GPTScan_But_Bigger: Extending the Smart Contract Vulnerability Detection Tool

Owen Joslin
otj8625@rit.edu

Rochester Institute of Technology
Golisano College of Computing and Information Sciences
Department of Cybersecurity

William Joslin
wjp3799@rit.edu

CSEC759: Advanced Software Security Analysis
December 8th, 2024

***Abstract*—Smart contracts are susceptible to various vulnerabilities, potentially resulting in significant financial losses. Existing analysis tools primarily focus on vulnerabilities with predefined control- or data-flow patterns, such as reentrancy and integer overflow. However, recent studies on Web3 security bugs reveal that approximately 80% of these issues remain undetectable by current tools due to the lack of domain-specific property descriptions and verification methods.**

Leveraging advances in Large Language Models (LLMs), this paper expands on the prior work "GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis" by Sun et al. in many ways. Their paper explored how Generative Pretraining Transformers (GPT) can assist in identifying logic vulnerabilities. Their tool, GPTScan, was the first tool to integrate GPT with static analysis for detecting smart contract logic vulnerabilities where—unlike other approaches that relied solely on GPT—which are prone to high false positive rates and limited by the model's pre-trained knowledge—GPTScan employed GPT as a powerful code comprehension assistant alongside static analysis.

Our additions to this project are four-fold. We first refactored the code base to make it actually usable. This process involved code commenting and refactorization, repository documentation improvements, and shell script instantiation. Second was the addition of vulnerability classification based on a CVSS v2.0-like scoring system. Third was the addition of generated code vulnerability remediation recommendations. By utilizing the power of the GPT, we are able to generate code that might solve any issues found. Lastly, we expand the LLM model set from 3.5-Turbo, used by Sun et al., to utilize any OpenAI model (e.g., GPT-4o, GPT-4o-mini, GPT-o1, and GPT-o1-mini).

I. Introduction

Smart contracts, the backbone of decentralized finance (DeFi), offer programmable and automated solutions for financial transactions. However, their security remains a critical concern due to frequent breaches, resulting in billions of dollars in financial losses [1], [4], [6], [12]. These incidents jeopardize the safety of users' assets and the stability of the entire DeFi ecosystem. Despite the availability of numerous analysis tools, these solutions primarily target vulnerabilities with fixed control- or data-flow patterns, such as reentrancy [29], [34], integer overflows [33], and access control flaws [10], [13], [22]. A study by Zhang et al. [37] revealed that approximately 80% of vulnerabilities remain undetected by these tools because they fail to address the business logic underpinning smart contracts. Traditional static and dynamic analysis methods lack the ability to comprehend complex logic, model contract functionalities, or account for the roles of variables and functions.

Advances in Large Language Models (LLMs) [27], such as Generative Pre-training Transformers (GPT) [16], [28], have demonstrated significant potential in understanding and processing code. Previous attempts to leverage GPT for vulnerability detection used high-level inquiries, but these approaches suffered from high false positive rates (around 96%) and required the advanced reasoning capabilities of GPT-4, making them cost-prohibitive. A more efficient and accurate solution is needed—one that combines GPT's code comprehension capabilities with static analysis to address the limitations of existing tools. The ability to detect vulnerabilities tied to business logic could significantly enhance the security of smart contracts, prevent financial losses, and serve as a valuable complement to human auditors.

Our paper outlines four main objectives:

1) Streamline and Debricking is the first objective since it was the first issue we encountered with GPTScan. It would not run out of the box, so this objective specifies making it usable.
2) Second is the addition of vulnerability classification based on a Common Vulnerability Scoring System (CVSS) v2.0-like scoring system. Sun et al. [31] do not classify the vulnerabilities in their tool, opting to manually review and assign severity in their commercial reports instead. Adding the feature can direct a non-security expert towards the most dangerous issues.
3) Third is the addition of vulnerability remediation. By utilizing the GPT's power, we can generate code that might solve any detected issues.
4) Fourth, we expand the LLM model set from 3.5-Turbo used by Sun et al. [31] to utilize any OpenAI model (e.g, GPT-4o, GPT-4o-mini, GPT-o1, and GPT-o1-mini). This ability to use other, in most cases, more advanced or new models may yield better results and ensures the utility of this tool over time.

The paper is laid out as follows: In Section 2, we present the issues associated with GPTs, focusing on those noted by Sun et al. [31]. In Section 3, we provide insights into how GPTScan operates at a high level as well as defining our datasets and baselines. In Section 4, we show the results and findings of our implementations. In Section 5, we provide insights into the novelty of our contributions to the project. Then, in Section 6, we draw our conclusions and provide the next steps.

II. Background

Here, we address the issues with Smart Contracts and LLMs in general, as well as the limitations and recommendations identified in Sun et al's paper [31].

A. Smart Contract Vulnerability Types

Smart Contracts are self-executing programs deployed on blockchains and written in high-level languages like Solidity [21].

According to Zhang et al. [37], smart contract vulnerabilities can be categorized into three groups based on their characteristics and exploitability:

1) **Group 1: Hard-to-Exploit or Non-Functional Vulnerabilities**
   These vulnerabilities are either difficult to exploit, doubtful in nature, or not directly related to the core functionalities of a project.
2) **Group 2: General Vulnerabilities Detectable by Simple Oracles**
   This group includes vulnerabilities like reentrancy and arithmetic overflow, which do not require an in-depth understanding of code semantics. Such issues can be identified using existing tools like:
   a) Data Flow Tracing (e.g., Slither [11])
   b) Static Symbolic Execution (e.g., Solidity SMT Checker [20], Mythril [7])
   c) Other Static Analysis Tools [5], [15], [24]
3) **Group 3: High-Level Semantic Vulnerabilities**
   These vulnerabilities require a deeper understanding of code semantics and are closely tied to the business logic of smart contracts. Current static analysis tools are generally ineffective in detecting these issues. This group comprises six major types of vulnerabilities: Price Manipulation, ID-Related Violations, Erroneous State Updates, Atomicity Violations, Privilege Escalation, and Erroneous Accounting.

B. GPT & Its Application in Vulnerability Detection

Generative Pre-training Transformer (GPT) models, such as GPT-3.5 [28], are large language models (LLMs) trained on extensive text corpora, including programming languages and descriptions of vulnerabilities. These models can interpret source code and perform zero-shot learning [16], detecting vulnerabilities without requiring explicit examples. However, GPT has limitations that prevent it from fully replacing human auditors [26]. Challenges with GPT in vulnerability detection include:

1) **Limited Recall:**
   A study by David et al. [9] demonstrated that even when feeding entire projects to the GPT-4-32k model to detect 38 types of vulnerabilities, the results were unsatisfactory, performing worse than a random model in terms of recall.
2) **Content-Length Constraints:**
   GPT models have a maximum token limit. This makes analyzing complete projects or documents impractical, particularly for large smart contract projects.
3) **Logical Reasoning Limitations:**
   GPT's reasoning and logic capabilities are limited, leading to inaccurate results. Verification through additional methods is essential to reduce false positive rates and ensure reliability.

While GPT offers powerful code understanding capabilities, its application in vulnerability detection requires hybrid approaches that combine its strengths with complementary techniques to address its limitations.

C. Limitations Noted by Sun et al's Existing Work

1) Path Sensitivity: A significant limitation of GPTScan lies in its lack of path-sensitivity [31]. This means that vulnerabilities tied to specific execution paths—those triggered under particular sequences of conditions or function calls—may go undetected. Path sensitivity is essential for capturing complex logic issues that depend on dynamic interactions within the code; i.e. code coverage. For instance, determining whether a certain branch is reachable under specific conditions or tracking the state changes along an execution path is critical for identifying subtle vulnerabilities. The current static analysis approach, which relies on simple control flow and data dependence graphs, is insufficient for this task. Incorporating symbolic execution engines, which simulate code execution with symbolic rather than concrete values, could address this limitation. Such enhancements would enable GPTScan to systematically explore execution paths and significantly improve its precision in detecting vulnerabilities with path-specific triggers.

2) Pre-Defined Whitelist Filtering: Another limitation of GPTScan is its reliance on its white-listing approach for filtering modifiers with access control [31]. While this method is straightforward, it lacks the flexibility to account for custom or dynamic implementations of access control mechanisms. This can result in both false positives, where legitimate code is flagged as vulnerable, and false negatives, where actual vulnerabilities are missed. For example, custom modifiers may implement nuanced access checks that fall outside the scope of the whitelist. To improve accuracy, GPTScan could benefit from a more sophisticated approach that retrieves the definitions of modifiers and performs semantic analysis. By understanding the underlying logic of these modifiers, the tool would be better equipped to accurately assess access control mechanisms and reduce errors in its findings.

3) Vulnerable to GPT's Inherent Challenges Like Hallucination: GPTScan's dependence on GPT models introduces inherent vulnerabilities tied to the limitations of these models [31]. GPT is prone to issues such as hallucination, where it generates outputs that are inconsistent with the provided input or factual reality. This can lead to the misidentification of vulnerabilities or the inclusion of irrelevant information in the analysis. Additionally, GPT's outputs can be inconsistent, influenced by randomness or ambiguous prompts. While zero-temperature settings and mimic-in-the-background prompting have been implemented to reduce variability, these measures do not eliminate the problem entirely. Augmenting GPT with static confirmation layers, as done in GPTScan, partially addresses these issues, but the tool still requires careful human validation in critical contexts to mitigate the risk of errors. Sun et al. also state that it's worth looking into how other models respond (e.g Claude 3.5 by Anthropic, Grok-1 by xAI, Gemini 1.5 by Google DeepMind, and Llama 3.1 by Meta AI). While these are all LLMs with the same hallucination issue, it would be interesting to see how different development teams mitigate this risk in their respective models.

III. Methodology

Here, we provide a high-level overview of GPTScan's inter-workings, the datasets used, and our baselines.

A. GPTScan

GPTScan is an advanced tool designed to identify vulnerabilities in smart contracts through GPT-based analysis and static verification. Its workflow begins by accepting smart contract projects, which can be standalone Solidity files or frameworks with multiple files. The tool decomposes these projects, constructs call graphs to assess function reachability, and filters out irrelevant components to extract candidate functions. GPTScan then employs a selected Chat GPT model to match these functions with predefined scenarios and properties representing various vulnerability types.

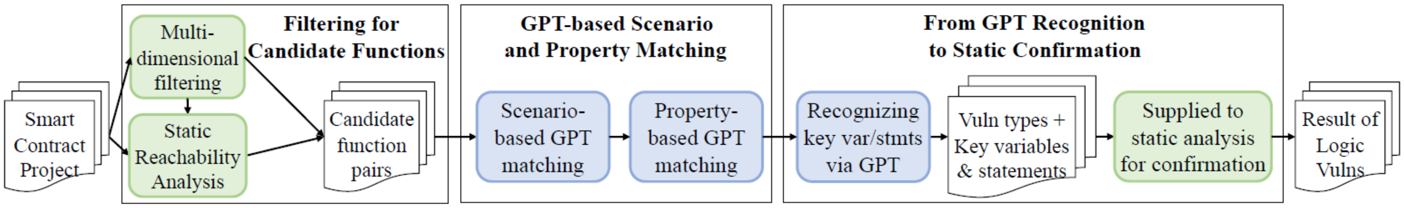

Figure 1: A high-level overview of GPTScan, with blue blocks denoting GPT tasks and green blocks representing static analysis. [31]

Table I: Breaking down ten common logic vulnerability types into scenarios and properties.

Vulnerability Type	Scenario and Property	Filtering Type	Static Check
Approval Not Cleared	**Scenario:** add or check approval via require/if statements before the token transfer **Property:** and there is no clear/reset of the approval when the transfer finishes its main branch or encounters exceptions	FNI, FCCE	VC
Risky First Deposit	**Scenario:** deposit/mint/add the liquidity pool/amount/share **Property:** and set the total share to the number of first deposit when the supply/liquidity is 0	FCCE	DF, VC
Price Manipulation by AMM	**Scenario:** have code statements that get or calculate LP token's value/price **Property:** based on the market reserves/AMMprice/exchangeRate OR the custom token balanceOf/totalSupply/amount/liquidity calculation	FNK, FCCE	DF
Price Manipulation by Buying Tokens	**Scenario:** buy some tokens **Property:** using Uniswap/PancakeSwap APIs	FNK, FCE	FA
Vote Manipulation by Flashloan	**Scenario:** calculate vote amount/number **Property:** and this vote amount/number is from a vote weight that might be manipulated by flashloan	FCCE	DF
Front Running	**Scenario:** mint or vest or collect token/liquidity/earning and assign them to the address recipient or to variable **Property:** and this operation could be front run to benefit the account/address that can be controlled by the parameter and has no sender check in the function code	FNK, FPNC, FPT, FCNE, FNM	FA
Wrong Interest Rate Order	**Scenario:** have inside code statements that update/accrue interest/exchange rate **Property:** and have inside code statements that calculate/assign/distribute the balance/share/stake/fee/loan/reward	FCE, CEN	OC
Wrong Checkpoint Order	**Scenario:** have inside code statements that invoke user checkpoint **Property:** and have inside code statements that calculate/assign/distribute the balance/share/stake/fee/loan/reward	FCE, CEN	OC
Slippage	**Scenario:** involve calculating swap/liquidity or adding liquidity, and there is asset exchanges or price queries **Property:** but this operation could be attacked by Slippage/Sandwich Attack due to no slip limit/minimum value check	FCCE, FCNCE	VC
Unauthorized Transfer	**Scenario:** involve transfering token from an address different from message sender **Property:** and there is no check of allowance/approval from the address owner	FNK, FCNE, FCE, FCNCE, FPNC	VC

Matched functions undergo further analysis, where GPT identifies relevant variables and statements passed to static analysis modules for vulnerability confirmation. Implemented using Python and Java/Kotlin, GPTScan is optimized to minimize costs and improve determinism. It integrates static analysis tools, such as ANTLR [3] and cryptic-compiler [8] to construct control flow and data dependency graphs, combining GPT's semantic understanding with precise static checks for robust and efficient vulnerability detection. (This is a high-level overview; for a deeper understanding, view Section 4 in Sun et al's paper [31].) We now look at each component as seen in Figure 1:

1) **Filtering for Candidate Functions:**
To address challenges with high-level vulnerability descriptions, GPTScan breaks down vulnerabilities into actionable code-level scenarios and properties. Scenarios describe the functional context where a vulnerability might occur, while properties capture specific code attributes. Table I showcases how ten common logic vulnerability types are broken down into said scenarios and properties. These vulnerability types were selected from [37], a study on smart contract vulnerabilities that require high-level semantic oracles [36]. The study summarizes six categories of logic vulnerabilities (see Section 2), and ten representative cases from these categories were chosen. Using yes-or-no prompts, the tool sequentially evaluates scenarios and properties to minimize ambiguity and reduce unnecessary processing. Additionally, randomness in GPT outputs is mitigated by using deterministic settings (temperature = 0) and a "mimic-in-the-background" technique inspired by the successful usage of "Let's think step by step" in the zero-shot chain-of-thought prompting [16], which ensures consistent and reliable answers through multiple iterations of the same query. This uses the instructed GPT to learn the output JSON format for multiple-choice scenario matching, leveraging GPT's instruction learning capability and minimizing randomness on the output [28].
2) **GPT-based Scenario and Property Matching:**
The tool employs multi-dimensional filtering to refine its analysis. It begins with file-level filtering to exclude non-Solidity files, test files, and third-party libraries like OpenZeppelin [4]. At the function level, rule-based filtering specifications, such as keyword matching and content-based rules, are applied to select functions relevant to specific vulnerabilities. Reachability analysis further narrows down the scope by retaining only those functions accessible to potential attackers, considering access control modifiers and custom permissions.
3) **From GPT Recognition to Static Confirmation:**
Once candidate functions are identified, GPTScan performs static vulnerability confirmation. Static analysis tools

Table II: Sun et al.'s GPTScan accuracy evaluation.

Dataset	TP	TN	FP	FN	SUM
Top200	0	283	13	0	296
Web3Bugs	40	154	30	8	232
DefiHacks	10	19	1	4	34

validate specific vulnerability attributes, such as data dependencies, value comparisons, execution order, and user-controllable function arguments. GPT outputs are key in extracting relevant variables and statements, validated using techniques like static data flow tracing and symbolic execution. This integration ensures the precise identification of vulnerabilities while leveraging GPT's semantic capabilities for context interpretation.

B. Datasets & Baselines

In this section, we compare our revised version of GPTScan against the results given in Sun et al.'s paper [31]; their results, our baselines, are denoted in Table II. Using a few of the same metrics, accuracy, performance, and effectiveness of its static confirmation, we attempt to use the same datasets and testing methods detailed in their report. The experiments were conducted on three datasets comprising real-world smart contracts.

Top200 Dataset: This dataset consists of 303 open-source smart contract projects from six major Ethereum-compatible chains, representing contracts with the top 200 market capitalizations. These well-audited contracts, encompassing 555 files and 134,322 lines of code, are assumed to have minimal vulnerabilities. The dataset is a benchmark to stress-test GPTScan's false-positive rate in highly scrutinized contracts. [18], [35].

Web3Bugs Dataset: Derived from the Web3Bugs dataset, this collection includes 72 out of 100 Code4-rena-audited projects that could be directly compiled. It contains 2,573 files, 319,878 lines of code, and 48 known vulnerabilities. Projects excluded from this dataset lacked necessary library dependencies or configuration files. In our case, compared to the Web3 Github by MetaTrust, we did not use any contracts marked as a Logic Vulnerability due to this issue. [36], [19], [2].

DefiHacks Dataset: Sourced from the DeFi Hacks dataset, this collection focuses on vulnerable token contracts in past attack incidents. It includes 13 projects, 29 files, and 17,824 lines of code, with 14 known vulnerabilities covering the 10 vulnerability types addressed by GPTScan. [25], [17], [32] or its fork [30].

C. Evaluation Metrics

TP is the number of true positives. One true positive is counted when GPTScan successfully detects a ground-truth vulnerable function for the tested vulnerability type.

TN is the number of true negatives. One true negative is counted when GPTScan correctly does not report any vulnerable function for the tested vulnerability type.

FP is the number of false positives. One false positive is counted when GPTScan incorrectly reports one or more vulnerable functions for the tested vulnerability type with no corresponding ground-truth vulnerabilities in the tested project.

FN is the number of false negatives. One false negative is counted when GPTScan fails to detect the ground-truth vulnerable function for the tested vulnerability type.

Table III: GPTScan accuracy evaluation with GPT-3.5-Turbo.

Dataset	TP	TN	FP	FN	SUM
Top200	0	172	25	0	197
Web3Bugs	9	27	12	8	56
DefiHacks	5	1	0	0	6

Table IV: GPTScan accuracy evaluation with GPT-4.

Dataset	TP	TN	FP	FN	SUM
Top200	0	179	18	0	197
Web3Bugs	11	28	13	4	56
DefiHacks	5	1	0	0	6

IV. Results & Evaluation

Here, we review our results and issues with our modified version of GPTScan.

A. Tool Capabilities

One of the biggest issues we faced with the tool's output was related to the cryptic-compiler and falcon instances failing to successfully compile and build a project's workspace. This can be seen very extensively in our "results" directory in our open-source repository[1] for this project. Due to this compiler failure occurring at the initialization of the project or during the tool's attempted static analysis of a contract, most of the sources in the datasets provided were not able to be tested. As seen in our confirmation of results from Sun et al.'s paper [31], the Tables III and IV, showcase the lackluster number of working results due to these errors. In most cases, the compilation error reduced the usable projects in every dataset across the board by about 30-40%. In the worst of cases, i.e., the DeHiHacks Dataset, it reduced the successful completion of analysis by 83%.

This is particularly interesting due to the fact that our modifications did not touch the core of this program. The static analysis, cryptic, and flacon compiling sections of this code are all unmodified, begging the question of whether the authors of the original paper had this same issue. And if they did, how they obtained such numbers as a result. We cannot be sure since the documentation and code commenting, as previously mentioned, was non-existent. And there is no mention of these issues in their original work.

In the cases where it does work, it works very well. As seen in Appendix B and our "example_data_source" directory in the repository[1], a fully compiled and analyzed file with found vulnerabilities showcases extensive documentation and insight. The rate at which the tool functioned as intended was also objectively higher for the Web3Bugs dataset. We believe that this is likely due to the higher quality, more consistent dataset that Code4-rena has put together.

We also noticed and confirmed that the better model, GPT-4, performs better compared to GPT-3.5-Turbo when looking at the baseline dataset from Top200. Here, we corroborate Sun et al.'s [31] work, which makes the argument that the better model, while having around 30% better performance, might not outweigh the additional cost of usage. However, this may not be the case for even more powerful models. Should an increase in 30% continue to be the trend, a more powerful model like o1 may be worth it for its accuracy.

[1] https://github.com/wpj3799/GPTScan-Bigger-Model

B. Other Issues Found

Other problems we found include a tokenization issue with newer models of GPT. It seems like the API does not like the prompt engineering used by the original authors and as such, will not tokenize the input correctly. As a future work, figuring out this API and model input issue is the only roadblock identified that would prevent other researchers from utilizing OpenAI's other models.

V. Novelty & Contribution

Here, we cover the four main objectives/features we implement in GPTScan.

A. Objective 1 - Streamline and Debrick

The original project by Sun et al. [31] faced several significant challenges, including a messy, unstructured codebase that was difficult to understand, a broken out-of-the-box experience, and poor documentation that left users and developers without adequate guidance. To address these issues, we undertook a comprehensive refactoring of the codebase, resolving inefficiencies, redundancies, and unnecessary complexities while enforcing consistent coding standards and adding meaningful comments. This effort made the code more readable, maintainable, and scalable, significantly improving the overall development experience.

To streamline the use of this tool and "debrick" the original implementation, we first had to fix the issues that plagued its setup. After fixing its Python3 dependency list, we introduced shell script-based initialization, which automates environment configuration and tool execution (see Appendix C and Appendix D). This script provides a seamless out-of-the-box experience, ensuring the tool repository can run immediately after being cloned without requiring manual troubleshooting. Additionally, robust error handling and clear debug messages were implemented to assist users in resolving any potential issues during initialization. This improvement in output also includes clearly marked, independent file analysis results, which were combined into a single master file before.

We also added the much-needed multi-threading feature, which allows the program to initialize multiple HardHat instances simultaneously. This enhancement eliminates the sequential nature of file analysis, where users previously had to wait for the step-by-step initialization of a contract's HardHat [14] environment. As a result, the time required to begin analyzing a large number of contracts is significantly reduced, thereby speeding up the overall analysis process.

We also streamlined user input where all tool-specific parameter configurations like the OpenAI API key and model type are configured in a configuration file (see Appendix E), where the user only needs to specify the contracts' directory upon execution. This addition of OpenAI model extensibility ensures this tool's longevity as more advanced models are released by the company.

We also overhauled the documentation to further enhance the project, updating the repository's README file to provide system requirements, dependencies, and step-by-step instructions for installation, configuration, and usage. All things that were severely lacking in the prior version.

B. Objective 2 - Vulnerability Classification

Sun et al's tool [31] initially lacked a standardized approach to classify and prioritize vulnerabilities, making it challenging to assess the severity of issues and allocate resources effectively. Rather, they provide this information in detailed reports compiled by MetaTrust [23], which utilized this tool. These assessments are made by expert human reviewers, which requires said individuals to be paid for their time. The issue with this approach is that, by necessity, a commercial approach must be taken. For an open-source tool made to enhance the security of Web3 smart contracts, we believe that locking this information behind a paywall does more harm than good. As such, we implemented a vulnerability classification system similar to the CVSS v2.0 framework. By utilizing GPT capabilities, the LLM can classify the issue by Low, Medium, and High severity levels based on their impact, exploitability, and other critical factors; it then assigns a risk value to the contract (see Appendix A and Appendix B).

This classification framework helps developers make informed decisions and direct their mitigation efforts. These developers can allocate the appropriate resources to the most severe issues by addressing the most critical vulnerabilities first. As a result of this feature, the tool gains a structured approach to vulnerability management, enhancing both its usability and utility.

C. Objective 3 - Vulnerability Remediation

The tool also initially faced a critical gap in vulnerability remediation, with no systematic approach for addressing identified security issues. To tackle this, we introduced a proactive remediation strategy by leveraging the GPT's LLM capabilities, which are already being used. For each identified vulnerability, we provided recommended code fixes tailored to the specific issue, ensuring actionable and precise solutions.

This approach enables a potentially rapid and effective resolution of vulnerabilities, as the AI-generated fixes are generally based on best practices and a comprehensive understanding of coding standards. By integrating this process into the workflow, we potentially accelerate the remediation timeline and reduce the burden on developers, providing a usable first step toward secure coding. Also of note is that, as seen in the next objective, the performance of programming capabilities that LLMs provide is expected to significantly increase, and as such, the recommendations provided should only improve with time. Ultimately, like the vulnerability classification feature, this addition significantly improves the tool's usability and utility.

D. Objective 4 - Model Extension

Sun et al. [31] used GPT-3.5-Turbo and GPT-4 models for generating results, which constrained its versatility and adaptability to different use cases. To expand its capabilities, we extended the system to support additional models (e.g. GPT-4o, GPT-4o-mini, GPT-o1, and GPT-o1-mini) through the tool's configuration files and API integration (see Appendix E). This enhancement provides greater flexibility by allowing users to select models based on specific requirements such as performance, cost, response time, and availability.

By incorporating these additional models, GPTScan gains the ability to cater to a broader range of scenarios, from lightweight applications requiring fast and cost-effective solutions to complex tasks needing high-performance models. This extension improved scalability and empowered users to tailor the system to their unique needs, enhancing overall functionality and user satisfaction. This integration unshackles the tool, allowing it to use any model that OpenAI releases to its API, enabling the tool to remain usable in the foreseeable future.

VI. Conclusion

This paper highlights our contributions to GPTScan, a tool that integrates GPT with static analysis for smart contract vulnerability detection. We address limitations in usability, scalability, and functionality that were unaddressed in the original implementation, providing a refined version of the tool that not only streamlines the user experience but also introduces critical new features.

Our contributions include refactoring the codebase to enhance usability, implementing a CVSS-inspired vulnerability classification system for prioritizing issues, and integrating GPT-generated remediation recommendations to help developers address these issues before releasing code into the wild. Additionally, we have ensured the tool's adaptability to future advancements in OpenAI technology by enabling compatibility with additional API models.

These improvements reduce technical complexity and enhance developer accessibility, allowing said developers to identify and address high-level vulnerabilities often overlooked by other analysis tools. However, while our enhancements have strengthened GPTScan's capabilities, challenges identified by Sun et al. [31], including compiler issues, tokenization problems with newer OpenAI API models, and the exploration of alternative LLMs, remain as avenues for future work. Addressing these issues would further establish GPTScan as a critical resource for the Web3 community.

References

[1] Code 423n4. 2021-11-yaxis, 2021. Accessed: 2024-12-07.

[2] Code 423n4. Code 423n4: A platform for secure code review contests, 2023. Accessed: 2024-12-07.

[3] ANTLR. Antlr: Another tool for language recognition, 2023. Accessed: 2024-12-07.

[4] OpenZeppelin Blog. On the parity wallet multisig hack, 2023. Accessed: 2024-12-07.

[5] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. Ethainter: A smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 454–469, 2020.

[6] CoinDesk. Understanding the dao attack, 2016. Accessed: 2024-12-07.

[7] Consensys. Mythril: Security analysis tool for ethereum smart contracts, 2023. Accessed: 2024-12-07.

[8] Crytic. Crytic compile: A multi-compiler wrapper for smart contract compilation, 2023. Accessed: 2024-12-07.

[9] Isaac David, Liyi Zhou, Kaihua Qin, Dawn Song, Lorenzo Cavallaro, and Arthur Gervais. Do you still need a manual smart contract audit? *arXiv preprint*, arXiv:2306.12338, June 2023.

[10] Yuzhou Fang, Daoyuan Wu, Xiao Yi, Shuai Wang, Yufan Chen, Mengjie Chen, Yang Liu, and Lingxiao Jiang. Beyond "protected" and "private": An empirical security analysis of custom function modifiers in smart contracts. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023.

[11] Josselin Feist, Gustavo Grieco, and Alex 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)*, pages 8–15. IEEE, 2019.

[12] FreeCodeCamp. What is yaml? the yml file format, 2022. Accessed: 2024-12-07.

[13] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. Achecker: Statically detecting smart contract access control vulnerabilities. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, 2023.

[14] Hardhat. Hardhat: Ethereum development environment for professionals, 2023. Accessed: 2024-12-07.

[15] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2018.

[16] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:22199–22213, 2022.

[17] MetaTrust Labs. Gptscan-defihacks: Scanning results of metascan's ai gptscan engine for 13 defihacks projects under 10 logic vulnerability types, 2023. Accessed: 2024-12-07.

[18] MetaTrust Labs. Gptscan-top200, 2023. Accessed: 2024-12-07.

[19] MetaTrust Labs. Gptscan-web3bugs: Scanning results of metascan's ai gptscan engine for 72 web3bugs projects under 10 logic vulnerability types, 2023. Accessed: 2024-12-07.

[20] Solidity Language. Smtchecker: The solidity formal verification tool, 2023. Accessed: 2024-12-07.

[21] Solidity Language. Solidity documentation, 2023. Accessed: 2024-12-07.

[22] Ye Liu, Yi Li, Shang-Wei Lin, and Cyrille 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)*, pages 716–727, 2022.

[23] MetaTrust. Metatrust: Ai-driven security solutions for blockchain and smart contracts, 2023. Accessed: 2024-12-07.

[24] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Greico, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2019.

[25] Notion. Resource on smart contracts and web3 vulnerabilities, 2023. Accessed: 2024-12-07.

[26] Trail of Bits. Codex and gpt-4 can't beat humans on smart contract audits, 2023. Accessed: 2024-12-07.

[27] OpenAI. Chatgpt, 2023. Accessed: 2024-12-07.

[28] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. *arXiv preprint*, arXiv:2203.02155, 2022.

[29] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019*. The Internet Society, February 24–27 2019.

[30] Skyonedot. Defihacklabs: Fork of defihacklabs for reproducing defi hacking incidents using foundry, 2023. Accessed: 2024-12-07.

[31] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, page 1–13. ACM, April 2024.

[32] SunWeb3Sec. Sunweb3sec: Enhancing web3 security, 2023. Accessed: 2024-12-07.

[33] Bryan Tan, Benjamin Mariano, Shuvendu K. Lahiri, Isil Dillig, and Yu Feng. Soltype: Refinement types for arithmetic overflow in solidity. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2022.

[34] Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*, pages 1029–1040. IEEE, September 21–25 2020.

[35] Xiao Yi, Yuzhou Fang, Daoyuan Wu, and Lingxiao Jiang. Blockscope: Detecting and investigating propagated vulnerabilities in forked blockchain projects. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2023.

[36] Zhuo Zhang. Web3bugs: A repository for blockchain vulnerabilities and exploits, 2023. Accessed: 2024-12-07.

[37] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. Demystifying exploitable bugs in smart contracts. In *2023 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, 2023.

APPENDIX

All of the files included in this appendix can be found on our GitHub repository: https://github.com/wpj3799/GPTScan-Bigger-Model

A. *Old Output of a Detected Vulnerability*

Type	Description	Affected Files		
Insecure LP Token Value Calculation	Liquidity token value/price can be manipulated to cause flashloan attacks.	File Path	Line Range	Code
		/home/owen/Documents/GitHub/GPTScan-Bigger-Model/eval_data/2021-05-yield-main/contracts/oracles/...	40 - 53	function _peek(bytes6 base, bytes6 kind) private view returns (uint price, uint updateTime) { uint256 rawPrice; address source = sources; require(source != address(0), "Source not found"); if (kind == "rate") { rawPrice = CTokenInterface(source).borrowIndex(); } else if (kind == "chi") { rawPrice = CTokenInterface(source).exchangeRateStored(); } else { revert("Unknown oracle type"); } require(rawPrice > 0, "Compound price is zero"); price = rawPrice * SCALE_FACTOR;
		File Path	Line Range	Code
		/home/owen/Documents/GitHub/GPTScan-Bigger-Model/eval_data/2021-05-yield-main/contracts/oracles/...	95 - 99	function get(bytes32 base, bytes32 quote, uint256 amount) public virtual override view returns (uint256 value, uint256 updateTime) { uint256 price; (price, updateTime) = _peek(base.b6(), quote.b6()); value = price * amount / 1e18;
Unsafe First Deposit	First depositor can break minting of shares or drain the liquidity of all users.	File Path	Line Range	Code
		/home/owen/Documents/GitHub/GPTScan-Bigger-Model/eval_data/2021-05-yield-main/contracts/yieldspa...	231 - 298	function _mintInternal(address to, bool calculateFromBase, uint256 fyTokenToBuy, uint256 minTokensMinted) internal returns (uint256 baseIn, uint256 fyTokenIn, uint256 tokensMinted) { // Gather data uint256 supply = _totalSupply; (uint112 _baseCached, uint112 _fyTokenCached) = (baseCached, fyTokenCached); uint256 _realFYTokenCached = _fyTokenCached - supply; // fyToken cache includes the virtual fyToken equal to the supply // Initialize variables uint256 baseReturned; // Check if supply is zero (initializing the pool) if (supply == 0) { require(calculateFromBase && fyTokenToBuy == 0, "Pool: Initialize only from base"); baseIn = base.balanceOf(address(this)) - _baseCached; tokensMinted = baseIn; // Initialize the pool with baseIn, no fyToken

Figure 2: Old Output of a Detected Vulnerability

B. New Output of a Detected Vulnerability

Type	Description	Affected Files			Secure Code Examples	Best Practices	CVSS Score	Recommendation
		File Path	Line Range	Code				
Insecure LP Token Value Calculation	Liquidity token value/price can be manipulated to cause flashloan attacks.	/home/owen/Documents/GitHub/GPTScan-Bigger-Model/eval_data/2021-05-yield-main/contracts/oracles/...	40 - 53	function _peek(bytes6 base, bytes6 kind) private view returns (uint price, uint updateTime) { uint256 rawPrice; address source = sources; require(source != address(0), "Source not found"); if (kind == "rate") { rawPrice = CTokenInterface(source).borrowIndex(); } else if (kind == "chi") { rawPrice = CTokenInterface(source).exchangeRateStored(); } else { revert("Unknown oracle type"); } require(rawPrice > 0, "Compound price is zero"); price = rawPrice * SCALE_FACTOR;	To make this code more secure, you can use a secure price oracle to fetch the price data. Here is an example using Chainlink Price Feeds: ```solidity // SPDX-License-Identifier: MIT pragma solidity ^0.8.0; import "@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol"; contract PriceOracle { AggregatorV3Interface internal priceFeed; constructor(address _priceFeed) { priceFeed = AggregatorV3Interface(_priceFeed); } function getLatestPrice() public view returns (uint80, int256, uint256) { (, int256 price, , uint256 timestamp,) = priceFeed.latestRoundData(); return (price, timestamp); } } contract YourContract { PriceOracle public priceOracle; constructor(address _priceFeed) { priceOracle = new PriceOracle(_priceFeed); } function peek(bytes32 base, bytes32 quote, uint256 amount) public view returns (uint256 value, uint256 updateTime) { (int256 price, uint256 timestamp) = priceOracle.getLatestPrice();	To mitigate vulnerabilities in the provided code snippet, you should consider the following best practices: 1. **Input Validation**: Ensure that the input parameters are properly validated to prevent unexpected behavior or vulnerabilities. For example, validate that `amount` is not negative. 2. **Safe Math Operations**: Use safe math libraries to perform arithmetic operations to prevent overflow or underflow errors. Consider using libraries like OpenZeppelin's SafeMath to perform safe arithmetic operations. 3. **Consistent Naming Conventions**: Maintain consistent naming conventions for variables and functions to improve code readability and reduce the chances of errors. 4. **Use of Constants**: Consider using constants for values that do not change, such as the divisor in the calculation, to improve code readability and maintainability. 5. **Access Control**: Ensure that the `peek` function is appropriately restricted based on who should be able to call it. Consider applying access control modifiers such as `public`, `external`, `internal`, or `private` based on the intended access requirements. 6. **Documentation**: Provide clear and comprehensive documentation for the function, including details about the input parameters, return values, and any other important information that users of the function need to be aware of. Here's an example of how you can apply some of these best practices to the code snippet: ```solidity // SPDX-License-Identifier: MIT	5.2	Ensure proper input validation to prevent potential vulnerabilities such as input manipulation or injection attacks. Use safe arithmetic operations to avoid issues like integer overflow or underflow. Consider implementing additional security measures such as input sanitization and validation checks.
		File Path	Line Range	Code				
		/home/owen/Documents/GitHub/GPTScan-Bigger-Model/eval_data/2021-05-yield-main/contracts/oracles/...	95 - 99	function get(bytes32 base, bytes32 quote, uint256 amount) public virtual override view returns (uint256 value, uint256 updateTime) { uint256 price; (price, updateTime) = _peek(base.b6(), quote.b6()); value = price * amount / 1e18;				
Unsafe First Deposit	First depositor can break minting of shares or drain the liquidity of all users.	File Path	Line Range	Code	To provide a more secure alternative for the provided code, you can consider the following improvements: 1. **Use SafeMath Library**: Replace direct arithmetic calculations with SafeMath library functions to prevent integer overflow and underflow vulnerabilities. 2. **Reduce Complexity**: Break down the function into smaller, more manageable functions with specific responsibilities to improve readability and maintainability. 3. **Input Validation**: Ensure that input validation is robust to prevent unexpected behavior. 4. **Access Control**: Implement proper access control mechanisms to restrict who can call certain functions. 5. **Error Handling**: Use require statements for input validation and error handling to revert transactions when conditions are not met. 6. **Event Logging**: Emit events for important state changes to provide transparency and allow for easier monitoring. Here is a high-level example of how you can refactor the code to improve security: ```solidity // SPDX-License-Identifier: MIT // Import SafeMath library import "@openzeppelin/contracts/utils/math/SafeMath.sol";	To mitigate vulnerabilities in the provided code, here are some best practices and recommendations: 1. **Input Validation**: - Ensure that all input parameters are properly validated before processing. This includes checking for valid ranges, types, and ensuring that the inputs do not lead to unexpected behavior. - Validate external contract calls to prevent potential attacks like reentrancy or malicious contract interactions. 2. **Safe Math Operations**: - Use safe math libraries to prevent integer overflow and underflow vulnerabilities. Replace standard arithmetic operations with safe math functions to handle calculations involving token amounts and balances. 3. **Access Control**: - Evaluate the use of access control mechanisms to restrict who can call certain functions. Consider implementing modifiers like `onlyOwner` or `onlyAdmin` to control access to critical functions. 4. **Error Handling**: - Use meaningful error messages in `require` statements to provide clear feedback to users when transactions fail. This helps in debugging and understanding the reason for transaction failures. 5. **Code Simplification**: - Simplify the logic in the function to reduce complexity and make it easier to review and analyze for potential vulnerabilities. Consider breaking down complex operations into smaller, more manageable functions.	8.3	Update the code to include proper input validation checks for potential arithmetic overflow and underflow scenarios. Implement bounds checking and ensure that all calculations are performed securely to prevent vulnerabilities.
		/home/owen/Documents/GitHub/GPTScan-Bigger-Model/eval_data/2021-05-yield-main/contracts/yieldspa...	231 - 298	function _mintInternal(address to, bool calculateFromBase, uint256 fyTokenToBuy, uint256 minTokensMinted) internal returns (uint256 baseIn, uint256 fyTokenIn, uint256 tokensMinted) { // Gather data uint256 supply = _totalSupply; (uint112 _baseCached, uint112 _fyTokenCached) = (baseCached, fyTokenCached); uint256 _realFYTokenCached = _fyTokenCached - supply; // fyToken cache includes the virtual fyToken equal to the supply // Initialize variables uint256 baseReturned; // Check if supply is zero (initializing the pool) if (supply == 0) { require(calculateFromBase && fyTokenToBuy == 0, "Pool: Initialize only from base"); baseIn = base.balanceOf(address(this)) - _baseCached; tokensMinted = baseIn; // Initialize the pool with baseIn, no fyToken				

Figure 3: New Output of a Detected Vulnerability

C. Setup Contracts Script

```
#!/bin/bash

# Usage: ./script.sh /path/to/parent_directory

# Check if the parent directory is provided
if [ -z "$1" ]; then
    echo "Please provide the parent directory containing the project directories."
    echo "Usage: $0 /path/to/parent_directory"
    exit 1
fi

PARENT_DIR="$1"

# SPDX License Identifier (Change as needed)
SPDX_IDENTIFIER="// SPDX-License-Identifier: MIT"

process_directory() {
    DIR="$1"
    echo "Processing directory: $DIR"

    # Change into the subdirectory
    cd "$DIR" || { echo "Failed to access directory: $DIR"; return; }

    # Initialize npm project
    if [ ! -f "package.json" ]; then
        echo "Initializing npm project in $DIR..."
        yes "" | npm init -y || { echo "npm init failed in $DIR"; return; }
    else
        echo "npm project already initialized in $DIR."
    fi

    # Install Hardhat locally
    echo "Installing Hardhat in $DIR..."
    npm install hardhat --save-dev || { echo "Failed to install Hardhat in $DIR"; return; }

    # Initialize Hardhat with default settings
    if [ ! -f "hardhat.config.js" ]; then
        echo "Initializing Hardhat project in $DIR..."
        yes "" | npx hardhat || { echo "Failed to initialize Hardhat in $DIR"; return; }

        # Delete the default Lock.sol file
        echo "Removing Lock.sol from contracts directory in $DIR..."
        rm -f contracts/Lock.sol || echo "No Lock.sol file found to delete."
    else
        echo "Hardhat already initialized in $DIR."
    fi

    # Move all .sol files in the current directory to the contracts directory
    echo "Moving .sol files to contracts/ in $DIR..."
    mkdir -p contracts
    find . -maxdepth 1 -type f -name "*.sol" -exec mv {} contracts/ \;

    # Extract Solidity versions from .sol files in contracts/ and update hardhat.config.js
    echo "Extracting Solidity versions from .sol files in $DIR..."
    SOLIDITY_VERSIONS=()
    for SOL_FILE in contracts/*.sol; do
        if [ -f "$SOL_FILE" ]; then
            VERSION_LINE=$(grep -E "^pragma solidity" "$SOL_FILE" | head -n 1)
            if echo "$VERSION_LINE" | grep -qE "^pragma solidity [^;]+;"; then
                VERSION=$(echo "$VERSION_LINE" | sed -E 's/^pragma solidity[^0-9]*([0-9]+\.[0-9]+\.[0-9]+).*/\1/')
                SOLIDITY_VERSIONS+=("$VERSION")
            fi
        fi
    done

    # Remove duplicates from SOLIDITY_VERSIONS
    UNIQUE_VERSIONS=($(echo "${SOLIDITY_VERSIONS[@]}" | tr ' ' '\n' | sort -u | tr '\n' ' '))

    # Update hardhat.config.js with the extracted versions
    if [ ${#UNIQUE_VERSIONS[@]} -gt 0 ]; then
        echo "Adding Solidity versions to hardhat.config.js in $DIR..."
        COMPILERS_STRING=$(printf '{ version: "%s" },\n            ' "${UNIQUE_VERSIONS[@]}")
        COMPILERS_STRING=${COMPILERS_STRING%, } # Remove trailing comma

        HARDHAT_CONFIG="require(\"@nomicfoundation/hardhat-toolbox\");

    /** @type import('hardhat/config').HardhatUserConfig */
    module.exports = {
        solidity: {
            compilers: [
                $COMPILERS_STRING
            ]
        }
    };"

        echo "$HARDHAT_CONFIG" > hardhat.config.js
    else
```

```
        echo "No Solidity versions found in .sol files in $DIR."
    fi

    # Compile the smart contracts
    echo "Compiling contracts in $DIR..."
    npx hardhat compile || echo "Compilation failed in $DIR."

    # Return to the parent directory
    cd "$PARENT_DIR" || exit 1
}

export -f process_directory  # Export function to be used with xargs

# Find directories and process them concurrently
find "$PARENT_DIR" -mindepth 1 -maxdepth 1 -type d | xargs -P 8 -I {} bash -c 'process_directory "{}"'

echo "Script completed."
```

Listing 1: Setup Contracts Script

D. Run GPTScan Script

```
#!/bin/bash

# Check if a directory parameter was provided
if [[ -z "$1" ]]; then
    echo "Usage: $0 <test_directory>"
    exit 1
fi

# Use the provided directory parameter
test_directory="$1"
main_script_path="main.py"

# Run setup script with the provided directory
./setup_contracts.sh "$test_directory"

# Iterate over all directories in the test directory
for dir_path in "$test_directory"/*; do
    # Ensure it's a directory
    if [[ -d "$dir_path" ]]; then
        # Define the output file path
        output_file="$dir_path/gptscan_results.md"

        # Construct and execute the command, redirecting output to the file
        echo "Running command: python3 $main_script_path -s $dir_path"
        python3 "$main_script_path" -s "$dir_path" | tee "$output_file"

        # Check if the command failed
        if [[ $? -ne 0 ]]; then
            echo "Error while running command for directory: $dir_path" | tee -a "$output_file"
        fi
    fi
done
```

Listing 2: Run GPTScan Script

E. Example Configuration File

```python
import logging
import datetime

# OpenAI API configuration
OPENAI_API_KEY = "YOUR_OPENAI_API_KEY" # Replace with your actual OpenAI API key
MODEL = "gpt-3.5-turbo"  # Specify the model parameter for GPT's API (e.g., gpt-3.5-turbo, gpt-4, etc. see more at
    https://openai.com/api/pricing/)

# Optional configuration for other services or tokens
GITHUB_TOKEN = "NOT NEEDED"

# Logging configuration
LOGGING_LEVEL = logging.INFO
LOGGING_FORMAT = "%(name)s: %(message)s"
# LOGGING_TARGET = datetime.datetime.now().strftime("logs/main.py-output-%Y%m%d-%H%M%S.log")

# File handling configuration
STATEMENT_FILE = "output/statements.csv"
WRITE_STATEMENTS_INTO_FILE = True
BACKUP_STATEMENTS = True

# Feature toggles
ENABLE_STATIC_ANALYSIS = True

# Pricing (for internal cost tracking or reference)
SEND_PRICE = 0.0015 / 1000
RECEIVE_PRICE = 0.002 / 1000
```

Listing 3: Example Configuration File