Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 178 (2020) 75–83

Procedia Computer Science

www.elsevier.com/locate/procedia

9th International Young Scientist Conference on Computational Science (YSC 2020)

RegularMutator: A Mutation Testing Tool for Solidity Smart Contracts

Y. Ivanova[a], A. Khritankov[a]

[a]*Moscow Institute of Physics and Technology (National Research University), 9 Institutskiy per., Dolgoprudny, 141701, Russian Federation*

Abstract

With the growing popularity of smart contracts, the problem of validation of their correctness becomes more and more relevant, but at the moment there are no universally recognized tools for their testing. In this paper, we propose to apply the mutational analysis to improve reliability of Solidity smart contracts. We identified wide spread errors in the source code of existing contracts and developed a RegularMutator tool for mutation analysis. It has shown its effectiveness in testing a large smart contracts project. As a result of our analysis, we managed to improve the test suites of the project to find the discovered defects and, increase the quality of the test suite.

Keywords: mutation testing; smart contract; blockchain; verification; solidity

1. Introduction

1.1. Introduction to smart contracts

The scope of this study is the verification and testing of smart contracts. Smart contracts are special algorithms and protocols designed to facilitate, verify and implement negotiations, as well as to execute the contract, allowing for making irreversible transactions without the participation of third parties [1]. Smart contracts are designed to be executed on distributed registry platforms (blockchain). In this paper, we will focus on smart contracts written in Solidity programming language for the Ethereum blockchain platform.

According to Ethereum developers, their main intent was to create a protocol for building decentralized applications (dApps), providing a large class of decentralized applications with ease of development, while at the same time

* Y. Ivanova. Tel.: +7-995-904-1104
E-mail address: ivanova.yua@phystech.edu

10.1016/j.procs.2020.11.009

allowing them to share an economic environment and blockchain security [2]. DApps are hosted at the platform by distributed autonomous organizations (DAOs). The concept of DAOs allows organization stakeholders to make collective decision about the allocation of certain resources and influence the company's performance.

Currently, dApps based on smart contracts are present in such areas as:

- Banking (SETL payment and settlement platform [3])
- Legal (Propy [4])
- Healthcare (MedRec [5])

1.2. Problem relevance

An important feature of smart contracts executed on the Ethereum platform is that their bytecode is publicly available and immutable after publication. As a result, special attention should be paid to testing and debugging of such programs. However, as practice shows, developers are not always able to guarantee the safety and reliability of their product, and users suffer losses because of this. For example, the world-famous attack ”The DAO” led to losses of $60m [6].

There are a lot of verification methods developed to check the correctness and reliability of programs, including automated testing. In general, testing of a software product is a process of executing a program with the aim of finding discrepancies in its execution with a set of requirements. An important task is to estimate the quality of the given requirements, namely, to estimate the ability of the test sets to detect errors of program work.

Currently, widespread metric of estimating the quantity and quality of tests is line code coverage. This metric is calculated as a percentage of the source code lines (SLOC) that have been executed during the testing process. It is applicable at several stages of testing including unit testing and integration testing. The disadvantage of using the metric is that there is no direct connection between the metric value and the quality of the test suite. In practice, even full line coverage of the code does not guarantee the absence of defects [1]. At the same time, the significant advantages of this approach are the speed and simplicity of its calculation.

1.3. Proposed solution

An alternative to line metrics is mutation testing (analysis). This approach involves automatic changes of the program syntax and creation of semantic variants of the program (artificial defects) [7]. These changes are called mutations and are based on mutation operators, programs with defects (variants) are named mutants. Test cases which can distinguish the behaviour of mutant programs from the original program correspond to test targets. If a test distinguishes mutant behaviour from the original program, we say the mutant is ”killed” or ”detected”; in another case, we say the mutant is ”alive”. The test suite quality metric is calculated as a ratio of killed mutants to all mutants generated. It can be used alongside line coverage to estimate the quality of the test sets.

Mutations mainly model real errors which can be made by programmers [8], that is why it is convenient to use them to find new test goals which have not been considered yet and create test cases for them. In general, this approach is considered the best method of test effectiveness evaluation [9].

2. Problem statement

At the moment, there are no generally accepted tools for assessing the quality test suites of smart contracts, despite their increasing popularity, it is a new area of research. At the moment, there is not enough experimental data, there are no or insufficient standard data sets/code for comparison of different methods. Various methods from other fields are also being tested and new scientific problems are being identified, so there is no generally accepted algorithm for evaluating test sets yet. To improve the quality of test suites, developers perform a manual review of tests and source code, use static code analyzers and calculate a line coverage. The contract studied in the Experiment section was also analyzed by a security company, with subsequent correction of the found vulnerabilities [10]. In this article, we investigate the fundamental applicability of the mutation testing method for the analysis of smart contracts:

1. Is mutation analysis applicable to improve the quality of test suites of smart contracts?
2. Which mutation operators are more effective for Solidity smart contracts?
3. How does mutation testing compare with other existing methods in terms of error detection and the final value of the code coverage metric?

The following tasks were set to answer the research questions:

- Identify common errors in the code of existing smart contracts, make a list of possible mutation operators
- Develop a tool for mutating smart contracts
- Test the tool on an existing large project containing smart contracts and test suites for them. Compare line coverage and mutation score values

3. Methods

As a first step, we investigated common errors in the code base of existing projects in order to create a list of mutation operators based on them. To do this, a dataset of 100 projects was compiled with source code hosted on github.com, containing smart contracts. Then commits in these repositories were manually analysed. Next we highlighted the most frequent changes found in commits that correct errors in the project's smart contracts code. We also used the "Smart Contract Weakness Classification and Test Cases" classification scheme [11] as a source of errors. Among the most common errors found, we selected those whose mutation operators can be represented as regular expressions for further substitution in the source code. This step is necessary due to the fact that the chosen technology of application of mutations involves writing regular expressions. In the future, this restriction can be overcome. The last part of the used operators are represented by a set of program-level mutation operators, described by P. Amman and J. Offutt in the book [7]. In total, all the operators used can be classified as follows.

1. Absolute Value Insertion - modification of arithmetic expressions by the function that returns the absolute value of the expression
2. Relational Operator Replacement - an occurrence of the relational operator ($<$, $\leq$, $>$, $\geq$, $==$) is replaced by other relational operator
3. Arithmetic Operator Replacement - an occurrence of the arithmetic operator ($+$, $-$, $*$, $/$, $**$, $\%$) is replaced by other operator
4. Conditional Operator Replacement - an occurrence of the logical operator (and, or) is replaced by other operator
5. Deleting a line of source text
6. Solidity specific operators - for example, replacing the function state from "view" to "pure". The full list is provided in Appendix A

We implemented a software tool RegularMutator using Python programming language. Given a Truffle [12] project with Solidity smart contracts and test suites RegularMutator generates mutants for each source file in the project. Mutation injection is implemented using the regular expressions library. After generating mutant files, RegularMutator substitutes mutant files instead of the original ones in a row and executes project test suites. The test output is analysed and the mutant is assigned one of the statuses:

1. "SURVIVED", if all test suites passed successfully
2. "KILLED", if one or more tests found an error and failed
3. "COMPILATION ERROR", if the project failed to compile

The mutation score is a ratio of killed mutants (2) to all mutants generated (1+2) except those with compilation error status. After running all mutants RegularMutator prints out the testing report to the console.

4. Experiment

In order to assess the effectiveness of RegularMutator tool, an experiment was conducted, during which the work of the tool was tested on the example project. We systematically searched for candidate projects among the smart contract dataset and selected the project ”POA Bridge Smart Contracts” [13]. Requirements to the candidate projects were as follows:

1. Active and ongoing support from developers
2. At least 50 Solidity source files
3. Sufficient number of test cases (line coverage at least 90%)
4. Simplicity of deployment and test suites’ execution

During the experiment, the project version v5.0.0-rc0 was used, corresponding to commit id 7ce4441ab77e1c3e4d01017d862c53516933645 in the project repository on github.com [13]. The value of the line coverage metric for this project version was equal to 96%. As a result of RegularMutator operation, 129 files were tested. The test report is represented in the Table 1.

Table 1. Test report.

Overall mutant number	Survived	Killed	Compilation error	Mutation score	Line coverage
871	110	25	736	18.5%	96%

After the experiment we examined 50 mutants manually and found that mutants obtained using Relational Operator Replacement and Solidity specific operators are most likely to survive. Among the mutants obtained by replacing logical operators, we selected one presented in the Table 2.

Table 2. Mutant selected for investigation.

Original line	Mutated line
require(_to ! = blockRewardContract);	require(_to > blockRewardContract);

The mutant represents a typical programmer error when implementing conditional statements. In this case, conditional statement executes the rule that reward for the block calculation cannot be sent directly. If a similar error occurred elsewhere in the source code, a number of valid operations would be rejected unnecessarily. In order to check the project for such errors, we updated the test suites so that the tests only pass if there is a strict inequality of the compared variables. Although we did not find any new errors as a result of running updated tests, we can assume that the quality of the test suites was improved. That is because the project is still actively developing and such an error may occur in the future, so tests should prevent them. The value of line coverage in turn did not change after the tests were changed.

5. Results and discussion

As a result of the experiment, we obtained a fairly low mutation score with a high line coverage value for the ”POA Bridge Smart Contracts” project and at the same time a high line coverage value. We managed to improve the quality of the test suites by modifying them in accordance with the mutation analysis results. This answers the first research question and shows that mutation testing is applicable to Solidity smart contracts. The line coverage value remained unchanged. We would propose POA project to apply RegularMutator on the project and rely on mutation score rather than line coverage when evaluating the quality of test suites.

We also have answered the second research question, what are the most effective mutational operators for Solidity. During the work of RegularMutator, it was found that mutation operators from the Relational Operator Replacement class and, Solidity specific operators have the highest efficiency. Namely, among the surviving mutants, there were 18% ROR and 26% Solidity Specific operators. It is also worth noting that half of the Solidity Specific operators are represented by replacing ”pure” function state with ”view” and vice versa. This means that it makes sense to focus on these types of mutations in the future.

To address the third research question we compared the mutation rate with the line coverage metric and found that the value of the coverage metric obtained by mutation analysis was significantly lower than the line coverage metric. At the same time, mutation analysis revealed vulnerability and improved test suites. Also, in the Related work section, we have compared RegularMutator to the existing tools.

5.1. Error analysis

It is worth noting that not all mutants from the survived ones are useful and lead to an increase in the quality of test suites. For example, mutations made over comments inside the source code cannot affect the execution of the program, so they must be eliminated. Also, the number of mutants that cause compilation errors was high, as well as the time spent on checking them.

5.2. Shortcomings, things to improve

Machine time spent on conducting the experiment was about 50 hours. Indeed, mutation analysis is computationally complicated and therefore it is not applicable in some areas. Besides, a lot of surviving mutants should be investigated after the experiment and analysed by manually. However, given the immutability of smart contracts after publication and potential financial losses due to project vulnerabilities, we can consider the increased computational cost of testing justified.

6. Related work

6.1. Other existing tools overview

Mutation analysis of smart contracts is a new and active area of research. In the work by S. Akca et al. [14] the SolAnalyser tool is presented, in which mutation operators are based on 8 main types of vulnerabilities found in existing smart contracts. The authors demonstrate in an experiment that their work is superior to five other previously developed tools for mutation testing of smart contracts: Oyente [15], Securify [16], Maian [17], SmartCheck [18] and Mythril [19]. Recent work also include MuSC developed by Li Zixin et al [20]. It performs mutation operations at the AST (Abstract Syntax Tree) level and also provides users with a graphical interface. There are also tools offered by commercial organizations. For instance, ”Smart Contract Verification Platform” offered by ChainSecurity [21] and ”Smart Contract Analysis and Verification package” by Runtime Verification Inc [22].

Still, there are some problems that remain unsolved. The main problem is high computational cost of executing a set of tests when generating numerous mutants. In order to solve this problem, it is necessary to develop methods for selecting the most effective mutations, as well as to investigate existing mutation optimization methods with regard to the smart contracts area. Research on the applicability of methods for determining equivalent mutants is also relevant.

6.2. Comparison with RegularMutator

Among the tools listed above, we chose SolAnalyser [14] and ”Smart Contract Analysis and Verification package” by Runtime Verification Inc [22] and compared their approaches and algorithms with those of RegularMutator. The first one was chosen because it has experimentally proven to outperform many other existing tools. The second – as it is a commercial product, which is already used by smart contracts developers. Although it is a product for formal program verification, rather than evaluating the quality of tests, it was interesting to see if mutation analysis could help to improve test suites for a project which was previously formally verified.

6.2.1. SolAnalyser

First SolAnalyser statically analyses source code of Solidity smart contracts to find potentially vulnerable locations. Then it uses the AST representation to insert vulnerabilities. The following types of vulnerabilities are considered [14]:

1. Integer overflow/underflow
2. Division by zero
3. Timestamp dependency
4. Authorisation through tx.orgin
5. Unchecked send
6. Repetitive call function
7. Out of gas

From the vulnerabilities presented above, we implemented ”Integer overflow/underflow”, ”Timestamp dependency” and ”Authorisation through tx.orgin” insertion. However, we also implemented other solidity specific mutations along with classical ones as stated in section Methods. Since Ethereum is relatively young, developers of smart contracts often confuse certain specific details. For example, the addresses of contracts and external owners accounts play such an important role in smart contracts that there are several ways to specify addresses and it can confuse the developer. For this reason we added ”tx.origin” to ”msg.sender” and vice versa mutations. There are several papers that discuss typical errors made by smart contract developers, including [23]. Therefore, we will not discuss the description of each mutation in detail further.

As the set of mutation operators in RegularMutator contains more operators including various Solidity specific ones, it can be considered more complete than in SolAnalyser. Moreover, as shown in the Results and discussion section, the most effective operators were Relational Operator Replacement (ROR) and solidity specific mutations. However, ROR mutations and most of the mutations from the solidity specific operators list were not implemented in SolAnalyser. Though, the use of AST code representation should be considered more effective in terms of execution time and therefore preferable. It is also worth noting that in contrast to our work in the article ”SolAnalyser: A Framework for Analysing and Testing Smart Contracts” [14], the authors did not compare mutation operators for their effectiveness in identifying vulnerabilities and did not explore the possibility of using the results to improve tests.

6.2.2. ”Smart Contract Analysis and Verification package” by Runtime Verification Inc

Methods of operation of the ”Smart Contract Analysis and Verification package” are discussed in the article ”A Formal Verification Tool for Ethereum VM Bytecode” [24]. Before the verification process starts the developers' team needs to define a formal specification of smart contract functionality. After that, the binary or low-level code (such as EVM binary) generated by the compiler from high-level smart contract code (such as Solidity) is compared with the specification. This is done with the help of the K-framework [25] and its verification infrastructure. The main difference between this tool and RegularMutator is that in our case the whole process is automated and does not require any additional knowledge from the user. At the same time, to use the ”Smart Contract Analysis and Verification package”, the developer should manually describe the specifications, which takes a considerable amount of time.

7. Conclusion

The article presents a new tool for improving the reliability of smart contracts written using Solidity language, RegularMutator. In addition to the classic mutation operators, in the tool were implemented language-specific operators that correspond to common errors made during the development of smart contracts. Injection of mutations to the program code is implemented using regular expressions.

The tool proved its effectiveness during an experiment in which mutation testing of an existing large project containing smart contracts was performed. As a result of testing, we were able to detect vulnerabilities in the source code of the project and improve the quality of test suites. The line coverage value for the project was equal to 96%, and the mutation score – 18.5%. It was concluded that it is more reliable to consider mutation analysis instead of line coverage when evaluating the quality of test samples.

In the future work, we propose to replace the method of inserting mutations using regular expressions with building an abstract syntax tree and making mutations that will not cause compilation errors. Special attention will be paid to the selection of mutants in order to use the most effective ones.

Appendix A. Solidity specific operators

Table A.3. Solidity specific operators.

Original string	Substituted string
true	false
false	true
uint	int
ufixed	fixed
int16	int8
int32	int16
int64	int32
memory	storage
storage	memory
view	pure
pure	view
constant	pure
payable	
msg.sender	tx.origin
tx.origin	msg.sender
ether	wei
minutes	seconds
days	minutes
weeks	days
years	weeks
block.timestamp	0
msg.value	0
msg.value	1
addmod	mulmod
mulmod	addmod
call	delegatecall
call	callcode
delegatecall	call
delegatecall	callcode
callcode	delegatecall
callcode	call

References

[1] Yang Q, Li JJ, Weiss DM. A survey of coverage based testing tools. The Computer Journal; 2009; 52(5); 589-597. doi: 10.1093/comjnl/bxm021.
[2] Buterin V. A Next-Generation Smart Contract and Decentralized Application Platform. White paper. 2014 Jan;3(37).
[3] Huynh C. Global Banks Acquire Stake in The Blockchain Based Payment Startup SETL, https://news.coinsquare.com/business/banks-acquire-stake-setl/; 2018 [accessed 20.04.20]
[4] Keeffe D. How Propy Works: Propys Transaction Management Platform, https://propy.com/blog/how-propy-works-transaction-management-software/; 2019 [accessed 20.04.20]
[5] MedRec. What is Medrec, https://medrec.media.mit.edu/; 2016 [accessed 20.04.20]
[6] del Castillo M. The DAO Attacked: Code Issue Leads to $60 Million Ether Theft, https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft; 2016 [accessed 20.04.20]
[7] Ammann P, Offutt J. Introduction to Software Testing. 1st ed. Cambridge; Cambridge University Press; 2008 [chapter 5]. doi:10.1017/9781316771273.
[8] Yue J, Harman M. An Analysis and Survey of the Development of Mutation Testing. IEEE Transactions on Software Engineering; 2011; 37(5); 649-678. doi: 10.1109/TSE.2010.62.
[9] Andrews JH, Briand LC, Labiche Y, Namin AS. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. IEEE Transactions on Software Engineering; 2006; 32(8); 608-624. doi: 10.1109/TSE.2006.83.
[10] Nikashin B. TokenBridge (by POA Network) Smart Contracts Security Analysis, https://blog.smartdec.net/tokenbridge-by-poa-network-smart-contracts-security-analysis-156c509abe5f; 2019 [accessed 25.08.20]
[11] SmartContractSecurity. SWC Registry, https://swcregistry.io/; 2020 [accessed 20.04.20].
[12] Truffle Blockchain Group. Truffle, https://www.trufflesuite.com/truffle; 2020 [accessed 20.04.20].
[13] Barinov I, Fedoseev K, Nardelli G. Smart contracts for TokenBridge, https://github.com/poanetwork/tokenbridge-contracts; 2020 [accessed 01.05.20].
[14] Akca S, Rajan A, Peng C. SolAnalyser: A Framework for Analysing and Testing Smart Contracts. 26th Asia-Pacific Software Engineering Conference (APSEC), Putrajaya, Malaysia; 2019; 482-489, doi: 10.1109/APSEC48747.2019.00071.
[15] Luu L, Chu D-H , Olickel H, Saxena P, Hobor A. Oyente. Making Smart Contracts Smarter. The 2016 ACM SIGSAC Conference; 2016; 254-269. doi:10.1145/2976749.2978309.
[16] Tsankov P, Dan A, Cohen DD, Gervais A, Buenzli F, Vechev M. Securify: Practical Security Analysis of Smart Contracts. arXiv:1806.01143; 2018.
[17] Nikolic I, Kolluri A, Sergey I, Saxena P, Hobor A. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. Proceedings of the 34th Annual Computer Security Applications Conference; 2018; arXiv:1802.06038
[18] Tikhomirov S, Voskresenskaya E, Ivanitskiy I, Takhaviev R, Marchenko E, Alexandrov Y. SmartCheck: Static Analysis of Ethereum Smart Contracts. IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), Gothenburg, Sweden; 2018; 9-16.
[19] MythX. Mythril Classic: Security analysis tool for Ethereum smart contracts, https://github.com/ConsenSys/mythril-classic; [accessed 30.09.19].
[20] Li Z, Wu H, Xu J, Wang X, Zhang L and Chen Z. MuSC: A Tool for Mutation Testing of Ethereum Smart Contract. 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA; 2019; 1198-1201. doi: 10.1109/ASE.2019.00136.
[21] Smart Contract Verification Platform. ChainSecurity, https://chainsecurity.com/platform/; 2020 [accessed 22.08.20].
[22] Smart Contract Analysis and Verification. Runtime Verification Inc, https://runtimeverification.com/smartcontract/; 2020 [accessed 22.08.20].
[23] Atzei N, Bartoletti M, Cimoli T. A survey of attacks on Ethereum smart contracts (SoK). 6th Conf. on Principles of Security and Trust (POST); 2017; volume 10204 of LNCS; 164186; Uppsala, Sweden; doi: 10.1007/978-3-662-54455-6 8.
[24] Park D, Zhang Y, Saxena M, Daian P, Ros G. A Formal Verification Tool for Ethereum VM Bytecode. Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 18); Lake Buena Vista, FL, USA; doi: 10.1145/3236024.3264591.
[25] Rosu G, Serbanuta T. An Overview of the K Semantic Framework. The Journal of Logic and Algebraic Programming; 2010; 79; 397434. doi: 10.1016/j.jlap.2010.03.012.