GPTScan_But_Bigger: Smart Contract Vulnerability Detection with GPT and Static Analysis

Owen Joslin
otj8625@rit.edu

William Joslin
wjp3799@rit.edu

Rochester Institute of Technology
Golisano College of Computing and Information Sciences
Department of Cybersecurity
CSEC759: Advanced Software Security Analysis

December 7, 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 paper "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 a shell script instantiation. Second was the addition of vulnerability classification based on the CVSS v2.0 scoring system. Third was the addition of vulnerability remediation. 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 as used by Sun et al. to utilize OpenAI's newer models (e.g., GPT-4o, GPT-4o-mini, GPT-o1-preview, and GPT-o1-mini).

1 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, 3, 5, 10]. 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 [25, 28], integer overflows [27], and access control flaws [8, 11, 19]. A study by Zhang et al. [30] 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) [23], such as Generative Pre-training Transformers (GPT) [?], 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 how to make it usable.

2. Second is the addition of vulnerability classification based on the CVSS v2.0 scoring system. Sun et al. do not classify the vulnerabilities in their tool, opting to manually review and asign severity in their reports. 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 issues.

4. Fourth, we expand the LLM model set from 3.5-Turbo used by Sun et al. to utilize OpenAI's newer models (e.g, GPT-4o, GPT-4o-mini, GPT-o1-preview, and GPT-o1-mini). This ability to use other, in most cases, more advanced or new models may yield better results.

The paper is layed out as follows: In Section 2, we present the issues associated with GPTs, focusing on those noted by [26]. In Section 3 we provide insight into how GPTScan operates at a high level as well as defining our datasets and baselines. In Section 4 we show our results and findings from our implementations. In Section 5 we provide insight into the novelty with our contributions to the project. Then in Section 6 we draw our conclusions and next steps.

2 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 [26].

2.1 Smart Contract Vulnerability Types

Smart contracts are self-executing programs deployed on blockchains and written in high-level languages like Solidity [18]. According to Zhang et al. [30], 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 [9])
   b) Static Symbolic Execution (e.g., Solidity SMT Checker [17], Mythril [6])
   c) Other Static Analysis Tools [4, 12, 20]

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.

2.2 GPT & Its Application in Vulnerability Detection

Generative Pre-training Transformer (GPT) models, such as GPT-3.5 [24], 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 [13], detecting vulnerabilities without requiring explicit examples. However, GPT has limitations that prevent it from fully replacing human auditors [22]. Challenges with GPT in Vulnerability Detection:

1. Limited Recall:
   A study by David et al. [7] demonstrated that even when feeding entire projects to the GPT-4-32k model to detect 38 types of vulnerabilities, while an older model compared to today, 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.

2.3 Limitations Noted by Sun et al's Existing Work

2.3.1 Path Sensitivity

A significant limitation of GPTScan lies in its lack of path-sensitivity [26]. 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.

2.3.2 Pre-Defined Whitelist Filtering

Another limitation of GPTScan is its reliance on a whitelisting approach for filtering modifiers with access control [26]. 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.

2.3.3 Vulnerable to GPT's Inherent Challenges Like Hallucination

GPTScan's dependence on GPT models introduces inherent vulnerabilities tied to the limitations of these models [26]. 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. Furthermore, GPT's lack of path sensitivity and its occasional inability to follow precise logical constructs further undermine its reliability. 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). And while these are all LLMs with the same hallucination issue, different approaches are implemented to mitigate this risk.

3 Methodology

Here we provide a hihg-level overview of GPTScan's interworkings as well as the datasets used and our baselines.

3.1 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. 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 [21] and cryptic-compiler [?] 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 [26]) We now look at each component:

1. Filtering for Candidate Functions:
   To address challenges with high-level vulnerability descriptions, GPTScan adopts a novel approach by breaking 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. 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 [44], 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. [50]

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. 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 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.

3.2 Datasets & Baselines

In this section, we compare our revised version of GPTScan's against the results given in Sun et al's paper [26], these results, our baselines, are denoted in Table 2. Looking at the same metrics, accuracy, performance, financial overhead, the effectiveness of its static confirmation, and its ability to uncover new vulnerabilities, 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. ADD IN WHAT ISSUES WE HAD WITH CERTAIN DATASETS

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. [15][62] STATE ISSUES WITH DATASET

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 contract listed as a Logic Vulnerability due to this issue. [29, 16, 2] STATE ISSUES WITH DATASET

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. [21, 14] STATE ISSUES WITH DATASET

4 Results and Evaluation

Type	Description	Affected Files: File Path	Affected Files: Line Range	Affected Files: 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 ki`
		/home/owen/Documents/GitHub/GPTSca Bigger-Model/eval_data/2021-05-yield-main/contracts/oracles/...	95–99	`function get(bytes32 base, bytes32 quote, uint256 amou`

5 Novelty & Contribution

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

5.1 Objective 1 - Streamline and Debrick

The original project 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. 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 feature, multi-threading, that allows GPTScan to execute its analysis on many smart contracts at once. This removed the series nature of file analysis where the user would have to wait for the initialization of a contract's HardHat [?] environment and then its analysis to complete before the next file could be analyzed; very time-consuming when analyzing multiple contracts, some of which contain upwards of 50 or more files. We also streamlined user input where all tool-specific parameter configurations like the OpenAI API key and model type—adding in extensibility for other GPT models—are configured in a configuration file, where the user only needs to specify the contracts' directory upon execution.

We overhauled the documentation to further enhance the project, updating the repository README files 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.

5.2 Objective 2 - Vulnerability Classification

Sun et al's tool [26] 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, which utilized this tool. These assessments are made by a human reviewer. The issue with this approach is that it's based on a commercial approach where an expert is paid to validate the vulnerabilities found. 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 Common Vulnerability Scoring System (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.

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 useability and utility.

5.3 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 that 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 accelerated the remediation timeline and reduced 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.

5.4 Objective 4 - Model Extension

Sun et al. [26] 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-preview, and GPT-o1-mini) through the tools configuration files and API integration. This enhancement provided 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, this enables the tool to remain usable into the foreseeable future.

6 Conclusion

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

[4] 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.

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

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

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

[19] 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.

[20] 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.

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

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

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.