SmartBugs 2.0: An Execution Framework for Weakness Detection in Ethereum Smart Contracts

di Angelo, Monika ; Durieux, Thomas; Ferreira, João F.; Salzer, Gernot

DOI
10.1109/ASE56229.2023.00060

Publication date
2023

Document Version
Final published version

Published in
Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)

Citation (APA)
di Angelo, M., Durieux, T., Ferreira, J. F., & Salzer, G. (2023). SmartBugs 2.0: An Execution Framework for Weakness Detection in Ethereum Smart Contracts. In J. Gurrola (Ed.), *Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 2102-2105). IEEE. https://doi.org/10.1109/ASE56229.2023.00060

2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00060

SmartBugs 2.0: An Execution Framework for Weakness Detection in Ethereum Smart Contracts

Monika di Angelo
TU Wien
Vienna, Austria
0000-0002-4217-4530

Thomas Durieux
TU Delft
Delft, Netherlands
0000-0002-1996-6134

João F. Ferreira
INESC-ID and IST, University of Lisbon
Lisbon, Portugal
0000-0002-6612-9013

Gernot Salzer
TU Wien
Vienna, Austria
0000-0002-8950-1551

***Abstract*—Smart contracts are blockchain programs that often handle valuable assets. Writing secure smart contracts is far from trivial, and any vulnerability may lead to significant financial losses. To support developers in identifying and eliminating vulnerabilities, methods and tools for the automated analysis of smart contracts have been proposed. However, the lack of commonly accepted benchmark suites and performance metrics makes it difficult to compare and evaluate such tools. Moreover, the tools are heterogeneous in their interfaces and reports as well as their runtime requirements, and installing several tools is time-consuming.**

In this paper, we present SmartBugs 2.0, a modular execution framework. It provides a uniform interface to 19 tools aimed at smart contract analysis and accepts both Solidity source code and EVM bytecode as input. After describing its architecture, we highlight the features of the framework. We evaluate the framework via its reception by the community and illustrate its scalability by describing its role in a study involving 3.25 million analyses.

***Index Terms*—Bytecode, EVM, Solidity, Security, Vulnerability**

I. Introduction

Smart contracts are a fundamental part of blockchain technology, particularly on platforms like Ethereum, where they enable the development of decentralized applications. Benefits like transparency, trust, and security are paired with potential risks, as malicious actors can exploit vulnerable smart contracts and cause substantial financial losses. Therefore, there is a pressing need for automated tools that help identify such vulnerabilities.

The goal of this paper is to present SmartBugs 2.0, a modular execution framework that simplifies the execution of analysis tools for smart contracts, facilitates reproducibility, and supports large-scale experimental setups. It is open-source and publicly available at https://github.com/smartbugs/smartbugs.

Methodology. SmartBugs supports three modes for analyzing smart contracts: Solidity source code, creation bytecode, and runtime code. It currently includes 19 tools encapsulated in docker images. With its standardized output format (via scripts that parse and normalize the output of the tools), it facilitates an automated comparison of the findings across tools. In the context of a bulk analysis, it allows for the parallel, randomized execution of tasks for the optimal use of resources.

Envisioned users. SmartBugs is intended for

- developers auditing smart contracts before deployment,
- analysts evaluating already deployed smart contracts,
- tool developers comparing selected tools,
- researchers performing large-scale analyses,

and thereby advances the state-of-the-art in the automated analysis of smart contracts.

Engineering challenges and new features. Compared to the original version, SmartBugs 2.0 offers the following improvements that overcome several engineering challenges:

- support for bytecode as input
- 8 additional tools
- modular integration of new tools
- support for multiple versions of the same tool
- generic architecture
- increased robustness and reliability
- detection and reporting of tool errors and failures
- SARIF as output format
- mapping of tool findings to the SWC taxonomy[1]

By adding bytecode as accepted input format, the range of smart contracts that can be analyzed by SmartBugs has been extended to programs without source code, including all smart contracts already deployed. Due to its modular structure, SmartBugs 2.0 can easily be extended with further tools. The standardized output format and the mapping to a tool-independent taxonomy both facilitate the integration of a comprehensive vulnerability analysis into the development cycle.

Validation studies. To showcase the capabilities of SmartBugs 2.0, we present a typical use case that demonstrates how SmartBugs 2.0 has supported the largest experimental setup to date, both in terms of the number of tools and the number of analyzed smart contracts.

II. Architecture

Figure 1 depicts the architecture of SmartBugs. It can be started from the command line or called from Python programs. The main arguments to provide are a specification

This project was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under project UIDB/50021/2020. The project was also partially supported by the CASTOR Software Research Centre.

[1] https://swcregistry.io/

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00060

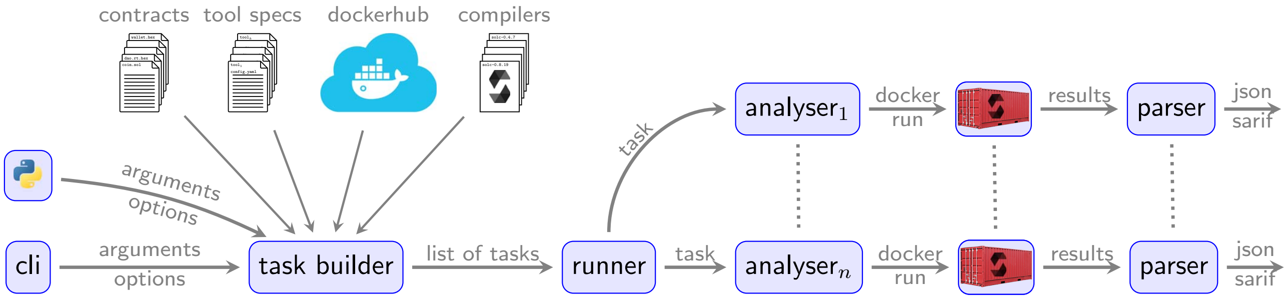

Fig. 1: The Architecture of SmartBugs.

of the smart contracts to process and a list of tools to execute. For a mass analysis, it is also important to specify the number of parallel processes as well as resource bounds per process.

Task builder. For each smart contract matching the specification, the task builder selects those tools that fit the format of the smart contract (source code, creation bytecode, or runtime code) and pulls their Docker images. Moreover, it determines a unique folder for the output of each run. Sometimes the naming scheme specified by the user leads to collisions, meaning that the output of different smart contracts or tools would end up in the same folder. The task builder resolves conflicts in a deterministic way such that any restart of SmartBugs with the same arguments after an interrupt leads to the same output folders.

Most tools analyzing Solidity source code either contain a compiler for a fixed Solidity version or download an appropriate compiler on the fly. Both approaches are problematic in the context of a bulk analysis. In the first case, the integrated compiler is not able to handle smart contracts written for a different version, whereas in the second case an adequate compiler will be downloaded, but used only once and then discarded together with the container of the tool, which leads to redundant downloads during the analysis. Therefore, the task builder inspects the smart contracts and downloads the corresponding compilers beforehand. Later on, during analysis, a compiler matching the smart contract is injected into the container such that the tool is able to compile the contract without attempting to download the compiler itself. Overall, the task builder downloads all resources and detects problems before actually starting the analysis. This prevents racing conditions, errors popping up only during the analysis phase, and minimizes network traffic.

Runner. The runner receives a list of tasks, where each task contains the information for applying a single tool to a single smart contract. The length of the list is roughly the product of the number of smart contracts and the number of tools. To improve the utilization of server resources, the runner randomly permutes the task list. Then it starts the requested number of parallel analyzers, which process the tasks from the list one after the other.

Analyzers. Each analyzer picks a task from the queue of the runner, copies the smart contract, the Solidity compiler (if necessary) and auxiliary scripts to a temporary volume and runs the Docker image of the tool with this volume mounted. Once the Docker container has terminated, the analyzer extracts the result files and writes them to the designated output folder. It adds a file with meta information like the execution time, the arguments of the Docker run, and the version of the tool.

Parsing. In the parsing step, the individual output of a tool is transformed into a standardized format. Each tool has its own parsing script that scans the tool output for *findings* (mostly weaknesses), *errors* (irregular conditions reported by the tool) and *failures* (exceptions not caught by the tool). This information is written to JSON files and — to facilitate the integration of SmartBugs into CI workflows — to SARIF files.

III. Features

Output format SARIF. SmartBugs 2.0 can provide the results in SARIF (Static Analysis Results Interchange Format), an OASIS standard that defines a common reporting format for static analysis tools [1]. SARIF is JSON-based and allows IDEs to access the analysis reports in a uniform way. By adopting a common format that can be parsed by readily available tools, the cost and complexity of aggregating the results of analysis tools into common workflows diminishes. For example, it becomes trivial to integrate SmartBugs into GitHub workflows, since GitHub automatically creates code scanning alerts in a repository using information from SARIF files.[2] For an example of the integration of SARIF produced by SmartBugs and GitHub, we refer the reader to the repository `smartbugs/sarif-tests`.[3]

Bytecode input. On Ethereum, smart contracts are deployed by sending a transaction containing the *creation bytecode*. When executed by the Ethereum Virtual Machine, this code initializes the environment of the new contract and returns the *runtime code* that is actually stored on the chain. In most cases, the creation bytecode is the result of compiling Solidity source code. A significant enhancement of SmartBugs 2.0 is its ability to integrate tools that analyze the creation bytecode and runtime code directly, obviating the need to

[2] https://docs.github.com/en/code-security/code-scanning/integrating-with-code-scanning/uploading-a-sarif-file-to-github

[3] https://github.com/smartbugs/sarif-tests/

procure Solidity sources first. In fact, for many smart contracts deployed on the chain, their source code is not available. Of the 19 tools currently included in SmartBugs, 13 are able to process creation bytecode and/or runtime code.

Provision of proper compiler versions. Another important addition to SmartBugs 2.0 is its ability to select an appropriate compiler for each smart contract. Solidity has seen a rapid development over the past years, with numerous breaking changes. Therefore, programmers are strongly advised to include a pragma that specifies the language version that a smart contract was developed for. Analysis tools have three strategies to cope with this situation. Experimental tools (proofs-of-concept) may come with just a specific compiler version, restricting its applicability. Other tools implicitly assume that the compiler on the command search path matches the smart contract to be analyzed. The most versatile tools inspect the smart contract and download an appropriate compiler before starting analysis. As none of these approaches fits the needs of an unsupervised bulk analysis, the task builder (see its description above) inspects the smart contracts, downloads each required compiler version once before the actual analysis, and then injects the correct one into every container. This allows the tool to run the correct compiler version without the need for on-the-fly downloads, which would cost time and increase the network traffic. As another benefit, this improvement enhances the reproducibility and uniformity of the analyses, as the same compiler version is used consistently across all runs.

Tool integration. With SmartBugs 2.0, it is now possible to incorporate new tools without touching the code of SmartBugs. The details of adding a new tool are described in the wiki of SmartBugs[4]. In essence, a few lines in a configuration file are needed to specify the docker image of the tool and its interface. Moreover, for extracting the findings and errors from the result files, a Python script has to be added. This new flexibility in adding tools also allows researchers to compare the behavior of different versions of the same tool, which is particularly useful for evaluating performance over time, or for ensuring that performance does not degrade with an update.

Mapping to weakness taxonomies. To compare findings across tools, the idiosyncratic labels assigned by each tool need to be mapped to a common frame of reference. SmartBugs 1.0 maps the findings to the vulnerability taxonomy DASP TOP 10[5]. The new version adds a mapping of all findings (including those of the new tools) to the weakness taxonomy of the SWC registry.[6] The SWC registry is a community-driven catalog of software weaknesses in smart contracts, whose granularity is finer than the one of DASP TOP 10. As SmartBugs is modular, further mappings can be included easily[7]. Any mapping provides additional information about the weaknesses found by the tools, which are added to the SARIF output, in order to be displayed in the context of the source or bytecode.

TABLE I: Supported tools.

Tool	Version	New	Contract format		
			Solidity	Creation	Runtime
ConFuzzius	#4315fb7	✓	✓		
Conkas	#4e0f256		✓		✓
Ethainter		✓			✓
eThor	2021 (CCS'20)	✓			✓
HoneyBadger	#ff30c9a		✓		✓
MadMax	#6e9a6e9	✓			✓
Maian	#4bab09a		✓	✓	✓
Manticore	0.3.7		✓		
Mythril	0.23.15		✓	✓	✓
Osiris	#d1ecc37		✓		✓
Oyente	#480e725		✓		✓
Pakala	#c84ef38	✓			✓
Securify			✓		✓
sFuzz	#48934c0	✓	✓		
Slither			✓		
Smartcheck			✓		
Solhint	3.3.8		✓		
teEther	#04adf56	✓			✓
Vandal	#d2b0043	✓			✓
19 tools		8	13	2	13

Supported tools. The tools currently in SmartBugs 2.0 are listed in Table I. Check marks in black (✓) indicate new additions, while the gray check marks in column 'Solidity' identify the capabilities of the old version. We added 8 new tools as well as bytecode support for seven of the old tools. In most cases, bytecode support refers to runtime code. Only two tools are able to handle the creation bytecode as well.

IV. Evaluation

Reception. The appreciation of SmartBugs by the community on GitHub is reflected in the following metrics. With 13 contributors, it received over 400 stars, 81 issues (17 since December 2022) were filed, and 110 users/organizations have forked the repository, with 50 unique cloners in the weeks from May 09 to 22, 2023.

SmartBugs is not only used by developers and security companies, but also in academic studies [2], [3], [4] or master theses [5], [6], [7], [8]. Moreover, components of it have been used to build a ML-based tool [9].

Use case. In the largest experimental study[8] to date [10], we used SmartBugs 2.0 to execute 13 tools on almost 250 000 runtime bytecodes. The tools reported over 1.3 million weaknesses in total. With a resource limit of 30 min and 32 GB, the execution took a total of 31 years. More than half of the tools could run on just 4 GB for the vast majority of the bytecodes and with less than 3 min on average per bytecode, while three tools ran into the limits for more than 1 000 bytecodes.

[4] https://github.com/smartbugs/smartbugs/wiki/Adding-new-analysis-tools

[5] https://dasp.co/

[6] https://swcregistry.io/

[7] Caution is advised when applying such a mapping of tool findings to any taxonomy since it is inherently imprecise.

[8] data available at https://figshare.com/s/5efef6335fa98ddc3ae2

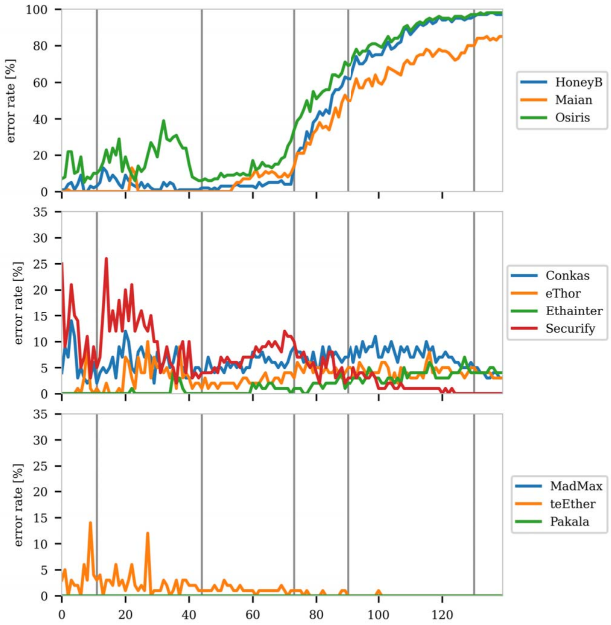

Fig. 2: Tool errors over time: percentage of errors encountered by the tools, in bins of 100 000 blocks.

The new feature in SmartBugs 2.0 of *reporting errors and failures* gives the user an indication, for which bytecodes a tool may be operating outside of its specification. This way, potential findings or non-findings are put in relation to the tools ability to properly analyze the bytecode.

Figure 2 depicts the error rate of each tool on a time line of blocks on the Ethereum main chain, where each data point represents the percentage of reported errors in bins of 100 000 bytecodes. Mythril, Oyente and Vandal report no errors and are not depicted. Apparently, HoneyBadger, Maian, and Osiris experience an increasing error rate after 7.5 million blocks. Conkas, eThor, Ethainter, and Securify report on average an error rate below 15 %. This information can be used to enhance the tools or make informed decisions about whether to use them for more recent smart contracts.

Moreover, tool failures may serve as a measure of robustness. For eight tools (Ethainter, HoneyBadger, MadMax, Maian, Mythril, Osiris, Oyente, Vandal), the failure rate was below 1 % of the bytecodes, whereas for one tool (teEther), the failure rate reached 25 %.

V. Related work

As documented in the previous sections, SmartBugs 2.0 is a major improvement over the original version of SmartBugs [11], which was released in 2019. To the best of our knowledge, the only other execution framework that implements similar ideas is USCV [12]. It comprises eight tools for the analysis of Solidity source code, with seven of them also covered by SmartBugs. USCV seems to be neither widely used nor maintained, as the latest of its 10 commits is from mid-2021 and no issues have been filed so far.

VI. Conclusion

SmartBugs 2.0 has proven to be useful for our own work as well as for fellow researchers and developers. Its extensive use has shown some limitations, partly resulting in enhancement requests by users. We consider the following extensions.

Support for historic compiler versions. SmartBugs supports Solidity 0.4.11 and above. By accessing another repository, we can include versions down to 0.4.0. Compiler versions older than that may be harder to come by.

Support for more complex formats of source code. At the moment, each smart contract has to be contained in a single file. However, complex projects are split into several files. SmartBugs could try to determine the dependencies and transfer them also into the container.

Use of source code mappings. Tools for bytecode input can analyze source code when feeding the compiled source code to the tool. The difficult part is to map the bytecode addresses of weaknesses back to source code lines.

Addition of new tools. With tools steadily emerging, we keep extending SmartBugs, not least with the help of the community contributing further tool configurations.

References

[1] OASIS Static Analysis Results Interchange Format (SARIF) Technical Committee, "Static Analysis Results Interchange Format (SARIF) Version 2.1.0, Oasis Standard," 2020, https://docs.oasis-open.org/sarif/sarif/v2.1.0/os/sarif-v2.1.0-os.html.

[2] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 Ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, no. November. New York, NY, USA: ACM, jun 2020, pp. 530–541.

[3] S. Chaliasos, M. A. Charalambous, L. Zhou, R. Galanopoulou, A. Gervais, D. Mitropoulos, and B. Livshits, "Smart contract and DeFi security: Insights from tool evaluations and practitioner surveys," *arXiv preprint arXiv:2304.02981*, 2023.

[4] I. Qasse, M. Hamdaqa, and B. Þ. Jónsson, "Smart contract upgradeability on the Ethereum blockchain platform: An exploratory study," *arXiv preprint arXiv:2304.06568*, 2023.

[5] B. Aryal, "Comparison of Ethereum smart contract vulnerability detection tools," Master's thesis, University of Turku, 2021. [Online]. Available: https://core.ac.uk/download/pdf/481513588.pdf

[6] N. M. O. Veloso, "Análise Estática de Smart Contracts," Master's thesis, Instituto Superior Técnico, Universidade de Lisboa (ULisboa), 2021.

[7] D. A. P. de Araújo, "A Static Analysis-based Platform-as-Service to Improve the Quality of Smart Contracts," Master's thesis, Instituto Superior Técnico, Universidade de Lisboa (ULisboa), 2021.

[8] J. T. S. Dinis, "Automatic Bug Prioritization of SmartBugs Reports using Machine Learning," Master's thesis, Instituto Superior Técnico, Universidade de Lisboa (ULisboa), 2022.

[9] J. Mandloi and P. Bansal, "A machine learning-based dynamic method for detecting vulnerabilities in smart contracts," *International Journal of Applied Engineering &Technology*, vol. 4, pp. 110–118, 2022.

[10] M. di Angelo, T. Durieux, J. F. Ferreira, and G. Salzer, "Evolution of automated weakness detection in Ethereum bytecode: a comprehensive study," *arXiv preprint arXiv:2303.10517*, 2023.

[11] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "Smartbugs: A framework to analyze Solidity smart contracts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, dec 2020, pp. 1349–1352.

[12] S. Ji, D. Kim, and H. Im, "Evaluating countermeasures for verifying the integrity of ethereum smart contract applications," *IEEE Access*, vol. 9, pp. 90 029–90 042, 2021.