How are Solidity smart contracts tested in open source projects? An exploratory study

Luisa Palechor
University of Alberta
Edmonton, AB, Canada
palechor@ualberta.ca

Cor-Paul Bezemer
University of Alberta
Edmonton, AB, Canada
bezemer@ualberta.ca

ABSTRACT

Smart contracts are self-executing programs that are stored on the blockchain. Once a smart contract is compiled and deployed on the blockchain, it cannot be modified. Therefore, having a bug-free smart contract is vital. To ensure a bug-free smart contract, it must be tested thoroughly. However, little is known about how developers test smart contracts in practice. Our study explores 139 open source smart contract projects that are written in Solidity to investigate the state of smart contract testing from three dimensions: (1) the developers working on the tests, (2) the used testing frameworks and testnets and (3) the type of tests that are conducted. We found that mostly core developers of a project are responsible for testing the contracts. Second, developers typically use only functional testing frameworks to test a smart contract, with Truffle being the most popular one. Finally, our results show that functional testing is conducted in most of the studied projects (93%), security testing is only performed in a few projects (9.4%) and traditional performance testing is conducted in none. In addition, we found 34 projects that mentioned or published external audit reports.

CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Smart contracts, Solidity, testing

ACM Reference Format:
Luisa Palechor and Cor-Paul Bezemer. 2022. How are Solidity smart contracts tested in open source projects? An exploratory study. In *IEEE/ACM 3rd International Conference on Automation of Software Test (AST '22), May 17–18, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524481.3527228

1 INTRODUCTION

Academic and industrial attention in the blockchain technology has exploded in recent years. Blockchain is a decentralized, distributed technology that enables trust between entities without the involvement of a third party by storing permanent and unalterable *blocks* that ensure the integrity of the stored data. New data can be stored on the blockchain in several ways, one of them being through smart contracts. A smart contract is a self-executing script of which the proper execution is given when predefined conditions are met [23].

AST '22, May 17–18, 2022, Pittsburgh, PA, USA
2022. ACM ISBN 978-1-4503-9286-0/22/05…$15.00
https://doi.org/10.1145/3524481.3527228

Ethereum is the most popular blockchain that stores smart contracts, with Solidity being the most popular high-level language for defining these contracts. Solidity is still a relatively immature language that is changing rapidly [8]. The rapid changes in Solidity add challenges to smart contract development and make it hard to thoroughly test smart contracts. Testing smart contracts is essential as bugs in smart contracts can lead to large financial losses (as demonstrated by the DAO bug which resulted in a loss of $60M [9]).

Several studies have proposed solutions to create high-quality smart contracts [10, 14, 16, 18–20]. In addition, there exist a small number of open source development platforms, e.g., `Truffle` [13] and `Hardhat` [17], to support the development and testing of smart contracts. However, three surveys [1, 4, 5] revealed that open source developers grumble about the lack of support for testing smart contracts compared to that for testing traditional applications.

In this paper, we perform an exploratory study on how Solidity smart contracts are tested in open source projects. We studied 139 Solidity open source projects from GitHub repositories focusing on the following research questions:

RQ1. Who are the developers involved in testing Solidity smart contracts? We found that mainly the core developers of a project are responsible for testing smart contracts, even though there are usually several non-core developers that contribute to the rest of the project.

RQ2. What are the preferred tools and testnets for testing Solidity smart contracts? We observed that `Truffle` and `Hardhat` are by far the most used tools for testing smart contracts. Few projects used other tools, in particular those stemming from academic research.

RQ3. What types of tests are performed on Solidity smart contracts? Smart contracts running in blockchain is a recent revolutionary technology. As in traditional applications, testing is essential to guarantee high-quality development. However, smart contract testing is affected by new technical challenges, e.g., the decentralization and immutability features. This research question shows smart contracts are tested mostly by functional test cases.

This paper is organized as follows. Section 2 discusses background concepts and related work. In Section 3, we describe our exploratory study setup and methodology. Section 4 presents our study results. Section 5 discusses the threats to the validity and Section 6 concludes our study.

2 BACKGROUND AND RELATED WORK

Blockchain and smart contracts. Blockchain is a decentralized, distributed technology that allows secure transactions to be made between entities without requiring a third party (such as a bank). An important property of a blockchain is that records (*blocks*) cannot

be altered once they are added to the blockchain. The most famous example of blockchain technology is the Bitcoin currency which runs on the Bitcoin blockchain. However, recently other implementations of blockchain have become popular, with the Ethereum blockchain [3] being the most notable example.

Ethereum is open source and allows users to develop their own programs that run on the blockchain. An example of such a program is a smart contract, a self-executing script (often written in Solidity) that executes in the Ethereum Virtual Machine (EVM) once the specified conditions in the contract are met. For example, the ownership of a house is transferred once a certain amount of money is transferred into the account that is specified in the contract. The performed transactions consume computational effort in the EVM that is measured in gas units. Since gas fees are paid in Ether (the Ethereum currency), gas represents real money. Hence, any transaction on the Ethereum blockchain costs money.

Testnets. To test a blockchain application, testnets can be used. Testnets are public blockchains that permit developers to test smart contracts in a staging environment with free gas. The most popular testnets for Ethereum are `Kovan`, `Rinkeby` and `Ropsten`, which allow to test smart contracts' functionalities and interactions in an EVM. Several EVM-based testnets exist that mimic the Ethereum blockchain but at the same time differ in technical configurations.

Tools for testing smart contracts. Most of the proposed testing tools focus on the security of smart contracts (e.g., `MythX` [6], `SmartCheck` [24], `ContractFuzzer` [14], `Harvey` [26]), `Madmax` [11], `Securify` [25], `Slither` [10], `Manticore` [20], `Echidna` [12], and, `ReGuard` [16]. Only a few frameworks focus on other aspects of testing smart contracts such as their functionality (e.g., `Truffle` [13], `Hardhat` [17], `dapp.tools` [7], and `Brownie` [2]).

3 METHODOLOGY

In this section, we explain the methodology of our exploratory study, which is depicted by Figure 1.

Gathering data: We queried the GitHub API to collect open source Solidity projects using a custom query in August, 2021. To ensure mature and active Solidity projects were considered, we limited our search to Solidity projects with at least ten stars and one hundred commits, likewise, we only considered non-archived projects. After these steps our dataset had 199 Solidity projects. Finally, we manually reviewed the Solidity projects and filtered out projects that were not intended for smart contract development. We excluded tutorials, repositories with no description, and testing frameworks, libraries and tools. At the end of this step, we obtained 139 repositories that contained at least one smart contract.

Identifying test files for smart contracts: The next step is to identify the test files in the projects. We were interested in actual test files and other files that reveal information about how testing is done in Solidity projects. To create our dataset, we first used heuristics to identify test files in each Solidity project. We included files that matched at least one of the next patterns: `*/test*/*`, `*/script/*`, `*mock*`, `*.spec.*` or `*t.sol*`. Second, the first author reviewed the obtained files and kept the files that provided data about the unit and integration testing, performance test cases and used testing tools. We ended up having 3,270 test files in our dataset. In addition, we manually reviewed the names of the folders in each project to make sure that we did not overlook relevant files. If we came across additional files that contained relevant information about the testing process, such as `Makefile` that specify the used testing framework, we considered those for RQ2 and RQ3 as well.

Finally, we reviewed YAML files to analyze if the test was performed within a continuous integration (CI) pipeline. We recorded the testing script and the CI tool name that was used.

Identifying configuration files: We manually identified configuration files for testing tools (e.g., `truffle.js`). Our final dataset is available online [21].

4 RESULTS OF OUR EXPLORATORY STUDY

In this section we present the motivation, approach and findings for our research questions.

4.1 RQ1. Who are the developers involved in testing Solidity smart contracts?

Motivation: We study which developers are involved in smart contract testing to get a better idea of how this task is approached in open source projects. Our hypothesis is that this type of testing is not a popular contribution to make for open source contributors.

Approach: We used the same metric as in our prior work [15] to identify if developers working on testing are core developers of a project. First, we ran the git command `git shortlog -s -n HEAD` to identify the developers that committed to each test file. Second, we determine the top-n developers working on the project, where n was the total number of developers coding test files. We identify these n developers as the core developers of the project. Finally, we calculated the ratio between the total number of developers working on the tests and the top-n developers of every project.

After obtaining the (test) developers names, we manually deduplicated them. For example, in the /balancer-labs/balancer-core project we merged the records for "fernandomartinelli" and "fernando martinelli" as they clearly point to the same person. In total, we merged 11% of the developers because they contributed to the profile under different names.

To compare the distributions, we utilized the Wilcoxon signed-rank test. The Wilcoxon signed-ranked test is a non-parametric statistical test of which the null hypothesis states that two input distributions are identical. We used Cliff's delta d to quantify the difference between the distributions, using the thresholds mentioned by Romano et al. [22]: negligible if $|d| \leq 0.147$; small if $0.147 < |d| \leq 0.33$; medium if $0.33 < |d| \leq 0.474$; and large if $0.474 < |d| \leq 1$.

Findings: **In the studied projects a median of 80% of the test developers was also part of the core team.** Figure 2 shows the portion of test developers who are core among the total of test developers. We observe a median of 0.80 for test developers who are part of the core team. This result suggests that testing smart contracts is a responsibility of core developers and having contributions from external developers is unlikely. The Wilcoxon signed-rank test shows that the two distributions are significantly different with a large effect size.

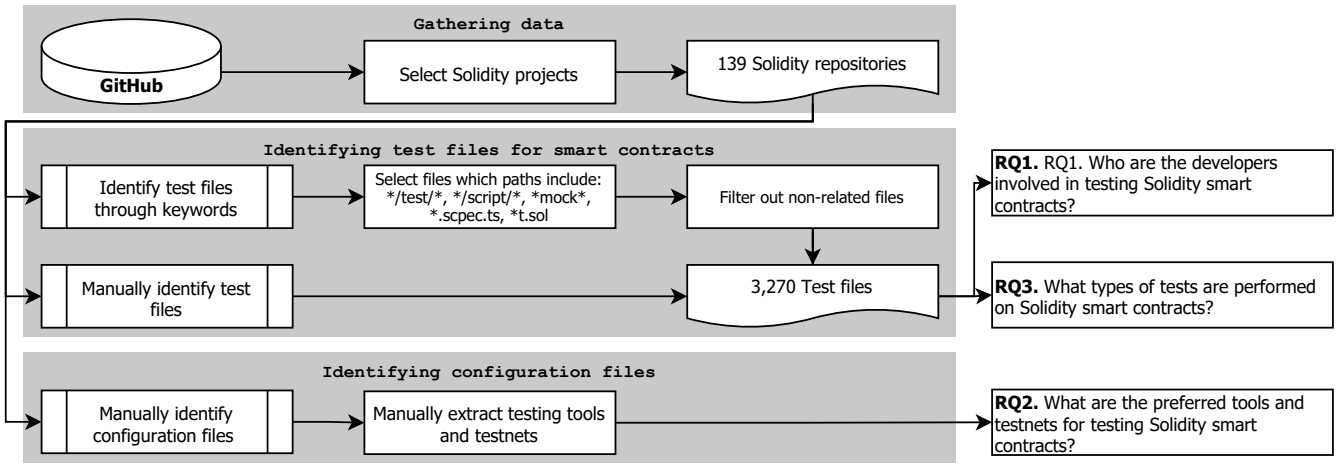

Figure 1: Overview of the steps taken in our methodology.

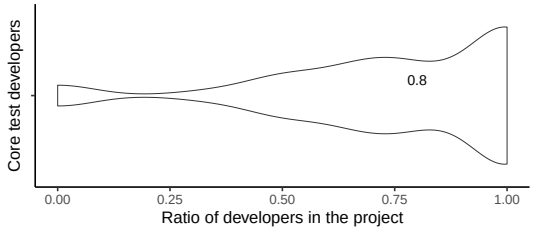

Figure 2: Portion of test developers who are core developers.

4.2 RQ2. What are the preferred tools and testnets for testing Solidity smart contracts?

Motivation: Investigating which tools and testnets developers use to test smart contracts reveals which testing approaches developers take in practice. Moreover, even though all studied contracts are intended to end up on the Ethereum blockchain or any blockchain that adopts EVM as its smart contract runtime, for testing this is usually avoided as executing transactions over an EVM-compatible blockchain is not free (i.e., because of the gas consumption). Instead, developers can use testnets which are public blockchain networks to test the smart contracts in a real-world environment.

Findings: **69 (50%) of the studied projects use `Truffle` for testing.** Table 1 shows that the most used testing frameworks are `Truffle`, `Hardhat` and `dapp.tools` which are mainly used for unit testing. Contrary, the least used testing tools in our data are `Manticore`, `Embark` and `Solgraph`.

48 (34.5%) of 139 projects do not provide information about using testnets as part of their testing process. This result suggests that these projects might test the smart contracts in a private blockchain without interaction with existing smart contracts, e.g., a private blockchain offered by `Truffle`, `Hardhat` or `dapp.tools`. Another explanation is that testnets are used but configured locally, e.g., within an integrated development environment (IDE).

Table 1: Testing tools used by Solidity open source projects.

Tool	Description	# of projects
`Truffle`	Testing framework	69
`Hardhat`	Testing framework	54
`dapp.tools`	Testing framework	15
`Waffle`	Testing framework	8
`Slither`	Security analysis	6
`Jest`	Testing framework	5
`Brownie`	Testing framework	4
`Echidna`	Security analysis	4
`pytest`	Testing framework	3
`Mythx`	Security analysis	2
`Cetora`	Security analysis	2
`Manticore`	Security analysis	1
`Embark`	Testing framework	1
`Solgraph`	Security analysis	1

Such configuration would imply that the tests are not intended for automated execution, e.g., in a CI pipeline.

While the median number of used testnets is 1, several projects test their smart contracts on more than one testnet. Figure 3 depicts the distribution of the number of testnets per project. The /sushiswap/sushiswap project uses 13 testnets for testing its smart contracts, /sushiswap/shoyu uses 8, /nftfy/nftfy-v1-core 7, and 3 projects perform their tests on 6 testnets. These projects all support smart contracts in multiple EVM-compatible blockchains (such as `Polygon`, `xDAI`, `Binance`, `Huobi`, `Avalanche Fuji` (testnet), `okexchain`, `arbitrum and celo`).

55 (39%) of 139 projects use `Kovan` and 48 (34%) use `Rinkeby`. Figure 4 shows the distribution of used testnets in the studied projects. The `Kovan`, `Rinkeby` and `Ropsten` testnets are the most popular. This is not surprising since these testnets are meant to test smart contracts in Ethereum which was the first blockchain to support smart contracts.

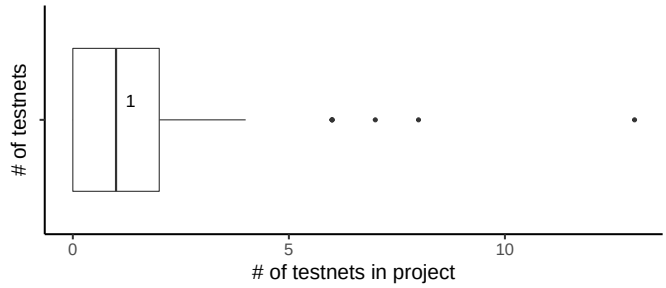

Figure 3: Number of testnets per project.

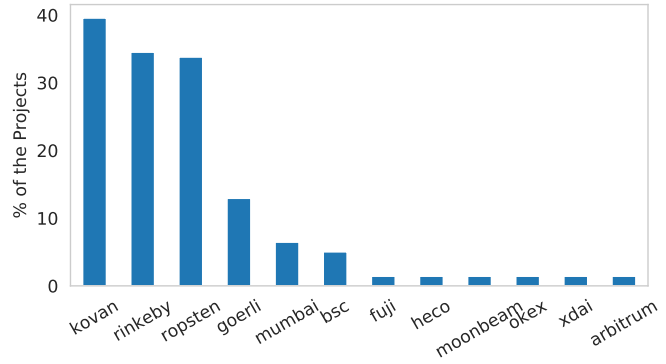

Figure 4: Popular testnets within OS projects.

4.3 RQ3. What types of tests are performed on Solidity smart contracts?

Motivation: Applications running in smart contracts are enhanced by features like decentralization and immutability. However, building a bug-free smart contract given these new features is tricky. This RQ investigates which tests developers conduct on smart contracts to overcome these challenges.

Approach: We manually went through the identified test files and classified the type of test as either a functional, security or performance test. We did not come across any other types of tests in the studied projects.

Findings: **9 (6.5%) projects do not provide details about how they test their smart contract(s).** After inspecting each of these projects, we did not find test-related code. We highlight, however, that 3 of the reviewed projects may include test files privately. For example, the /PancakeBunny-finance/Bunny project contains audit report files which specify the code coverage index for 4 smart contracts, the test report of 95 scenarios and a vulnerability analysis report. Furthermore, the `gitignore` file in the /yieldyak/smart-contracts project defines "test" within the list of untracked files. Finally, the /nftfy/nftfy-v1-core project includes a CI file that points to tasks related to testing the smart contracts of the project, e.g., `truffle test`.

130 (93.5%) of the projects in our data include functional testing as part of the testing strategy. Our results corroborate the findings of prior research [4, 5].

57 (41%) of the studied projects develop Solidity mocks to test the smart contracts by creating presumed scenarios. The mocks we found in the studied projects were developed to either override, add or reset internal functions to simulate particular scenarios within the blockchain application. For example, the 88mphapp/88mph-contract project has 5 mock files, that inherit from the ERC20 smart contract. Every mock file initializes the smart contract in a different way to simulate and test different scenarios.

81 (58%) of the studied projects include testing in the CI pipeline. We noticed that the CI tools used were `Actions`, `TravisCI`, and `CircleCI`, similar to traditional open source projects. We also found 3 projects that include security testing in their CI pipeline.

13 (9.4%) projects conduct security tests. We observed that fuzzy testing and audit reports are used to test the security of smart contracts. 9 (6.5%) of the projects conduct fuzzing testing by feeding the code with random and erroneous data. These projects attempt to execute test cases multiple times while changing a particular parameter's value. As a consequence, multiple sequences of transactions are generated and reviewed automatically.

34 (24.5%) of the studied projects include audit reports performed by third parties. The audit reports have different formats across projects, but overall, they show the smart contracts' potential vulnerabilities and provide recommendations for improvements. For example, the audit report for the /balancer-labs/balancer-core project discusses a vulnerability that allows assets to be stolen.

None of the projects implement traditional performance testing. We did not find traditional performance tests (e.g., execution time) for the smart contracts. However, we did find 41 (29.5%) projects that report gas consumption in their tests.

5 THREATS TO VALIDITY

Internal validity. We filter the Solidity projects based on their number of stars and number of commits to measure maturity. Future studies should investigate if our findings would vary for a different set of projects.

Construct validity. Our manual review of the test files for smart contracts could be biased. Similarly, there are testing tools that have command-line interface (CLI) options available that allow developers to test their smart contracts without leaving a trace in the source code repository. As a result, the numbers in our study may be lower estimates.

External validity. Our results are limited to Solidity projects and may not generalize to other languages for writing smart contracts. In addition, we focus on open source projects only. Future studies should extend our study to include other languages and proprietary projects.

6 CONCLUSION

This paper analyzes how Solidity smart contracts in 139 open source projects are tested. We found that testing files for smart contracts do not receive many contributions from developers who are not already part of the core development team of a project. In addition, we found that functional testing is the most common for smart contracts, followed by security testing. The most popular testing tools are `Truffle` and `Hardhat`, and several projects use a third party to conduct an audit of their smart contracts.

REFERENCES

[1] Amiangshu Bosu, Anindya Iqbal, Rifat Shahriyar, and Partha Chakraborty. 2019. Understanding the motivations, challenges and needs of blockchain software developers: A survey. *Empirical Software Engineering* 24, 4 (2019), 2636–2673.

[2] Brownie. 2021. *Brownie - Brownie 1.17.2 documentation.* https://eth-brownie.readthedocs.io/en/stable/

[3] Vitalik Buterin. 2014. A next-generation smart contract and decentralized application platform. *white paper* 3 (2014), 1–36.

[4] Partha Chakraborty, Rifat Shahriyar, Anindya Iqbal, and Amiangshu Bosu. 2018. Understanding the Software Development Practices of Blockchain Projects: A Survey. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, Article 28, 10 pages.

[5] Jiachi Chen, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. 2021. Maintenance-related concerns for post-deployed Ethereum smart contract development: issues, techniques, and future challenges. *Empirical Software Engineering* 26, 6 (2021), 1–44.

[6] ConsenSys. 2021. *ConsenSys MythX.* https://mythx.io/

[7] dapp.tools. 2022. *dapp.tools.* https://dapp.tools/

[8] Ethereum. 2021. *Solidity v0.8.11.* https://docs.soliditylang.org/en/v0.8.11/

[9] Etherscan. 2022. *TheDAO smart contract.* http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code

[10] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.

[11] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2020. MadMax: Analyzing the out-of-Gas World of Smart Contracts. *Commun. ACM* 63, 10 (sep 2020), 87–95.

[12] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 557–560.

[13] ConsenSys Software Inc. 2022. *Truffle Suite.* https://trufflesuite.com/

[14] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 259–269.

[15] Philipp Leitner and Cor-Paul Bezemer. 2017. An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ACM, 373–384.

[16] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. ReGuard: Finding Reentrancy Bugs in Smart Contracts. In *IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. 65–68.

[17] Nomic Labs LLC. 2022. *Hardhat - Ethereum development environment for professionals.* https://hardhat.org/

[18] Yao Lu, Xinjun Mao, Zude Li, Yang Zhang, Tao Wang, and Gang Yin. 2016. Does the role matter? an investigation of the code quality of casual contributors in github. In *23rd Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 49–56.

[19] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the ACM SIGSAC conference on computer and communications security*. 254–269.

[20] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1186–1189.

[21] L. Palechor and CP. Bezemer. 2022. *How are Solidity smart contracts tested in open source projects? An exploratory study.* https://doi.org/10.5281/zenodo.5862800

[22] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, Jeff Skowronek, and Linda Devine. 2006. Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen'sd indices the most appropriate choices. In *annual meeting of the Southern Association for Institutional Research*. Citeseer, 1–51.

[23] Nick Szabo. 1997. The idea of smart contracts. *Nick Szabo's papers and concise tutorials* 6, 1 (1997).

[24] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. ACM, 9–16.

[25] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 67–82.

[26] Valentin Wüstholz and Maria Christakis. 2020. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 1398–1409.