MuSC: A Tool for Mutation Testing of Ethereum Smart Contract

Zixin Li[1], Haoran Wu[1], Jiehui Xu[1], Xingya Wang[1*], Lingming Zhang[2], Zhenyu Chen[1]
[1]State Key Laboratory for Novel Software Technology, Nanjing University, China
[2]Department of Computer Science, University of Texas at Dallas, USA
*corresponding author: xingyawang@nju.edu.cn

***Abstract*—The smart contract cannot be modified when it has been deployed on a blockchain. Therefore, it must be given thorough test before its being deployed. Mutation testing is considered as a practical test methodology to evaluate the adequacy of software testing. In this paper, we introduce MuSC, a mutation testing tool for Ethereum Smart Contract (ESC). It can generate numerous mutants at a fast speed and supports the automatic operations such as creating test nets, deploying and executing tests. Specially, MuSC implements a set of novel mutation operators w.r.t ESC programming language, Solidity. Therefore, it can expose the defects of smart contracts to a certain degree. The demonstration video of MuSC is available at https://youtu.be/3KBKXJPVjbQ, and the source code can be downloaded at https://github.com/belikout/MuSC-Tool-Demo-repo.**

***Keywords*—Blockchain, Ethereum Smart Contract, Mutation Test, Mutation Operator**

I. Introduction

Blockchain technology, which was proposed by Nakamoto Satoshi in 2008 [1], has been a research hot spot [2]. In 2014, a Turing complete blockchain platform named Ethereum came into being [3], enabling the blockchain to support smart contract. Smart contract can be automatically and correctly executed by a network of mutually distrusting nodes without the need of an external trusted authority [4]. It greatly reduces the cost of implementing protocols and improves the security and transparency of protocols, and thus make it easier for users to control the digital assets. However, the code of the smart contract is immutable once deployed onto the blockchain. Therefore, to ensure the quality of the smart contract, as well as the blockchain system, it must be thoroughly tested [5].

Mutation testing is a fault-based software testing technique that can effectively evaluate the adequacy of test cases [6]. Its goal is to help testers identify the weaknesses of the test data and locate the code that is rarely (or never) covered during the execution. Mutation testing has been extensively studied for more than 30 years for various applications (such as test generation [7], fault localization [8]–[10], and program repair [11], [12]), contributing a range of tools such as PIT [13] and MuCPP [14]. However, to the best of our knowledge, there is no research work concerns the ESC mutation testing.

In this paper, we present MuSC, a robust and easy-to-use mutation testing tool for ESC. It performs mutation operations efficiently and precisely at the AST (Abstract Syntax Tree) level and supports user-defined creation of testnet to meet the needs of different smart contract developer. The major contributions of this paper are: it is the first mutation testing tool for ESC, and it implements a series of mutation operators specific for the most commonly used ESC development language, Solidity. All these make MuSC an ideal testing tool for ESC developers.

II. Mutation Testing for ESC

A. ESC Testing

ESC is transaction-driven program [5]. Ethereum users rely on accounts to carry out transactions. Ethereum contains two types of accounts, External Account (EA) and Contract Account (CA): the former is controlled by the users with private keys, and the latter is controlled by the ESC code. A complete ESC testing includes the creation of a CA and a series of transactions between EAs and the CA. An Ethereum transaction can be denoted by a quadruple $tx = \langle from, to, value, data \rangle$, where $from$ and to denote the addresses of sender's account and receiving account respectively, and $value$ denotes the number of Wei to be transferred to to. tx is an CA creation transaction when $to = \varnothing$. At this time, $data$ denotes SCUD. Otherwise (i.e., $to \neq \varnothing$), if to is a CA, $data$ denotes the input to to's corresponding ESC code. ESC deployment can be deemed as the process of CA creation.

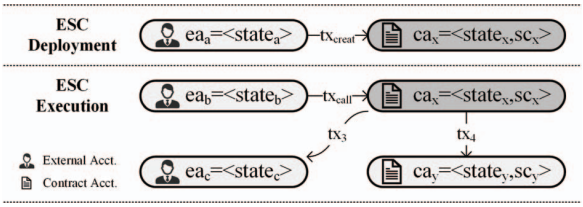

Fig. 1: ESC Deployment and Execution

Fig. 1 depicts deploying and executing the ESC sc_x:

(1) ESC Deployment. EA ea_a launches a CA creation transaction $tx_{create} = \langle addr_a, \varnothing, value_1, sc_x \rangle$. Once tx_{create} is packed into a block and the block is persisted in Ethereum, CA ca_x creation and sc_x deployment are finished.

(2) ESC Execution. EA ea_b launches a message call transaction $tx_{call} = \langle addr_b, addr_x, value_2, input_x \rangle$. Once tx_{call} is packed into a block and the block is persisted in Ethereum, sc_x takes $input_x$ as input and starts to run. In this example, executing sc_x results in two additional transactions, tx_3 and tx_4.

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00136

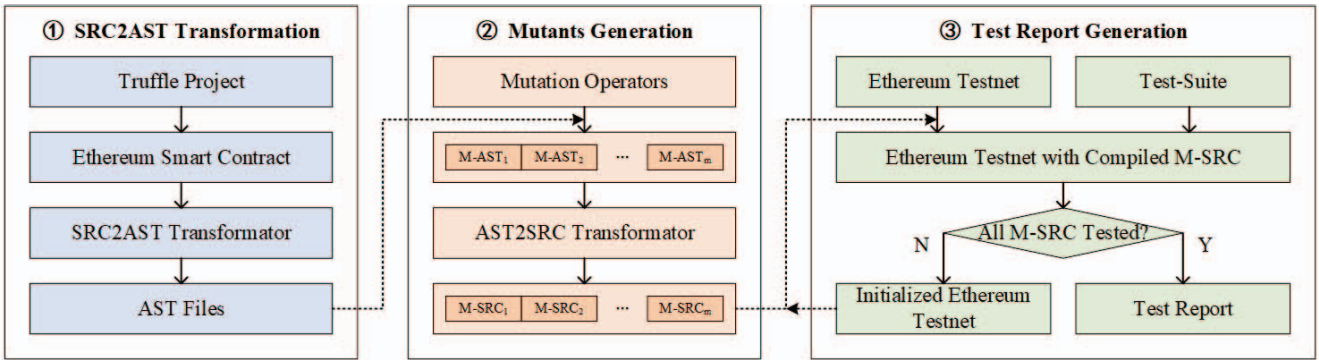

Fig. 2: Overall Flow of MuSC

B. ESC Mutation Operators

The effectiveness of mutation testing mainly depends on the design of the mutation operator [6]. These so-called mutants are based on well-defined mutation operators that either simulate typical application errors or force a validity test. Mutation testing has been shown to subsume other test criteria by incorporating appropriate mutation operators [15]. Thus, mutation operator designation becomes one and even the most critical task in mutation testing tool development.

TABLE I: Traditional mutation operators

Operator	Description
AOR	Arithmetic Operator Replacement
AOI	Arithmetic Operator Insertion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
LOR	Logical Operator Replacement
ASR	Assignment Short-cut Operator Replacement
SDL	Statement Deletion
RVR	Return Value Replacement
CSC	Condition Statement Change

Since the language structure of Solidity is influenced by JavaScript and is similar in syntax [16], we studied the existing mutation operators for JavaScript [17]. Table I presents the traditional mutation operators that can be used for ESC mutation testing. AOR can be further divided into two kinds of mutation operators, AOR_B (Binary Arithmetic Operator Replacement) and AOR_S (Short-cut Arithmetic Operator Replacement). AOR_B replaces basic binary arithmetic operators ($+$, $-$, $*$, $/$, and $\%$) with other binary arithmetic operators, and AOR_S replaces short-cut arithmetic operators (op$++$, $++$op, op$--$, and $--$op). ROR replaces relational operators ($>$, $>=$, $<$, $<=$, $==$, and $!=$) with other relational operators or replaces the entire predicate with *true* and *false*. COR replaces binary conditional operators (&&, ||, &, |) with other binary conditional operators. ASR replaces short-cut assignment operators ($+=$, $*=$, $/=$, $\%=$, and $\&=$) with other short-cut assignment operators. SDL can delete an executable statement by commenting it out, and CSC can force the statement in a conditional judgment to be *true* or *false*.

Specially, to improve the effectiveness of mutation testing for ESC, we also defined a set of new mutation operators according to the characteristics of ESC such as keyword, global variable and function, variable unit and error handling. To achieve this, We read the Solidity documentation and investigated the issues related to smart contracts on GitHub and Stack Exchange to design the mutation operators for the specific defects that may be generated by smart contracts. As depicted in Table II, MuSC implements 15 ESC specific mutation operators, including six related to ESC keyword (FSC, FVC, DLR, VTR, PKD, DKD), three related to ESC global variable/function (GVC, MFR, AVR), two related to ESC variable unit (EUR, TUR), and four related to error handling in ESC (RSD, RSC, ASD, ASC). For each ESC specific mutation operator, we detail it in [18].

TABLE II: ESC Specific mutation operators

Type	Operator	Description
Keyword	FSC	Function State Keyword Change
	FVC	Function Visibility Keyword Change
	DLR	Data Location Keyword Replacement
	VTR	Variable Type Keyword Replacement
	PKD	Payable Keyword Deletion
	DKD	Delete Keyword Deletion
Global Variable and Function	GVC	Global Variable Change
	MFR	Mathematical Functions Replacement
	AVR	Address Variable Replacement
Variable Unit	EUR	Ether Unit Replacement
	TUR	Time Unit Replacement
Error Handling	RSD	Require Statement Deletion
	RSC	Require Statement Change
	ASD	Assert Statement Deletion
	ASC	Assert Statement Change

III. Implementation Details

For each Smart Contract Under Test (SCUT), MuSC first transforms its source files to AST version, and then performs mutant generation on it. Next, these generated AST mutants are transformed back into source files for compilation, execution and testing. Since smart contract testing requires deployment, MuSC supports user-defined testnet or uses a default testnet. MuSC can automatically deploy mutants and execute test cases. Finally, a detailed test report is generated when all mutants have been tested. As Fig. 2 depicted, the whole process can be divided into the following three phases.

A. SRC2AST Transformation

MuSC accepts a Truffle project [19] with its contracts and test suite as input. The smart contracts in *contracts* folder

under the project path are SCUT candidates. Users should choose the and the mutation operators to use. They are allowed to use all mutation operators or a subset of them to generate mutants. The source files of the selected smart contracts are then transformed into the AST format. Redundant statements in source files, such as comments and blank lines and spaces may affect the accuracy of mutations. Thus, MuSC delete them, and the transformed results only retain the necessary sentence structure. It is more conducive to the generation of mutants and improve the accuracy of mutants. To ensure the reliability of parsing result, *solidity-parser-antlr* [20], a mature Solidity parser built on top of a robust ANTLR4 grammar, is selected for transforming.

B. Mutants Generation

MuSC conducts mutating on AST formatted SCUT. The outputs of *solidity-parser-antlr* are in JSON format. We implement a Java project to convert JSON content into corresponding Java objects. For each operator, we apply the relevant objects, operators by changing the member objects of the corresponding object. For example, for the FVC mutation operator, we mutate the original content by changing its $visibility$ member variable. Afterward, each AST mutant will be transformed into a source file version, which has the identical semantic with SCUT except being injected with a pre-defined fault. In the restore step, we will call the output method in the object of the top-level class $SourceUnit$ to output the restored Solidity statement. Each model class in the project has an output method to output the restored Solidity statement, which is called recursively from top to bottom, and finally outputs the complete Solidity file.

The form in which the mutants are stored corresponds to the efficiency of the mutation test. Rather than the complete mutant, MuSC stores the line number, type and the code of the mutated line for each mutation operation. As shown in Listing. 1, each line is called a piece of mutation information and corresponds to a mutant. This storage format can effectively reduce the space for storage as well as the time for file reading. Furthermore, during the process of mutants execution, we only need to record each variant information and the corresponding execution results, so that the test results can be conveniently counted. MuSC also provides an interface to display mutants. For example, Fig. 3 shows an ROR mutant generates by MuSC. Both the mutant and the original program are displayed on the interface. MuSC marks the line of mutated code in red to make it easier for testers to locate the mutated place.

C. Test Report Generation

In this phase, MuSC first deploys the smart contract onto the blockchain, and then executes the tests in the *test* directory. ESC runs on Ethereum blockchain, its execution result depends on the state of the blockchain, such as account balance and block height. To avoid the influence of blockchain state, it should be ensured that SCUT and the mutants have the same initial test chain state. To achieve this, MuSC re-deploy all of the migrations at the beginning of every test file to ensure there is a fresh set of contracts to test.

Listing 1: Mutant Storage Format of MuSC

```
//Mutant Format
8 ROR if(makeAddress != takerAddress){
13 ROR if(expiration <= now){
24 ROR if(takerToken != takerAmount){
25 ROR if(msg.value != takerAmount){
37 ROR if(msg.value > 0){
42 ROR if(takerAddress > msg.sender){
55 ROR if(fills[hash] != false){
```

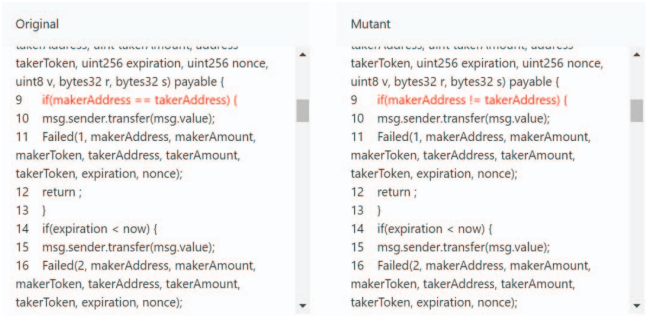

Fig. 3: Example of an ROR Mutant

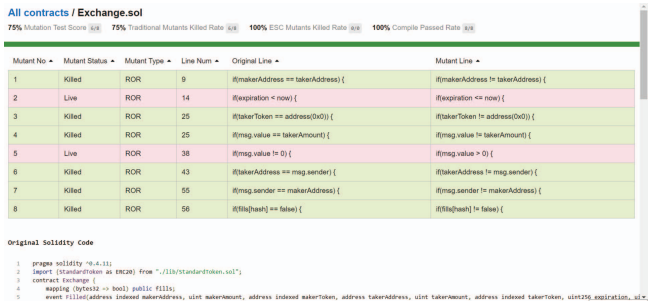

Fig. 4: Example of Test Report

MuSC supports user-defined creation of testnet. Users are allowed to create a customized testnet, as long as the port of the testnet is the same as the configuration in the truffle configuration file under the project folder. During test, mutants that cause compilation errors are discarded immediately and not used in the following testing. For each mutant, we record its executing result (i.e., passed or failed). Finally, a test report will be generated, displaying the total mutation score and the testing result of each mutant. Fig. 4 depicts an example of ESC mutation test report.

IV. Performance Evaluation

To demonstrate the applicability and the performance of MuSC, we applied MuSC to a set of smart contracts in four different real-world Ethereum DApps (i.e., SkinCoin, SmartIdentity, AirSwap and CryptoFin). SkinCoin is a universal cryptocurrency for instant trading skins in games and making bets on e-sports events. SmartIdentity relies on Ethereum blockchain to represent an identity using a smart contract.

TABLE III: Experiment Result

DApp	Traditional Mutants			ESC Mutants			Average Time (s) Per Mutant
	Mutants	Killable	Time(s)	Mutants	Killable	Time(s)	
SkinCoin	74	23	1642	110	21	2371	21.8
SmartIdentity	42	19	1481	33	15	759	29.8
AirSwap	32	18	672	59	15	1321	21.9
Cryptofin	37	12	964	84	60	2806	31.1

CryptoFin is a collection of Solidity libraries, with an initial focus on arrays. AirSwap is a peer-to-peer trading network built on Ethereum. These DApps are selected because each of them not only provides a set of ESCs but also is accompanied by a well-designed test-suite. Thus, we do not need to manually design an ESC test-suite, which is too subjective to generate a fair experimental result and conclusion. Tests that failed on the original DApp are removed from the subject (e.g., we removed 15 test cases from AirSwap's test-suite, which contains 32 test cases in total). Table IV summarizes the characteristics of DApps used in the experiments. For each DApp, its name (Col. 1), lines of code (Col. 2), branches of code (Col. 3), size of test-suite (Col. 4) and websites (Col. 5) are described. All smart contracts have been open sourced on GitHub.

TABLE IV: Experimental Subject

DApp	LOC	BOC	STS	Website
SkinCoin	225	66	35	www.britchicks.com
SmartIdentity	180	34	87	www.deloitte.co.uk/smartid
AirSwap	330	76	17	www.airswap.io
CryptoFin	348	58	44	www.blog.cryptofin.io

Table III presents the experimental results of MuSC on the four projects. We separate the traditional mutation operator and the ESC mutation operator for statistics. In terms of the number of variants, both types of mutation operators generate a considerable number of mutants, indicating that the mutation operators in MuSC are effective. Compared to the test result, which is related to the quality of the test suite, we pay more attention to the efficiency of mutant generation and execution. The generation time of mutants is very short (usually only a few seconds), and the average time for each mutant is almost negligible. This indicates that our tools are very efficient at error injection.

Intuitively, the more the mutants are, the longer the mutation testing takes. As can be seen from Table IV, the average time is between approximately 20 and 30 seconds, which is associated with the number of test cases and the performance of the testnet. However, considering that the quality requirements of smart contracts exceed the general procedures, such execution time is worthwhile.

V. Conclusion and Future Work

This paper presents MuSC, the first mutation testing tool w.r.t Ethereum smart contract. MuSC offers the following major advantages: It provides a graphical user interface that is very convenient to use and a set of novel mutation operators specific for Solidity language; it supports testing smart contracts on user-defined testnet. The basic functions of our mutation tool have been implemented and the source code is open on GitHub. The mutation operators of MuSC are perfected continuously, and some features such as error handling will be improved in future versions.

Acknowledgment

The work is partly supported by the National Natural Science Foundation of China (61802171) and the Jiangsu Planned Projects for Postdoctoral Research Funds (2018K028C).

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.
[2] "Libra white paper," https://libra.org/en-US/white-paper/, 2019.
[3] V. Buterin, "Ethereum white paper: A next-generation smart contract and decentralized application platform," 2014.
[4] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: A survey," *International Journal of Web and Grid Services*, vol. 14, no. 4, pp. 352–375, 2018.
[5] X. Wang, H. Wu, W. Sun, and Y. Zhao, "Towards generating cost-effective test-suite for Ethereum smart contract," in *SANER*, 2019, pp. 549–553.
[6] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. LeTraon, and M. Harman, "Mutation testing advances: An analysis and survey," *Advances in Computers*, vol. 112, no. 2019, pp. 275–378, 2019.
[7] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. De Halleux, and H. Mei, "Test generation via dynamic symbolic execution for mutation testing," in *ICSM*, 2010, pp. 1–10.
[8] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," in *OOPSLA*, 2013, pp. 765–784.
[9] M. Papadakis and Y. Le Traon, "Metallaxis-FL: Mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.
[10] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," in *OOPSLA*, 2017, pp. 92:1–92:30.
[11] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *ISSTA*, 2019, pp. 19–30.
[12] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *ICST*, 2010, pp. 65–74.
[13] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: A practical mutation testing tool for java," in *ISSTA*, 2016, pp. 449–452.
[14] P. Delgado-Pérez, I. Medina-Bulo, F. Palomo-Lozano, A. García-Domínguez, and J. J. Domínguez-Jiménez, "Assessment of class mutation operators for C++ with the MuCPP mutation system," *Information and Software Technology*, vol. 81, no. 2017, pp. 169–184, 2017.
[15] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *FSE*, 2014, pp. 654–665.
[16] "Solidity docs," https://solidity.readthedocs.io/en/v0.5.8/, 2019.
[17] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Efficient javascript mutation testing," in *ICST*, 2013, pp. 74–83.
[18] H. Wu, X. Wang, J. Xu, W. Zou, L. Zhang, and Z. Chen, "Mutation testing for ethereum smart contract," *arXiv:1908.03707*, 2019.
[19] "Truffle, a development environment, testing framework and asset pipeline for Ethereum," https://www.trufflesuite.com/truffle, 2019.
[20] "solidity-parser-antlr, a solidity parser built on top of a robust antlr4 grammar," https://github.com/federicobond/solidity-parser-antlr, 2019.