y_{raw}(\phi_1,\phi_2)$, and $Y_{summary}(\phi_1,\phi_2)$. Additionally, we computed the actual score $\hat{Score}(f,\phi_1) = Y_{summary}(\phi_1,\hat{\phi_1})$, where $\hat{\phi_1}$ is the known rule property of f in the Certora dataset. Our training aims to produce $Score(f,\phi_1)$ to approximate $\hat{Score}(f,\phi_1)$ using the four features mentioned in §V-C.

Model Training. We trained an Ordinary Least Squares (OLS) linear regression model using the above data and evaluated the model's performance using multiple metrics: Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Coefficient of Determination (R^2), Mean Absolute Percentage Error (MAPE), and Mean Deviation Error (MDE). Finally, we obtained the following weight coefficients and performance results:

- Coefficients: α: 0.134, β: 0.556, γ: 0.141, and η: 0.168
- Performance metrics: MAE: 0.0239, MSE: 0.0008, RMSE: 0.0291, R^2: 0.1294, MAPE: 2.7810, and MDE: -0.0022

These coefficient settings achieve relatively good performance. Additionally, we conducted other primitive experiments and found that combining fewer features in the prediction model led to a decline in performance metrics. Therefore, we believe that the selected four-feature combination can rank the generated properties with reasonably high accuracy.

TABLE VII: The raw information for all the 61 Certora projects.

Report Name	Year	Month	Included	#Property
Aave CLSynchronicity Price Adapter	2022	December	×	
Aave GHO Stablecoin	2023	March	✓	35
Aave Governance V2 Update	2022	September	×	
Aave L2 Bridge	2022	July	✓	42
Aave Proof of Reserve	2022	November	✓	3
Aave Protocol V2	2020	December	✓	17
Aave Rescue Mission Phase 1	2023	January	✓	1
Aave Staked Token v1.5	2023	February	✓	11
Aave Static aToken	2023	April	✓	24
AAVE Token V3	2022	September	×	
Aave V2 AStETH	2022	August	×	
Aave V3	2022	January	✓	59
Aave V3 BTC.b Listing Steward	2022	September	×	
Aave V3 MAI & FRAX Listing Stewards	2022	August	×	
Aave V3 PR #820	2023	March	×	
Aave V3 sAVAX Listing Steward	2022	July	×	
Aave V3 sUSD Listing Steward	2022	August	×	
Aave V3.0.1	2022	December	×	
Aave Vault	2023	June	✓	16
Aave-StarkNet L1-L2 Bridge	2022	October	✓	10
Balancer	2022	September	×	
Balancer V2	2021	April	×	
Balancer V2 (Issues only)	2021	April	×	
Balancer's Timelock Authorizer Verification Report	2023	May	×	
Benqi's Liquid Staking Contracts	2022	April	×	
Celo Core Contracts Release 4	2021	May	×	
Celo Governance Protocol	2020	May	✓	35
Compound V1 Price Oracle	2018	September	×	
Compound V3 Comet	2022	July	×	
Compound's MoneyMarket v2 formal verification report	2019	August	✓	41
Compound's Open-Oracle with Uniswap Anchor	2020	August	×	
Daoism	2022	October	×	
dcSpark	2022	December	×	
dForce Lending Protocol	2021	February	×	
Euler	2021	November	×	
Furucombo	2021	May	✓	20
Kashi Lending Protocol	2021	March	×	
Keep's Fully-backed bonding contract	2020	November	✓	13
Lido V2	2023	April	✓	1
Lyra	2021	May	×	
Master Chef V2	2021	April	×	
Notional Finance V2	2021	November	✓	30
OOPSLA'2020	2020	-	×	
Open Zeppelin	2022	April	✓	80
Open Zeppelin	2022	June	×	
OpenZeppelin Governance contracts	2021	December	×	
Opyn Gamma Protocol	2020	December	✓	33
Orchid's Smart Contracts	2019	December	×	
Origin OUSD Token	2021	February	✓	16
Popsicle V3 Optimizer	2021	November	✓	21
Radicle Drips	2023	January	✓	36
Rolla Finance	2021	August	×	
SaaS Verification Report by Blockswap Labs	2022	July	×	
SaaS Verification Report by Silo	2022	July	×	
Sushi BentoBox	2021	February	✓	22
Sushi Compound Strategy	2021	April	×	
SushiSwap ConstantProductPool	2021	November	×	
SushiSwap TridentRouter	2021	November	×	
Synthetix Multi-Collateral Loans	2020	December	×	
Trader Joe	2022	March	✓	98
Zesty	2021	July	×	