Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Original software publication

SolAR: Automated test-suite generation for solidity smart contracts

S.W. Driessen[a,*], D. Di Nucci[b], D.A. Tamburri[c], W.J. van den Heuvel[a]

[a] JADS/Tilburg University, the Netherlands
[b] University of Salerno, Italy
[c] JADS/Eindhoven University of Technology, the Netherlands

ARTICLE INFO

Keywords:
Blockchain
Smart contract
Testing
Solidity

ABSTRACT

Smart contracts have rapidly gained popularity as self-contained pieces of code, especially those run on the Ethereum blockchain. On the one hand, smart contracts are immutable, have transparent workings, and execute autonomously. On the other hand, these qualities make it essential to properly test the behavior of a smart contract before deploying it. In this paper, we introduce SOLAR, a tool and approach for **Sol**idity **A**utomated Test Suite Gene**R**ation. SOLAR allows smart contract developers to generate test suites for Solidity smart contracts optimized automatically for branch coverage using either a state-of-the-art genetic algorithm or a fuzzing approach. It enables a novel way to handle blockchain operations—or ChainOps—from a pipeline perspective, entailing a larger-scale as well as more manageable and maintainable service continuity.

Code metadata

Code metadata description	
Current code version	v1.2.0
Permanent link to code/repository used for this code version	https://github.com/ScienceofComputerProgramming/SCICO-D-22-00270
Permanent link to Reproducible Capsule	https://zenodo.org/record/7139982
Legal Code License	CC BY-NC-SA 4.0
Code versioning system used	git
Software code languages, tools, and services used	javascript, python, solidity, ganache, docker
Compilation requirements, operating environments and dependencies	
If available, link to developer documentation/manual	
Support email for questions	s.w.driessen@jads.nl

The code (and data) in this article has been certified as Reproducible by Code Ocean: https://codeocean.com/. More information on the Reproducibility Badge Initiative is available at https://www.elsevier.com/physical-sciences-and-engineering/computer-science/journals.

* Corresponding author.
E-mail address: s.w.driessen@jads.nl (S.W. Driessen).

https://doi.org/10.1016/j.scico.2023.103036
Received 20 October 2022; Received in revised form 3 October 2023; Accepted 3 October 2023

Available online 12 October 2023

1. Introduction

Decentralized finance (DeFi) has changed finance since the summer of 2020, with yield farming and the rise of tokens promising attractive yields to crypto traders. In DeFi, paperwork is replaced by blockchain smart contracts, which could be affected by software defects and vulnerabilities with astonishing effects.[1] Blockchain and smart contract technologies facilitate trusted computing by gracefully allowing for development in a distributed and decentralized manner [1,2].

On the one hand, smart contracts are capsules of code, similar to classes in object-oriented programming languages, e.g., Java and Python, deployed on distributed systems such as blockchains. In large part, the inherent qualities of blockchains, such as the immutability of data and ease of access to the data stored offer new possibilities previously unseen. At the same time, however, these properties render extensive testing of critical importance, especially before code deployment.

On the other hand, the most prominent and investigated blockchain that deploys smart contracts is *Ethereum*, whose smart contracts are often written in the *Solidity*[2] programming language. Different tools have been proposed to assist smart contract developers with testing their smart contracts.

Some approaches have applied fuzzing to generate inputs that cause errors or break specified invariants [3,4]. However, these fuzzing approaches do not allow testing of the intended behavior. They focus solely on generating inputs and do not create composite tests (i.e., tests that modify the state in a smart contract by calling one or more methods in succession). Manticore leverages symbolic execution to generate input data given a smart contract and a test property to verify [5]. Other tools are based on formal verification techniques to verify the behavior of the smart contract against a model of expected behavior [6]. Formal verification allows for extensive testing of the smart contract but relies heavily on the model provided by the developer. This technique is computationally intensive and does not scale well as the complexity of the code increases [7]. Finally, many tools focus exclusively on detecting certain known vulnerabilities, such as the famous reentrancy vulnerability. In doing so, they focus on detecting code patterns indicative of one or more vulnerabilities [2,8]. However, they do not provide test cases or suites that allow testing intended behavior or identifying bugs outside the specified vulnerabilities.

Previous studies have shown that the lack of such test suites is one of the major challenges hampering a successful technology transfer from academics to industry [9,10]. For Solidity developers, Zou et al. [11] report that 54.7% developers are hindered by the lack of powerful tools, including testing tools, for blockchain-specific development. Furthermore, existing tools do not consider all corner cases and scenarios.

To the best of our knowledge, only two tools currently aim at providing test suites covering all corner cases leveraging search-based algorithms. The tool by Wang et al. [12] is not publicly available for experimentation. In contrast, SynTest-Solidity by Olsthoorn et al. [13] follows a similar approach to our tool but does not appear to focus as much on extendibility towards multiple algorithms and blockchain manipulation techniques.

In the following, we present our tool called SolAR. SolAR aims to assist Solidity smart contract developers by automatically creating test suites for smart contracts optimized for branch coverage, effectively covering as many corner cases as possible. In addition to providing a test suite that can detect bugs and test intended behavior, the test suites can be used for different types of testing. For example, the test suites could be used to evaluate mutation testing frameworks (e.g. [14–16]) or combined with one or more existing oracles to detect vulnerabilities. Another use case that we are investigating for future work is to use SolAR to compare the effectiveness of different tools that create Control Flow Graphs (CFGs) from Solidity bytecode.

2. Solution overview and architecture

The full SolAR tool and an additional appendix on its usage can be found online.[3] To use it, the developer compiles their smart contract and prepares a local Ethereum simulation (e.g., ganache[4]). Afterward, they can launch SolAR through a CLI and set the tool's hyperparameters to fit their smart contract's testing requirements. A full list of the possible hyperparameter settings can be found online.[5] SolAR allows tests to be generated using either the genetic algorithm DynaMOSA [17] or a custom-made Fuzzer[6] In addition to the full tool, we provide docker images for the fuzzing and genetic algorithm approaches of SolAR online.[7] This docker is convenient for understanding the tool and comes with a prepared environment that includes a ganache blockchain environment.

The tool automatically generates test suites by going through the steps shown in Fig. 1. It starts by scraping the provided ABI (Application Binary Interface) for all the methods that can change the internal state of the contract (by default SolAR ignores pure functions such as views), as well as any hardcoded values that might be useful for test generation. Afterward, a control dependency graph is generated [18], whose nodes represent coherent blocks of code and edges corresponding to transitions between the blocks. By creating a test suite that traverses all the edges at least once, SolAR ensures that all logical paths through the code are covered.

Test suite generation happens in several steps: first, a random set of test cases is generated. In each test case a new smart contract instance is deployed, and one or more method calls are invoked by various accounts, with varying Ether balances and input

[1] https://rekt.news/leaderboard/.
[2] See http://solidity.readthedocs.io/.
[3] https://github.com/AGSolT/SolAR.
[4] https://trufflesuite.com/ganache/.
[5] https://github.com/AGSolT/SolAR/blob/master/DynaMOSA/SolMOSA/Config.ini.
[6] See: https://arxiv.org/abs/2102.08864.
[7] https://zenodo.org/record/7139982.

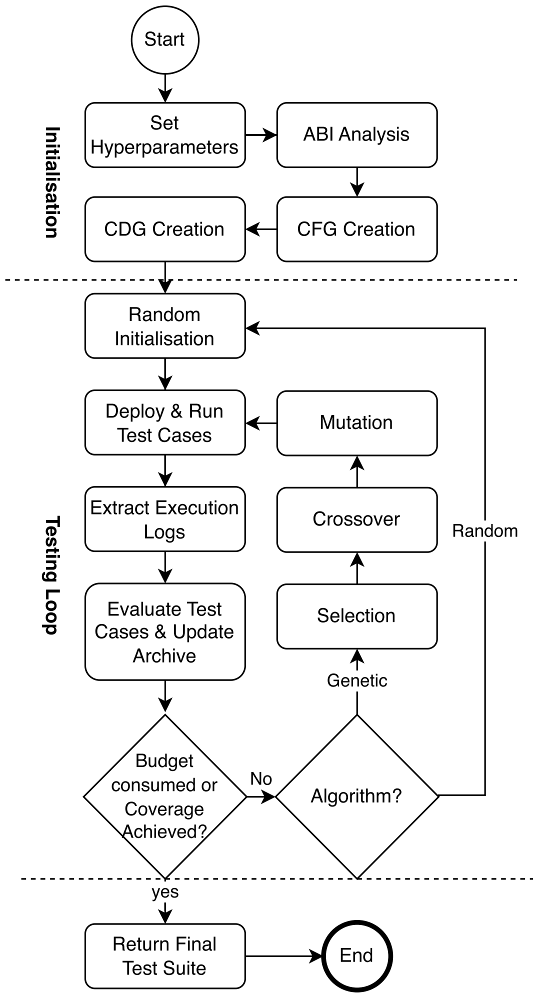

Fig. 1. A step-by-step overview of the SolAR tool.

parameters, thus modifying the state of the smart contract. Each test case is evaluated to check how close it is to covering each branch in the CDG, and the best test cases are saved for the final test suite. If the budget is not consumed or there are still branches to be covered, new test cases are generated, and the cycle begins again.

At the end of the process, the developer is presented with a ready-made test suite, which can be used for functional testing. Additionally, SolAR provides several insights into the process, such as log files for both the interaction with the blockchain and the off-chain processing of the test suites. The CFG and CDG are also provided, currently in text format, so the developer can see which parts of their code are (not) covered. Additional metrics about the performance of the smart contract and the selected algorithm are provided in a CSV file, which can aid in evaluating the tool.

Software architecture

The SolAR tool consists of several components shown in Fig. 2, which are also presented as a pre-packaged Docker file for convenient experimentation. The user interacts with SolAR back-end through a command line interface. The back-end then accesses the smart contract ABI location, uses the CFG-creation module,[8] and refines the resulting CFG into a CDG (using Lengauer and Tarjan's algorithm [19]) whose edges are covered by the final test suite. SolAR then generates test cases and executes these on the blockchain (ganache) environment. The blockchain environment returns the results of the method calls in each test suite, and a separate module keeps track of the state of the stack of the Ethereum Virtual Machine (EVM) while executing the method calls. Based on these results, SolAR evaluates which test cases contribute the most to covering all the branches and the cycle of generating test cases, executing them shown in Fig. 1 and evaluating them is repeated until a final test suite is returned.

[8] Currently the `evm_cfg_builder.cfg` python library.

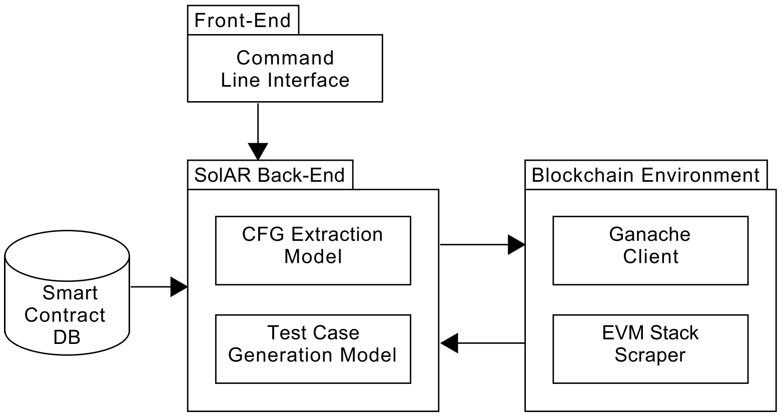

Fig. 2. SOLAR Architecture, an overview.

3. SOLAR characteristics and evaluation

To validate the usefulness of SOLAR, we scraped GitHub to obtain a set of 36 smart contracts from amongst the most starred projects containing Solidity files. We determined the appropriate hyperparameters for each of these contracts to test the intended functionality and its interaction with the blockchain environment. Because smart contracts run on a blockchain environment, their functionality depends on its properties, which are out of the control of the smart contract developer. In particular, this includes the fact that *all* existing wallets (both individuals and smart contracts) on the blockchain can attempt to interact with them. We identified seven patterns where Solidity smart contracts depend on the blockchain environment and provide options, in the form of hyperparameters, to help guide test suite generation to cover as many corner cases as possible. As far as we know, SOLAR is the only Solidity testing tool that explicitly considers these blockchain environment interactions. Below, we present a brief discussion of the patterns and how they affect test suite generation; examples of contracts with each of these patterns, explanations of where the interaction is logged and available for users, as well as a full list of hyperparameters can be found online.[3]

The *sender-* and *value dependencies* indicate whether the functionality of the smart contract depends on who sends transactions to them and what amount of Ether they include in these transactions. In order to properly account for these patterns, SOLAR can create different accounts with different amounts of Ether that can be included in the transactions. The *block-* and *time dependencies* indicate whether the contract relies on the block number or the blockchain time for its functionality. SOLAR can artificially manipulate the blockchain environment to simulate the passage of time required for smart contracts with these patterns. Finally, three patterns exist that result in dependence on external smart contracts. First, whenever *addresses* are passed *as variables*, the smart contract could potentially interact with other smart contracts. SOLAR allows users to deploy other smart contracts in the test environment and include their address in the input variables selected for test case generation. Second, passing a *non-existing* address as a variable can lead to unexpected behavior, which is why SOLAR is capable of doing exactly that. Finally, some smart contracts[9] behave differently when interacting with the so-called *zero-address* (0x00) which is a specially reserved address in Solidity.

To evaluate the effectiveness of SOLAR at generating test suites that cover all corner cases, we ran the tool with both the random search and the genetic algorithm that come with it. For each evaluation, the tool ran until full branch coverage was achieved or the tool had gone through 100 iterations of test suite generation. An extensive description of the experiments and an overview of the results can be found online.[2] However, we note two important insights. On the one hand, SOLAR is quite successful at achieving a high level of branch coverage for Solidity smart contracts (achieving full branch coverage for roughly two-thirds of the time). On the other hand, using SOLAR, we have discovered bugs in smart contracts currently published on the Ethereum blockchain. These contracts, along with a brief explanation of the problem are available in our online appendix.[10]

4. Impact

Since SOLAR was made available online, we have become aware of several initiatives that use it, both from industry and academia. As an example of an industrial application of SOLAR, we point to a project we are starting up with AstraKode,[11] a company that develops low-code tools for blockchain and smart contract development. We are looking to apply and evaluate SOLAR to test smart contracts developed with their clients to further validate its potential for practitioners. For researchers, we find that SOLAR is a good baseline for testing different aspects of Solidity smart contract testing. For example, we are currently working with the developers of the control-flow-graph creation tool EtherSolve [20] to evaluate its applicability for test suite generation. As another example, we

[9] See e.g., https://github.com/fravoll/solidity-patterns/blob/master/GuardCheck/GuardCheck.sol.
[10] https://github.com/AGSolT/SolAR/tree/master/DynaMOSA/Smart%20Contracts/ErrorContracts.
[11] https://www.astrakode.tech/.

know of at least one initiative currently underway that uses the test suites generated by SOLAR to apply mutation testing to Solidity smart contracts.

5. Conclusions and future work

In this work, we introduced the SOLAR tool, which allows for the automatic generation of test suites for Solidity smart contracts. SOLAR enables smart contract developers to test smart contract behavior and researchers to investigate the effectiveness of new algorithms and tools for testing smart contracts. We intend to continue expanding the tool as indicated in the previous section. For example, we aim to include symbolic execution to develop fitness functions that improve the efficiency of meta-heuristics.

CRediT authorship contribution statement

S.W. Driessen: Conceptualization, Data curation, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft. **D. Di Nucci:** Conceptualization, Methodology, Visualization, Writing – review & editing. **D.A. Tamburri:** Conceptualization, Supervision, Writing – review & editing. **W.J. van den Heuvel:** Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] B.-J. Butijn, D.A. Tamburri, W.-J. van den Heuvel, Blockchains: a systematic multivocal literature review, ACM Comput. Surv. 53 (3) (2020) 61:1–61:37, http://dblp.uni-trier.de/db/journals/csur/csur53.html#ButijnTH20.

[2] L. Anderson, R. Holz, A. Ponomarev, P. Rimba, I. Weber, New kids on the block: an analysis of modern blockchains, arXiv preprint, arXiv:1606.06530, 2016.

[3] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, A. Hobor, Making smart contracts smarter, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 254–269.

[4] T.D. Nguyen, L.H. Pham, J. Sun, Y. Lin, Q.T. Minh, sFuzz: an efficient adaptive fuzzer for solidity smart contracts, in: Proceedings – International Conference on Software Engineering, vol. 11, 2020, pp. 778–788.

[5] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, A. Dinaburg, Manticore: a user-friendly symbolic execution framework for binaries and smart contracts, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE, IEEE, 2019, pp. 1186–1189.

[6] I. Garfatta, K. Klai, W. Gaaloul, M. Graiet, A survey on formal verification for solidity smart contracts, in: ACM International Conference Proceeding Series, Feb. 2021.

[7] C. Baier, J.-P. Katoen, Principles of Model Checking, vol. 950, MIT Press, 2008, pp. I–XVII, 1–975.

[8] B. Jiang, Y. Liu, W.K. Chan, Contractfuzzer: fuzzing smart contracts for vulnerability detection, in: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering, ASE, IEEE, 2018, pp. 259–269.

[9] E. Daka, J. Campos, G. Fraser, J. Dorn, W. Weimer, Modeling readability to improve unit tests, in: 2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 – Proceedings, 2015, pp. 107–118.

[10] G. Grano, S. Scalabrino, H.C. Gall, R. Oliveto, An empirical investigation on the readability of manual and generated test cases, in: Proceedings – International Conference on Software Engineering, 2018, pp. 348–351.

[11] W. Zou, D. Lo, P.S. Kochhar, X.-B.D. Le, X. Xia, Y. Feng, Z. Chen, B. Xu, Smart contract development: challenges and opportunities, IEEE Trans. Softw. Eng. 5589 (2019) 1.

[12] X. Wang, H. Wu, W. Sun, Y. Zhao, Towards generating cost-effective test-suite for ethereum smart contract, in: SANER 2019 – Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering, 2019, pp. 549–553.

[13] M. Olsthoorn, D. Stallenberg, A. Van Deursen, A. Panichella, Syntest-solidity: automated test case generation and fuzzing for smart contracts, in: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, 2022, pp. 202–206.

[14] P. Chapman, D. Xu, L. Deng, Y. Xiong, Deviant: a mutation testing tool for solidity smart contracts, in: 2019 IEEE International Conference on Blockchain, Blockchain, IEEE, 2019, pp. 319–324.

[15] Y. Ivanova, A. Khritankov, Regularmutator: a mutation testing tool for solidity smart contracts, Proc. Comput. Sci. 178 (2020) 75–83.

[16] J.J. Honig, M.H. Everts, M. Huisman, Practical mutation testing for smart contracts, in: Data Privacy Management, Cryptocurrencies and Blockchain Technology: ESORICS 2019 International Workshops, DPM 2019 and CBT 2019, Luxembourg, September 26–27, 2019, Proceedings 14, Springer, 2019, pp. 289–303.

[17] A. Panichella, F.M. Kifetew, P. Tonella, Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets, IEEE Trans. Softw. Eng. 44 (2018) 122–158, https://doi.org/10.1109/TSE.2017.2663435.

[18] J. Ferrante, K.J. Ottenstein, J.D. Warren, The program dependence graph and its use in optimization, ACM Trans. Program. Lang. Syst. 9 (1987) 319–349.

[19] T. Lengauer, R.E. Tarjan, A fast algorithm for finding dominators in a flowgraph, ACM Trans. Program. Lang. Syst. 1 (1979) 121–141, https://doi.org/10.1145/357062.357071.

[20] F. Contro, M. Crosara, M. Ceccato, M.D. Preda, Ethersolve: computing an accurate control-flow graph from ethereum bytecode, in: IEEE International Conference on Program Comprehension 2021, May 2021, pp. 127–137.