Building a Labeled Smart Contract Dataset for Evaluating Vulnerability Detection Tools' Effectiveness

Open Source

Ryan Weege Achjian[1], Marcos Antonio Simplicio Junior[1]

[1]Laboratórtio de Arquitetura de Redes de Computadores (LARC)
Universidade de São Paulo (USP)
São Paulo – SP – Brazil

{ryan.achjian,msimplicio}@usp.br

Abstract. *In recent years, surveys on vulnerability detection tools for Solidity-based smart contracts have shown that many of them display poor capabilities. One of the causes for such deficiencies is the absence of quality benchmarking datasets, where bugs typically found in smart contracts are present in quantity and accurately labeled. VulLab's main aim is to help tackle this issue as a framework that incorporates both, state-of-the-art vulnerability insertion and vulnerability detection tools. Such capabilities empower users to seamlessly generate benchmark capable datasets from collected contracts and employ them to validate novel analysis tool and obtain an accurate comparison with current state-of-the-art solutions. The framework was able to, from 50 smart contracts collected from the Ethereum mainnet, generate an annotated dataset more than 300 entries which included 20 unique vulnerabilities, and use them to compare 14 analysis tools in approximately 24 hours. VulLab is open-source and is available at* ***https://github.com/lsRyan/vullab.***

1. Introduction and Motivation

Starting in 2008 with Bitcoin, digital currencies, a decentralized blockchain-powered technology, have become a reality for many businesses and individuals globally. One key aspect in this domain are the so-called "smart contracts" [Lin 2022], pieces of code designed to implement complex financial functionalities beyond simple transfers of digital assets among users. Although various programming languages can be used to develop such applications, Solidity, an object-oriented language created by the Ethereum Foundation, is among the most prevalent. However, as the number of everyday users and the volume of transactions through smart contracts increase, concerns about their security have grown among users, companies, and the community as a whole. In fact, past incidents have shown that smart contract vulnerabilities, malicious code, or development bugs can cause significant financial losses for the parties involved.

Security issues in smart contracts have drawn the attention of the scientific and developers communities, especially after the widely publicized "DAO attack"against an Ethereum contract in 2016 [Kushwaha et al. 2022a, Chu et al. 2023, Chen et al. 2020, Chaliasos et al. 2024]. Since then, one of the preferred strategies to help create a safer environment for smart contract users and developers has been the development of vulnerability detection tools [Chaliasos et al. 2024, Ren et al. 2021, Kushwaha et al. 2022b, Zhou et al. 2022]. Even though this is a promising strategy, evaluating and comparing the actual effectiveness of existing tools is hindered by the absence of a robust and verifiable framework, powered by an accurately labeled dataset. Indeed, most detection tools proposed in the literature are validated using distinct datasets, collected and labeled either manually or with automated mechanisms (e.g., existing detection tools) [Chu et al. 2023].

2. Objectives

VulLab's primary objective is to enable security developers to swiftly generate annotated smart contract datasets for evaluating their newly developed vulnerability detection tools. The framework's functionalities, besides facilitating the construction of a benchmark-capable database, also offers a comparison of the application under test against current state-of-the-art vulnerability detectors. Importantly, it is designed with modularity and granularity in mind as it enables selective execution of vulnerability insertion tools and the seamless addition of new ones. Such characteristics provide flexibility to the tool, allowing the developer to execute it on both, high-performance computing systems and also resource-constrained setups.

Another contribution of VulLab is to streamline the measurement of detection capabilities, particularly accuracy, across different vulnerability analysis mechanisms. This capability will allow users to rapidly obtain high quality benchmark datasets and use them to identify strengths and limitations in existing solutions. Such functionality will greatly assist researchers to conduct surveys in regard to bug detection tools, which contributes to the ability to obtain up-to-date overviews of current state-of-the-art in security tools' capabilities, and compare it to new tools under development. Ultimately, VulLab serves as a versatile tool for both researchers and developers, and will help accelerate the development of reliable new security applications that, in turn, will contribute to a safer Ethereum's smart contracts environment.

3. Architecture

To implement the functionalities described in Section 2, VulLab's architecture is divided into two main modules. The first is a Vulnerability Insertion Module, dedicated to the insertion of bugs into the smart contracts collected by the user. Simultaneously, it also provides labels for each insertion, ensuring that all issues are clearly documented and categorized. The second is a Vulnerability Detection Module, which focuses on the automatic execution of various detection tools. The reports resulting from the analyses are automatically compiled and graphically displayed, providing a comprehensive overview of the information obtained by each tested application. Figure 1 presents a diagram of the described architecture, which is further detailed in Sections 3.1 and 3.3.

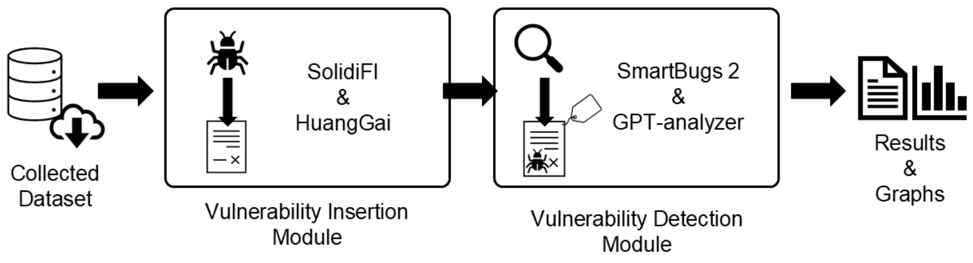

Collected Dataset
SolidiFI & HuangGai
Vulnerability Insertion Module
SmartBugs 2 & GPT-analyzer
Vulnerability Detection Module
Results & Graphs

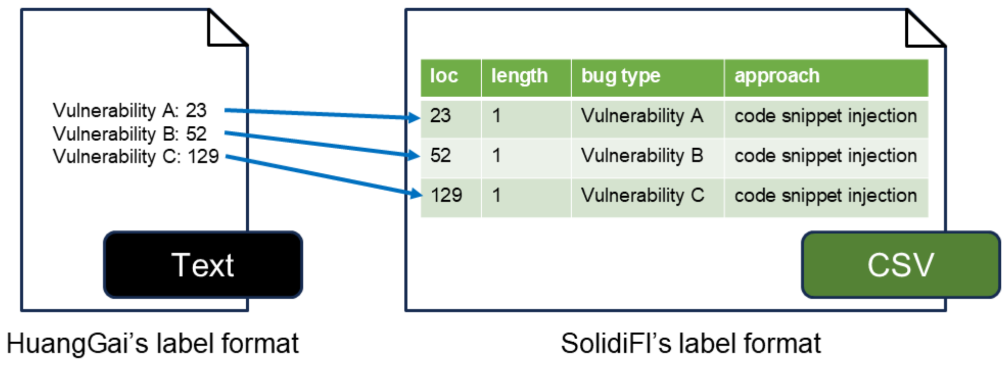
Figura 1. VulLab Architecture

3.1. Vulnerability Insertion Module

The vulnerability insertion module was implemented by combining the execution of two of two tools, SolidiFI [Ghaleb and Pattabiraman 2020] and HuangGai [Jiamao 2021]. SolidiFI was introduced in a 2020 research paper as a pioneer in vulnerability insertion for Solidity smart contracts. It implements a fast and structured method for inserting 7 common bug types, namely: timestamp dependency; unhandled exceptions; integer overflow/underflow; tx.origin usage; reentrancy; unchecked send; and transaction order dependency. HuangGai, in turn, is a more recent vulnerability insertion tool that builds upon the Slither analysis tool [Feist et al. 2019]. It is capable of inserting 20 different types of vulnerabilities in smart contracts, presenting itself as a more comprehensive application than SolidiFI. Although created by a developer from the Blockchain community and not formally published in any scientific venue, it has garnered significant attention and recognition, being commonly cited in recent research papers [Jin et al. 2023, Naeem and Alalfi 2024].

VulLab insertion module executes both SolidiFI and HuangGai independently and in parallel, using the containerized implementation offered by both. This approach offers the ability to speed up the insertion process with multiple instances of Docker containers running simultaneously. Each of them is configured to use a specific tool to try to insert a particular vulnerability into a contract, thus guaranteeing that the underlying tools try to insert all supported vulnerabilities in the target source code files. The user can also configure the number of virtual cores available for execution, which are automatically divided between each instance for a smooth operation in different hardware environments. In addition, this module also translates HuangGai reports into SolidiFI's more structured standard, organizing them in the system directory. This process consist simply of an automatic rewriting of the information provided by each report, which is a simple text file, into a CSV format, with the addition of the vulnerability insertion nature. Since all of the vulnerabilities in SolidiFI are stated to be inserted as "code sniped injection", the same pattern was adopted in the new labels. Besides, HuangGai does not provide vulnerabilities' length or end line. Hence, all are considered as being 1 line long. Figure 2 exemplifies the conversion:

Inspired by SmartBugs2' seamlessly extensible framework for detection tools, VulLab's Insertion Module was implemented with modularity as one of its major aspects. All data regarding each of the injection applications is read from JSON files under the configurations directory. Hence, extension consists of simply adding a



loc	length	bug type	approach
23	1	Vulnerability A	code snippet injection
52	1	Vulnerability B	code snippet injection
129	1	Vulnerability C	code snippet injection

Figura 2. Example of how VulLab's insertion module converts HuangGai's .txt labels into SolidiFI's .csv standard.

new directory with the configuration files in with the correct format and add another entry into the framework's main script: vullab.sh. The last step, despite the added complexity to the addition process, ensures that each of the supported tools can be utilized with an individual flag, which enables users to only execute the ones he or she finds more relevant.

3.2. Vulnerability Detection Module

One core component of this module is SmartBugs2 [Ferreira et al. 2021, di Angelo et al. 2023], a powerful aggregation and execution framework for vulnerability detection. Besides, as many AI-based vulnerability detection technologies, particularly LLMs, have been published in recent years, at least some usage of such kind of tool was adopted [Tann et al. 2019, Ashizawa et al. 2021, Sun et al. 2024, Shen et al. 2023, Wei et al. 2024]. Specifically, VulLab includes a simple GPT-based vulnerability detection framework. It was implemented as an API call which uses a specially crafted system prompt instructing the model to act as an analysis tool. Importantly, in context learning was employed to ensure that the adopted report standard, Sarif 2.1.0, is followed and that the detected tools are named by the SCWE standard.

As with vulnerability insertion, the detection module follows the same strategy of containerized parallel execution for improved performance. It can also easily integrate new vulnerability detection tools into its execution framework, so users can seamlessly use the framework to compare their application with the state of the art. To do that, the users needs to follow the steps presented at SmartBugs2 official repository, which comprises of simply the modification of some files.

The output of this module consists of: (1) a list of detected issues, structured in the JSON-derived Sarif 2.1.0 format [GitHub 2025], commonly used for bug detection reports regardless of the area; (2) listings automatically comparing the inserted vulnerability labels with the generated detection reports; and (3) a graphical representation of the results, in the form of a plot showing the number of detection for each vulnerability and tool under test.

3.3. Documentation

More information about Vullab can be found at `https://github.com/lsRyan/vullab/blob/main/readme.md`. The documentation includes installation options with steps, together with usage instructions. It also includes a demonstration video, which can also be directly accessed using the link `https://www.youtube.com/watch?v=IMw2kVApL3g`.

4. Experimental Results

VulLab was tested with a dataset collected from the following sources: DISL [Morello et al. 2024], Smart Contract Sanctuary [Ortner and Eskandari], and HuangGai's unlabeled dataset. This resulted in a database with more than 300.000 unique smart contracts. The complete execution of VulLab on this database is quite costly, though: even using 20 processing cores, the experiments indicate that one may need 1 hour on average to fully process about 100 contracts. Therefore, for easier reproducibility, in this section the results obtained for a subset of 50 smart contracts, while also limiting the runtime per contract, are presented. This subset was carefully curated to explore all capabilities of the framework, i.e., it enables the insertion of at least one among all supported vulnerabilities, as well as the analysis of all tools. Besides, the selected entries are comprised of contracts that utilize Solidity versions 0.4.x up to 0.6.x and vary from a few dozens to a fez hundred lines of code. Besides not containing contracts above Solidity version 0.7.0, such characteristics are indeed representative of the complete dataset.

VulLab was configured as follows: parallel threading was with 4 cores was adopted; the insertion phase uses SolidiFI, HuangGai, the latter with a 1 minute timeout; and the detection phase uses SmartBugs2 and GPT. SmartBugs2 was employed in its original configuration, without any additional vulnerability detection tools, with a timeout of 15 minutes per tool per contract. With the "all" flag, though, any extra analysis application can be executed. Regarding the GPT component, GPT-4o-mini was choosen as the underlying model; to use other models, though, one can simply modify the API call script.

The results for the 50 contracts subset is compiled in Figure 3. This figure graphically shows what was detected in smart contracts where at least three vulnerabilities were inserted, aiming to give a general overview of what could be expected from a quick execution of the proposed tool.

For a more comprehensive analysis of the framework, however, SmartBugs2 was also executed over a larger portion of the larger dataset, that contains 50.000 smart contracts, which took about a month in 20-core server. As a result, Solhint and Slither can be distinguished as the most capable detection solutions, showing a 75% or higher detection rate for most vulnerabilities. Smartcheck also displayed very promising results, although those were limited to a selection of bugs – especially those easily detected by static analysis, such as the usage of the tx.origin variable for authentication, or the use of floating pragma version.

Mythril, on the other hand, scored poorly across the board, with detection rates barely reaching 20%. This result is somewhat surprising, considering the fact

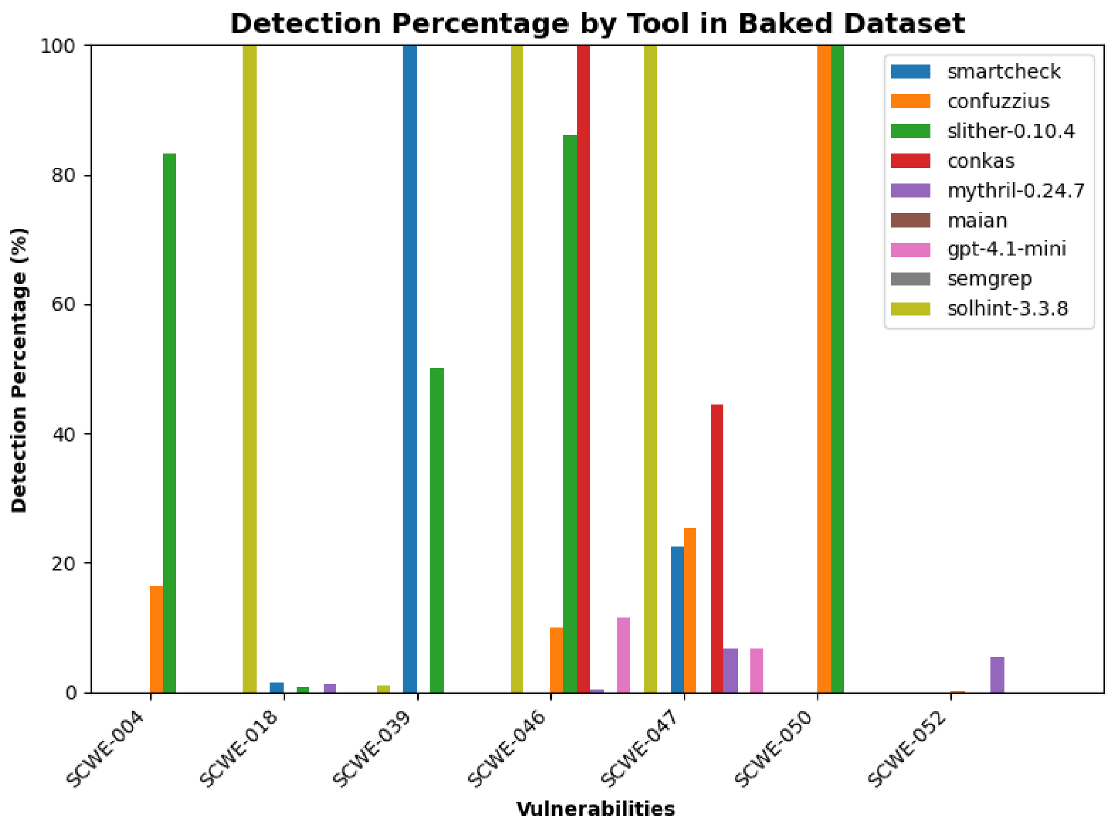


Figura 3. Vulnerability Detection Tools Results for each Vulnerability following the OWASP enumeration [OWASP 2025]

that this ConsenSys-provided tool is commonly employed for real-world audits of smart contracts, and is frequently updated by its open-source community. Similarly, HoneyBadger, Semgrep, Osiris, sFuzz, Oyente, Manticore, Securify, and Conkas presented reasonably poor detection capabilities compared with the alternatives. Some of them, like HoneyBadger, Oyente and sFuzz, have even displayed error messages during their execution, indicating that they do not support the newer versions of Solidity compiler used in the dataset. This is due to the fact that, with the exception of Mythril, Slither and Solhint, most tools do not receive frequent updates, most likely due to the finalization of the scientific work that proposed them. While understandable, such observation raises concerns in regard to the current state-of-the-art detection frameworks.

Another important reason for the poor coverage is that many of the detectors tend to focus only on a narrow selection of vulnerabilities. While not an issue by itself, this leads to some bugs being over-represented, such as SCWE-046 (Reentrancy), with many tools detecting them, while others, such as SCWE-079 (Insecure use of Transfer and Send), not included in the graph above but present in the project repository, do not have appropriate coverage. Importantly, another possible reason for poor performance capabilities is the insufficient validation of detectors, mainly due to the absence of high quality labeled datasets. Due to that, validation practices tend to rely on error-prone manual annotation or on previously published analysis tools, with bug injectors receiving little attention from the research and developer communities [Chu et al. 2023].

Something else that calls our attention is that dynamic analysis tools, such as fuzzers, provided worse detection results compared to their static counterparts. This result may be explained by the 15-minute execution timeout for each contract, though, since those tools usually require long execution times to provide more robust results. A longer execution time would most likely result in better detection capabilities for such tools, as they would be able to execute more tests. One possible next step in regard of this assumption would be the adoption of several different timeouts in order to obtain a more in-depth analysis of fuzzers and other dynamic detectors.

With respect to each individual vulnerability, several important gaps in the current state-of-the-art detection tools were noted. Potentially hazardous bugs such as unprotected or unexpected self-destruction, insecure hashing with multiple variables, and transaction order dependence scenarios (TOD) have been found to be insufficiently covered by all analysis tools. Besides, some practices that contribute to bad code quality were not adequately covered by the tools hereby evaluated. Examples include unprotected access to private variables, and functions declared public when they should be external. It is also noteworthy that most vulnerability analysis tools offered poor detection capabilities for integer underflow and overflow bugs inserted by HuangGai, which was not the case for contracts where the same vulnerabilities were inserted by SolidiFI. Fortunately, such discrepancy is not of outmost importance for modern contracts because such bugs have been resolved by Solidity itself, starting on version 0.8.0.

5. Demonstration

For the VulLab demonstration during the conference, a small dataset of approximately 5 smart contracts will be used. The primary reason for selecting a limited number of contracts is the time-consuming nature of the bug insertion process, even when multiple processes are executed at once. Given this constraint, a compact yet representative dataset is necessary to balance feasibility with the need for a meaningful evaluation. The database has been specifically curated to ensure that every type of vulnerability covered by HuangGai and SolidiFI can be inserted at least once. This selection process guarantees that the demonstration comprehensively showcases the capabilities of both tools.

For the vulnerability detection phase, only static analysis tools will be employed: since dynamic analysis tools typically require a significant execution time to provide meaningful results, their are unsuitable for quick demonstrations. During this step, ChatGPT's API may also be executed to demonstrate the use of GPT-4o-mini. It is important to note that excluding dynamic tools does not affect the demonstration, as their execution process in the framework is identical to that of static tools. Additionally, to prevent potential excessive runtime, Smartbugs2's timer function will be used to limit the execution of each tool to a maximum of one minute per smart contract. Although infinite loops are not expected to occur, this safeguard ensures that the demonstration remains time-efficient, allowing for a smooth and controlled evaluation of the tool's performance.

The framework will also be configured for a "verbose" execution mode, where every step of each process is printed on the screen. This approach enhances visualization, making it easier to follow the procedures in real-time and contributing to a clearer and more engaging demonstration. Due to the architecture of how VulLab was implemented, the vulnerability insertion tools do not inherently provide a visible terminal output. To address this, a visual interface that displays the operation state was implemented. In contrast, Smartbugs2 already includes a terminal-friendly interface that presents the execution process step-by-step.

Finally, the results obtained from executing the vulnerability detection tools on the dataset with inserted bugs will be compiled and presented in an automatically generated graph. This final step consolidates the findings, providing a clear and structured overview of VulLab's execution.

All of the execution process will be undertaken in the researchers laptop, with no need to any additional hardware. Since the insertion and detection tools are implemented in Docker containers, internet access will be required in order to download the container images from the official repository.

6. Conclusions and Future Works

This work shows that VulLab is a promising solution that aims to facilitate the task of benchmarking and comparing vulnerability detection tools. By leveraging the most up-to-date bug insertion frameworks, VulLab enhances researchers' and developers' capabilities of quickly building a large reliable smart contract database with labeled vulnerabilities for analysis. Furthermore, the selected approach does not rely on error-prone human labeling of smart contracts, ensuring a more rigorous constructed process. In addition to allowing the evaluation of existing tools, VulLab should also promote the development and analysis of new vulnerability detection solutions to address gaps in the literature. Finally, by leveraging Smartbugs2's extensibility features, the proposed tool provides a comprehensive and adaptable testing environment, ensuring that vulnerability detection tools can be continuously refined and rigorously assessed.

One possible direction for future work is the development of new vulnerability insertion techniques, along with implementations of these techniques to further enhance the quality and diversity of labeled datasets. Although this has not been a subject of intense development in recent studies, such approaches are acknowledged as promising methods for generating reliably labeled datasets, as further demonstrated in this work. Furthermore, it is useful to consider the creation of a framework similar to Smartbugs2, but designed to aggregate containerized insertion tools and orchestrate their execution through a unified configuration. Such a framework would streamline the integration of multiple such applications, providing a more flexible and scalable solution for future database generation.

Finally, inserting vulnerabilities into smart contracts naturally requires building a representative, unlabeled dataset. However, the analysis undertaken of the literature indicates the absence of a consistently maintained and widely cited dataset for Solidity smart contracts. Consequently, this has prompted many works to assemble their own datasets for validation, sometimes including databases that, albeit large, carry many outdated contracts. This encumbers the construction of solutions intended to promote the secure development of smart contracts, such as bug insertion and bug detection tools. Therefore, it would be important to create a framework that promotes the construction of a collaborative, open-access repository with up-to-date entries, which could then be further updated as new contracts are published. Those datasets could also include labeled entries for prompt usage by vulnerability detection tools.

Acknowledgments: This work was in part supported by the Brazilian CNPq (Research productivity grant 307732/2023-1), CAPES (Finance Code 001), and FAPESP (grant 2020/09850-0).

Referências

Ashizawa, N., Yanai, N., Cruz, J. P., and Okamura, S. (2021). Eth2vec: Learning contract-wide code representations for vulnerability detection on ethereum smart contracts. In *Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure*, BSCI '21, page 47–59, New York, NY, USA. Association for Computing Machinery.

Chaliasos, S., Charalambous, M., Zhou, L., Galanopoulou, R., Gervais, A., Mitropoulos, D., and Livshits, B. (2024). Smart contract and defi security tools: Do they meet the needs of practitioners? In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 705–717, Los Alamitos, CA, USA. IEEE Computer Society.

Chen, H., Pendleton, M., Njilla, L., and Xu, S. (2020). A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Comput. Surv.*, 53(3).

Chu, H., Zhang, P., Dong, H., Xiao, Y., Ji, S., and Li, W. (2023). A survey on smart contract vulnerabilities: Data sources, detection and repair. *Information and Software Technology*, 159:107221.

di Angelo, M., Durieux, T., Ferreira, J. F., and Salzer, G. (2023). Smartbugs 2.0: An execution framework for weakness detection in ethereum smart contracts.

Feist, J., Grieco, G., and Groce, A. (2019). 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.

Ferreira, J. a. F., Cruz, P., Durieux, T., and Abreu, R. (2021). Smartbugs: A framework to analyze solidity smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 1349–1352, New York, NY, USA. Association for Computing Machinery.

Ghaleb, A. and Pattabiraman, K. (2020). How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 415–427, New York, NY, USA. Association for Computing Machinery.

GitHub (Access in 2025). Sarif documentation.

Jiamao (2021). Huanggai. `https://github.com/xf97/HuangGai`.

Jin, L., Cao, Y., Chen, Y., Zhang, D., and Campanoni, S. (2023). Exgen: Cross-platform, automated exploit generation for smart contract vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 20(1):650–664.

Kushwaha, S. S., Joshi, S., Singh, D., and Kaur (2022a). Systematic review of security vulnerabilities in ethereum blockchain smart contract. *IEEE Access*, 10:6605–6621.

Kushwaha, S. S., Joshi, S., Singh, D., Kaur, M., and Lee, H.-N. (2022b). Ethereum smart contract analysis tools: A systematic review. *IEEE Access*, 10:57037–57062.

Lin (2022). A survey of application research based on blockchain smart contract. page 635–690.

Morello, G., Eshghie, M., Bobadilla, S., and Monperrus, M. (2024). Disl: Fueling research with a large dataset of solidity smart contracts.

Naeem, H. and Alalfi, M. H. (2024). Machine learning for cross-vulnerability prediction in smart contracts. In *2024 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 21–28.

Ortner, M. and Eskandari, S. Smart contract sanctuary.

OWASP (Access in 2025). Smart contract security weakness enumeration.

Ren, M., Yin, Z., Ma, F., Xu, Z., Jiang, Y., Sun, C., Li, H., and Cai, Y. (2021). Empirical evaluation of smart contract testing: what is the best choice? In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 566–579, New York, NY, USA. Association for Computing Machinery.

Shen, Z., Chen, Y., and Zhang, W. (2023). Gsvd: Common vulnerability dataset for smart contracts on bsc and polygon. pages 01–16.

Sun, Y., Wu, D., Xue, Y., Liu, H., Wang, H., Xu, Z., Xie, X., and Liu, Y. (2024). 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, New York, NY, USA. Association for Computing Machinery.

Tann, W. J.-W., Han, X. J., Gupta, S. S., and Ong, Y.-S. (2019). Towards safer smart contracts: A sequence learning approach to detecting security threats.

Wei, Z., Sun, J., Zhang, Z., Zhang, X., Li, M., and Hou, Z. (2024). Llm-smartaudit: Advanced smart contract vulnerability detection.

Zhou, H., Milani Fard, A., and Makanju, A. (2022). The state of ethereum smart contracts security: Vulnerabilities, countermeasures, and tool support. *Journal of Cybersecurity and Privacy*, 2(2):358–378.