Concolic Testing on Smart Contracts

by

Emirhan Saraç

Submitted to the Department of Computer
Engineering in partial fulfillment of
the requirements for the degree of
Bachelor of Science

Undergraduate Program in Computer Engineering
Boğaziçi University

Fall 2019

Concolic Testing on Smart Contracts

APPROVED BY:

Prof. Alper Şen .....
(Project Supervisor)

DATE OF APPROVAL: . .2020

ACKNOWLEDGEMENTS

I am very appreciate to Prof. Alper Şen for his mentorship and ideas about the project. By the help of him, I have improved myself about verification and blockchain and it continues. Without his mentorship, I would be lost in this complex area.

ABSTRACT

Concolic Testing on Smart Contracts

Recent attacks on Ethereum smart contracts made people to reconsider security issues about it. Billions of dollars has been lost and cryptocurrency idea has been shaken. Those events attracted people to think about solutions for this environment.

In my project, I will focus on these security issues by using verification methods. This issue based on Solidity code writers and their lack of awareness about Ethereum environement. After reading other verification tools, one can understand after looking statistics of these results, those tools are far from complete to gain full trust from users. Tools, in general, works on both Solidity code and EVM bytecode. It is harder to work on EVM bytecode but it is impossible to complete verification without understanding underlying code flow. Even if we understand underlying stuffs, finding Solidity codes that resides on EVM ledger are not easy. Therefore, we will mostly work on EVM byte code as well.

I have focused on understanding how other tools are doing verification and what are the methods of doing it. Our goal is to find a different approach to this issue and achieve better results. I have examined some tools such as Securify, ContractFuzzer, and Oyente. Then we have seen there is no concolic testing for smart contract which has a different approach from other tools. Concolic testing is hybrid software verification method. It uses symbolic executions method along concrete values for better code coverages. We will cover most common problems such as transaction-ordering dependence, timestamp dependence, mishandled exceptions, and reentrancy vulnerability in our project.

ÖZET

Akıllı Sözleşmede Concolic Test

Son zamanlarda ki Ethereum akıllı sözleşmeleri üzerindeki saldırılar, insanların bu konudaki güvenlik çekincelerini tekrardan düşünmeye başlattı. Milyarlarca dolar kaybedildi ve kripto para düşüncesi sarsıldı. Bu olaylar insanların dikkatini çekti ve bu çevrede çözümleri üzerinde düşünülmeye başlandı.

Projemde doğrulama yöntemlerini kullanarak bu güvenlik konularına odaklanacağım. Bu sorunun asıl nedeni Solidity dilini yazanlar ve onların Ethereum sistemi hakkındaki bilgi eksiklikleri. Diğer doğrulama araçlarını okuduktan ve onların sonuç istatistiklerine baktıktan sonra, bu araçların kullanıcıların güvenini tamamen kazanmaya yakın olmadığı anlaşılıyor. Bu araçlar genelde hem Solidity kodunda ve EVM bit kodunda da çalışırlar. EVM bit kodunda çalışmak zordur ama alt tarafta kodda dönen işleri anlamadan da doğrulama yöntemini yapmak imkansızdır. Altta dönen işleri anlasak bile EVM ortamında olan Solidity dilinde ki kodu bulmak kolay değildir. Bu yüzden, biz de genelde EVM bit kodu üzerinde çalışacağız.

Bu zamana kadar odağımda diğer araçların doğrulamayı nasıl yaptığı ve yapış yöntemlerinin neler olduğunu anlamak oldu. Bizim amacımız diğer araçlardan farklı bir yöntem kullanmak ve daha iyi sonuçlar almak. Bazı araçları inceledim ve bunlar Securify, ContractFuzzer, ve Oyente. Daha sonra fark ettik ki akıllı sözleşmelerde concolic test gibi farklı bir yaklaşım yapan bir software tool yok. Concolic test etmek melez bir yazılım doğrulama yöntemidir. Bu araç daha iyi kod kapsaması için somut değerlerle birlikte sembolik uygulama yöntemi kullanır. Bizim projemizde en önemli sorunları çözeceğiz işlem-siparişi bağlılığı, zaman damgası bağlılığı, kötü halledilmiş istisnalar, ve evrensel güvensizlik gibi.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

a_{ij}	Description of a_{ij}
$\mathbf{A}$	State transition matrix of a hidden Markov model
α	Blending parameter *or* scale
$\beta_t(i)$	Backward variable
Θ	Parameter set

LIST OF ACRONYMS/ABBREVIATIONS

2D	Two Dimensional
3D	Three Dimensional
AAM	Active Appearance Model
ASM	Active Shape Model

1. INTRODUCTION AND MOTIVATION

Last 10 years, power of blockchain environment has increased significantly. Ethereum is one of the blockchain-based distrubuted computing platform. Ethereum gets its power from smart contracts, execution of arbitrary programs, that interact without relying on trusted third parties. These mutually entrusted entities, smart contracts, can carry many cryptocurrencies worth of hundreds of dollars and have many security issues, because people do not understand overall execution of this environment. Another problem is that unlike traditional distributed environment Ethereum operates in open networks with many arbitrary participants can join. Therefore, adversaries can manipulate other smart contracts if not properly implemented. Another problem is deploying smart contracts into blockchain server is almost permanent. Updating existing smart contract on blockchain requires huge cost. Thus, smart contracts require verification, especially statically, before they are deployed to blockchain service.

In this project, we will work on the concolic testing of smart contracts. Especially on Solidity language, JavaScript like language, which is a language for writing smart contracts. Since security issues of smart contracts still has not received enough attention. Therefore, I have found it interesting to work on it. There are several verification tools for smart contracts, but none of them has reached their goal on this topic. There has been attack on some smart contracts which is called "Frozen funds" resulted on worth of 280 millions of USD. We all know after some point, crypocurrencies will be in our daily life and we should provide a secure environment for it. I have read many articles about verification tools for it. Some notable tools are symbolically analyzers, fuzzers, and static symbolic execution tools. We want to come up with another idea for verification which is concolic testing. Up to this point, my work on this topic generally was reading articles and understanding verification methods on smart contracts. Therefore, I have no concrete results and short codepieces for concolic testing.

2. STATE OF THE ART

2.0.1. Securify

Securify [1] is a security analyzer for Ethereum smart contracts that has a high precision on contract behavior safe/unsafe decision with respect to a give property. It symbolically analyzes contract's dependency graph and extracts semantic for it. It has a domain specific language for conditions. For each bug that might lead to problems, it has two patterns such as compliance and violation patterns. It decreases false negative and false positive warnings, warnings require manual check, for some type of attack types. It's decompiler from EVM byte code to EVM assmebly can be used in our project. It provides good explanations for some attack types and their detection methods that can also be useful. Article also mentions about semantics of EVM which will be very useful for my future work. Drawbacks of the project are still requires manual checks, requires professionals to write patterns for detection of future attack types since domain specific language created for it, assumes generic structures for contracts so that it does not work on all smart contracts.

2.0.2. ContractFuzzer

ContractFuzzer [2] is a fuzzer for testing Ethereum smart contracts. It generates millions of random inputs for testing. Not all are random, especially address locations, requires dynamic execution for linking. It creates pools for understanding which contract can interact with. It also uses test oracle which is used for guessing about the result of test. Since we are randomly generating inputs, we should know the results for each one. For each attack type, results of the test examined and result is given. Its main goal is reducing false negative and false positive cases. Pooling method will be useful for my project, since I need random inputs. Drawbacks are it requires at least 15 minutes for initial logs that makes is undesirable and doesn't perform well at each attack type.

2.0.3. Oyente

Oyente [3] is a symbolic execution tool for smart contracts. It creates CFG for given contract and then creates basic blocks, which means one way in - one way out, then starts execution. Inputs are executed as symbol and for each branch, program divides itself for each path and adds conditions to their path. After execution is completed, it checks path is possible or not. Our future goal is similar to this, so we will get help from its symbolic execution implementation ,how it creates basic blocks, and how it models them. It uses Z3 to check path validation, we might use the same as well. It is the one of the very first verification tools for EVM so it does not cover many of the attacks.

2.0.4. LLVMVF: A Generic Approach for Verification of Multicore Software

It is a novel framework for LLVM Verification Framework(LLVMVF) which is implemented in purely functional language for verification of multicore systems [4]. In this project, different from others, it uses LLVM to verify program. It converts bit code to LLVM and then creates well-defined model(LLVM IR). Then uses SMT model checker to verify it.

2.0.5. Symbolic Execution Techniques

In this article [5], it defines all of the symbolic execution methods and their basic engine methodologies. It also mentions design principles of problematic cases like memory modeling, path selection, interaction with environment, and path explosion. It explains the basic engine behavior and also mentions concolic testing which is a different approach of symbolic execution. Then it mentions advantages and disadvantages of different approaches. It mentions tools that are required to complete symbolic execution such as constraint solvers like Z3.

2.0.6. Heuristics for Concolic Software Testing

It is a master thesis of Yavuz Köroğlu. It also mentions steps of concolic testing. However, it focuses on better performance of constraint solvers since it is the bottleneck of concolic testing. Strategy is basically increasing constraints but keeping them as small which is called Incremental Partial Path Constraints (IPPC). It uses this technique on CREST which is a concolic testing for C language.

3. METHODS

First we need a EVM disassembler to retrieve EVM opcodes from EVM byte code. I have used disassembler of Oyente but since solidity version has changed too much and there are some problems on getting right program counters(PC), I have implemented my own disassembler and calculated correct PC's by comparing with online tools. Even if they are not completely same, it is enough for now.

Then I have created basic blocks from opcodes. Basic blocks are one way to in and out parts of the code part. However, since our basic concolic engine is not working, we cannot obtain all the basic blocks which are run-time basic blocks. We are currently obtaining static basic blocks. Those blocks will be used to understand branching or terminating points. To do that, I have studied on opcodes of the EVM, since I should know which opcodes point basic block end point.

Then I have worked on finding beginning and end of function program counters because we will use concolic engine on each function and results will be generated from there. After completing this phase, I have started to design, how I am going to represent Stack, Memory, and variables in the simulation. Also, those design has been done for symbolic execution part. Then, I have determined what should I keep in my hand in order to handle repeatable executions in the Global State. Stack has been represented as array, Memory has represented as GrowingList, and Storage represented as dictionary in Python.

After completing design, I have started to write concrete execution simulation for all OPCODES of Solidity language. There are over 100 opcodes and some of them are impossible to simulate since they are not even explained in the Ethereum yellow paper. After completing concrete execution simulation, I have started to to write symbolic execution simulation for all OPCODES. However, not all opcodes can be simulated with symbolic execution since z3, which is solver that we are using in our project, cannot solve complex instructions and it take too much time to find solutions. First,

z3 cannot handle operations like exponential such as $a^b = 22$. Second, solidity language uses 256 bit values for all values even if it is defined as int64, int8, even boolean and that makes z3 operations take too much time.

Then, I have researched about initial conditions and predefined values for Solidity codes. Such as accessing a non-defined storage point should return 0 instead of error or address of the contract and more. I have created them and give a appropriate initial values to them. Then, it was time to simulate all functions. I have given maximum value for each parameter for initial conditions and run the simulation for corresponding function. After each execution trace, we should run the function if there is any non covered branch and branch information is collected during all execution traces.
Currently my program detects three common bugs.

TIME DEPENDENCE BUG

If critical operation, like sending an ether, done by user has a dependence on time stamp value of the contract, program raises TIME DEPENDENCE BUG. The reason for this bug is time stamp of the contract can be easily changeable and malicious users can use this for their favor.

BLOCK NUMBER DEPENDENCE BUG

If critical operation, like sending an ether, done by user has a dependence on current block number value of contract, program raises BLOCK NUMBER DEPENDENCE BUG. The reason for this bug is block number of the contract can be easily changeable and malicious users can use this for their favor.

TRANSACTION-ORDERING DEPENDENCE(TOD) BUG

If critical operation, like sending an ether, done by user has a dependence on the static variables that can be changed from other functions, program raises TOD BUG. The reason for this bug is with simple example: If user has function that sends ether to caller if he finds the correct result of the puzzle and prize of the correct result is a static variable defined in the contract. If there is any other function that changes this value, it can be used for lowering prize value even setting as 0.

4. RESULTS

```
function coverage(int256 par1, int256 par2, int256 par3)returns (uint256){
    if(par1 == 12) {
        if(par1 < 10) {
            return 0;
        }else if(par1 - par2 == 4) {
            return 1;
        }else if(par1 + par2 >= 44) {
            return 2;
        }else {
            return 3;
        }
    }else {
        if(par3*par2 == 500){
            return 4;
        }else {
            return 5;
        }
    }
}
```

Figure 4.1. Example of Branch Coverage

```
68 <-> 3922603777 (0xe9ce2f01) class starts at pc  0
{3922603777: ['int256', 'int256', 'int256']}
FUNCTION 0xe9ce2f01 will be tested
EXECUTION PARS --> 0xe9ce2f01111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111
[parameter_3 = 1,
 parameter_2 = 500,
 parameter_1 = 115792089237316195423570985008687907853269984665640564039457584007913129639923]
EXECUTION PARS --> 0xe9ce2f01fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff300000000000000000000000000000000000000000000000000000000000001f40000000000000000
000000000000000000000000000000000000000000000001
[parameter_1 = 12]
EXECUTION PARS --> 0xe9ce2f01000000000000000000000000000000000000000000000000000000000000000c11111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111
[parameter_2 = 57896044619500596045133949997927298148104355791371442782933184893991052640252,
 parameter_1 = 12]
EXECUTION PARS --> 0xe9ce2f01000000000000000000000000000000000000000000000000000000000000000c8000000007fffffffffffffffffffffffffffffffffffffffffffffffffffffc1111111111111111
111111111111111111111111111111111111111111111111
[parameter_1 = 12, parameter_2 = 8]
EXECUTION PARS --> 0xe9ce2f01000000000000000000000000000000000000000000000000000000000000000c00000000000000000000000000000000000000000000000000000000000000081111111111111111
111111111111111111111111111111111111111111111111
UNSATISFIED TRACE DETECTED
```

Figure 4.2. Execution Result

Above there are 6 branches to cover, but one of them is logically impossible and in the results it has been stated. (EXECUTION PARS is the hex value of calling function both function id and parameters are included.)

```
function timestamp(address getter, uint256 c){
    uint256 temp = now + c;  // Now is the timestamp of the contract
    if(temp > 100000) {
        getter.call.value(20).gas(10)(abi.encodeWithSignature("register(string)", "MyName"));
    }else {
        getter.call.value(10).gas(10)(abi.encodeWithSignature("register(string)", "MyName"));
    }
}
```

Figure 4.3. Timestamp Bug Example

```
70 <-> 3114812229 (0xb9a84345) class starts at pc  0
{3114812229: ['uint160', 'int256']}
FUNCTION 0xb9a84345 will be tested
EXECUTION PARS --> 0xb9a8434511111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
[IH_TIMESTAMP = 34736, parameter_2 = 65280, my_balance = 19]
EXECUTION PARS --> 0xb9a843451111111111111111111111111111111111111111111111111111111111111111000000000000000000000000000000000000000000000000000000000000ff00
[IH_TIMESTAMP = 34464, parameter_2 = 65536]
EXECUTION PARS --> 0xb9a84345111111111111111111111111111111111111111111111111111111111111111100000000000000000000000000000000000000000000000000000000000010000
[IH_TIMESTAMP = 33442, parameter_2 = 65536, my_balance = 9]
EXECUTION PARS --> 0xb9a84345111111111111111111111111111111111111111111111111111111111111111100000000000000000000000000000000000000000000000000000000000010000


[{'function': '0xb9a84345', 'warning': 'TIMESTAMP DEPENDENCY BUG'}]
```

Figure 4.4. Timestamp Bug Result

```
function block_number(address getter, uint256 c){

    uint256 temp = block.number;  // block.number is the BLOCK NUMBER OF THE CONTRACT
    if(temp > 100000) {
        getter.call.value(123).gas(10)(abi.encodeWithSignature("register(string)", "MyName"));
    }else {
        getter.call.value(321).gas(10)(abi.encodeWithSignature("register(string)", "MyName"));
    }
}
```

Figure 4.5. Block Number Bug Example

```
70 <-> 829492797 (0x31710e3d) class starts at pc  0
{829492797: ['uint160', 'int256']}
FUNCTION 0x31710e3d will be tested
EXECUTION PARS --> 0x31710e3d11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
[IH_NUMBER = 0, my_balance = 320]
EXECUTION PARS --> 0x31710e3d11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
[IH_NUMBER = 100032]
EXECUTION PARS --> 0x31710e3d11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
[IH_NUMBER = 100032, my_balance = 122]
EXECUTION PARS --> 0x31710e3d11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111


[{'function': '0x31710e3d', 'warning': 'BLOCKNUMBER DEPENDENCY BUG'}]
```

Figure 4.6. Block Number Bug Result

```
uint256 prize = 50;

function setPrize(uint256 pp) {
    prize = pp;
}

function opp(address getter, int256 solution) {
    if(solution == 44) {
        getter.send(prize);
    }else {
        getter.send(prize);
    }

}
```

Figure 4.7. Transaction-Ordering Dependence Bug Example

```
81 <-> 1822946234 (0x6ca7f3ba) class starts at pc  0
126 <-> 2652195096 (0x9e154918) class starts at pc  0
{1822946234: ['int256'], 2652195096: ['uint160', 'int256']}
FUNCTION 0x6ca7f3ba will be tested
EXECUTION PARS --> 0x6ca7f3ba1111111111111111111111111111111111111111111111111111111111111111


FUNCTION 0x9e154918 will be tested
EXECUTION PARS --> 0x9e15491811111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
[my_balance = 0,
 STORAGE_0 = 1,
 parameter_2 = 115792089237316195423570985008687907853269984665640564039457584007913129639891]
EXECUTION PARS --> 0x9e1549181111111111111111111111111111111111111111111111111111111111111111ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffd3
[parameter_2 = 44]
EXECUTION PARS --> 0x9e1549181111111111111111111111111111111111111111111111111111111111111111000000000000000000000000000000000000000000000000000000000000002c
[my_balance = 0, STORAGE_0 = 1, parameter_2 = 44]
EXECUTION PARS --> 0x9e1549181111111111111111111111111111111111111111111111111111111111111111000000000000000000000000000000000000000000000000000000000000002c


[{'function': '0x9e154918', 'warning': 'TOD DEPENDENCY BUG with STORAGE_0'}]
```

Figure 4.8. Transaction-Ordering Dependence Result

5. CONCLUSION AND DISCUSSION

First, I want to compare my project with other tools for Solidity. There are two main types of testing for Solidity, symbolic and concrete. Oyente uses fully symbolic testing and program runs slow and cannot handle difficult execution since all system relies on the Z3 solver. Moreover, Z3 cannot solve all operations in the Solidity opcodes. The other one is ContractFuzzer, which uses only concrete execution so there is no sophisticated approach. It is basically randomly running functions millions of times, called Random Bomb. Here it may miss lots of branches but it is fast since there is no usage of Z3 in ContractFuzzer. However, in my project I am both using sophisticated approach of Oyente by using Z3 and concrete approach of ContractFuzzer. Therefore, it can be called sophisticated approach of ContractFuzzer with eased Z3 weight on Oyente.

6. FUTURE WORK

Currently, my project only detect the bugs. I am thinking to detect two more bugs such as Reentrancy and Mishandled Exception bug. After that, I can go into more rare bug solutions or I can improve my project so that it can handle infinite loops.

I expect to forge a another powerful tool for verification of smart contracts. It might be a notable tool if it is handled well. Since there is no closely perfect tool for smart contracts, I have been always excited about it. It looks like for me that it is a point in the dark that nobody has ever noticed but the solution is that. Even this idea makes me encouraging about this project.

REFERENCES

1. Tsankov, P., A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli and M. T. Vechev, "Securify: Practical Security Analysis of Smart Contracts", *CoRR*, Vol. abs/1806.01143, 2018, http://arxiv.org/abs/1806.01143.

2. Jiang, B., Y. Liu and W. K. Chan, "ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection", *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pp. 259–269, ACM, New York, NY, USA, 2018, http://doi.acm.org/10.1145/3238147.3238177.

3. Luu, L., D.-H. Chu, H. Olickel, P. Saxena and A. Hobor, "Making Smart Contracts Smarter", pp. 254–269, 10 2016.

4. Sousa, M. and A. Sen, "LLVMVF: A Generic Approach for Verification of Multicore Software", *J. Electron. Test.*, Vol. 29, No. 5, pp. 635–646, Oct. 2013, http://dx.doi.org/10.1007/s10836-013-5405-9.

5. Baldoni, R., E. Coppa, D. C. D’elia, C. Demetrescu and I. Finocchi, "A Survey of Symbolic Execution Techniques", *ACM Comput. Surv.*, Vol. 51, No. 3, pp. 50:1–50:39, May 2018, http://doi.acm.org/10.1145/3182657.

APPENDIX A: DATA AVAILABILITY STATEMENT

- Describe how your data can be accessed by others in this appendix. You can look at the following pages to see examples of data availability statements:

https://www.springernature.com/gp/authors/research-data-policy/data-availability-statements/12330880
https://academic.oup.com/brain/pages/data_availability

APPENDIX B: STANDARDS, LAWS, REGULATIONS AND DIRECTIVES

- If you have used or referred to any standards, laws, regulations and/or directives or did work related to standardization, list and describe them briefly in this appendix. Examples: GDPR, WiFi (802.11a/g).