8-2020

Software Quality Control through Formal Method

Jialiang Chang
Western Michigan University, aiolos404@gmail.com

SOFTWARE QUALITY CONTROL THROUGH FORMAL METHOD

by

Jialiang Chang

A dissertation submitted to the Graduate College
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
Computer Science
Western Michigan University
August 2020

Doctor Committee:

Zijiang Yang, Ph.D., Chair
Steve Carr, Ph.D.
Jun Sun, Ph.D.

SOFTWARE QUALITY CONTROL THROUGH FORMAL METHOD

Jialiang Chang, Ph.D.
Western Michigan University, 2020

With the improvement of theories in the software industry, software quality is becoming the most significant part of the procedure of software development. Due to the implicit and explicit vulnerabilities inside the software, software quality control has caught more researchers and engineers' attention and interest.

Current research on software quality control and verification are involving various manual and automated testing methods, which can be categorized into static analysis and dynamic analysis. However, both of them have their own disadvantages. With static analysis methods, inputs will not be taken into consideration because the software system isn't executed so we do not know the behavior of the concurrent software system, while dynamic analysis methods would be extremely expensive because of the enumeration of inputs and paths/interleaving states of the concurrent program. In this research, the formal method brings rigorous mathematical proof into the software quality control, to verify that the requirements for the system being developed have been completely and accurately specified.

This study focuses on the application of the formal method, associated with other software verification method, including symbolic execution, fuzzing, and static analysis methods, to verify the software vulnerabilities and security issues which negatively impact the software quality. For the purpose of presenting the benefits of the utilization of formal methods in software quality control, this study summarizes the research results on the

different software platforms, including memory reuse distance measurement and blockchain smart contracts security verification on Ethereum and Hyperledger Fabric.

ACKNOWLEDGMENTS

At this time approaching to the end of long journey, I truly feel grateful and lucky that there are my family, my teachers and my friends who selflessly show their supports and love during this 5 year long chapter. At the last chapter of this fruitful book, I wrote down every letter in this dissertation not only to complete my Ph.D. program, but also to show my appreciation to all of you. I cannot make this far without you.

I would like to express my special appreciation and thanks to my advisor, Dr. Zijiang Yang, whose invaluable mentorship on both research and my career guided me to become a qualified Ph.D.

I would also extend my thanks to my committee members, Dr. Steve Carr and Dr. Jun Sun for serving as my committee members. I would like to thank you for the research opportunities and advices you gave me in my research projects.

I would also like to thank my parents, Sheng and Kefen, who support me to pursue this academic honor from the beginning of my life, and whose love continuously encourages me through these years.

Special thanks to my wife, Qian, who witness my hardworking, support me without any return and has been with me all these years. The presence of you is the finest of rewards to me.

Contents

1 Introduction **1**

1.1 Background 1

1.2 Formal Methods in Software Quality Control 2

1.3 Research Domains 5

1.4 Dissertation Contributions 6

1.4.1 Memory Distance Measurement for Concurrent Programs 6

1.4.2 sCompile: Critical Path Identification and Analysis for Smart Contracts . . 7

1.4.3 Hyperledger Fabric Chaincode Quality Control with Fuzz Testing 7

2 Memory Distance Measurement for Concurrent Programs **9**

2.1 Introduction 9

2.2 Background: Execution of Concurrent Programs 13

2.3 Memory Distance Measurement Based on Symbolic Execution 15

2.4 Memory Distance Measurement with Random Scheduling 15

2.4.1 PCT Algorithm 17

2.4.2 Measure Memory Distance with Random Scheduling 18

2.4.3 Probabilistic Guarantee Inheritance 19

2.5 Experiments 20

2.5.1 Implementation 20

2.5.2 Comparison between DisConPro and DisConSym on Small Programs . . . 20

2.5.3 DisConPro on Public Benchmarks 22

2.6 Related Works 25
2.7 Conclusion 27

3 sCompile: Critical Path Identification and Analysis for Smart Contracts 28
3.1 Introduction 28
3.2 Illustrative Examples 31
3.3 Approach 33
3.3.1 Constructing CFG 33
3.3.2 Identifying Monetary Paths 36
3.3.3 Identifying Property-Violating Paths 37
3.3.4 Ranking Program Paths 40
3.3.5 Feasibility Checking 41
3.4 Implementation and Evaluation 42
3.4.1 Implementation 42
3.4.2 Experiment 42
3.5 Related Works 49
3.6 Conclusion 50

4 Hyperledger Fabric Chaincode Quality Control with Fuzz Testing 51
4.1 Introduction 51
4.2 Motivating Example 54
4.2.1 Chaincode 54
4.2.2 Motivation Example 55
4.3 Fuzzing Test Approach 60
4.3.1 Fuzzing Policy Generation 60
4.3.2 CFG Generation 61
4.3.3 Instrumentation 62
4.3.4 Feedback Guided Fuzzing Mechanism 63
4.3.5 Input Mutation 68

4.4 Warning Oracle 70
4.5 Experiment and Evaluation 73
4.6 Related Works 78
4.7 Conclusion 83

5 Conclusion and Future Works **84**

Bibliography **86**

List of Tables

2.1 Memory reference of a program execution 10
2.2 Motivating example 11
2.3 Memory distance results in different interleavings 11
2.4 Four threads with eight memory accesses. 17
2.5 Impact of the number of global variables comparing with DisConSym and DisConPro 21
2.6 Tracked paths and time cost result for DisConSym and DisConPro 22

3.1 Definition of α_{pr} 40
3.2 Loop bound definitions among three tools 43
3.3 Comparison on vulnerable contracts 44
3.4 Average number of program paths 44
3.5 Statistics and results of surveyed contracts 48

4.1 Minimization of candidate input 65
4.2 Mutation of candidate input 68
4.3 List of security warnings 70
4.4 Analysis report on evaluated chaincodes 75
4.5 Crash result 77
4.6 Warning result 78

List of Figures

1.1 Symbolic execution example code 3

2.1 Code snippet of a concurrent program and its generalized interleaving graph(GIG). 14
2.2 Parsec results 23
2.3 MySQL result 24

3.1 Illustrative contracts 32
3.2 Overall workflow of sCompile 33
3.3 Control flow graph of the contract *toyDAO* 35
3.4 Guardless suicide 38
3.5 A non-greedy contract 41
3.6 Execution time of sCompile vs. Oyente vs. MAIAN 44
3.7 Ambiguous cases between sCompile and MAIAN 46
3.8 Contract of owner change 47

4.1 Chaincode necessary interfaces 54
4.2 Motivating example 56
4.3 Policy file for Motivating example 77
4.4 Control flow graph of chaincode in Fig. 4.2 78
4.5 Snippet of instrumented chaincode in Fig. 4.2 79
4.6 inputs that can call private functions 79
4.7 difference between statical and dynamical CFG generation 79
4.8 Example for sonar instrumentation 79

4.9 Vulnerabilities explaination example 80
4.10 Overall workflow of afuzzer 81
4.11 Chaincode coverage rate 81

Chapter 1

Introduction

Software quality[78] is a commonly applied standard that applied in description of the degree of matching between the implementation of software and its initial customer requirement. The definitions of software quality can be illustrated from different perspectives, including correctness, robustness, security and usability. In this dissertation, software quality is representing the correctness and security of the software, which is the main features that studied in software verification research area.

1.1 Background

Software quality has been a huge concern in many research and industry areas. Many accidents that related to software quality, which could-be-avoided, have happened. In order to verify its correctness and security, software testing method is introduced conduct examinations on quality of software, which can significantly improve the software quality. There are some fatal cases among those casesthat happened due to vulnerabilities of software:

- A female driver got killed in a car accident caused by falure of autopilot function in 2018[11].
- The fatal overdosed radiation exposure to patients by Therac-25 medical equipment. It was involved in at least six accidents between 1985 and 1987, in which patients were given massive overdoses of radiation[14].

- A notorious bug called reentrancy attack happened on Ethereum blockchain smart contract, which caused 60 million us dollars financial loss[12].

Aforementioned and many other security issues that dangers software quality in terms of its correctness and security. The best practical methods that can control and improve the software quality is testing, which has been considered as an essential step of the software life cycle[77]

When the software testing[72] is introduced for the first time, all of the methods are conducted manually to control the software quality. In manual testing, people have to create specific test cases for each software, which consumes lot of time and labors but cannot check all runtime possibilities of software exhaustively. With progressive development of research and theory, automated software testing[54] is introduced to assist manual testing with automated methods and theories. Automatic testing tried to replace the traditional manual testing but yet completely obsolete it at all. This is because automated testing cannot prove all detected vulnerabilities are positively true and its system is extremely expensive to design and implement due to highly requirement of professional experiences. Formal method[35] is one of highly specialized mathematical based method that applied on verifying the correctness and security of the software and hardware. In this dissertation, I mainly focus on formal method with integration of automated fuzzing and static analysis methods to improve the quality of different types of software.

1.2 Formal Methods in Software Quality Control

In the past, due to lack of software quality control method, it is quite difficult for software engineer to control the software quality during the software life circle. Formal method is one of many theories that introduced as tools to solve this issue. Formal method is fundamentally consists of three steps: formal model abstraction, property/specification generation and property/specification checking. Formal model abstraction will abstract the software implementation into a formal model which is basically a model that is described by a specific type of language, such as XML[24] or Ethereum virtual machine[29] abstract tree. However, building a formal method based software quality control tool requires highly professional knowledge. Therefore, my research interest in throughout my Ph.D.

program is to develop formal method-based tools to support the debugging, analysis and verification of complex systems, and integrated with the automated technique. I would like to introduce the methods that are applied in this dissertation which bring universal proof of software security and solution to improve and better control the software quality.

Formal method is a mathematical technique which integrated programming theories, aims to construct the formal model to describe the behaviors of the software. With formal method, constructed model can be used to fully explore and verify the software quality. Symbolic execution is the main formal method that applied in the dissertation. Symbolic execution[61] is one of the most widely well-accepted formal methods that applied to analyze the software quality. With symbolic execution, program could be analyzed without worrying of concrete inputs to the software.

In order to apply symbolic execution method, symbolic variables will be inserted into software program at the beginning. The software program will then be abstracted into a specific type of formal method that contains path constraints, which are collected from each possible conditional branches in the software program. Satisfiability modulo theories(SMT) constraint solver will be applied to analyze the path constraints to check if any paths are feasible or infeasible in the software, and generate input case or counter input example as test case. Fig. 1.1 shows a small example to shown how symbolic execution works in analyzing the software.:

```
1 x = read();
2 if (x == 12)
3     fail();
4 else
5     print("OK");
```

Figure 1.1: Symbolic execution example code

Above code snippet is an example to better illustrate the procedure of symbolic execution. With given input of the program, x, there are two possible path that may be executed in runtime, fail() will be executed when the value of x is exactly equal to 12, otherwise, print("OK") will be executed. In manual testing, software engineer has to look inside the program in order to create test cases to cover all runtime possibilities. However, given a more complex software, manual inspection must require more time and labor to generate enough test cases.

With symbolic execution, the variable of x will be set as the symbolic variable. A static

analysis is required to capture the conditions of the selection statement, e.g. if-else statements in line 2 to line 5. In above code snippet, the captured condition is (x == 12). Once all possible conditions are captured, SMT constraint solver will be invoked to solve the captured condition and check its satisfiability. If there is at least one solution to the condition can be found to make this condition satisfied, then we can guarantee the true branch, where fail() will be executed, can be executed in runtime. Otherwise if there's no solution existing to satisfy the condition, it thus indicates only false branch will be executed in runtime.

The evaluation criteria for software testing method is the coverage rate. Coverage rate is the percentage that the software is examined. There are multiple types of coverage rate to represent the degree of the software examined in different perspectives:

- Path coverage: the coverage rate based on the number of paths checked to the total number of paths
- Statement coverage: the coverage rate based on the number of statements checked to the total number of statements
- Branch coverage: the coverage rate based on the number of branches checked to the total number of branches

The most discussed topic is how to efficiently get a high path coverage rate in the analysis process. With the developing of the software engineering, path explosion and constraint solving effectiveness are two of the key challenges in symbolic execution research are. The more paths explored, the more program behavior will be covered. However, the exponential growth of the number of paths would require a lot of computation ability and tons of time. Besides path explosion problem, the constraint solver is another bottleneck in symbolic execution. The constraints solvers lost its ability when solving large scale program. It is a very large restriction in concurrent program symbolic execution area. This dissertation is going to study the necessity of exhaustive path coverage for fault detection in testing parallel program.

Besides the formal method, fuzzing and static analysis methods are also applied in controlling the software quality in my Ph.D. researches and this dissertation.

Fuzzing[44, 45, 86] is an automated practical testing method that is widely used in software testing. By mutating and generating random inputs, fuzzing aims to execute the target program reasonable times in order to cover as many program paths as possible and find potential vulnerabilities.

With the same code snippet shown above, traditional fuzzing will randomly generate input as value of x, then execute the code snippet with input. Every time an execution is finished, a random input of x will be generated and used in next execution. The aim of fuzzing is to achieve high code coverage through multiple program executions. However, it is difficult to cover all possibilities of program with random inputs. Like in the code snippet above, the searching space for fuzzing engine to generate 12 as next input is huge without knowing the condition of the selection statement through static analysis. Thus, in order to effectively get high code coverage through fuzzing, the static information of the program must be utilized.

Static analysis methods are a set of method that perform the software analysis and testing to control its quality without executing the software, including control flow graph construction, abstract syntax tree analysis, etc. It was first gradually brought into sight in 1970's when inspection and review of the program code is regarded as necessary part of software quality[59]. It is faster than dynamic software analysis, which requires execution of the software. However, static analysis has its own limitation, such as higher false positive rates than dynamic analysis.

In this dissertation, to control the software quality, I conducted researches to test and verify the software security and correctness of different types of software with the combination of symbolic execution, fuzzing and static analysis methods for the purpose of software quality control.

1.3 Research Domains

In this dissertation, Aforementioned formal method, fuzzing and static analysis methods are primarily applied on software on following platforms and domains:

- Introduced a memory reuse distance measurement approach[65] that is based on randomized executions for sequential program and a probabilistic guarantee method of observing all possible interleaving without repeated executions for concurrency program, evaluated the

first one on Parsec benchmark suite and a large industrial-size benchmark MySQL, which confirms that the randomized execution-based approach is effective and practical.

- Developed automatic verification framework[33] to test smart contract on Ethereum platform. Framework identify paths which involve monetary transaction as critical paths and prioritize those which potentially violate important properties. The framework is evaluated that can capture vulnerabilities if user inspects fewer program paths and considering explicit or implicit loops due to fallback function.

- Implemented first-in-the-world Hyperledger Fabric chaincode fuzzing test engine to verify chaincode security issues, including detections of all types common runtime crashes and 10 types static warnings.

1.4 Dissertation Contributions

With the methods discussed in 1.2 and research domains listed in 1.3, this dissertation presents the solutions to projects that provided in each of following platform: memory reuse distance prediction, Ethereum smart contract security verification and Hyperledger Fabric chaincode fuzzing test.

1.4.1 Memory Distance Measurement for Concurrent Programs

Memory distance analysis, the number of unique memory references made between two accesses to the same memory location, is an effective method to measure data locality and predict memory behavior. Many existing methods on memory distance measurement and analysis consider sequential programs only. With the trend towards concurrent programming, it is necessary to study the impact of memory distance on the performance of concurrent programs. Unfortunately, accurate measurement of concurrent program memory distance is non-trivial. In fact, due to non-determinism, the reuse distance of memory references may differ with the same input set across multiple runs. Since memory distance measurement is fundamental to analysis, we propose a measuring approach that is based on randomized executions. Our approach provides a probabilistic guarantee of observing all

possible interleaving without repeated executions. In order to evaluate our approach, we propose a second symbolic execution-based approach that is more rigorous but much less scalable than the first approach. We have compared the two approaches on small programs and evaluated the first one on Parsec benchmark suite and a large industrial-size benchmark MySQL. Our experiments confirm that the randomized execution-based approach is effective and practical.

1.4.2 sCompile: Critical Path Identification and Analysis for Smart Contracts

Ethereum smart contracts are an innovation built on top of the blockchain technology, which provides a platform for automatically executing contracts in an anonymous, distributed, and trusted way. The problem is magnified by the fact that smart contracts, unlike ordinary programs, cannot be patched easily once deployed. It is important for smart contracts to be checked against potential vulnerabilities. In this work, we propose an alternative approach to automatically identify critical program paths (with multiple function calls including intercontact function calls) in a smart contract, rank the paths according to their criticalness, discard them if they are infeasible or otherwise present them with user friendly warnings for user inspection. We identify paths which involve monetary transaction as critical paths and prioritize those which potentially violate important properties. For scalability, symbolic execution techniques are only applied to top ranked critical paths. Our approach has been implemented in a tool called sCompile, which has been applied to 36,099 smart contracts. The experiment results show that sCompile is efficient, i.e., 5 s on average for one smart contract. Furthermore, we show that many known vulnerabilities can be captured if user inspects as few as 10 program paths generated by sCompile. Lastly, sCompile discovered 224 unknown vulnerabilities with a false positive rate of 15.4% before user inspection.

1.4.3 Hyperledger Fabric Chaincode Quality Control with Fuzz Testing

Since the invention of blockchain technology, both academia and industry area raised highly interests in researching blockchain theory seeking for utilization of its high transparency and high distribution

features. However, the rapid development and improvement of blockchain comes with the huge concerns of smart contract security. In this paper, we introduced a state-of-the-art software testing approach with fuzzing formal method technique on chaincode, which is the smart contract on the well-known blockchain platform, Hyperledger Fabric [19]. Fuzzing is a automated formal method that is widely used in software testing. By mutating and generating random inputs, fuzzing aims to execute the target program reasonable times in order to cover as many program paths as possible and find potential vulnerabilities. Based on the approach, we designed the fuzzing test tool to help chaincode user to verify the correctness of the chaincode. In our tool, chaincode that written in golang is interpreted into abstract syntax tree format, which is abstract representation of the chaincode. Interpreted chaincode thus is executed with inputs that are mutated and generated based on the feed-back guidance of code-coverage and vulnerabilities that collected statistically in previous executions. Until the a specific coverage criteria is satisfied, 10 types of chaincode vulnerabilities checking applies and there is report that contains chaincode safety information submitted to user. With experiment results of 24 golang written chaincodes, our tool is proved that can correctly examine 10 types of vulnerabilities and verifies 20 real world runtime Hyperledger Fabric chaincodes crashes in a reasonable time.

Chapter 2

Memory Distance Measurement for Concurrent Programs

This chapter presents the contribution in the research: Memory Distance Measurement for Concurrent Programs. The 30th International Workshop on Languages and Compilers for Parallel Computing (LCPC). Section 2.1 and 2.2 serves as introduction and background knowledge to the research. In section 2.3 and 2.4, symbolic execution and random scheduling methods are introduced in memory reuse distance measurement research area. The evaluation of the research is performed in section 2.5. Finally, related works study and conclusion of the research are given in section 2.6 and 2.7.

2.1 Introduction

Nowadays, widespread multicore hardware has put us at a fundamental turning point in software development. Although we have seen incrementally more programmers writing multithreaded programs in the past decade, the vast majority of applications today are still single-threaded and cannot benefit from the hardware improvement without significant redesign. Applications will need to be well-written concurrent software programs in order to benefit from the advances in multicore processors.

Table 2.1: Memory reference of a program execution

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Reference accessed	A	B	C	A	C	C	B	A	C	B	A	C	B	B	A	C
Memory distance	∞	∞	∞	2	1	0	2	2	2	2	2	2	2	0	2	2

The main reason to develop concurrent programs, which are much more sophisticated than sequential programs, is to enhance the performance of an application. To achieve the performance, developers usually make extra effort to hand tune the programs. One aspect of performance enhancement is data locality because of its significant effect on cache. In order to manage locality, developers need to measure the memory distance of their programs.

The *memory distance* of a reference is a dynamic quantifiable distance in terms of the number of different memory references between two accesses to the same memory location [46]. It is a widely accepted concept in analyzing program cache performance. The speed gap between the processor and memory has resulted in what is known as the memory wall. To overcome this wall and speed up program performance, data locality is an important factor that developers must consider. Memory distance analysis [43, 46, 47, 70] is an effective method to measure data locality and predict memory behavior.

Much existing work on memory distance measurement and analysis considers sequential programs only. With the trend towards concurrency, we need to do such measurement on concurrent programs. Unfortunately, adapting existing approaches that were designed for sequential programs is not feasible. Due to the inherent non-deterministic behavior under fixed inputs for concurrent programs, measuring concurrent memory distance is fundamentally different from that of sequential programs.

Consider the example shown in Table 2.1. The first row lists the indices of the events in a program execution under input vector v. The second row gives the symbolic memory address being accessed and the third row computes the memory distance. In the following, we use an index as the superscript to differentiate the instances of the same memory addresses in the execution trace. The memory distance of A^1, denoted as $\Delta_v(A^1)$, is ∞ because it is the first appearance of A. For the same reason we have $\Delta_v(B^2) = \Delta_v(C^3) = \infty$. $\Delta_v(A^4) = 2$ because there are two accesses to other

memory locations between the current access and the previous access to A. Note that $\Delta_v(B^7) = 2$, because although there are four accesses between B^2 and B^7, three out of the four access visit the same memory location. It can be easily observed that the minimal and maximal memory distances under v are 0 and 2 (not considering ∞), respectively. All the existing memory analysis approaches are in general based on such computation, with minor variants [1].

Table 2.2: Motivating example

index	1	2	3	4	5	6	7	8
Memory references in thread 1	A	B	C	A	C	C	B	A
Memory references in thread 2	C	B	A	C	B	B	A	C

However, the minimal and maximal memory distances under v may not be 0 and 2 if the program under analysis is concurrent. For example, the trace in Table 2.1 may be from a concurrent program with two threads as shown in Table 2.2. That is, the first eight memory accesses are from Thread 1 and the remaining eight are from Thread 2. The execution trace in Table 2.1 corresponds to the case where Thread 2 starts its execution after Thread 1 completes. However, this is not the only possibility. Many other interleavings are possible, as illustrated in Table 2.3.

Table 2.3: Memory distance results in different interleavings

idx	Reference accessed	Memory distance
1	$\{C_2,B_2,A_2,C_2,B_2,B_2,A_2,C_2,A_1,B_1,C_1,A_1,C_1,C_1,B_1,A_1\}$	$\{\infty,\infty,\infty,2,2,0,2,2,1,2,2,2,1,0,2,2\}$
2	$\{C_2,B_2,A_2,C_2,A_1,B_1,C_1,A_1,C_1,C_1,B_1,A_1,C_2,B_2,B_2,A_2\}$	$\{\infty,\infty,\infty,2,1,2,2,2,1,0,2,2,2,2,0,2\}$
3	$\{A_1,B_1,C_1,C_2,B_2,A_2,C_2,A_1,C_1,C_1,B_1,A_1,C_2,B_2,B_2,A_2\}$	$\{\infty,\infty,\infty,0,1,2,2,1,1,0,2,2,2,2,0,2\}$
4	$\{A_1,C_2,B_2,B_1,C_1,A_2,C_2,A_1,C_1,C_2,B_2,C_1,B_1,A_1,B_2,A_2\}$	$\{\infty,\infty,\infty,0,1,2,1,1,1,0,2,1,1,2,1,1\}$
5	$\{A_1,C_2,B_1,B_2,C_1,A_2,C_2,A_1,C_1,C_2,C_1,B_2,B_1,B_2,A_1,A_2\}$	$\{\infty,\infty,\infty,0,1,2,1,1,1,0,0,2,0,0,2,0\}$
6	$\{C_2,B_2,A_1,B_1,A_2,C_2,C_1,A_1,B_2,B_2,C_1,C_1,A_2,C_2,B_1,A_1\}$	$\{\infty,\infty,\infty,1,1,2,0,1,2,0,2,0,2,1,2,2\}$
7	$\{C_2,A_1,B_2,B_1,A_2,C_1,A_1,C_2,B_2,B_2,C_1,A_2,C_1,C_2,B_1,A_1\}$	$\{\infty,\infty,\infty,0,1,2,1,1,2,0,1,2,1,0,2,2\}$

This simple example illustrates the challenge in measuring memory distance for concurrent programs. Multiple executions of a concurrent program with the same input might exercise different sequences of synchronization events possibly producing different results each time. To obtain accurate memory distances for a given input, *all execution traces* permissible under that input must be examined. However, in current execution environments a developer has no control over the

[1] For example, some approaches may report $\Delta_v(B^7) = 4$ because there are four accesses between B^2 and B^7 regardless same memory locations are accessed.

scheduling of threads. Furthermore, when executing a concurrent program by running it repeatedly on a lightly-loaded machine, the same thread interleaving, with minor variations, tend to be exercised since thread schedulers generally switch among threads at the same program locations. The net effect of these impediments is that only a few interleavings end up being examined. This leads to an incomplete picture of memory distances. Even if it were possible to control thread scheduling, it would still be infeasible to explicitly test all interleavings. For example, a program with 3 threads and 50 lines of code per thread may have more than 10^{69} different interleavings. In general, the number of possible interleavings of a multi-threaded program with n threads, each executing at most k steps, can be as large as $(nk)!/(k!)^n \geq (n!)^k$, a complexity that is exponential in both n and k.

There are two well accepted method to measure the reuse distance: stack based method and tree based method. In stack based method, a stack is used for calculating reuse distance When a program is executed and encountered an address, this address will be searched in the stack. If there is no such address in the stack, then this new address will be pushed onto the top of the stack and the reuse distance of this memory reference will be set to infinity. This method takes O(N) time and is relatively slow. On the other hand, in tree based method, a tree is used to store four variables in the structure: basic pointers to the left and right child, the memory address and the the number of nodes under this node in the tree. This is a much more efficient algorithm and takes O(log N) time.

In this paper, we present an approach to measure memory distance of concurrent programs. Given the fact that we cannot possibly explore all the thread interleavings of a concurrent program, our approach introduces randomness in repeated executions. By adapting a method called PCT [26], our approach provides a mathematical guarantee to detect memory distances of given triggering depths. That is, if there exists a memory distance d between memory accesses to m with triggering depth δ_m^d(definition to be given in Section 2.4), our approach guarantees its detection with probability of $1/(n \times k^{\delta_m^d - 1})$, where n and k are the approximated number of threads and the approximated number of events, respectively, of the given program. We have implemented our method in a tool called DisConPro(Memory Distance measurement of Concurrent Programs with Probabilistic Guarantee).

In order to validate the effectiveness of DisConPro, we propose a more rigorous but much

less scalable approach to measure memory distance based on symbolic execution. The second approach utilizes the symbolic execution engine that we developed to exhaustively explore all intra-thread paths and inter-thread interleavings. We name this tool DisConSym (Memory Distance measurement of Concurrent Programs based on Symbolic Execution Guarantee). DisConSym can only handle small programs due to its inherent path explosion. By comparing DisConPro against DisConSym on small programs, we are able to determine if DisConPro covers a similar memory distance spectrum as DisConSym.

The contributions of this paper include the following:

1. To the best of our knowledge, we are the first to propose a feasible approach to measure the memory distance of concurrent programs. Our approach is based on randomized executions and provides probabilistic guarantees.

2. We propose a second approach that is more rigorous but less scalable than the first approach. Although such a symbolic execution based approach can only handle small benchmarks, it allows us to evaluate the effectiveness of the first approach.

3. We have implemented two prototypes DisConPro and DisConSym and conducted experiments on medium-sized Parsec[23] benchmarks and a large industrial size benchmark MySQL with DisConPro.

The rest of the paper is organized as follows. The background knowledge of concurrent program execution is described in Section 2.2, followed by the explanation of our two approaches in Sections 2.3 and 2.4, respectively. The experimental results are given in Section 2.5. Section 2.6 discusses the related work. Finally Section 2.7 concludes the paper.

2.2 Background: Execution of Concurrent Programs

Figure 2.1 gives a code snippet of a concurrent program with two threads. Depending on the values of a and b, different branches in the two threads can be observed across executions. Depending on the synchronization and operating system scheduling policies, different interleavings can also be

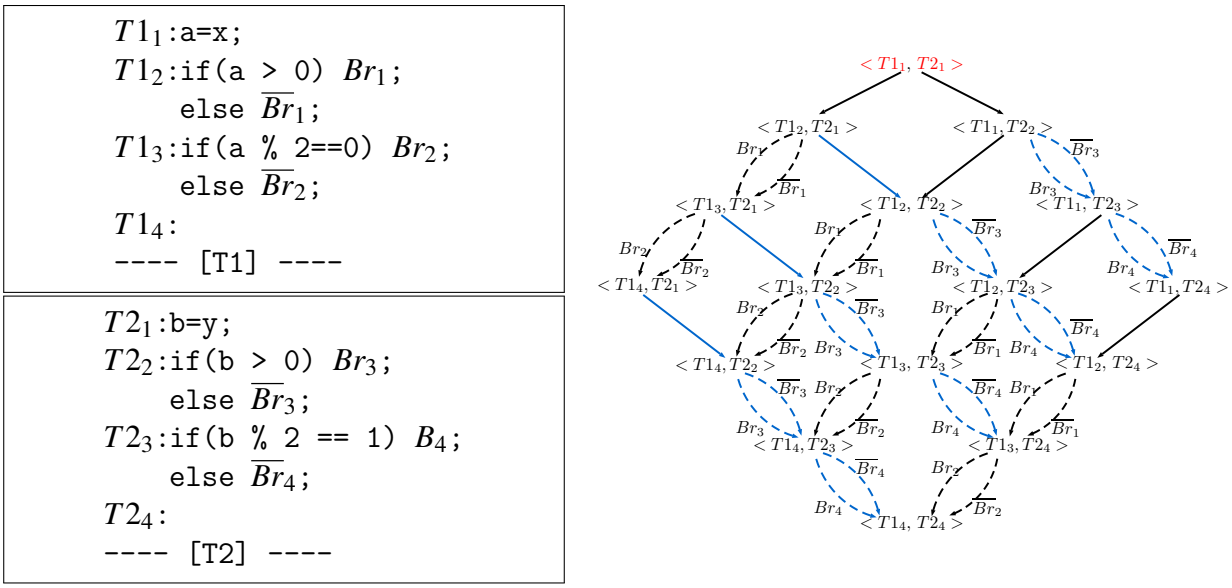


Figure 2.1: Code snippet of a concurrent program and its generalized interleaving graph(GIG).

observed. In order to present intra-thread paths and inter-thread interleavings, we use the *generalized interleaving graph*(GIG) [50, 90] to illustrate all possible executions of a concurrent program.

Figure 2.1 shows a program and its GIG. For simplicity, we assume `a=x++` is atomic. The root node (a_1, b_1) corresponds to the starting points of the two threads. The terminal node (a_5, b_5) corresponds to the end of the two threads. Nodes such as (a_1, b_1) are i-PP nodes, where we can execute either thread 1 which leads to (a_2, b_1), or thread 2 which leads to (a_1, b_2). In contrast, nodes such as (a_2, b_1) are b-PP nodes, where we can take either the `assume`$(a = 0)$ branch, leading to the code segment A_1, or the `assume`$(a \neq 0)$ branch, leading to the code segment $\overline{A}_1$.

Figure 2.1 depicts the GIG of the code snippet on its left, where black and blue edges represent an execution step of Threads $T1$ and $T2$, respectively. The dashed lines with the same source (defined as b-PP node) denote a branch within a thread and the solid lines with the same source (defined as i-PP node) denote a context switch between two threads. Note that a node can be both b-PP and i-PP. In order to measure memory distance accurately, all the paths in a GIG must be considered. This is what our symbolic execution based approach, described in Section 2.7, attempts to accomplish. However, enumerating all possible executions is obviously impractical. Thus, we present a practical approach in Section 2.4.

2.3 Memory Distance Measurement Based on Symbolic Execution

In this section, we present a symbolic execution based approach that is able to systematically explore all the intra-thread branches and inter-thread interleavings. The pseudo-code is shown in Algorithm 1, which is based on the symbolic execution algorithm proposed in [50], and follows the Concolic [80] framework. The algorithm uses a recursive procedure TRACKSTATE to explore paths. The first path is randomly chosen. When a new b-PP node with condition c is encountered, TRACKSTATE checks whether the current path condition appended with c if satisfiable. If so, it continues the execution along the branch while pushing the other branch $\neg c$ on the stack S. The satisfiability is checked by an SMT solver such as Z3 [38]. If the SMT solver fails to find a solution, it indicates that no inputs or interleavings can continue the execution along the branch. In this case, the current execution backtracks by popping its stack S. If an i-PP node is first encountered, TRACKSTATE randomly choose one interleaving while pushing the other one on the stack. For a more detailed explanation, please refer to [50]

The measurement of memory distance occurs during backtrack. That is, when the current execution reaches an end state $normal\_end\_state$ or reaches an infeasible branch. In GETMEMDIST, a path is treated as a sequence of member accesses $\langle acc_1, \ldots, acc_n \rangle$. Each acc_i is a pair $(addr, d)$ of memory address and distance. All the global memory accesses are analyzed to calculate the memory distance. Initially the memory distance of any memory access is set to -1. The algorithm continuously checks the next access acc_j. If acc_j accesses a memory address different from $acc_i.addr$, $acc_j.addr$ is added to the set $memorySet$. Otherwise,the size of $memorySet$ is the memory distance between acc_i and acc_j.

2.4 Memory Distance Measurement with Random Scheduling

In this section, we present our main approach that computes memory distances with random scheduling. We begin with the concept of memory distance minimal depth δ_m^d. Given a memory

Algorithm 1: SymbolicExecution(P)

```
    let Stack S ← ∅ be the path constraints of a path;
 1: TRACKSTATE(s)
 2:     S.push(s);
 3:     if (s is an i-PP node or b-PP node)
 4:         while (∃t ∈ (s.enabled \ s.done \ s.branch))
 5:             s' ← NEXT(s,t);
 6:             TRACKSTATE(s');
 7:             s.done ← s.done ∪ {t};
 8:     else if (s is an local thread node)
 9:         t ← s.next;
10:         s' ← NEXT(s,t);
11:         TRACKSTATE(s');
12:     path ← S.pop();
13: NEXT(s,t)
14:     let s be ⟨pcon, M⟩;
15:     if ( t instanceof halt )
16:         s' ← normal_end_state;
17:         GetMemDist(path);
18:     else if ( t instanceof branch(c) )
19:         if ( s.pcon is unsatisfiable under M )
20:             s' ← infeasible_state;
21:             GetMemDist(path);
22:         else
23:             s' ← ⟨pcon ∧ c, M⟩;
24:     else if ( t instanceof X = Y op Z )
25:         s' ← ⟨pcon, M[X]⟩;
26:     return s';
27: GETMEMDIST(path)
28:     let path be ⟨acc_1, ..., acc_n⟩;
29:     for (int i ← 0, i < n − 1, i++ )
30:         memorySet ← ∅;
31:         for (int j ← 1, j < n, j++ )
32:             if (acc_i.addr = acc_i.addr
33:                 acc_i.d ← memorySet.size();
34:                 break;
35:             else
36:                 memorySet.insert(acc_j.addr);
```

location m, δ_m^d is defined as the minimal number of constraints for any pair of accesses to m that have a memory distance of d. Consider the example given in Figure 2.4. There are four threads with eight events $e_1, \ldots, e_8$ that access four memory locations A, B, C and D. Among the total 2520 interleavings, the memory distances between a pair of accesses to A range from 0 to 3. The memory

Table 2.4: Four threads with eight memory accesses.

Thread1	Thread2	Thread3	Thread4
$< e1, A >$	$< e2, B >$	$< e3, C >$	$< e4, D >$
$< e5, A >$	$< e6, A >$	$< e7, A >$	$< e8, A >$

distance of 3 occurs only if $e_1 \prec e_2 \wedge e_1 \prec e_3 \wedge e_1 \prec e_4$, where $\prec$ denotes the happens-before relation. That is, $\delta_A^3 = 3$ because there are three constraints.

2.4.1 PCT Algorithm

We adapt the PCT [26] algorithm that was proposed to detect concurrent bugs with a probabilistic guarantee. The basic idea is to add a random scheduling control mechanism to randomize scheduling to avoid redundant executions.

In [26], a concurrent bug depth is defined as the minimum number of order constraints that are sufficient to guarantee to find the bug. The algorithm attempts to find the concurrent bug with depth of d by controlling the thread scheduling as the following.

- The scheduling is controlled by giving each thread a priority. A thread executes only if it has the highest priority or the threads with higher priorities are waiting.
- It assigns n initial priorities d, $d+1$, $d+2$....$d+n-1$ to the n threads.
- It randomly picks $d-1$ change points from k instructions, where k is the estimated number of instructions. The program is then executed with the following rules.
  - Each time only the enabled instruction from the thread with the highest priority can be executed. During execution all the instructions are counted.
  - If the instruction to be executed is counted as the number k-th and k is equal to any of k_i, change the priority value of the current thread to i. This causes a context switch.

2.4.2 Measure Memory Distance with Random Scheduling

We propose an approach called DisConPro, which adapts the PCT [26] algorithm to measure the memory distance in concurrent programs. As demonstrated above, memory distances may be different with different interleavings.

The basic idea of DisConPro is to measure the memory distance in multiple executions with the PCT scheduling control mechanism. At the beginning, DisConPro generates a random schedule following PCT [26]. Then it executes the program following the schedule. The memory distance is measured during the execution. Each memory access is recorded in a memory access trace and memory distances are calculated based on the memory access traces.

In practice, the statically computed scheduling is not always feasible. However, the infeasible cases only deviate from the planned interleavings but do not lead to execution error. For example, DisConPro may attempt to execute an instruction that is disabled by the operating system. In this case, the execution will choose the next thread with highest priority until an enabled instruction is found.

Algorithm 2: DisConPro(P,n,k,d,m)

```
1  Input: P is a program
2  Input: n is the number of threads
3  Input: k is the number of events
4  Input: d is memory distance minimum depth
5  Input: m is a memory address on which memory distance is measured
6  Var: Trace is a list that records every memory access events
7  Var: Distance is an array of memory distances
8  Distance[i] is the memory distance between i-th and (i+1)-th access to m
9  Trace = Empty List
10 Generate a random schedule S based on PCT algorithm
11 Schedule n threads based on S and execute
12 those k events
13 for each memory access event e do
14 |   Trace.add(e)
15 end
16 Calculate Distance based on Trace
```

2.4.3 Probabilistic Guarantee Inheritance

The PCT algorithm provides a probabilistic guarantee to find a concurrent bug. By adapting it, our approach can provide a probabilistic guarantee to find a particular memory distance d with a depth of δ_m^d. The probability is at least $1/(n \times k^{\delta_m^d - 1})$. Now we now give the proof by adapting the proof for finding a concurrent bug found in [26].

Definition 1. *$DisConPro(m,n,k,P)$ is defined as a set of memory distances of a memory object m. DisConPro finds memory distances during one execution of program P, containing n threads and k instructions.*

Theorem 1 (Probabilistic Guarantee Theorem)**.** *If there exists a memory distance d with a minimum depth memory distance of δ_m^d, the probability of DisConPro finding it in one execution is*

$$Pr(d \in DisConPro(m,n,k,P)) > 1/(n \times k^{\delta_m^d - 1}) \tag{2.1}$$

Proof. We define an assert statement $assert(m,d)$ as that d is not the memory distance of memory object m in the execution. We define a bug B that can be flagged if the assertion fails. If bug B is detected, d is found as the memory distance of memory m. We define event E_1 as *DisConPro* finds bug B and event E_2 as DisConPro finds d as the memory distance of m. Base on the definition of E_1 and E_2, we can argue that $E_1 \equiv E_2$. Let *Cons* be a minimum set of constraints that are sufficient for E_1 to happen. We argue that *Cons* is one of the minimum set of constraints that are sufficient for E_2 to happen. This bug B is not different from other concurrent bugs hidden in rare schedules. The depth of B equals δ_m^d, which is the size of *Cons*. We defineE_3 as PCT algorithm find B in one execution. Since DisConPro adapts PCT algorithm, we can argue that $Pr(E_2) = Pr(E_3)$. By the definition, we have

$$Pr(E_1 : d \in DisConPro(m,n,k,P)) = Pr(E_2 : DisConPro\ finds\ B) \tag{2.2}$$

$$Pr(E_2 : DisConPro\ finds\ B) = Pr(E3 : PCT\ \ finds\ B) \tag{2.3}$$

It has been proved that(see[26])

$$Pr(E_3 : PCT\ finds\ B) > 1/(n \times k^{\delta_m^d - 1}) \tag{2.4}$$

Then,

$$Pr(E_1 : d \in DisConPro(m,n,k,P)) > 1/(n \times k^{\delta_m^d - 1}) \tag{2.5}$$

□

2.5 Experiments

2.5.1 Implementation

We implement DisConPro using PIN [67], a dynamic binary instrumentation(DBI) framework that allows users to insert analysis routines to the original program in binary form. DisConSym is based on Cloud9 [34], a symbolic execution engine built upon LLVM [63, 64] and KLEE [30]. DisConSym has an extension for analyzing concurrent programs since Cloud9 only partially supports concurrency. The extension of Cloud9 follows the algorithm and implementation given in [82]. With the extension, DisConSym can analyze the interleavings not only due to synchronization primitives, which is also supported by Cloud9, but also due to global variables. The latter is essential and a prerequisite to analyze the memory distance of a concurrent program.

2.5.2 Comparison between DisConPro and DisConSym on Small Programs

We compare DisConPro with DisConSym to answer the following questions.

- Can DisConPro discover the same memory reuse range as DisConSym does?
- Can DisConPro cover all valid tracks as DisConSym does?
- Is DisConPro more scalable than DisConSym?

Table 2.5: Impact of the number of global variables comparing with DisConSym and DisConPro

	Thread number = 3	2 *mem_global*	3 *mem_global*	4 *mem_global*	5 *mem_global*
DisConSym	*mem_global* 1	-1,0,1	-1,0,1,2	-1,0,1,2,3	-1,0,1,2,3,4
	mem_global 2	-1,0,1	-1,0,1,2	-1,0,1,2,3	-1,0,1,2,3,4
	mem_global 3	N/A	-1,0,1,2	-1,0,1,2,3	-1,0,1,2,3,4
	mem_global 4	N/A	N/A	-1,0,1,2,3	-1,0,1,2,3,4
	mem_global 5	N/A	N/A	N/A	-1,0,1,2,3,4
DisConPro	*mem_global* 1	-1,0,1	-1,0,1,2	-1,0,1,2,3	-1,0,1,2,3,4
	mem_global 2	-1,0,1	-1,0,1,2	-1,0,1,2,3	-1,0,1,2,3,4
	mem_global 3	N/A	-1,0,1,2	-1,0,1,2,3	-1,0,1,2,3,4
	mem_global 4	N/A	N/A	-1,0,1,2,3	-1,0,1,2,3,4
	mem_global 5	N/A	N/A	N/A	-1,0,1,2,3,4

Since DisConSym is not scalable, we compare the two tools on several small concurrent programs with an adjustable number of threads and global variables. All the programs have less than 100 lines of code. Table 2.5 gives the experimental results. In the experiments we set the number of threads to 3, as indicated by the heading of Column 2, and the number of global variables to be 2-5. DisConSym is not able to handle a program with more threads and global variables. Columns 3-6 indicate the number of global variables created in each group of experiments. Each row in the table gives the memory distance observed for each individual global variable. When a variable does not exist in an experiment, e.g. *mem_global*3 in an experiment with only two global variables in Column 3, N/A is given. In the table, the top half of the rows give the results under DisConSym and the bottom half show the results under DisConPro. For all the experiments done by DisConPro, we set depth to be 5 and run each program 100 times. The table indicates that memory distances can be affected by the number of global variables. It can also be observed that for the small programs DisConPro can find as many memory distances as DisConSym.

Although for small programs DisConSym and DisConPro generate the same results in measuring memory distance, the cost is significantly different. Table 2.6 gives the number of paths and time usage of the seven groups of experiments with various numbers of threads and global variables. It can be observed that even for such small programs DisConPro is more than 1000 times faster. As concurrent programs become larger, the gap will be wider. Although we cannot guarantee DisConPro can detect as many memory distances as DisConSym does for non-trivial programs, we

Table 2.6: Tracked paths and time cost result for DisConSym and DisConPro

Threads and *mem_global* setting	Approach	# Paths	Time (seconds)
3 threads, 2 *mem_global*	DisConSym	90	2
	DisConPro	100	25
3 threads, 3 *mem_global*	DisConSym	1680	27
	DisConPro	100	25
3 threads, 4 *mem_global*	DisConSym	34650	930
	DisConPro	100	25
3 threads, 5 *mem_global*	DisConSym	>200000	>6794
	DisConPro	100	25
2 threads, 3 *mem_global*	DisConSym	20	1
	DisConPro	100	25
4 threads, 3 *mem_global*	DisConSym	>200000	>9609
	DisConPro	100	25
5 threads, 3 *mem_global*	DisConSym	>200000	>11473
	DisConPro	100	25

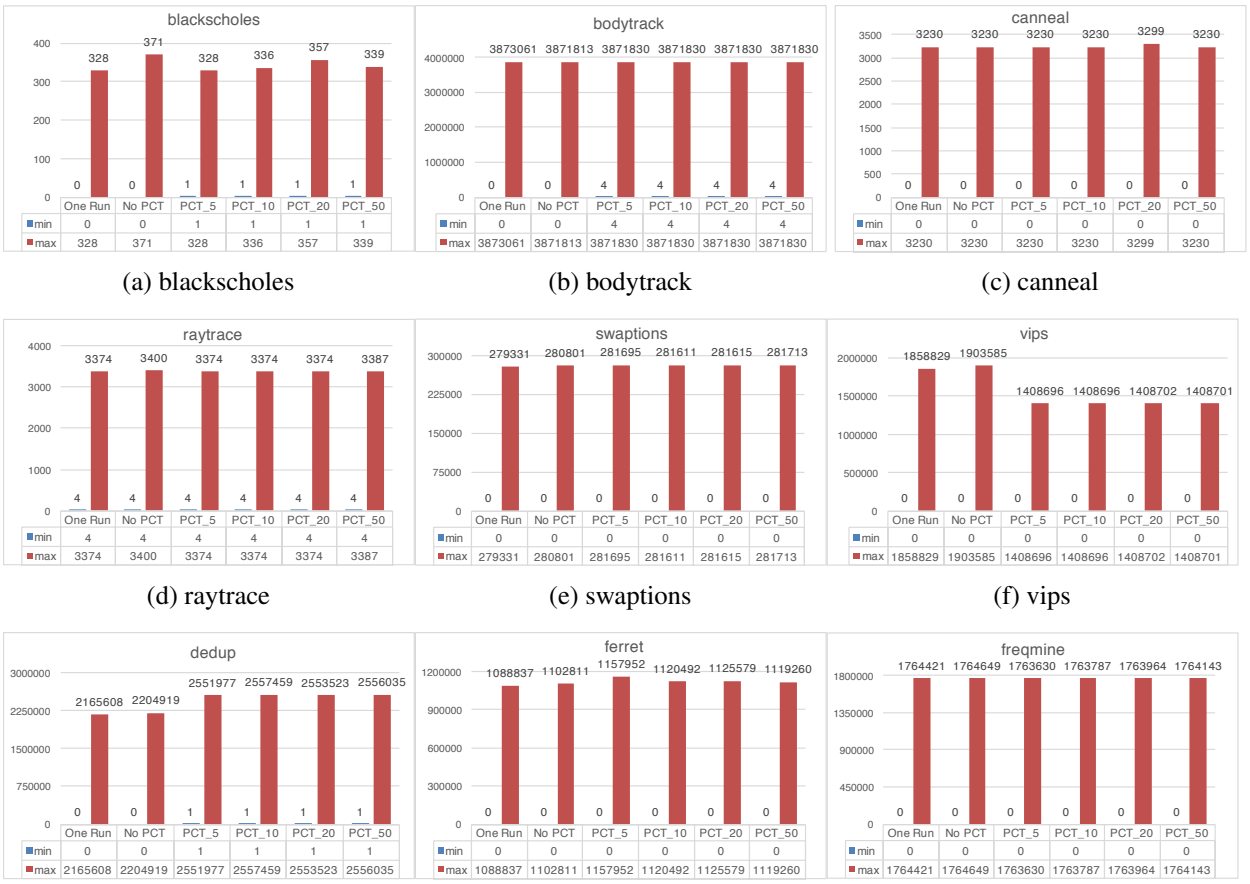
believe DisConPro achieves a nice trade-off between accuracy and efficiency.

2.5.3 DisConPro on Public Benchmarks

We evaluate DisConPro with 9 applications in the Parsec benchmark suite [23], as well as the real-world application MySQL with more than 11 million lines of code. For each application, we conduct 6 groups of experiments. Group one measures the memory distance by only running the test cases once without scheduling control. Groups 2 to 6 measure the memory distances by running each test case 30 times. Group 2 uses random scheduling. Groups 3-6 set the predefined depth to 5, 10, 20 and 50, respectively. For each application, we perform memory distance analysis on global variables only. For a large application with too many global variables, we randomly choose several global variables to measure their memory distances.

Parsec Benchmark.

Figure 2.2 gives the results of the experiments on Parsec [23]. The data in the sub-tables and sub-figures present the range of memory distances. Each column gives the minimum and maximum distances of all the global variables we evaluate. The figures show that in most cases the ranges



(a) blackscholes (b) bodytrack (c) canneal

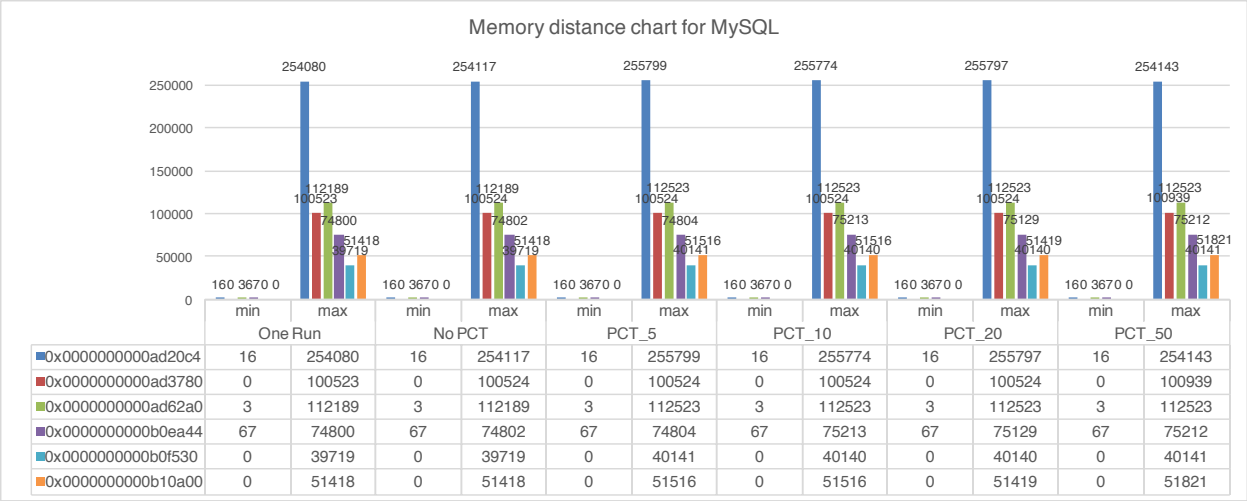
(d) raytrace (e) swaptions (f) vips

(g) dedup (h) ferret (i) freqmine

Figure 2.2: Parsec results

that DisConPro finds are larger than those detected by Random_Schedule, which in turn are larger than the ranges discovered by Single_Run. However, the range gaps achieved by PCT are not comparable to those obtained by random algorithm or even single runs. This is because the ranges reported in the figure are for all the global variables that we have evaluated. Assume that there exists a global variable that is accessed at the beginning of an execution and is re-accessed before the program terminates, its memory distance span is large and does not change much under all the possible interleavings. In this case, this variable hides the differences of the ranges exhibited in other variables.

For the application *vips*, single and random executions without PCT detect a larger memory distance span. Since PCT randomly generates change points to enforce context switches, it may disturb program executions significantly. For this reason, PCT may observe memory distances that



	One Run		No PCT		PCT_5		PCT_10		PCT_20		PCT_50	
	min	max	min	max	min	max	min	max	min	max	min	max
0x0000000000ad20c4	16	254080	16	254117	16	255799	16	255774	16	255797	16	254143
0x0000000000ad3780	0	100523	0	100524	0	100524	0	100524	0	100524	0	100939
0x0000000000ad62a0	3	112189	3	112189	3	112523	3	112523	3	112523	3	112523
0x0000000000b0ea44	67	74800	67	74802	67	74804	67	75213	67	75129	67	75212
0x0000000000b0f530	0	39719	0	39719	0	40141	0	40140	0	40140	0	40141
0x0000000000b10a00	0	51418	0	51418	0	51516	0	51516	0	51419	0	51821

Figure 2.3: MySQL result

are less diverse than those without PCT. This phenomenon is further amplified by the facts that we aggregate all variables in the same figure. To understand the performance of PCT algorithms further, we choose to illustrate the data per variable in MySQL experiments.

MySQL

Figure 2.3 gives the experimental results on MySQL. We randomly choose 6 memory objects whose addresses are listed in the table. The figure depicts the ranges of the minimum and maximum memory distances that we have observed from each group of experiments. It can be observed that the memory ranges in Groups 2 to 6 are larger than that in Group 1. By comparing the results of Group 2 to Groups 3-5, we can conclude that DisConPro is more effective than the random scheduling algorithm. The best performance algorithms for the six memory objects are PCT_5, PCT_50, PCT_5 or PCT_10 or PCT_50, PCT_10, PCT_5 or PCT_10, PCT_50, respectively. For measuring the memory distances of individual variables, DisConPro can find a range that is 30% larger than Random_Schedule.

2.6 Related Works

Cache performance heavily depends on program locality. In the past there were studies that indirectly measure program locality by simulating its execution on a set of probable cache configurations. Such simulations are not only time consuming but also inaccurate. In [42], Ding and Zhong proposed to measure program locality directly by the distance between the reuses of its data because data reuse is an inherent program property and does not depend on any cache parameters. They designed two algorithms with one targeting efficiency and the other one targeting accuracy. Their work inspired further improvements that exploits sampling [82] and statistical methods [46]. These methods work well for sequential programs. However, they do not consider that non-deterministic thread scheduling and thus not applicable to concurrent programs.

In recent years there has been research on multicore reuse distance analysis [56, 79, 92, 93]. Schuff et. al. [79] propose a sampled, paralleled method of measuring reuse distance profiles for multithreaded programs. Whereas previous full reuse distance analysis tracks every reference, sampling analysis randomly selects individual references from the dynamic reference stream and yields a sample for each by tracking unique addresses accessed until the reuse of that address. The sampling analyzer can account for multicore characteristics in much the same way as the full analyzer. The method allows the use of a fast-execution mode when no samples are currently active and allows parallelization to reduce overhead in analysis mode. These techniques result in a system with high accuracy that has comparable performance to the best single-thread reuse distance analysis tools. While our work also conducts reuse distance analysis of multithreaded programs, there exists fundamental difference between their approach and ours. Schuff et al. focus on the hardware while we focus on software. Their goal is to efficiently measure the distance on a more sophisticated multicore. Thus efficiency is a major concern of their research. With the help of the their findings a system designer may design a better cache. We aim to provide a feasible approach that measures the reuse distance of a particular multithreaded program. Therefore non-deterministic thread scheduling is the major concern of our work. With our approach we hope to let programmers understand the behavior of their multithreaded programs regardless of the cache configurations. The two methods

are orthogonal and can potentially be integrated. While we strive to diversify the executions of a multithreaded program, the approach proposed in [79] can be used to monitor each execution.

The goal of the approaches in [56, 92, 93] are similar to that of [79]. They apply reuse distance analysis to study the scalability of multicore cache hierarchies, with the goal to help architects design better cache systems. In particular, Jiang et. al. [56] introduce the concept of concurrent reuse distance (CRD), a direct extension of the traditional concept of reuse distance with data references by all co-running threads (or jobs) considered. They reveal the special challenges facing the collection and application of CRD on multicore platforms, and present the solutions based on a probabilistic model that connects CRD with the data locality of each individual thread. Wu et. al. [93] present a framework based on concurrent reuse distance and private reuse distance (PRD) profiles for reasoning about the locality impact of core count. They find that interference-based locality degradation is more significant than sharing-based locality degradation. Wu and Yeung [92] extend [93] by using reuse distance analysis to efficiently analyze multicore cache performance for loop-based parallel programs. They provide an in-depth analysis on how CRD and PRD profiles change with core count scaling, and develop techniques to predict CRD and PRD profile scaling. As we mentioned, our focus is to examine program behavior rather than the cache performance. Thus we measure memory distance from a completely different perspective from [56, 92, 93].

There exists work that studies reuse distance from other perspectives. Keramidas et al. [60] propose a direct way to predict reuse distance and apply their method to cache optimization. Zhong et al. [96] focus on the effect of input on reuse distance. They propose a statistical, pattern-matching method to predict reuse distance of a program based on executions under limited number of inputs. Shen et al. [81] introduce the time-efficiency model to analyze reuse distance with time distance. Retaining the high accuracy of memory distance, their approach significantly reduces the reuse-distance measurement cost. Niu et al. [76] present the first parallel framework to analyze reuse distance efficiently. They apply a cached size upper bound to restrict a maximum reuse distance to get a faster analysis. Although these approaches are not optimized for multithreaded programs, many of their ideas can potentially be adopted to extend our work.

Our repeated executions of a multithreaded program relies on PCT [26, 27], a randomized

algorithm originally designed for concurrent program testing. The advantage of PCT over total randomized algorithms is that PCT provides a probabilistic guarantee to detect bugs in a concurrent program. There has been recent work that adopts PCT for various purposes. For example, Liu et al. [66] introduce a pthread library replacement that applies PCT to support analyzing data races and deadlocks in concurrent programs deterministically. Cai and Yang [32] propose to add a radius to the PCT algorithm so the revised algorithm can efficiently detect deadlocks. However, to the best of our knowledge, we are the first to apply PCT in applications that are not intended to detect concurrency bugs.

2.7 Conclusion

In this paper, we have presented an approach to measure the memory distance of concurrent programs. Given the fact that we cannot possibly explore all the thread interleavings of a concurrent program, our approach introduces randomness in repeated executions. By adapting the scheduling method PCT, our approach provides a mathematical guarantee to detect memory distances of given triggering depths.

Chapter 3

sCompile: Critical Path Identification and Analysis for Smart Contracts

This chapter presents the contribution in the research: sCompile: Critical Path Identification and Analysis for Smart Contracts. 21st International Conference on Formal Engineering Methods(ICFEM 2019). At the beginning of this research, a introduction of the research topic is shown in section 3.1. To better present the research topic, an illustrative example is given in section 3.2. In section 3.3, the detailed approach and method to conduct the research is described. Experiment and evaluation details are written in section 3.4. Finally, related works and conclusion are shown in section 3.5 and 3.6.

3.1 Introduction

Built on top of cryptographic algorithms [40, 41, 57] and the blockchain technology [25, 51, 74], cryptocurrency like Bitcoin has been developing rapidly in recent years. Many believe that it has the potential to revolutionize the banking industry by allowing monetary transactions. Smart contracts bring it one step further by providing a framework which allows any contract to be executed in an autonomous, distributed, and trusted way. Smart contracts thus may revolutionize many industries. Ethereum [91], an open-source, blockchain-based cryptocurrency, is the first to

integrate the functionality of smart contracts. Due to its enormous potential, its market cap reached at $26.13 billion as of Jun 11th, 2019 [15].

In essence, smart contracts are computer programs which are automatically executed on a distributed blockchain infrastructure. A majority of smart contracts in Ethereum are written in a programming language called Solidity [13]. Like ordinary programs, Solidity programs may contain vulnerabilities, which potentially lead to attacks. The problem is magnified by the fact that smart contracts, unlike ordinary programs, cannot be patched easily once they are deployed on the blockchain.

In recent years, these attacks exploit security vulnerabilities in Ethereum smart contracts and often result in monetary loss. One notorious example is the DAO attack [2], i.e., an attacker stole more than 3.5 million Ether (about $45 million USD at the time) from the DAO contract on June 17, 2016.

The problem of analyzing and verifying smart contracts is far from being solved. Some believe that it will never be, just as the verification problem of traditional programs. Solidity is designed to be Turing-complete which intuitively means that it is very expressive and flexible. The price to pay is that almost all interesting problems associated with checking whether a smart contract is vulnerable are undecidable [89]. Consequently, tools which aim to analyze smart contracts *automatically* either are not scalable or produce many false alarms. For instance, Oyente [68] is designed to check whether a program path leads to a vulnerability or not using a constraint solver to check whether the path is feasible or not. Due to the limitation of constraint solving techniques, if Oyente is unable to determine whether the path is feasible or not, the choice is either to ignore the path (which may result in a false negative, i.e., a vulnerability is missed) or to report an alarm (which may result in a false alarm).

Besides, we believe that manual inspection is unavoidable given the expressiveness of Solidity. However, given that smart contracts often enclose many behaviors (which manifest through different paths), manually inspecting every path is overwhelming. Thus, sCompile further aims to reduce the manual effort by identifying a small number of critical paths and presenting them to the user with easy-to-digest information.

Overall, sCompile works as follows:

- sCompile firstly constructs a control flow graph (CFG) which captures all possible control flow including those due to the *inter-contract* function calls. sCompile then systematically generates paths (with a bounded sequence of function calls).
- To address path explosion, sCompile then statically identifies paths which are 'critical'. In this work, we define paths involving monetary transactions as critical paths, which is often sufficient in capturing vulnerabilities in smart contracts.
- We then define a set of (configurable) money-related properties based on existing vulnerabilities and identify all paths that potentially violate our properties. Considering that different properties have different criticalness and a long path may be unlikely feasible than a short one, sCompile ranks all paths by computing a criticalness score for each path based on the two factors.
- Finally, for top ranked paths,sCompile automatically checks whether it is feasible using symbolic execution techniques. And, the feasible paths are presented to the user for inspection.

We have implemented sCompile and applied it to 36,099 smart contracts gathered from EtherScan [7]. Our experiment shows that sCompile can efficiently analyze smart contracts, i.e., it spends 5 seconds on average to analyze a smart contract (with a bound on the number of function calls 3). Furthermore, we show that sCompile effectively prioritizes programs paths which reveal vulnerabilities in smart contracts, i.e., it is often sufficient to capture the vulnerability by inspecting the reported 10 or fewer critical paths. Overall, sCompile identified 224 vulnerabilities. The false positive rate of sCompile (before the results are reported for user inspection) is 15.4%, which is also generally acceptable. A further user study result shows that with sCompile 's help, users are more likely to identify vulnerabilities in smart contracts.

The rest of the paper is organized as follows. Section 3.2 illustrates how sCompile works through a few simple examples. Section 3.3 presents the details of our approach step-by-step. Section 3.4 shows evaluation results on sCompile . Section 3.5 reviews related work and lastly Section 3.6 concludes with a discussion on future work.

3.2 Illustrative Examples

In this section, we present multiple examples to illustrate vulnerabilities in smart contracts and how sCompile helps to reveal them. The contracts are shown in Fig. 3.1.

Example 1: Contract *EnjinBuyer* is a token managing contract. It has 2 inherent addresses for *developer* and *sale*. In function *purchase_tokens()*, the balance is sent to the sale's address. There is a mistake on the sale's address and as a result the balance is sent to a non-existing address and is lost forever. Note that any hexadecimal string of length not greater than 40 is considered a valid (well-formed) address in Ethereum and thus there is no error when function *purchase_tokens()* is executed. Given this contract, the most critical path reported by sCompile is one invoking function *purchase_tokens()*. The path will be labeled by sCompile with a message stating that the address does not exist on Ethereum mainnet. With this, the user captures the vulnerability.

Example 2: Contract *toyDAO* is an invariant one of the DAO contract. Mapping *credit* is a map which records a user's credit amount. Function *donate()* allows a user to top up its credit with 100 wei (which is a unit of Ether). Function *withdraw()* by design sends 20 wei to the message sender (at line 1) and then updates *credit*. However, when line 1 is executed, the message sender could call function *withdraw()* through its fallback function, before line 2 is executed. Line 1 is then executed again and another 20 wei is sent to the message sender. Eventually, all Ether in the wallet of this contract is sent to the message sender.

In sCompile , inspired by common practice in banking industry, assume that the user sets the limit to be 30. Given the contract, a critical path reported by sCompile is one which executes line 0, 1, 0, and 1. The path is associated with a warning message stating that the accumulated amount transferred along the path is more than the limit. We remark that existing approaches often check such vulnerability through a property called reentrancy, which often results in false alarms [58, 68].

Example 3: Contract *Bitway* is another token management contract. It receives Ether (i.e., cryptocurrency in Ethereum) through function *createTokens()*. Note that this is possible because function *createTokens()* is declared as *payable*. However, there is no function in the contract which can send Ether out. Given this contract, sCompile identifies a list of critical paths for user inspection.

```
contract EnjinBuyer {
    address public developer = 0x0639C169D9265Ca4B4DEce693764CdA8ea5F3882;
    address public sale =      0xc4740f71323129669424d1Ae06c42AEE99da30e;
    function purchase_tokens() {
        require(msg.sender == developer);
        contract_eth_value = this.balance;
        require(sale.call.value(contract_eth_value)());
        require(this.balance==0);
    }
}
contract toyDAO{
    address owner;
    mapping (address => uint) credit;
    function toyDAO() payable public {
        owner = msg.sender;
    }
    function donate() payable public{
        credit[msg.sender] = 100;
    }
    function withdraw() public {
0       uint256 value = 20;
1       if (msg.sender.call.value(value)()) {
2           credit[msg.sender] = credit[msg.sender] - value;
        }
    }
}
contract Bitway is ERC20 {
    function () public payable {
        createTokens();
    }
    function createTokens() public payable {
        require(msg.value > 300);
        ...
    }
    ...
}
```

Figure 3.1: Illustrative contracts

The most critical one is a path where function *createTokens()* is invoked. Furthermore, it is labeled with a warning message stating that the smart contract appears to be a “black hole” contract as there is no path for sending Ether out, whereas this path allows one to transfer Ether into the wallet of the contract. By inspecting this path and the warning message, the user can capture the vulnerability. In comparison, existing tools like Oyente [68] and MAIAN [75] report no vulnerability given the

contract. We remark that even although MAIAN is designed to check similar vulnerability, it checks whether a contract can receive Ether through testing[1] and thus results in a false negative in this case.

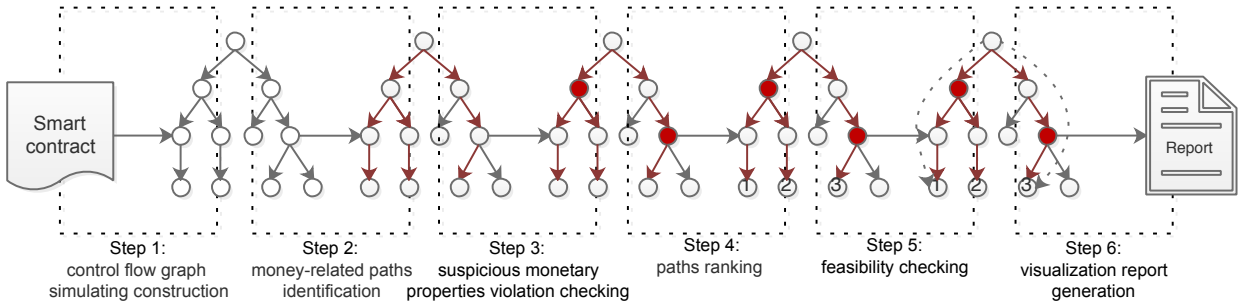


Figure 3.2: Overall workflow of sCompile

3.3 Approach

Fig. 4.10 shows the overall work flow of sCompile . Firstly, given a smart contract, sCompile construct a control flow graph (CFG) [16], based on which we can systematically enumerate all paths. Secondly, we identify the monetary paths based on the CFG up to a user-defined bound on the number of function calls. Thirdly, we analyze each path in order to check whether it potentially violates any of the pre-defined monetary properties. Next, we compute a criticalness score for each and rank the paths accordingly. Afterwards, we apply symbolic execution to filter infeasible critical paths. Lastly, we present the results along with the associated paths to the user for inspection.

3.3.1 Constructing CFG

sCompile constructs a CFG for a smart contract (the compiled EVM opcode with a single entrance for whole and for each function) to capture all possible paths. Formally, a CFG is a tuple $(N, root, E)$ such that

- N is a set of nodes, where each node is a basic block of opcodes.
- $root \in N$ is the first basic block of opcodes.

[1]MAIAN sends a value of 256 wei to the contract deployed in the private blockchain network

- $E \subseteq N \times N$ is a set of edges, where each edge (n, n') corresponds to exactly a control directly from flow n to n'.

We also consider inter-contract functions calls, where there is a CALL to a foreign function that is assumed to call the current function including third-part contract.

For instance, Fig. 3.3 shows the CFG of the contract *toyDAO* shown in Fig. 3.1. Each node is in the form of *Node_m_n*, where *m* and *n* are the indices of the first and the last opcodes of the basic block, respectively. The red diamond node at the top is the *root* node; the blue rectangle nodes represent the first node of a function. Note that a black oval represents a node that can be redirected to the root due to inter-contract function calls. The black solid edges represent the normal control flow. The red dashed edges represent control flow due to a new function call, e.g., the edge from *Node_88_91* to *Node_0_12*. That is, for every node *n* such that *n* ends with a terminating opcode instruction (i.e., STOP, RETURN), we introduce an edge from *n* to *root*. The red dotted edges represent control flow due to the inter-contract function call. That is, for every node which ends with a CALL instruction to an external function, an edge is added from the node to the root.

Given a bound *b* on the number of function calls, we can systematically unfold the CFG so as to obtain all paths during which only *b* or fewer functions are called. For instance, with a bound 2, the set of paths include all of those which visit *Node_81_87* or *Node_102_109* no more than twice.

Statically constructing the CFG is non-trivial due to *indirect jump*s in the bytecode generated by the Solidity compiler. For instance, part of the bytecode for contract *toyDAO* is shown as follows.

```
..........                  |       .......
 92 JUMPDEST                |       300 SHA3
 93 PUSH2 0x0064  //  100   |       301 DUP2
 96 PUSH2 0x0070  //  112   |       303 SSTORE
 99 JUMP                    |       304 POP
                            |
100 JUMPDEST                |       305 JUMPDEST
101 STOP                    |       306 POP
```

```
.......                    |       307 JUMP
112 JUMPDEST               |       ........
113 PUSH1 0x00             |
115 PUSH1 0x14             |
.......                    |
```

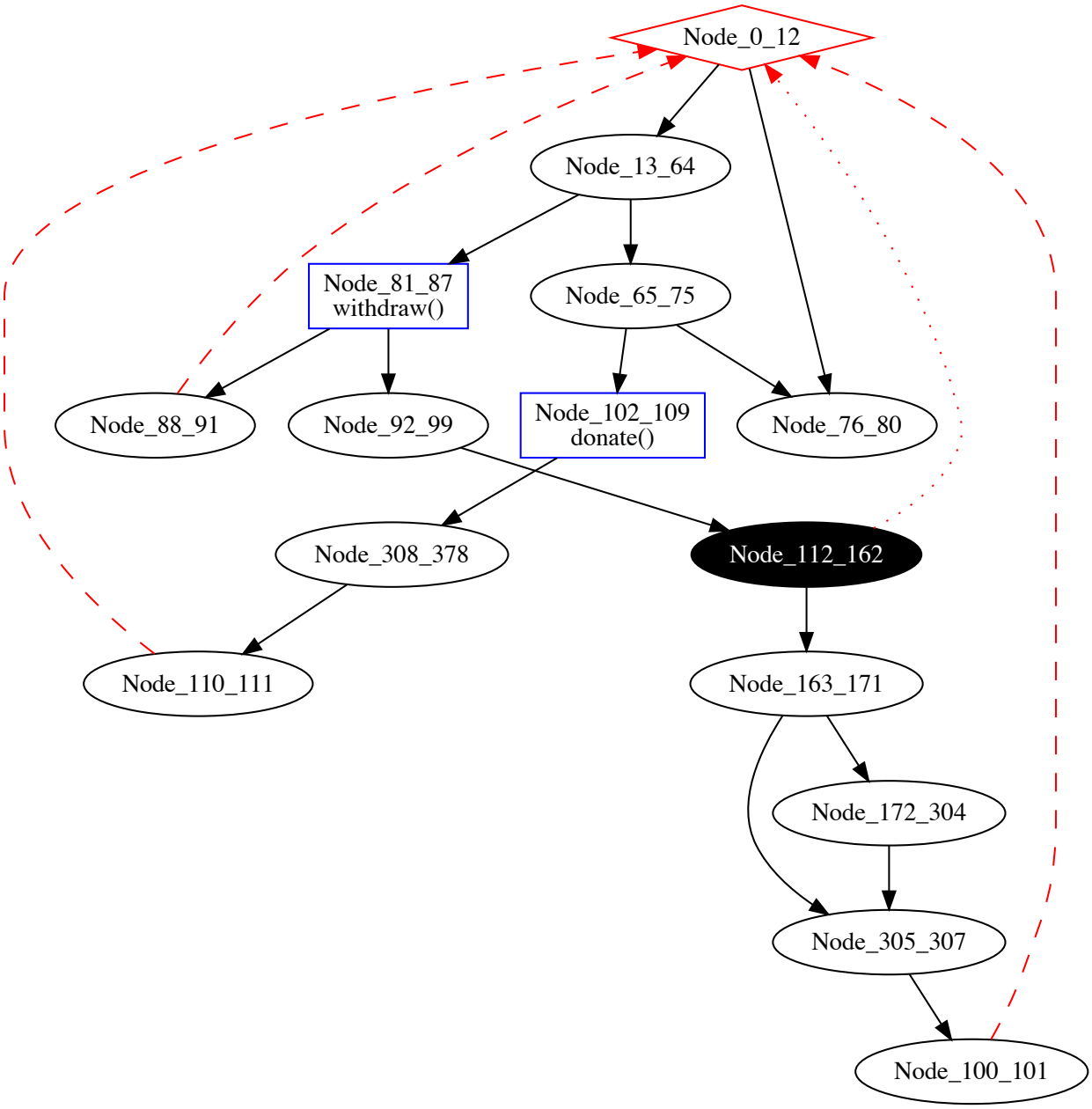


Figure 3.3: Control flow graph of the contract *toyDAO*

Considering that Solidity compiler use templates and often introduces indirect jumps (e.g., PUSH), we actually construct CFGs from EVM opcode as follows:

- Disassemble the bytecode to a sequence of opcode instructions.
- Identify all basic blocks (BBL) from the opcode instructions as nodes of a CFG, where the boundaries among BBLs are branching instructions JUMP and JUMPI, JUMPDEST, call instructions CALL, and terminal instructions such as RETURN, STOP, and REVERT.)
- Connect basic blocks with edges (e.g., direct jumps) which are statically decided from the opcode instructions.
- Use stack simulation to complete the CFG with edges for indirect jumps.

In the above, whenever there are indirect jumps, their targets cannot be decided by checking the proceeding instructions and we have missing edges. These nodes are known as *dangling blocks* and we introduce *stack simulation* to find the successor of them. Stack simulation is similar to define-use analysis except that dangling blocks which are reachable from the entry BBL are processed first. That is, we find all the paths from the entry BBL to the dangling blocks (e.g., the two paths from $Node\_0\_12$ to $Node\_305\_307$) and simulate the instructions in each path following the semantics of the instruction on the stack. Note that a dangling block ends with JUMP may have multiple successors in the CFG. When we reach the JUMP or JUMPI in the dangling block, the content of the top stack entry shall be determined and we connect the dangling block with the BBL which starts at the address as in the top stack entry. For instance, for the dangling block $Node\_305\_307$, there is only one successor $Node\_100\_101$ in both paths which is pushed by the instruction at address 093. We repeat the above step until all dangling blocks are processed.

3.3.2 Identifying Monetary Paths

Given a bound b on the number of call depth (i.e., the number of function calls) and a bound on the loop iterations, there are still many paths in the CFG to be analyzed. For instance, there are 6 paths in the *toyDAO* contract with a call depth bound of 1 (and a loop bound of 5) and 1296 with a call depth bound of 4. This is known as the path explosion problem [18]. In this work, we focus on the paths which are money-related to avoid path explosionhttps://github.com/Microsoft/verisol, as

almost all vulnerabilities [20] are 'money'-related.

A node is money-related if and only if its BBL contains any of following opcode instructions: `CALL`, `CREATE`, `DELEGATECALL` or `SELFDESTRUCT`. In general, one of these instructions must be used when Ether is transferred from one account to another. A path which traverses through a money-related node is considered money-related. [2]

3.3.3 Identifying Property-Violating Paths

Next, sCompile prioritizes paths that violate critical properties. The objective is to prioritize those paths which may trigger violation of critical properties for user inspection. The properties are designed based on previously known vulnerabilities and they can be configured and extended in sCompile .

Property: Respect the Limit In sCompile , we allow users to set a limit on the amount of Ether transferred out of the contract's wallet. For each path, we statically check whether Ether is transferred out of the wallet and whether the transferred amount is potentially beyond the limit. To do so, for each path, we use a symbolic variable to simulate the remaining limit. Each time an amount is transferred out, we decrease the variable accordingly and check whether the remaining limit is less than zero. If so, the path potentially violates the property. Note that if we are unable to determine the exact amount to be transferred , we conservatively assume the limit may be broken.

Property: Avoid Non-Existing Addresses Any hexadecimal string of length no greater than 40 is considered a valid (well-formed) address in Ethereum. If a non-existing address is used as the receiver of a transfer, the Solidity compiler does not generate any warning and the contract can be deployed on Ethereum successfully. If a transfer to a non-existing address is executed, Ethereum automatically registers a new address (after padding 0s in front of the address so that its length becomes 160bits). Because this address is owned by nobody, no one can withdraw the Ether in it since no one has the private key.

[2]Note that each opcode instruction in EVM is associated with some gas consumption which technically makes them money-related. Gas[91] is the cost of any transaction that can be utilized to measure actions on Ethereum platform. However, the gas consumption alone in most cases does not constitute vulnerabilities and therefore we do not consider them money-related. In Fig. 3.3, we visualize money-related nodes with black background (e.g., the node *Node*_112_162 with a `CALL` statement `msg.sender.call.value(value)()`).

For every path which contains instruction CALL or SELFDESTRUCT, sCompile checks whether the address in the instruction exists or not. This is done with the help of EtherScan Ethereum [7] (which can check whether an address is registered or not). A path which sends Ether to a non-existing address is considered to be violating the property. Currently, to minimize the number of requests to EtherScan, we only query external transactions, thus may lead to false positives when the address has only internal transactions. Of course, users can configure sCompile to also check internal transactions.

Property: Guard Suicide sCompile checks whether a path would result in destructing the contract without constraints on the date or block number, or the contract ownership. A contract may be designed to "suicide" (with the opcode SELFDESTRUCT) after certain date or reaching certain number of blocks, and often by transferring the Ether in the contract wallet to the owner. A famous example is Parity Wallet which resulted in an estimated loss of tokens worthy of $155 million [1].

We thus check whether there exists a path which executes SELFDESTRUCT and whether its path condition is constituted with constraints on date or block number and contract owner address. While checking the former is straightforward, checking the latter is achieved by checking whether the path contains constraints on instruction TIMESTAMP or BLOCK, and checking whether the path condition compares the variables representing the contract owner address with other addresses. A path which calls SELFDESTRUCT without such constraints is considered a violation of the property.

```
   contract StandardToken is Token {
1     function destroycontract(address _to) {
2        require(now > start + 10 days);
3        require(msg.sender != 0);
4        selfdestruct(_to);
5     }
6     ...
7  }
8  contract Problematic is StandardToken { ... }
```

Figure 3.4: Guardless suicide

One example is the *Problematic* contract[3] shown in Fig. 3.4. Contract *Problematic* inherits

[3]We hide the names of the contracts as some of them are yet to be fixed.

contract *StandardToken*, where one of the functions is *destroycontract()* allowing one to destruct the contract. sCompile can report that line 4 potentially violates the property.

Property: Be No Black Hole In a few cases, sCompile analyzes paths which do not contain CALL, CREATE, DELEGATECALL or SELFDESTRUCT. For instance, if a contract has no money-related paths (i.e., never sends any Ether out), sCompile then checks whether there exists a path which allows the contract to receive Ether. The idea is to check whether the contract acts like a black hole for Ether. If it does, it is considered a vulnerability.

To check whether the contract can receive Ether, we check whether there is a *payable* function. Since Solidity version 0.4.x, a contract is allowed to receive Ether only if one of its public functions is declared with the keyword *payable*. When the Solidity compiler compiles a non-payable function, the following sequence of opcode instructions are inserted before the function body.

```
1  CALLVALUE
2  ISZERO
3  PUSH XX
4  JUMPI
5  PUSH1 0x00
6  DUP1
7  REVERT
```

At line 1, the instruction CALLVALUE retrieves the message value (to be received). Instruction ISZERO then checks if the value is zero, if it is zero, it jumps (through the JUMPI instruction at line 4) to the address which is pushed into stack by the instruction at line 3; or it goes to the block starting at line 5, which reverts the transaction (by instruction REVERT at line 7). Thus, to check whether the contract is allowed to receive Ether, we go through every path to check whether it contains the above-mentioned sequence of instructions. If all of them do, we conclude that the contract is not allowed to receive Ether. Otherwise, it is. If the contract can receive Ether but cannot send any out, we identify the path for receiving Ether as potentially violating the property and label it with a warning messaging stating that the contract is a black hole.

Above properties are designed based on reported vulnerabilities. Of course, sCompile is designed to be extensible, i.e., new properties can be easily supported by providing a function which takes a path as input and reports whether the property is violated.

To further help users understand paths of a smart contract, sCompile supports additional analysis. For instance, sCompile provides analysis of gas consumption of paths. However, without trying out all possible inputs, users may not be aware of the existence of certain particularly gas

consuming paths. The gas consumption of a path is estimated based on each opcode instruction in the path statically.

3.3.4 Ranking Program Paths

To allow user to focus on the most critical paths as well as to save analyses efforts, we prioritize the paths according to the likelihood they reveal critical vulnerability. For each path, we calculate a criticalness score and rank the paths according to the scores. The criticalness score is calculated as follows: let *pa* be a path and V be the set of properties which *pa* violates.

$$criticalness(pa) = \frac{\Sigma_{pr \in V} \alpha_{pr}}{\varepsilon * bound(pa)} \tag{3.1}$$

where α_{pr} is a constant which denotes the criticalness of violating property *pr*, $bound(pa)$ is the depth bound of path *pa* (i.e., the number of function calls) and ε is a positive constant. Intuitively, the criticalness is designed such that the more critical a property the path violates, the larger the score is; and the more properties it violates, the larger the score is. Furthermore, it penalizes long paths so that short paths are presented first for user inspection.

Table 3.1: Definition of α_{pr}

	transfer limit	non-existing addr.	suicide	black hole
Likelihood	1	1	2	3
Severity	2	3	3	2
Difficulty	2	2	3	2
α_{pr}	4	6	18	12

To assess the criticalness of each property, we use the technique called failure mode and effects analysis (FMEA [83]) which is a risk management tool widely used in a variety of industries. FMEA evaluates each property with 3 factors, i.e., *Likelihood*, *Severity* and *Difficulty*. Each factor is a value rating from 1 to 3, i.e., 3 for *Likelihood* means the most likely; 3 for *Severity* means the most severe and 3 for *Difficulty* means the most difficult to detect. The criticalness α_{pr} is then set as the product of the three factors. After ranking, only paths which have a criticalness score larger than certain threshold are subject to further analysis, reducing the number of paths significantly.

In order to identify the threshold for criticalness, we adapt the k-fold cross-validation[39, 62] idea in statistical area. We collected a large set of smart contracts and split them into a training data set(10,452 contracts) and a test data set (25,678 contracts). We repeated the experiments 20 times which took more than 5,700 total hours of all machines and optimizes those parameters.The adapted parameters are shown in Table 3.1, and ε is set to be 1 and the threshold for criticalness is set to be 10.

3.3.5 Feasibility Checking

Not all the paths are feasible. To avoid such false alarms, we filter infeasible paths through symbolic execution [53, 61]. The basic idea is to symbolically execute a given program.Symbolic execution has been previously applied to Solidity programs in Oyente [68] and MAIAN [75]. In this work, we apply symbolic execution to reduce the paths which are to be presented for users' inspection. Only if a path is found to be infeasible by symbolic execution, we remove it. In comparison, both Oyente and MAIAN aim to fully automatically analyze smart contracts and thus when a path cannot be determined by symbolic execution, the result may be a false positive or negative.

```
  contract GigsToken {
1   function createTokens() payable {
2     require(msg.value > 0);
3     uint256 tokens = msg.value.mul(RATE);
4     balances[msg.sender] = balances[msg.sender].add(tokens);
5     owner.transfer(msg.value);
6   }
7   ...
  }
```

Figure 3.5: A non-greedy contract

For instance, Fig. 3.5 shows a contract which is capable of both receiving (since the function is *payable*) and sending Ether (due to *owner.transfer(msg.value)* at line 5), and thus sCompile does not flag it to be a black hole contract. MAIAN however claims that it is. A closer investigation reveals that because MAIAN has trouble in solving the path condition for reaching line 5, and thus

mistakenly assumes that the path is infeasible. As a result, it believes that there is no way Ethers can be sent out and thus the contract is a black hole.

3.4 Implementation and Evaluation

3.4.1 Implementation

sCompile is implemented in C++ with about 8K lines of code and is available online[4]. The symbolic execution engine in sCompile is built based on the Z3 SMT solver [37].

3.4.2 Experiment

We aim to to answer research questions (RQ) regarding sCompile 's efficiency, effectiveness and usefulness in practice. Our test subjects contain all 36,099 contracts (including both the training set and the set) with Solidity source code downloaded from EtherScan. sCompile can directly take EVM code as input and the source code is used for our manual inspection for experiment purpose.

All experiment are done on an Amazon EC2 C3 xlarge instance installed with Ubuntu 16.04 and gcc 5.4. The timeout set for sCompile is: global wall time is 60 seconds and Z3 solver timeout is 100 milliseconds. The limit on the maximum number of blocks for a single path is set to be 60, and the limit on the maximum iterations of loops is set to be 5, i.e., each loop is unfolded at most five times.

RQ1: Is sCompile efficient enough for practical usage? In this experiment, we evaluate sCompile in terms of its execution time. We systematically apply sCompile to all the benchmark programs in the training set.

The results are summarized in Figure 3.6. In sub-table of Figure 3.6, the second, third and fourth row show the execution of sCompile with call depth bound 1, 2, and 3 respectively.For comparison, the fifth row shows the execution time of Oyente (the latest version 0.2.7) with the same timeout. We remark that the comparison should be taken with a grain of salt. Oyente does

[4]The link is removed for anonymity.

not consider sequences of function calls, i.e., its bound on function calls is 1. Furthermore, it does not consider initialization of variables in the constructor (or in the contract itself). The next columns show the execution time of MAIAN (the latest commit version on Mar 19). Although MAIAN is designed to analyze paths with multiple (by default, 3) function calls, it does not consider the possibility of a third-party contract calling any function in the contract through inter-contract function calls and thus often explores much fewer paths than sCompile . Furthermore, MAIAN checks only one of the three properties (i.e., suicidal, prodigal and greedy) each time. Thus, we must run MAIAN three times to check all three properties. The different bounds used in all three tools are summarized in Table 3.2.

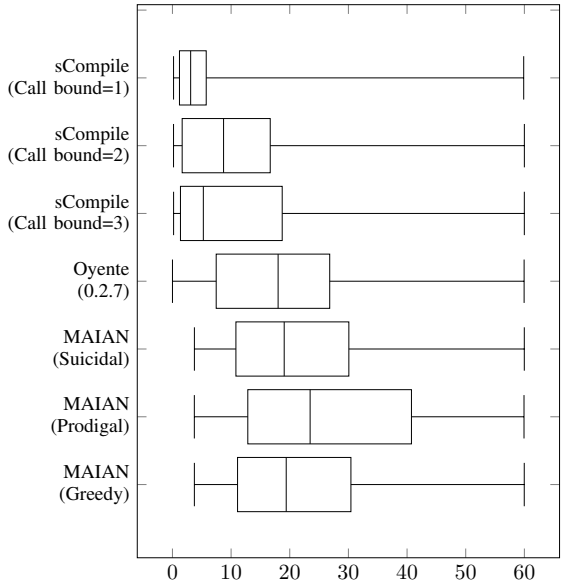
Table 3.2: Loop bound definitions among three tools

Tool	call bound	loop bound	timeout	other bound
sCompile	3	5	60 s	60 cfg nodes
Oyente	1	10	60 s	N.A.
MAIAN	3 (no inter-contract)	N.A.	60 s	60 cfg nodes

In sub-table of Figure 3.6, the second column shows the median execution time and the third column shows the number of times the execution time exceeds the global wall time (60 seconds). We observe that sCompile almost always finishes its analysis within 10 second. Furthermore, the execution time remains similar with different call depth bounds. This is largely due to sCompile 's strategy on applying symbolic execution only to a small number of top ranked critical paths. We do however observe that the number of timeouts increases with an increased call depth bound. A close investigation shows that this is mainly because the number of paths extracted from CFG is much larger and it takes more time to extract all paths for ranking. In comparison, although Oyente has a call depth bound of 1, it times out on more contracts and spends more time on average. MAIAN spends more time on each property than the total execution of sCompile . For some property (such as *Greedy*), MAIAN times out fewer times, which is mainly because it does not consider inter-contract function calls and thus works with a smaller CFG.

The sub-figure in Figure 3.6 visualizes the distribution of execution time of the tools in plot-box. The x-axis represents the execution time (in seconds). From the figure, we can conclude

that sCompile is efficient.



	median(s)	timeout #
sCompile (Call bound = 1)	3.106	1145
sCompile (Call bound = 2)	8.717	1737
sCompile (Call bound = 3)	5.267	2597
Oyente	18.015	2223
MAIAN (Suicidal)	19.053	1561
MAIAN (Prodigal)	23.472	6186
MAIAN (Greedy)	19.397	1081

Figure 3.6: Execution time of sCompile vs. Oyente vs. MAIAN

Table 3.3: Comparison on vulnerable contracts

	sCompile			**MAIAN**		
	alarmed	true positive	false positive	alarmed	true positive	false positive
Avoid non-existing address	37	32	5	N.A.	N.A.	N.A.
Be no black hole	57	57	0	141	56	85
Guard suicide	42	38	4	66	30	36

Table 3.4 shows the statistics on the number of processed path, including the estimated total number of paths on average (in the second column), the number of symbolic-executed (based on CFG), and the number passed to users. It can be observed that only a small fraction of the paths are symbolically analyzed. Furthermore, the number of symbolically executed paths remain small even when the call depth bound is increased. This is because only the top ranked critical paths are analyzed by symbolic execution.

Table 3.4: Average number of program paths

	In total	**Symbolic-executed**	**To user**
call depth 1	48.92	37.51	1.49
call depth 2	6177.21	144.24	12.46
call depth 3	31346.62	121.23	12.62

RQ2: Is sCompile effective to practical usage? In the second experiment, we aim to investigate the effectiveness of sCompile . We apply sCompile to all 36,099 contracts and manually inspect the critical paths reported by sCompile to check whether the path, together with the associated warning message, reveals a true vulnerability in the contract. Note that not all properties checked by sCompile readily signals a vulnerability. We only focus on those results produced by sCompile which are directly related to vulnerabilities in the following, i.e., paths which are deemed to violate property "avoid non-existing addresses", "be no black hole" and "guard suicide". Note that two of the properties (i.e., the latter two) analyzed by sCompile are supported by MAIAN as well. We can thus compare sCompile 's performance with that of MAIAN for these two properties. The results are shown in Table 3.3. In the following, we discuss the detailed findings[5].

For *Property: Be no Black Hole*, there are 57 contracts in the training set are marked vulnerable by sCompile . We manually confirmed that they are all true positives. In comparison, MAIAN identified 141 black hole contracts and 56 contracts among them are true positives, 43 of which overlap with sCompile 's results. For 13 missed contracts by sCompile but detected by MAIAN, all of them took more than 60 seconds and thus sCompile timed out before finishing analyzing.

The other 85 identified by MAIAN are false positives and 62 of them are library contracts. We randomly choose 5 contracts from the remaining for further investigation. We find Z3 could not finish solving the path condition in time and thus MAIAN conservatively marks the contract as vulnerable. After extending the time limit for Z3 and the total timeout, 4 of the 5 false positives are still reported. The reason is that these contracts can only send Ether out after certain period, and MAIAN could not find a feasible path to send Ether out for such cases, and mistakenly flags the contract as a black hole.

For *Property: Guard Suicide*, sCompile reports a program path if it leads to `SELFDESTRUCT`, without a constraint on the ownership of the contract or the date or the block number, i.e., a guard to prevent an unauthorized users from killing the contract. Among the analyzed contracts, sCompile

[5]We have informed all developers whose contact info are available about the vulnerabilities in their contracts and several have confirmed the vulnerabilities and deployed new contracts to substitute the vulnerable ones. Some are yet to respond, although the balance in their contracts are typically small.

identified 42 contracts which contain at least one path which violates the property. Many of the identified contracts violate the property due to contract inheritance as shown in Fig. 3.4.

The remaining 4 cases reported by sCompile are false positives. We manually investigated into them and found that they are belonged two uncommon coding cases (where 3 of them are originated from the same contract) and three of them can be detected by sCompile by slightly revising its implementation.

MAIAN identified 66 contracts violating the property. 30 of them are true positives, 13 of which are also identified by sCompile . The other 36 are false positives. The contract *MiCarsToken* shown in Fig. 3.7 shows a typical false alarm. There are 2 constraints before SELFDESTRUCT in the contract. sCompile considers such a contract safe for there is a guard of *msg.sender* == *owner* (or the other condition), whereas MAIAN reports a vulnerability as the contract can also be killed if the msg.sender is not the owner when the second condition is satisfied.

```
contract MiCarsToken {
    function killContract () payable external {
      if (msg.sender==owner ||
              msg.value >=howManyEtherInWeiToKillContract)
          selfdestruct(owner);
    }
    ...
}
```

Figure 3.7: Ambiguous cases between sCompile and MAIAN

We further analyzed the 17 cases which were neglected by sCompile . 6 of them are alarmed for owner change as exemplified in Fig. 3.8. In this contract, *selfdestruct* is well guarded, but the developer makes a mistake so that the constructor becomes a normal function, and anyone can invoke *mortal()* to make himself the owner of this contract and kill the contract.

For *Property: Avoid Non-existing Address*For the contracts in the training set, all addresses identified are of length 160 bits. However, there are 37 contracts identified as non-existing addresses (i.e., not registered in Ethereum mainnet). They may be used for different reasons. For example, in contract *AmbrosusSale*, the address of TREASURY does not exist before the function `specialPurchase()` or `processPurchase()` is invoked (which will cost more gas for its first

```
contract Mortal {
    address public owner;
    function mortal() { owner = msg.sender; }
    function kill() {
      if (msg.sender == owner) suicide(owner); }
}
```

Figure 3.8: Contract of owner change

user). And there are 5 addresses registered by internal transactions.

We further analyzed 25,647 contracts newly uploaded in EtherScan from February 2018 to July 2018. For *"Be no Black Hole"*, there are 109 vulnerabilities out of 139 alarms generated by sCompile . Applying MAIAN on these contracts, 84 of them are marked vulnerable, 77 of which are true vulnerabilities overlapping with those found by sCompile and 7 library contracts are marked vulnerable mistakenly. Among the 139 contracts, 25 vulnerable ones are missed by MAIAN according to our manual check. For *"Guard Suicide"*, there are 83 vulnerabilities out of 114 alarms generated by sCompile . Applying MAIAN on these contracts, 42 are marked vulnerable, all of which overlap with those found by sCompile . For *"Avoid Non-existing Addresses"*, there are 80 vulnerabilities out of 87 alarms generated by sCompile . The 7 false alarms are due to internal transactions.

In total, sCompile identifies 224 vulnerabilities from the 36,099 contracts consisting of 46 *Black Hole* vulnerabilities, 66 *Guardless Suicide* vulnerabilities and 112 *Non-existing Address* vulnerabilities.

RQ3: Is sCompile useful to contract users? Different from other tools which aim to fully automatically analyze smart contracts, sCompile is designed to facilitate human users. We thus conduct a user study to see whether sCompile is helpful to them.

The study takes the form of an online test. Once a user starts the test, first the user is briefed with necessary background on smart contract vulnerabilities (with examples). Then, 6 smart contracts (selected at random each time from a pool of contracts) are displayed one by one. For each contract, the source code is first shown. Afterwards, the user is asked to analyze the contract and answer the two questions. The first question asks what is the vulnerability the contract has.

The second question requires user to identify the most gas consuming path in contract (with one function call).

For the first three contracts, the outputs from sCompile are shown alongside the contract source code as a hint to the user. For the remaining 3 contracts, the hints are not shown. The contracts are randomized so that not the same contracts are always displayed with the hint. The goal is to check whether users can identify the vulnerabilities correctly and more efficiently with sCompile 's results.

We distribute the test through social networks and online professional forums. We also distribute it through personal contacts who we know have some experience with Solidity smart contracts. In three weeks we collected 48 successful responses to the contracts (without junk answers)[6]. Table 3.5 summarizes the results. Recall that sCompile 's results are presented for the first three contracts. Column LOC and #paths shows the number of lines and paths in each contract. Note that in order to keep the test manageable, we are limited to relatively small contracts in this study. Columns Q1 and Q2 show the number of correct responses (the numerator) out of the number of valid responses (the denominator). We collect the time (in seconds) taken by each user in the Time column to answer all the questions. In the end of the survey we ask the user to give us a score (on the scale of 1 to 7, the higher the score the more useful our tool is) on how useful the hints in helping them answer the questions. The value in column Usefulness is the average score over all responses because all responses are shown half the hints.

Table 3.5: Statistics and results of surveyed contracts

Contract	LOC	#paths	Q1	Q2	Time	Usefulness
C1 (w)	33	8	7/8	3/8	119	5
C2 (w)	52	16	7/8	2/8	98	
C3 (w)	67	38	7/8	2/8	233	
C4 (w/o)	87	59	2/8	1/8	414	
C5 (w/o)	103	13	3/8	1/8	397	
C6 (w/o)	107	27	4/8	1/8	420	

The results show that for the first three contracts for which sCompile 's analysis results are shown, almost all users are able to answer Q1 correctly using less time. For the last three contracts

[6]There are about 80 people who tried the test. Most of the respondents however leave the test after the first question, which perhaps evidences the difficulty in analyzing smart contracts.

without the hints, most of the users cannot identify the vulnerability correctly and it takes more time for them to answer the question. For identifying the most gas-consuming path, even with the hints on which function takes the most gas, most of the users find it difficult in answering the question, although with sCompile 's help, more users are able to answer the question correctly. The results show that gas consumption is not a well-understood problem and highlight the necessity of reporting the condition under which maximum gas consumption happens. All the users think our tool is useful (average score is $5/7$) in helping them identify the problems.

3.5 Related Works

sCompile is related to existing work on identifying vulnerabilities in smart contracts that can be roughly categorized into 3 groups according to the level at which the vulnerability resides at: Solidity-level, EVM-level, and blockchain-level [20**?**]. In addition, existing work can be categorized according to the techniques they employ to find vulnerabilities: symbolic execution [36, 55, 68, 75, 87], static-analysis based approaches [88] and formal verification [22, 58]. Our approach works at the EVM-level and is based on static analysis and symbolic execution, and is thus closely related to the following work.

Oyente [68] formulates the security bugs as intra-procedural properties and uses symbolic execution to check these properties. However, Oyente does not perform inter-procedural analyses to check inter-procedural or trace properties as did in sCompile .

MAIAN [75] is recently developed to find three types of problematic contracts in the wild: prodigal, greedy and suicidal. It formulates the three types of problems as inter-procedural properties and performs bounded inter-procedural symbolic execution. It builds a private testnet to valid whether the contracts found by it are true positives by executing the contracts with data generated by symbolic execution. However, sCompile differs from MAIAN in following aspects. First, sCompile makes a much more conservative assumption about a call to third-party contract which we assume can call back a function in current contract. sCompile is designed to reduce user effort rather than to analyze smart contracts fully automatically. Secondly, sCompile supports

more properties than MAIAN. Thirdly, sCompile checks properties in ways which are different from MAIAN. sCompile treates the value used for a transaction as symbolic variable and does not introduce false negatives. For checking suicidal contracts, MAIAN performs symbolic execution and generates inputs to trigger the suicide of the contract in the validation phase; sCompile checks guardless suicide with either ownership checking or date checking which are often hard for SMT solver to generate values. Other symbolic execution based tools [36, 87] perform intra-procedural symbolic analysis directly on the EVM bytecode as what Oyente does.

The tool Securify [88] is based on static analysis to analyze contracts. It specifies both compliance and violation patterns for the property. The vulnerability detection problem is then reduced to search the patterns on the inferred data and control dependencies information. The use of compliance pattern reduces the number of false positives in the reported warnings. In the ranking algorithm, our approach rely on syntactic information to reduce paths for further symbolic analysis to improve performance. We analyze the extracted paths with symbolic execution which is more precise than the pure static analysis as adopted by Securify.

Other attempts on analyzing smart contracts include formal verification using either model-checking techniques [58] or theorem-proving approaches [22]. They in theory can check arbitrary properties specified manually in a form accepted by the model checker or the theorem prover. It is known that model checking has limited scalability whereas theorem proving requires an overwhelming amount of user effort.

3.6 Conclusion

We proposed a practical approach named sCompile to reveal “money-related“ paths in smart contract and to further detect vulnerabilities among critical ones. In our experiment among 36,099 smart contracts, it detected 224 new vulnerabilities. All the new vulnerabilities are well defined in our approach and could be presented to the user in well-organized information within a reasonable time frame. A comparison with two existing approaches also demonstrated that sCompile is both efficient and effective.

Chapter 4

Hyperledger Fabric Chaincode Quality Control with Fuzz Testing

This chapter presents the contribution in the research about Hyperledger Fabric chaincode quality control with fuzz testing method. At the beginning, introduction of the research topic and a motivating example is shown in section 4.1 and 4.2. After that, in section 4.3, the entire fuzzing approach is described in 5 steps and the details of these steps are completely described. The vulnerabilities detection analysis is illustrated in section 4.4. In order to evaluate the research project performance, a well designed experiment and its result evaluation is shown in section 4.5. At last, related works and conslusion is given in section 4.6 and 4.7.

4.1 Introduction

Smart contract is an revolutionary software category built upon the blockchain platform, which has been well-accepted in documentation keeping and broadcasting messages that introduced in recent decade. Smart contract is designed as a programmable and safe implementation for the purpose of satisfying user's requirements, such as voting and trading cryptocurrency. However, in most recent years, there are plenties of attacks that caused huge economic losses. The notorious DAO attack make the Ethereum owner suffer a loss of $ 60M. More hazardously, the same type of attack that

had been discovered and happened, which cannot be prevented completely.

Hyperledger Fabric is a promising blockchain platform that maintained by Linux foundation. It has great features on distribution and privacy which are the reason of being widely accepted and applied in business industry. It's smart contract, chaincode, designed with upgradeable and high maintainable status, which is also extensively utilized and integrated into the business system.

With the development and adoption of Hyperledger Fabric, there's incremental demands for the security analysis. However, the security of Hyperledger unfortunately, hasn't not gain much attention from neither industry nor research area. Noticing this surging security requests and driven by curiosity of software security research on chaincode, we developed an approach for automated analyzing chaincode vulnerabilities with fuzz testing, which is proved as a quality software testing technique to discover software security issues.

To the best of our knowledge, we didn't find any effective fuzzing analysis tool that can truly verify chaincode security issues and detect chaincode vulnerabilities and crashes. There are researches[3, 94] that focused on static analysis for Hyperledger Fabric chaincode. However, there're natural limitations to statically analyze program as static analysis cannot prove the found warnings really exist in runtime or could lead to runtime bug in practical. Moreover, static analysis also bring high amount of false positive and false negative in analysis result.

Besides static analysis, fuzzing is another widely used software testing technique in practical industry. Building a efficient fuzzer is quite challenging due to following difficulties:1, difficulty to generate new inputs; 2, difficulty to create a execution method for specific type of program;3, difficulty to achieve high coverage rate and 4, difficulty to reveal potential vulnerabilities in test result. In order to efficiently utilize fuzzing technique in chaincode testing, our approach take above difficulties into considerations and is implemented into afuzzer , which is modularized fuzzer that handled difficulties to guarantee figure out any potential warnings and security issues that hidden inside the chaincode. afuzzer inherits advantages from go-fuzz [6], which is well-known golang program fuzzer, and extensively adopts feed-back seed mutation strategy and static analysis on chaincode security issues. This make afuzzer a first and unique Hyperledger Fabric chaincode fuzzer to bring insight to chaincode developer with signals and alerts of when and how they should

start focusing on verify security of chaincode in order to increase the software quality.

afuzzer is designed and implemented based on go-fuzz, which collected wisdom of American fuzzy lop(AFL)[9]. Besides the inherited fuzzing architecture, afuzzer integrate chaincode static analysis from previous researches[3, 94], and improved and extended their chaincode security definitions to further increase the analysis accuracy on chaincode security issues. In order to prove the correctness and efficiency of afuzzer , 24 chaincodes are systematically analyzed with the tool. As experimental result, afuzzer can finish all Hyperledger Fabric chaincode analysis with 1 minute and 10 minutes wall time each to reveal all pre-defined chaincode warnings, which are in total of 52 chaincode warnings. At the same time, 20 runtime crashed are found amount these chaincodes, and those that related to official Hyperledger Fabric projects had been reported.

Our main contributions in this article are

- Design fuzzing approach to reveal chaincode vulnerability warnings and runtime chaincode crashes.
- Integrate existing chaincode static analysis as module to enhance fuzzing test process. Static analysis module can optional be stand-alone analysis module during or after the fuzz execution.
- Improved existing chaincode static analysis with more accurate chaincode vulnerability warnings definitions.
- Design input mutation seed prioritizing algorithm with factors of static analysis result and functions which had been fuzzed at runtime.
- Implementation of afuzzer based on fuzzing approach, which is state-of-the-art distributed Hyperledger Fabric chaincode test platform.
- Evaluation of afuzzer with opcode sourced chaincodes. This proves correctness and efficiency of afuzzer

We organized the paper as follows. In Section 4.2, we present how afuzzer works through a motivating example. In Section 4.3 we show the details of fuzz testing that containing inside the

afuzzer . Section 4.5 illustrate the implementation of afuzzer and shows the results on experiment and evaluating afuzzer . Then section 4.6 reviews related work. Finally, Section 4.7 concludes the paper.

4.2 Motivating Example

4.2.1 Chaincode

Hyperledger Fabric chaincode is a smart contract program designed to interactive with ledger state of Hyperledger Fabric blockchain. It can be implemented with various kinds of languages including Go, Java and Node.js. Chaincode runs on peer nodes of blockchain and is called to initial or invoke transactions to update the status of ledger and assets. A typical chaincode is as following:

```
1  import (
2    "github.com/hyperledger/fabric/core/chaincode/shim"
3    pb "github.com/hyperledger/fabric/protos/peer"
4  )
5
6  type ChaincodeAsset struct{}
7
8  func (t *ChaincodeAsset) Init(stub shim.ChaincodeStubInterface) pb.Response{
9      //statements
10 }
11
12 func (t *ChaincodeAsset) Invoke(stub shim.ChaincodeStubInterface) pb.Response{
13     //statements
14 }
```

Figure 4.1: Chaincode necessary interfaces

As shown in Fig. 4.1, to implement a chaincode, there are two interfaces(shows in Fig. 4.1), *Init* and *Invoke*, must be implemented in the chaincode. After that, the chaincode can be installed on the peer node and called through these two implemented functions when user would like to create new transactions. Please note that there are only *Init* and *Invoke* can be called through the commands of Hyperledger Fabric whereas in Ethereum, each functions can be called from external call independently in Ethereum smart contract. In this setting, in order to call other functions in the Hyperledger Fabric chaincode, these functions have to be called through *Init* and/or *Invoke* function. The commonly method is implement a selection statement to control which function to be called by

setting with different values of input arguments.

Being shown the chaincode skeleton, we shows a motivating example to further explain how warning and crash hide inside the chaincode and revealed by our approach.

4.2.2 Motivation Example

In this section, we first introduce the basic of Hyperledger Fabric chaincode. Then we show a illustrative chaincode example that contains chaincode security warnings and implicit runtime crash, and the steps how our approach analyze the example and reveal all warnings and crashes.

During the research process of Hyperledger Fabric chaincode(smart contract), we found vulnerabilities in several examples that come with the Hyperledger Fabric Software Development Kit(SDK) and thus motivated by them. In order to illustrate what our approach can analyze, we designed following chaincode based upon example04.go, one of Hyperledger Fabric SDK official example[1].

Warning and Crash: In the chaincode in Fig. 4.2, there are 4 functions: *Init*, *Invoke*, *query*, and *toChaincodeArgs*. Among these 4 functions, *Init*, *Invoke* and *query* are methods that designed for *ME* struct, which is short name of Motivating Example. According to the design requirement of chaincode, *Init* and *Invoke* are the functions that must be implemented based upon the Hyperledger Fabric chaincode interface. The difference between them is that *Init* will only be called once when chaincode's initialization and *Invoke* could be called multi-times when every time the chaincode is invoked after initialization. Besides these two functions, *query* is the function to call another chaincode with a set of given arguments formalized with help of *toChaincodeArgs* function. Please note that the *shim.ChaincodeStubInterface* is the chaincode required argument for any function in the chaincode, but not necessary to be assigned to the function when invoking any function of the chaincode other than *Init* and *Invoke*.

In the motivating example, there are 2 warnings and 1 crash hidden inside:

- The first warning is at **line 13** where the length of args is not checked and could cause

[1]Found crashes in example04.go as well as other 4 vulnerable chaincodes have been reported to Hyperledger Fabric bug bounty website (https://hackerone.com/hyperledger)

```
1  package MotivatingExample
2
3  import (
4    "strconv"
5    "github.com/hyperledger/fabric/core/chaincode/shim"
6    pb "github.com/hyperledger/fabric/protos/peer"
7  )
8
9  type ME struct{}
10
11 func (t *ME) Init(stub shim.ChaincodeStubInterface) pb.Response {
12   _, args := stub.GetFunctionAndParameters()
13   event := args[0]//warning 1
14   eventVal, _ := strconv.Atoi(args[1])//warning 2
15   err := stub.PutState(event, []byte(strconv.Itoa(eventVal)))
16   if err != nil {
17     return shim.Error(err.Error())
18   }
19   return shim.Success(nil)
20 }
21
22 func (t *ME) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
23   function, args := stub.GetFunctionAndParameters()
24   if function == "query" {
25     return t.query(stub, args)
26   }
27   return shim.Error("Invalid function name")
28 }
29
30 func (t *ME) query(stub shim.ChaincodeStubInterface, args []string) pb.Response {
31   if len(args) > 3 {
32     chainCodeToCall := args[1]
33     queryKey := args[2]
34     channel := args[3]
35     invokeArgs := toChaincodeArgs("query", queryKey)
36     //crash if input is {"Args":["query","a","","",""]}
37     stub.InvokeChaincode(chainCodeToCall, invokeArgs, channel)
38   }
39   return shim.Success([]byte("Query success"))
40 }
41
42 func toChaincodeArgs(args ...string) [][]byte {
43   bargs := make([][]byte, len(args))
44   for i, arg := range args {
45     bargs[i] = []byte(arg)
46   }
47   return bargs
48 }
```

Figure 4.2: Motivating example

uncheckedArgument warning since a none exist member of array may be called. If the input to *Init* is {"*Args*" : []}, then *args[0]* may lead to an **out of bound** error if there's no member in the array of *args[0]*

- The second warning is at **line 14** which is a *unhandledError* warning. The trigger input to this warning could be {"*Args*" : [""," *string*"]}. The return value of *eventVal* will be 0 if the value of *args[1]* is a non-integer string. To prevent this, the second return value of function

strcvonv.Atoi, which is an error, should be handled correctly in order to make sure the value that stored into the ledger is correct in **line 15**.

- The crash happens at **line 37**. When the invoke argument set to {"*Args*" : ["*query*","*a*","","","",""]}, the *Invoke* function will be called and argument will be parsed by the method *GetFunctionAnd-Parameters*. After the execution of **line 23**, the value of *function* will be assigned with *query*. The values of *args* will be {*"a","","","",""*} and then passed into function *query*. After that, the args pass the if statement in **line 32** in function *query* and second element of *args*(*arg[1]*) will be assigned to variable *chainCodeToCall*. Since the value that assigned to *chainCodeToCall* is null, thus there is error in call *stub.InvokeChaincode* as *chainCodeToCall* is its first argument and cannot be null. In this crash, the length of args is guarded by line 31, which can prevent following statements from *unhandledError* warnings. However, there's still a crash at **line 37**. This crash informs us that: A warning may not have to cause a crash and a crash may happen even if the warning has been proved.

Approach Steps: In order to detect all implicitly existing crashes and expose warnings that may lead to crashes, our approach is inspired from [49] and fuzzing test the Hyperledger Fabric chaincode with following 4 main steps:

1. The first step is to generate required policy file. The policy file will be used as the entry of the fuzzing test and assist to validate the mutated-generated input to each chaincode execution.

As shown in Fig. 4.3, there is a snippet of a policy file generated based on aforementioned chaincode in Fig. 4.2. There are 3 functional steps implemented in the policy: *filter invalid fuzz-generated inputs* (line 2-11), *parse fuzz-generated inputs into correct format*(line 12-21) and *invoke chaincode with inputs* (line 23-27). With policy file, our approach can feed mutated-inputs through policy file to the chaincode, and avoid the warnings and crashes that caused by the illegal format of inputs. Please note that we applied *Mock* API comes with Hyperledger Fabric SDK to test each single execution of targeted chaincode since our approach aims to provide an efficient method to fuzzing test chaincode. In this case, our approach tries best to simulate the practical running environment of chaincode and ignored the illegal inputs which will be handled in practical runtime

environment of Hyperledger Fabric blockchain. That's the reason our approach considered and ignored any malicious activities caused by the illegal inputs as they won't happen in the runtime. More details will be given in section 4.3.

2. The second step is to construct the **control flow graph** of chaincode. To figure out if there's any implicit warnings in the chaincode, there are two criteria needed to check: path information for each unique path and path that are executed in runtime. For the uniqueness of each path and to collect path information, it is necessary to build the CFG as reference of the chaincode execution.

The CFG is built statically based on chaincode abstract syntax tree before the chaincode is instrumented. Fig. 4.4 shows the control flow graph of chaincode in Fig.4.2.

Note that there are only two functions that can be called externally: Init and Invoke. The difference of these two functions is that *Init* function can only be called *once* when the chaincode is deployed on Hyperledger Fabric blockchain. On the other hand, *Invoke* could be called every time chaincode is invoked with inputs. This is due to design of chaincode and will affect the control flow graph that shows in 4.4

In Fig. 4.4, it shows all 5 paths that could be executed during the fuzz testing procedure. The CFG consists of *Node*s and *Edge*s. There are 4 types of *Node*: The diamond node with lines inside is the *root* node, which represents the beginning of chaincode execution. The diamond node without inside lines are for *Init* and *Invoke* function as these two functions are entry of the chaincode when it is called. The ellipse node represents the basic blocks in the function and finally, the square node is the terminal node, which is last node and exit of the function. If there's relations between two nodes, they are connected with *Edge*. If one node is the start node of two edges, it shows this specific node has branches that may lead to different child nodes.

Since the call to chaincode can only be through *Init* and *Invoke*, we thus categorize these paths into two call processes. In call process 1, path information that starts with function call *Init* will be collected. Meanwhile, information of warning 1 and warning 2 will be also collected and could be detected potentially if the paths that includes **line 13** and **line 14** are executed in later steps. Similarly, our approach collects path information in call process 2 to analyze any path that walk

through *Invoke* function and monitor if any information are related to potential warnings and crashes.

3. The third step is to instrument source code of chaincode into instrumented program. Our approach walk through abstract syntax tree(ast) of the each file that are consists of target chaincode. The comments are trimmed and a customized label is added into source code and form the instrumented chaincode. For each block of ast, a CoverTab size is given in front of the beginning of the block as instrumentation information. This information will be used to analyze path information as well as coverage rate in following steps. The code snippet of instrumented source code of motivating chaincode example is shown in Fig. 4.5.

4. In the fourth step, instrumented program then is feeded with fuzzing-generated inputs and executed. In each time of execution, if the new mutated input triggers and discovers a new path, then this specific execution path and its associated data will be recorded. This is because every time the different input is feeded into the program, the executed chaincode path may (or not) be same path. Next we could start the execution process and waiting until the wall time or desired paths information are collected. Ideally, all feasible paths are tracked in a given time and fuzzing step can then be terminated.

The aim of step four is to reveal all implicit warnings and generate test cases for any potential crashes in maximized number of covered branched in the chaincode in efficient way. Towards this aim, we designed algorithm to guide seed mutation with feed-back of every previous history chaincode execution record. Feed-back algorithm take given input as initial seed, prioritize the seed which is most promising in mutating new input that can discover new branch in each execution, and generate new input based on selected seed. This algorithm repeatedly monitor the fuzzer to guarantee efficiency of fuzzer until fuzzer's termination.

In the motivating example, we thoroughly present the entire procedure of our approach step by step. In next section, the technical details about our approach will be illustrated in each step.

4.3 Fuzzing Test Approach

In this section, we introduce our state-of-the-art fuzz testing method for Hyperledger Fabric chaincode in details.

4.3.1 Fuzzing Policy Generation

As first step, the target chaincode will be given as input to create policy file, which will contain the necessary data that need by fuzzer in following steps, such as the number and the types of variables that accepted by chaincode. Moreover, a specific template will be applied to generate a policy file which can thus be recognized as fuzzing configuration used by following steps.

However, there are challenges in generating a well-formed, meaningful policy file.

- How to design an entrance to accept input that in the form that chaincode can accept?
- How to invoke chaincode with given input in policy file?

With such consideration, a fuzzing policy is generated with 2 functional parts: a validator for input and interface to call chaincode with given input.

As any inputs will be considered as data in form of *[]byte* due to the versatility consideration of go-fuzz. We adopted the idea of go-fuzz [6] in which there's a *Fuzz* function working as start point and takes all possible inputs that generated through afuzzer in format of []byte, like the variable *data* in line 1 of code snippet in Fig. 4.3.

According to design of the chaincode, each chaincode must implement two interfaces, *Init* and *Invoke*. *Init* will be invoked when chaincode is initialized or upgraded. The *Invoke* function is called when chaincode is invoked to process transaction. As the nature of handling multiple functions as a program and only *Invoke* could be the entry of the chaincode after chaincode initialization, it is non-deterministic for any fuzz input generator to generate a valid invoke input automatically at the beginning of the chaincode test. The format of input of chaincode must strictly follow a specific JSON [10] format. In Fig. 4.6, the snippet of *Invoke* function shows there are only three types of input set can successfully call private functions. For example, input {*"Args":["invoke",args...]*}

will lead the snippet to walk through line 3 and 4 while {*"Args":["query",args...]*} will guide chaincode to the line 7 and 8. However, invalid inputs like {"*Args*" : ["*update*2","*a*","200"]} or {"*Args*2" : ["*query*","*a*","200"]} may also be generated by fuzzer and passed to chaincode during fuzzing process if there's no policy file to filter them, which will eventually decrease fuzzer's efficiency.

Thus, a validator is integrated in the policy file, such as line 2 to line 2 in the policy file in Fig. 4.3, which only allow the input with format that accepted by chaincode pass.

Once the input argument is validated and parsed, *Mock API* is utilized to simulate chaincode execution procedure. Taking Fig 4.3 as example, the chaincode object is created at line 23 and the internal state map of chaincode is initialized through the constructor at line 24. At line 26, there's an optional code to apply to invoke the *Init* before *Invoke* if the Invocation of the chaincode depends on the status of initialization. Otherwise, chaincode could be invoked through line 27 directly.

4.3.2 CFG Generation

After the setup of fuzz policy file and initial corpus, a control flow graph(CFG) [17] is necessary to guide the path data analysis. A CFG consists of 3 elements: nodes, edges and relations between nodes and nodes. Given a chaincode, the CFG will contain all possible chaincode paths information.

Given a chaincode, there are only two functions, Init and Invoke, can be invoked by external calls. We thus construct a graph starts from a abstract *root* node (representing a function call for initializing variables). Then there are always two nodes, *Init* and *Invoke*, are connecting to the *root* node. The *Invoke* node is then connected to nodes representing calling of any of the private functions. Each private function f is then analyzed such that if it calls a function f', there is an edge from f to f'. Any function which can terminate without calling another function is connected to every child of the *root* node, which allows a sequence of call of the private functions. Note that the paths that identified in CFG may not be all feasible in runtime since it is unknown to figure out the branch conditions of chaincode statically.

We take advantage of methods of *go/ast* function provided by golang to get necessary data to contract CFG by default. The algorithm of CFG construction is presented as shown in algorithm 3

Algorithm 3: Algorithm *constructCFG*

```
1  Let G = (N, root, E) be an empty control flow graph
2  Let source be source code of chaincode
3  Let AST be abstract tree of chaincode
4  AST = gen_ast(source)
5  Blocks = parse_blocks(AST)
6  while block ∈ Blocks do
7      while stmt ∈ block do
8          N = update_n(N, stmt)
9          E = update_e(E, N, stmt)
10         if stmt = fst then
11             root = add_r(N, stmt)
12         end
13     end
14 end
15 G = update_g(G)
```

The generation of CFG happens statically before the instrumentation of chaincode. Since the statically process, the CFG is over-approximated comparing with the CFG that constructed dynamically in some cases. For example, there may be only one branch(if branch) could be tracked for a selection statement(if-else-if branches) when CFG is built dynamically with certain input. Such as example in 4.7, else-if branch will not be tracked dynamically as the square of input i is always greater than or equal to 0, which, however, could be added to CFG since effect of input will not be considered if CFG is built statically. To sum up, the CFG built is an over-approximation of the set of reachable paths of chaincode in runtime.

4.3.3 Instrumentation

The third step is chaincode instrumentation. In this step, we adopted go-fuzz solution. Original chaincode and policy file will be given into instrumentation module. Instrumentation module will insert a label for each line and column of code and a instrumented source code of chaincode will be created.

During the procedure of instrumentation, there two types of instrumentation, cover and sonar. Cover instrumentation is for coverage measurement purpose. It will be generated to guide

the process of fuzzing and keep the information of tracked paths. And sonar is to find out all the comparison operations in source code code. When there's comparison operation happens, intercept will keep and pass it into runtime. The purpose of sonar instrumentation can help to mutate the input more effectively with the current input.

As shown in Fig. 4.8, there are two possible paths for the code snippet. One will go with return "Error" when v1 is not equal v2 and another one will go with return "Success". Assuming the current fuzz input is v1 and v1 is not equal to v2, the current tracked path is going with return "Error". If we choose to randomly generate another fuzz input, it probably won't track another path since a new v1 may still not equal to v2. Sonar instrumentation can help to replace the v1 with v2's value from the chaincode and force tracking another path, which will go with return "Success" in next execution.

Eventually, the instrumentation and source code of chaincode will be passed to the next Execution step.

4.3.4 Feedback Guided Fuzzing Mechanism

With instrumented source code of chaincode, execution module will start the fuzzing execution. During the execution step, the most distinguished secret to specific fuzzing engine is its input feed-back guidance. As to our approach, we designed the algorithm to select the seed which will most likely discover new path on top of two-level input mutation strategy inherited and extended from go-fuzz, aiming to achieve a high coverage rate.

The overall work flow of execution is shown in algorithm 4. The input of the algorithm is a set of initial input seed, which is represented as S. With the input, the expecting output are a result set for crashes, C, a result set for warnings, W, and a result set for input seed, which is updated S set. A candidate input seed, *cand*, will be randomly picked from the initial input seed S at the beginning, and algorithm iteratively execute the chaincode and update every candidate input with two-level mutation strategy, $computeMinimization$ and $computeMutation$ respectively, until pre-set time is out or is manually terminated. At line 8, *cand* is feeded into instrumented chaincode, and generate a set of crash c, a set of warning w and a set of path information i as results. All of these three sets

Algorithm 4: Fuzzing execution

```
1  Let C be an empty set for crashes
2  Let W be an empty set for warnings
3  Let S be initial input seed set
4  Let I be information of the discovered paths
5  Let cand be candidate input to next execution
6  cand = randomPick(S)
7  cand.minimized = false
8  while timeout or terminated manually do
9      c, w, i, eureka := exec(cand)
10     C = C ∪ c
11     W = W ∪ w
12     I = I ∪ i
13     if eureka then
14         S = S.insert(cand)
15     end
16     minimizedCand = cand
17     while ! cand.minimized do
18         minimizedCand = computeMinimization(minimizedCand)
19         C, W, I, S = exec(minimizedCand, C, W, I)
20         if minimizedCand == M then
21             cand.minimized = true
22         end
23     end
24     cand.score=computeScore(cand,I)
25     sortBySeedScoreAscOrder(S)
26     cand=computeMutation(S[0])
27 end
28 return C, W, S
```

will be stored into C, W, and I respectively. If there is a new path tracked by *cand*, *eureka* will be *true* which represents a new path is found by executing chaincode with *cand*. When a new path is found, current *cand* will be stored in S. Note that our approach integrated plugin to parse all internal functions in the chaincode except *Invoke* and *Init* in with regular expression method. As the format of chaincode input is strict to a certain type, i.e., $\{\text{"}Args\text{"} : [arg0, arg1, arg2...]\}$, in order to make sure the input could walk through *Init* and *Invoke* and into the function that we would like to test with fuzzer, first argument of the chaincode, such as $arg0$, must be exactly be a case string of switch statement to guarantee appropriate input is generated. As for other arguments in generated input, the

number and values of them are randomly generated to prevent bias caused by inputs in fuzz testing process. Lastly, for each internal function, a certain number of inputs may be generated depending on the user's customization. With this plugin, we reduced the search space for randomly generating initial inputs from scratch and focused on analyzing internal functions ignoring the effects from *Invoke* and *Init* functions.

From line 15 to line 21, there are three functions to generate candidate input for next execution cycle, *computeMinimization*, *computeScore* and *computeMutation*. each candidate will first be mutated by *minimization* way to generate every another new candidate until candidates cannot be minimized, all of the minimized candidates will be feeded in to chaincode for execution and analysis. Then a score will be computed for the candidate that cannot be minimized with function *computeScore*. Every input seed that has been executed with will have a score in factors of crashes, warnings and path information in the path that this input triggered. Computed score will help to decide which input is most promising candidate in terms of discovering a new path of chaincode. After that, the set of inputs S is sorted by scores of inputs and the input with smallest score will be given as mutation seed. With selected input, function *computeMutation* will generate candidate input to next chaincode execution. We will describe details of *computeMinimization*, *computeScore* and *computeMutation* in following paragraphs.

Function *computeMinimization* assists to explore candidate input's potential in terms of discovering new path, crashes and warnings. We adopted go-fuzz minimization theory to minimize candidate input with minimization methods listed in table 4.1 and execute chaincode with each minimized input. There are three methods used to minimize candidate input: remove each individual byte, remove each subset of bytes and replace each individual byte with '0'.

Table 4.1: Minimization of candidate input

computeMinimization(cand) methods
Remove each individual byte
Remove each subset of bytes
Replace each individual byte with '0'

For example of remove each subset of bytes, candidate input {"*Args*" : ["*query*","*a*","200"]} would be firstly minimized into {"*Args*" : ["*query*","*a*"]} , then {"*Args*" : ["*query*"]} and finally{"*Args*" :

[]}. In afuzzer , the minimized input is set to {"*Args*" : []} and *cand.minimized* will be flagged as true at this time. All method will be applied to candidate input to generate new input for execution. Any crashes, warnings and new path information found in executions will also be stored in the C, W, I respectively.

Function *computeScore* is designed to compute a score for each input and evaluate the potential of the seed in terms of tracking a new path, which will eventually discover all crashes and warnings in all paths of chaincode. The score will be used to decide which seed will be selected as mutation candidate to pass to the *computeMutation* function. Therefore, *computeScore* will play key role in our feedback guided fuzz analysis, which needs to be scrutinized carefully and designed finely.

$$Score(cand) = \frac{\Sigma(Times_f / Complexity_f)}{\Sigma(Weight_w * Number_w)}$$

where:

$Score(cand)$ = computed score for current input *cand*
$Times_f$ = accumulated called times of invoked function f
$Complexity_f$ = function complexity of invoked function f by *cand*
$Weight_w$ = weigh of warning of w
$Number_w$ = times of warning w reported in current path

Above algorithm shows how our approach compute scores for each candidate inputs. Basically, the score is affected by the times of functions are called, the complexity of functions, customized weights of each types of warnings and the number of each type of warning that are reported. As described in above algorithm variable definitions, *Score* is proportional to *Times* and is inversely proportionally to another three factors, *Complexity*, *Weight* and *Number*. The thinking behind these relations is: *the candidate input that invoked under-investigated, more complicated functions and reported more warnings should be selected as mutation seed.* This is because through insight regarding to software development, we observed that more complicated function will always

have more number of paths, which needs continuously to be investigated. If the times of high complexity function is less that, it also indicate the specific function needs to be further tested. Nonetheless, the input that report more high-impact warnings should also be prioritized as next mutation input seed. With consideration of this, the smaller score of input get, the more likely this input is selected as mutation seed. Therefore, function *sortBySeedScoreAscOrder* is used to sort the inputs by their scores to facility selecting the input that with smaller score.

More in details, as for *Times*, invoked times of each function will be recorded in each execution. The more a function is invoked, the less likely the input that will invoke this function is selected as mutation seed. To compute the *Complexity*, we adopted cyclomatic complexity [71] as function complexity measurement, which shows as following:

$$Complexity(f) = \begin{cases} 1 & \text{if complexity is 0 initially} \\ Complexity(f) + 1 & \text{if found 'if','for','case','\&\&','||'} \end{cases}$$

Moreover, *Weight* is a set of customizable pre-defined values for each type of warnings. It combined with the number of warning of each type, *Number*, affect the score of input as well. More details about warnings will presents in section 4.4.

Function *computeMutation* is similar with function *computeMinimization* to mutate the given input seed. There are 20 mutation strategies applied in *computeMutation* as methods, which shows in table 4.2. One mutation method will be selected each time randomly to mutate the input. Note that the generated mutation will be discarded if the test case has been generated previously or is a minimized input, i.e., {"*Args*" : []} in afuzzer .

When fuzzing process is terminated, crashes, warnings and input seeds(combined with initial seeds as well as generated seed during fuzzing) will be stored for further analysis. Besides *minimization* and *mutation*'s assistance to mutate the input, there's another structure mutation also

Table 4.2: Mutation of candidate input

computeMutation(cand) methods
Remove a range of bytes.
Insert a range of random bytes
Duplicate a range of bytes
Copy a range of bytes
Random bit flip
Set a byte to a random value
Swap 2 bytes
Add/subtract from a byte
Add/subtract from a uint16
Add/subtract from a uint32
Add/subtract from a uint64
Replace a byte with an random value
Replace an uint16 with an random value
Replace an uint32 with an random value
Replace an ascii digit with another digit
Replace a multi-byte ASCII number with another number
Splice another input
Insert a part of another input
Insert a random literal
Replace a random literal

improved the efficiency of input mutation, which is described in next sub section in details.

4.3.5 Input Mutation

The execution step is iterative since multiple inputs will be generated continuously based on original input corpus, mutation(bitflip) algorithm and chaincode execution feedback. They will be input into the chaincode to check if crash happens. Any crash that happens will be recorded. Note that an input filter is necessary to only pass the satisfied input to the chaincode since go-fuzz does not provide API to generate complicated structure inputs.

Low-level implies in where the valid input will guide the generation of next input. It utilized the metadata from chaincode instrumentation in last step and help to mutate the input in low level, such as the mutation from input of $\{$"*Args*" : $[$"*update*","*a*","200"$]\}$ to input of

{"*Args*" : ["*update*","*a*1","2000"]}, which mutates current input and generate a non-structure mutation.

Besides the lower-lever input mutation based on metadata of instrumentation, there's another high-level input mutation design inside go-fuzz, which name is *Versifier*. *Versifier* is based on the reverse engineering algorithm that can recognize current input and learn from the validation of current input. *Versifier* does four steps:

- Catch current input.
- Figure out if current input is a valid.
- Check out if current input can be take into consideration in increased priority or in lower priority.
- Based on above steps, decide whether add current input to the corpus of fuzz testing, and then mutate the current input with a certain direction.

Assuming a current input is {"$function\_name$","$arg1$","$arg2$"} in a form of $\{string, int, int\}$, *Versifier* detects and stores the current input. *Versifier* will firstly check if current input should be stored into corpus or not based on policy setting. Then *Versifier* will go on to generate next input. If current input is valid, then *Versifier* will more likely mutate next input with current form. Otherwise, if the current input is invalid, then *Versifier* may consider to generate next input in another form, such as $\{string, int, string\}$ or $\{int, int\}$.

With effective input mutation, the procedure of analysis execution is a iterative event. There's a execution report will be printed on screen in a customizable time interval. The time interval is set to 3 seconds by default. At the beginning of the execution, user can set the number of workers for the execution and wall time for terminating the fuzzing execution. Then in each interval, the number of corpus, crashes, restarts and executions will be reported. The cover in the figure represents how many statements are executed in the chaincode so far.

While in the process of fuzz execution of the chaincode, each unique tracked path will be saved into designated location. Our approach can apply warning analysis method on each stored

path information to figure out if there's any warnings. Note that the analysis method does not have to wait until the end of the fuzz execution but generate warning report during the process of fuzz execution.

4.4 Warning Oracle

As a part of path information that stored in the process of execution, a coverage profile is generated for each new discovered path. Coverage profiles are kept in the form of golang coverage profile and can be used directly to generate coverage statistics with go cover tool [4]. The benefit of selecting this form is that it can be natively supported by official golang coverage analysis tool as well as other open source tools that support this format.

Besides the well-supported third-party coverage statistic tools, we followed the factory design pattern and designed a oracle factory to provide 10 types of warnings to analysis path that are executed and tested. Table 4.3 shows the list of vulnerabilities supported to be detected by our approach. The oracle factory also provides extension capability to allow further adding warning detection, working as plugin.

To better illustrate each vulnerabilities, we manually design a chaincode in Fig. 4.9 to present the each warnings and show the positions where the warnings are.

Global Variable Usage Global variables in chaincode are not saved on ledger of blockchain but just belonging to a single node. If this node is down, then the value of global variable will be reset

Table 4.3: List of security warnings

Global Variable	Global variable used in chaincode.
Blacklisted imports	Libraries which allow communication with the outside world, grant file access or can introduce non-determinism are used.
Concurrency of Program	Concurrency that may causes non-deterministic behavior such as race condition exists.
Field Declarations	Field declared at structure of chaincode.
Map range iterations	Map object as key value is returned in a random order in iterating through range operator.
Unhandled Errors	Ignore return values related to errors. Not all errors are handled.
Unchecked Input Arguments	Input arguments are not checked to prevent from out-of-bound error.
Read After Write	Read a value after write may cause reading output inconsistent when read and write are non-atomic.
Range Query Risk	Two functions to call range query cause phantom read of the ledger.
Cross Channel Invocation	Call chaincode across the channels where data are not handled correctly.

and cause inconsistent over all other peers. To prevent from this issue, global variable should not be used to read or write the ledger, or other operations should not depend on the global variables.

For example, *gv0*, *gv1* and *blocks* are global variables that declared in the chaincode. Meanwhile, *gv0* is read and write at line 41 in Fig. 4.9. As result, the read and write location of *gv0* will be reported as *Global Variable Usage* warning.

Blacklisted imports Using external libraries may bring inconsistent into the chaincode, such as *time* and *math/rand*, which will bring a time dependent value and generate random value. respectively.

For example, library *time* is imported into the chaincode and a external function *time.Now()* is used at line 39. This will cause endorsing peers not compute the same result from using a non-determinism function. With inconsistent computation, then the transaction may be invalid.

Concurrency of Program Using *go* currency function in the chaincode may lead to non-deterministic behavior, especially in read and write ledger operation. This results in non-consistent behavior amongst peers. Thus the *go* concurrency function should not be used in the chaincode.

From line 38 and 39 in Fig. 4.9, the concurrency usage of *function1* may cause the data race issue in writing the different value into ledger. In this case, any path that tracked line 38 and 39 will be reported that contains *Concurrency of Program* warning.

Field Declarations The chaincode object is created once when setting up the chaincode at first time. If chaincode object contains fields, then it is possible to sustain a global state in the program. Since these global states are not stored on the ledger, the chaincode object should not contain any fields.

At line 8, *VE* is declared as global chaincode structure, and includes two members, *dummyv* and *dummyw*. These two members may not sustain global state in chaincode because these global states are not stored on ledger. So it may cause inconsistency in the state of *VE*. As a result, a *Field Declarations* warning will be reported with VE's information.

Map range iterations As shown in Fig. 4.2 as well as line 24 in Fig. 4.9, the global map object *blocks* will cause this type of warning because of iteration over the entries of a map is not deterministic in execution of chaincode. Therefore, a *Map range iterations* warning will be warning to the user warning

Unhandled Errors In golang, the output of a function does not have to be saved to certain variable,

shown as line 43 in Fig. 4.9. The *function1*'s return values consists of two error type values. The first error return value is ignored with a underscore at line 43, which represent this value is not necessary in future operation. However, sometimes it is import to capture all errors and set related resolution to errors. So, there's a *Unhandled Errors* warning reported at Line 42 by our approach.

Unchecked Input Arguments For the chaincode invocation, the arguments of invocation is set as an array form and input into the *Invoke* function. If the length of the arguments are not checked with bound, then an out-of-bound error may happens. For example, *args0[2]* is used *function0* at line 42. Since the length of args0 is not guarded to be guaranteed to larger than 3. Then *Unchecked Input Arguments* should be reported as the number of chaincode arguments must be checked before usage to prevent from out-of-bound error.

Read After Write *Read After Write* warning is actually a data dependence issue which is due to the read and write operation is not atomic. The operation through *PutState* or *GetState* will always be through a transaction. In this case, *GetState* may return the old value from the ledger. For example, in line 21 and 22 of Fig. 4.9, *PutState* write a key-value pair to the ledger and *GetState* read this key-value pair from ledger. However, because they are in the same transaction, a new value caused by *PutState* may not affect the return value of *GetState* and *GetState* will return a old value. In this case, a *Read After Write* warning will be reported.

Range Query Risk The data reading from chaincode API GetHistoryOfKey or GetQueryResult will cause phantom reads, which will cause inconsistancy if the data is used to update ledger. For example, at line 19 and line 21, the value of *data* is coming from the *GetHistoryOfKey* function, it should not be written on the ledger. In this case, the *Range Query Risk* warning will be reported.

Cross Channel Invocation *channel* value in InvokeChaincode(chaincodeName string, args [][]byte, channel string) should not be any arbitrary value except value of GetChannelID(). Otherwise same channel chaincode call will not be guaranteed. If channel is not the same channel of caller chaincode, the transaction is just a "Query" and any state update that are made through chaincode will not work properly.

For example, there is a Cross Channel Invocation warning at line 56. If the value of *channel* is set to the value other than the value of channel of current chaincode itself. Thus current chaincode

across the different channels to call another chaincode through *stub.InvokeChaincode*, which may cause error on the global states.

For each chaincode, following outputs will be provided as a result set: coverage profile, warnings statistics, includes numbers of each types and locations, and crashes profile, includes trigger input argument and panic stack information.

4.5 Experiment and Evaluation

Based on the design of our approach, we implemented afuzzer , which consists of 4 main module components: *Policy generation module, Instrumentation module, Fuzz execution module and Warning analysis module*. Fig. 4.10 shows general structure of the tool. Each module has following functions:

- *Policy generation module*: Generate policy file according to the chaincode source code.
- *Instrumentation module*: Generate control flow graph and instrumentation source code and binaries.
- *Fuzz execution module*: Execute chaincode with mutated inputs iteratively, generate path information and store crashes information.
- *Warning analysis module*: Analyze warnings in each path and coverage rate of fuzzing test.

In order to evaluate the contribution of afuzzer , we designed the experiment to evaluate afuzzer and answer following research questions(RQ) in order to guarantee its robustness, soundness, efficiency and effectiveness.

- RQ1: How's the efficiency of afuzzer ? Fuzz testing is a test method focusing on generating valid/reasonable inputs as faster as possible. In this case, testing efficient is the first important standard that needs to be evaluated.

- RQ2: How's the effectiveness of afuzzer ? High coverage rate is another evaluation standard that we care about. High coverage rate implies high percentage of the chaincode has been tested. In order to find more implicit vulnerabilities and warnings, we designed this RQ and would like to answer by evaluating the testing coverage rate of each chaincode.

- RQ3: Will the afuzzer find out crashes and give sufficient warnings to users?

Experiment Setting We did the experiment on a platform of 64bit MacOS 10.14 with Hyperledger Fabric golang SDK 1.4 version and golang 1.12.1 version. From hardware perspective, all experiment has been done with an Intel Core i5-8400(4.0 GHz, 6 cores) and 16 GB of DDR4 2600 RAM.

Experiment benchmarks In order to answer above RQs and evaluate afuzzer , we collected Hyperledger Fabric chaincodes from multiple sources. Due to the nature of Hyperledger Fabric as consortium blockchain, there's no reliable method to collect all of existing chaincodes on all Hyperledger Fabric chains. For the purpose of collect as many chaincodes as possible, we focused on collecting the official examples and demonstration cases that are designed and contributed by the active entrepreneurs supporters and chaincode developers. We collect and manage these chaincodes as a benchmark suite as they has high possibility in being used in private Hyperledger Fabric chains. As a result, we collected 24 chaincode benchmarks and organized these chaincodes as shown in table 4.4. Noted that these tested chaincodes includes the cases that consists of single chaincode and that consists of multiple chaincode files. We continued to analyze the chaincode benchmarks that consists of multiple files for the purpose of evaluation.

In Fig. 4.4, the first and second columns on the left lists indexes and names of the chaincodes that are fuzz tested with afuzzer . All of them are collected through 3 sources: official Hyperledger Fabric chaincode tutorials offered by IBM [8], general open source libraries and most forked chaincode projects on open source platform. We managed them into a chaincode set as our benchmark suite, which is utilized to evaluate afuzzer and answer RQs.

RQ1: Efficiency: To evaluate the efficiency of afuzzer , we firstly tested benchmark suite in two

Table 4.4: Analysis report on evaluated chaincodes

Index	Name	Explored paths		Coverage rate		Crashes		Warnings		Initial inputs	LOC
		1 min	10 min	1 min	10 min	1 min	10 min	1 min	10 min		
1	cashloan/cash	17	49	73.00%	80.90%	0	0	0	0	3	283
2	cashloan/loan	10	10	75.30%	75.30%	2	2	0	0	2	166
3	crowdfunding/	10	10	80.20%	80.20%	0	0	5	5	5	259
4	drug/	10	10	65.40%	65.40%	0	0	0	0	5	235
5	eventsender/	6	6	71.90%	71.90%	0	0	0	0	2	91
6	example01/	2	5	73.30%	83.30%	0	0	5	8	1	72
7	example02/	6	12	79.60%	87.20%	0	0	0	0	3	177
8	example03/	2	5	76.50%	85.30%	0	0	0	0	1	81
9	example04/	9	11	75.60%	78.00%	1	2	0	0	3	155
10	example05/	8	8	49.00%	49.00%	2	2	0	0	2	196
11	fabcar/	10	10	74.30%	74.30%	0	0	1	1	5	203
12	farm/	51	127	47.20%	51.90%	0	0	3	3	10	745
13	maps/	60	86	59.40%	59.40%	6	6	7	7	15	464
14	marbles_chaincode/	41	267	51.20%	53.80%	1	1	3	3	13	754
15	morpheo/	32	77	62.40%	64.20%	0	0	8	8	10	1059
16	mrtgexchg/	37	156	69.10%	79.60%	2	2	1	1	14	722
17	passthru/	5	5	76.50%	76.50%	1	1	0	0	2	71
18	sleeper/	6	8	78.40%	78.40%	0	1	2	2	2	123
19	fabric_test/maps.go	14	18	56.10%	58.20%	1	1	4	4	6	203
20	fabric_test/mapkeys.go	26	37	54.70%	56.00%	1	2	6	6	11	305
21	fabric_test/samplecc/	7	9	82.10%	82.10%	0	0	0	0	2	180
22	fabric_test/sbe/	39	39	75.50%	75.50%	0	0	3	3	3	150
23	fabric_test/shim-vendored/	9	9	61.20%	61.20%	0	0	0	0	3	237
24	fabric_test/shimApiDriver/	9	9	81.40%	81.40%	0	0	1	1	5	212

group of wall time limit, 1 minute and 10 minutes, in order to check out two hypothesis: Will the number of explored paths increase with increased time period? Can afuzzer could explore many paths of the program in a minimal time period? To check these two hypothesis, based on data on the *Explored paths* and *Coverage rate* columns, the statistics of experiment are presented and grouped with wall time length on both columns. Note that the coverage rate represent the statement coverage rate which collected based on the explored paths information.

For all the chaincodes in Fig. 4.4, the number of explored paths are all identical or increased when the wall time extended from 1 minute to 10 minute. The increment of explored paths numbers represent afuzzer can guarantee test and explore the chaincode continuously. Meanwhile, for the chaincodes No.*2, 3, 4, 5, 10, 11, 17, 22, 23, 24*, the number of explored paths are same in both 1 min and 10 min wall time setting and the value of coverage rate are same too. This situation indicate that afuzzer can finish testing these chaincodes in minimal time since the coverage rate won't increase even if the wall time extended. Thus, afuzzer can efficiently explore as many paths as possible in a minimal time period.

Moreover, for the chaincodes No.*13, 18, 21*, the number of explored paths are increased while the coverage rate keep identical. This case represent that the statement of chaincode has been explored thoroughly but the branch coverage are still increasing. Following figure shows a small example to further explain this case:

Assuming there are two group of inputs: a=1, b=1 and a=-1,b=-1, thus the line *1,2,6,7* are explored with a=1, b=1 and the line *3,4,8,9* are explored with a=-1, b=-1. At the end, all of the statements are explored and statement coverage rate is 100%. However, if we continue to provide another group of inputs, a=1, b=-1, then the line *1,2,8,9* are explored. A new path is explored while the statements explored keep the same.

With the explanation of above examples, afuzzer can continuously and efficiently test the chaincode even if statements has been explored.

RQ2: Effectiveness: We evaluate the effectiveness based on the value of coverage rate of each chaincode. Fig. 4.11 shows the coverage rate for each chaincode in 1 minute and 10 minute time frame. The average of coverage rate is 68.64% for 1 minute and 71.21% for 10 minutes, which presents quite reasonable effectiveness of afuzzer .

RQ3: Crashes and Warnings: According to the statistics in Fig.4.4, afuzzer can detect crashes and give warnings to user in a timely manner like 1 minute and also could find out the bug that hided deeply in the chaincode in longer time frame, like 10 minutes. To further analysis the features of these crashes and warnings, we categorize them, respectively, according to the types and definitions described in section 4.4 and show the result in table 4.5 and table 4.6. Note that the warning categories that does not have any warnings in chaincodes are ignored.

As the date shown in table 4.5, for most of crashes generated during the experiment, we found they are caused by two reason: 1, for the index out of range crashes, it commonly happens when one of argument in argument array is read without checking its existence or length of the argument array. This crash reason is closely connected with the *Unchecked Argument* warning. 2, for the invalid memory address or nil pointer dereference caused crashes, the reason behind it is that

```go
func Fuzz(data []byte) int {
  if len(data) == 0 {
    return -1
  }
  var inp interface{}
  if err := json.Unmarshal(data, &inp); err != nil {
    return -1
  }
  if inp.(type) != map[string]interface{} {
    return -1
  }
  mp := inp.(map[string]interface{})
  argmp := make(map[string]interface{})
  for a := range mp {
    argmp[strings.ToLower(a)] = mp[a]
  }
  _, argsPresent := argmp["args"]
  _, funcPresent := argmp["function"]
  if !argsPresent || (len(mp) == 2 && !funcPresent) || len(mp) > 2 {
    return 0
  }

  scc := new(ME)
  stub := shim.NewMockStub("chaincodeName", scc)
  //optional if Init needs test or Invoke depends on initialization
  stub.MockInit("uuid", [][]byte{[]byte("Init"), []byte("a"),[]byte("200")})
  stub.MockInvoke("uuid", toArgsByteArray(argmp))

  return 1
}
```

Figure 4.3: Policy file for Motivating example

Table 4.5: Crash result

Name	index out of range		invalid memory address or nil pointer dereference		SIGABRT		Total	
	1min	10min	1min	10min	1min	10min	1 min	10 min
cashloan/loan	1	1	1	1	0	0	2	2
example04/	0	0	1	2	0	0	1	2
example05/	0	0	1	2	0	0	2	2
maps/	4	4	2	2	0	0	6	6
marbles_chaincode/	1	0	1	1	0	0	1	1
mrtgexchg/	0	0	2	2	0	0	2	2
passthru/	0	0	1	1	0	0	1	1
sleeper/	0	0	0	0	0	1	0	1
fabric_test/maps.go	1	1	0	0	0	0	1	1
fabric_test/mapkeys.go	1	2	0	0	0	0	1	2
Total	8	8	9	11	0	1	17	20

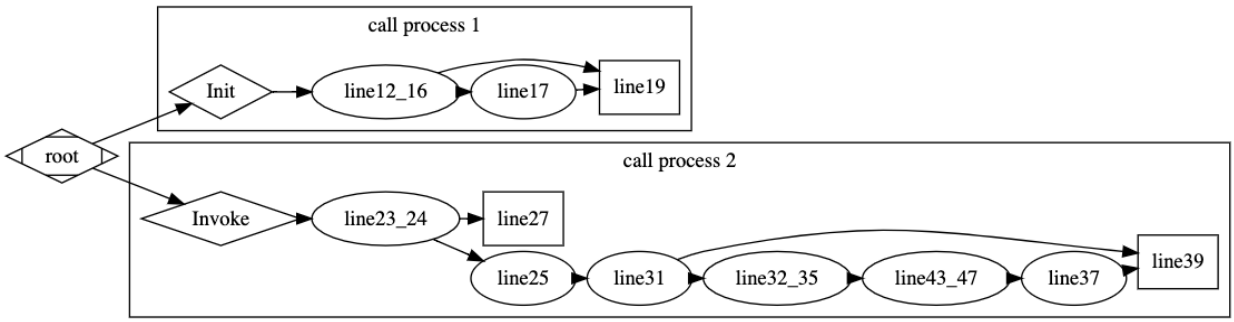


Figure 4.4: Control flow graph of chaincode in Fig. 4.2

Table 4.6: Warning result

Name	GlobalVariable Usage		Blacklisted Import		Unhandled Error		Unchecked Argument		ReadAfter Write		RangeQuery Risk		Total	
	1min	10min	1min	10min	1min	10min	1min	10min	1min	10min	1min	10min	1min	10min
crowdfunding/					5	5							5	5
example01/	5	8											5	8
fabcar/							1	1					1	1
farm/			1	1							2	2	3	3
maps/			1	1	3	3			3	3			7	7
marbles_chaincode/			1	1							2	2	3	3
morpheo/					5	5	3	3					8	8
mrtgexchg/			1	1									1	1
sleeper/			1	1	1	1							2	2
fabric_test/maps.go			1	1	1	1					2	2	4	4
fabric_test/mapkeys.go			1	1	2	2					3	3	6	6
fabric_test/sbe/	1	1					2	2					3	3
fabric_test/shimApiDriver/			1	1									1	1
Total	6	9	8	8	12	12	6	6	3	3	9	9	44	47

although the existence of the read argument is checked, developer did check if *null* can accepted and transfer to the function.

From table 4.6, the largest number of warnings appear in unhandled error category, which means it is quite common for chaincode developer to ignore handling all possible errors in the chaincode program.

4.6 Related Works

With the closer attention to smart contract testing techniques from researchers, related researches are recently proposed and published in smart contract verification area. In our approach, we adopted fuzzing test and introduced it into smart contract verification research area. To the best of

```
func (t *ME) query(stub shim.ChaincodeStubInterface, args []string) pb.Response {
  _afuzzer_dep_.CoverTab[51928]++
  if len(args) > 3 {
    _afuzzer_dep_.CoverTab[56144]++
    chainCodeToCall := args[1]
    queryKey := args[2]
    channel := args[3]
    invokeArgs := toChaincodeArgs("query", queryKey)

    stub.InvokeChaincode(chainCodeToCall, invokeArgs, channel)
  } else {
    _afuzzer_dep_.CoverTab[27618]++
  }
  _afuzzer_dep_.CoverTab[61116]++
  return shim.Success([]byte("Query success"))
}
```

Figure 4.5: Snippet of instrumented chaincode in Fig. 4.2

```
func (t *ME) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
  function, args := stub.GetFunctionAndParameters()
  if function == "invoke" { //{"Args":["invoke",args...]}
    return t.invoke(stub, args)
  }else if function == "update"{//{"Args":["update",args...]}
    return t.update(stub, args)
  }else if function == "query"{//{"Args":["query",args...]}
    return t.query(stub, args)
  }
  return shim.Error("Invalid function name")
}
```

Figure 4.6: inputs that can call private functions

```
//Assuming input i is in type of integer
if i * i >= 0 {
  //statement 1
}else if i * i < 0  {
  //statement 2
}
```

Figure 4.7: difference between statical and dynamical CFG generation

```
if v1 != v2 {
  return "Error"
}
return "Success"
```

Figure 4.8: Example for sonar instrumentation

```go
import (
    "fmt"
    "github.com/hyperledger/fabric/core/chaincode/shim"
    "github.com/hyperledger/fabric/protos/peer"
    "time"
)

type VE struct {
    dummyv string
    dummyw string
}

var blocks = map[int]int{1:1,2:5,3:10,4:50,}

var gv0 int = 111
var gv1 = 222

func function1(stub shim.ChaincodeStubInterface, data string)(e0 error,e1 error){
    iterator,_ := stub.GetHistoryForKey("key")
    data, _ := iterator.Next()
    stub.PutState("key", []byte(data.Value))
    stub.GetState("key")
    returnValue := 0
    for i, ii := range blocks{
        returnValue = returnValue * i - ii
    }
    args1 := stub.GetStringArgs()
    if len(args1) < 1 {
        fmt.Println("working guard")
        return e1, e0
    }
    _, e1 = stub.GetState(args1[2])
    return e0, e1
}

func (t *VE) function0(stub shim.ChaincodeStubInterface) {
    args0 := stub.GetStringArgs()
    go function1(stub, "data1")
    go function1(stub, "data2")
    fmt.Println(time.Now())
    gv0 = gv0 + 1
    resultT, _ := stub.GetState(args0[2])
    _ , e1 := function1(stub, "data")
    fmt.Println(e1)
}

func (m *VE) Init(stub shim.ChaincodeStubInterface) peer.Response {
    return shim.Success([]byte("Init Success"))

}

func (m *VE) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    m.function0(stub)
    _, args := stub.GetFunctionAndParameters()
    chainCodeToCall := args[1]
    channel := args[2]
    stub.InvokeChaincode(chainCodeToCall, [][]byte{}, channel)
    return shim.Success([]byte("Invoke Success"))
}

func main() {
    if err := shim.Start(new(VE)); err != nil {
        fmt.Printf("Error starting chaincode: %s", err)
    }
}
```

Figure 4.9: Vulnerabilities explaination example

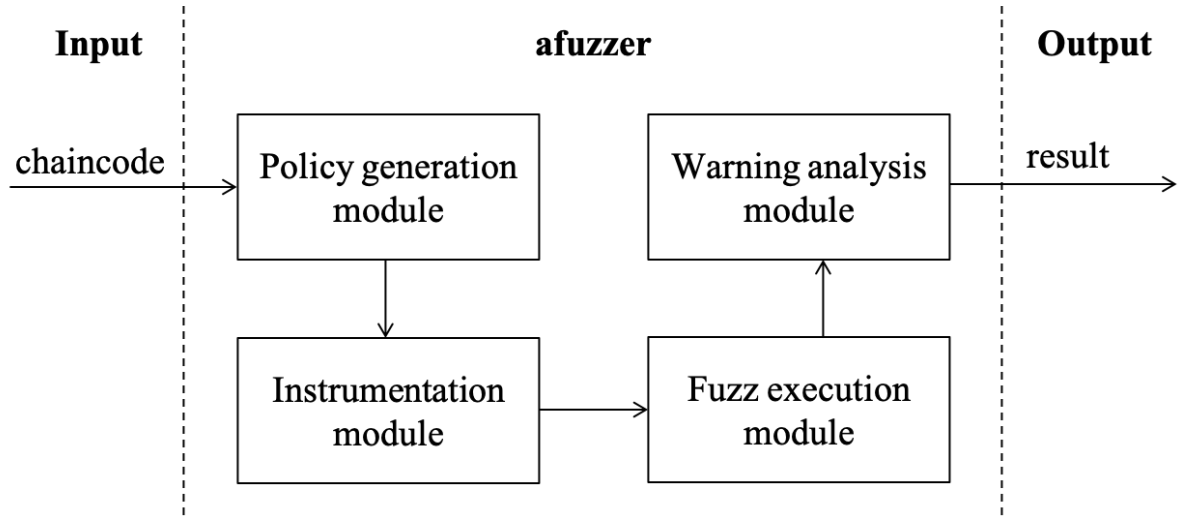


Figure 4.10: Overall workflow of afuzzer

```
if a > 0 {
  statement1
}else {
  statement2
}
if b > 0 {
  statement3
}else {
  statement4
}
```

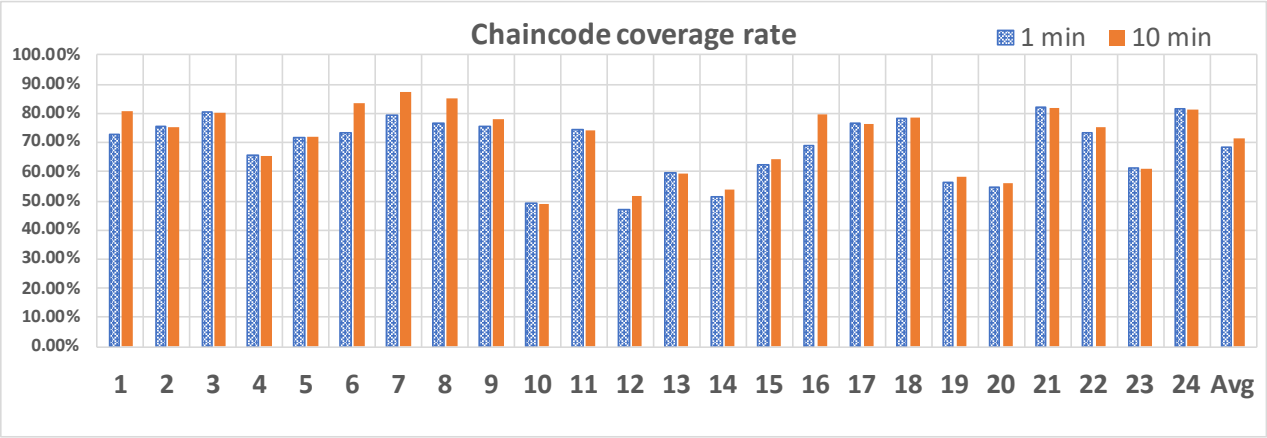


Figure 4.11: Chaincode coverage rate

knowledge, we systemly looked in the research works that focusing on (1)fuzzing test[31, 48, 49, 69] and (2)smart contract verification approaches, especially those works that aims to verify Hyperledger Fabric chaincode.

Fuzzing [84] test's aim to execute target program and monitor runtime status by feeding

inputs that are generated during the executions. The most challenges are (1)how to generate effective inputs and (2) how to increase the code coverage. American fuzzy lop(AFL) [95] is tool designed to solve these two questions and has been proved as efficient and extended as basis to multiple toolsbohme2017coverage,bohme2017directed. To generate effiective inputs, AFL maintains a minimum input queue by trimming the inputs that will not change the program behavior. New inputs are mutated iteratively through customized fuzzing strategies and pushed into the input queue. If any input triggers a new path that is not explored, then related output will be recorded. This coverage-oriented fuzzing test theory inspired following fuzzing test tools as well as our approach, to test different programs on various platforms. afuzzer take advantages of mutation based fuzzing test tool [6] focusing on golang that based on AFL theory. Beyond that, we introduced new method to validate the inputs and integrated vulnerability warning into the afuzzer for the purpose of providing one-stop fuzzing test service to users.

Smart contract theory is introduced two decades ago[85] and brought into public sightseeing since the popularity of bitcoin[73] raised. Most recent smart contract testing and verification researches are aggregating on Ethereum[28] platform with formal verification techniques, including symbolic execution and fuzz testing. Luu *et al.*[68] for the first time introduced symbolic execution method into smart contract verification. In this approach, loop interations and relations among functions are not considered during the symbolic execution procedure. Zeus[58] persuaded translating smart contract into llvm[64] and thus can be integrated with llvm based program verification tool to test smart contracts. Similar research logic, translate smart contract into other language then tested with existing tool, is also applied in other tools[21, 52]. Besides symbolic execution, fuzzing test is another practical method in testing vulnerabilities of smart contract. ContractFuzzer[55] and echidna[5] are utilizing fuzz testing techniques to verify Ethereum smart contracts. In contractFuzzer, fuzz testing is introduced to test smart contract for the first time, and 7 types of vulnerabilities can be detected. However, inputs are totally generated based on the source code and no input improvement could be guided to increase coverage rate due to lack of input mutation analysis.

For the perspective of Hyperledger Fabric chaincode testing, there are few researches presents

practical verification methods integrating with formal verification techniques. As references of analysis of chaincode security, in [3], authors presents tool to static analyze chaincode to investigate 9 types of vulnerabilities and send warnings to users. This is research provide insights to analyze dynamic analysis of chaincode and its theory is inherited by [94], which extend the detection capability beyond the investigated types of vulnerabilities. To the best of our knowledge, there's no other fuzzing test basis, even formal method basis chaincode verification platform existing or proposed yet.

4.7 Conclusion

In this work, we propose the first Hyperledger Fabric chaincode fuzzer to bring insight to chaincode developer with signals and alerts of when and how they should start focusing on verify security of chaincode in order to increase the software quality.

Chapter 5

Conclusion and Future Works

In the researches that are throughout my Ph.D. years, I always focus on the software qulity and verification. Software quality should always be top priority in the life cycle of the software development.

In the research of memory reuse distance, We repurpose PCT algorithm to mitigate path explosion problems in analyzing concurrent program and take its advantage to measure memory distance in an effective way with probabilistic guarantees and without losing its generality.

Blockchain is gradually becoming a hot research topic in research area, especially the software quality of smart contract. Smart contract is a evolutional software category built upon the blockchain platform, which has been well-accepted in voting, crowdfunding and broadcasting area due to information transparency and traceability. However, in most recent years, there are plenty of attacks that caused huge economic losses. The notorious DAO attack make the Ethereum owner suffer a loss of $60 M. More hazardously, the same type of attack that had been discovered and happened, which cannot be prevented completely. With insight of huge potential and market caps, I committed time into this promising research area and is concentrating on following research projects.

Hyperledger fabric is a another promising blockchain platform that maintained by Linux foundation. It has great features on distribution and privacy which are the reason of being widely accepted and applied in business industry. It's smart contract, chaincode, designed with upgradeable

and high maintainable status, which is also extensively utilized and integrated into the business system.

With the development and adoption of Hyperledger fabric, there's incremental demands for the security analysis. However, the security of Hyperledger fabric unfortunately, hasn't not gain much attention from neither industry nor research area. Noticing this surging security requests and driven by curiosity of software security research on chaincode, I developed an approach for automated analyzing chaincode vulnerabilities with fuzz testing, which is proved as a quality software testing technique to discover software security issues.

To the best of our knowledge, I didn't find any effective fuzzing analysis tool that can truly verify chaincodes security issues and detect chaincode vulnerabilities and crashes. There are researches that focused on static analysis for Hyperledger fabric chaincode. However, there're natural limitations to statically analyze program as static analysis cannot prove the found warnings really exist in runtime or could lead to runtime bug in practical. Moreover, static analysis also bring high amount of false positive and false negative in analysis result.

In the future, my current researches can be extended into new platform such as Libra smart contract test with static analysis and symbolic execution technique

Libra is another blooming blockchain platform introduced by Facebook, aiming to establish a safe, unbounded crypto-currency exchange platform, which will assist research and applications from any areas.

This project is, on one hand, very promising and worth research due to the increment of smart contract vulnerabilities and necessities of the audit verification, and application prospects as introduced by Facebook, who taken up large share of market and embraced 2.45 billion monthly active users in 2019. On the other hand, to the best of our knowledge, there's no any existing tools focusing on libra smart contract, which is written by move.

I propose a state-of-the-art libra smart contract testing framework with integration of static analysis and symbolic execution technique. This framework will automatically explore the security issues hidden inside the smart contracts and will definitely help both researchers and developers to build more safely smart contract, bring broader impact on the research on smart contract

vulnerabilities verification and attracts the investment from both scientific foundation and industries.

In testing concurrent software program, the difficulties are that I couldn't predict the inputs of the concurrent software program and which path or state will the concurrent software program will be in. So It is very hard to test a concurrent program because not only inputs have to be enumerated exhaustively but also program path interleaving states are very hard to predict.

In my future research plans, I will focus on blockchain cybersecurity area and other interesting research areas that can assist solving questions in software testing area better and more effectively. Contributing on current research projects, blockchain smart contract testing will be my research topic in next three years in order to generate applicable software testing application for most financially promising blockchain platforms and attract external funding from national funding supporter as well as industry companies. Besides considerations of research financial support, I will also evaluate my researches' impact to my community research institutes in terms of education and research activities, such as holding and joining research conferences. I will make contributions to my beloved software verification research area in the future.

Bibliography

[1] Another parity wallet hack explained. https://medium.com/@Pr0Ger/another-parity-wallet-hack-explained-847ca46a2e1c. (Accessed on 06/06/2018).

[2] Attack - the dao - the dao. https://daowiki.atlassian.net/wiki/spaces/DAO/pages/7209155/Attack. (Accessed on 06/12/2019).

[3] Chaincode scanner. https://chaincode.chainsecurity.com/. (Accessed on 09/02/2019).

[4] The cover story - the go blog. https://blog.golang.org/cover. (Accessed on 10/19/2019).

[5] crytic/echidna: Ethereum fuzz testing framework. https://github.com/crytic/echidna/. (Accessed on 10/28/2019).

[6] dvyukov/go-fuzz: Randomized testing for go. https://github.com/dvyukov/go-fuzz. (Accessed on 09/09/2019).

[7] Ethereum (eth) blockchain explorer. https://etherscan.io/. (Accessed on 06/30/2018).

[8] Hyperledger fabric tutorials – ibm developer. https://developer.ibm.com/components/hyperledger-fabric/tutorials/. (Accessed on 10/20/2019).

[9] lcamtuf.coredump.cx/afl/technical_details.txt. http://lcamtuf.coredump.cx/afl/technical_details.txt. (Accessed on 01/09/2020).

[10] Rfc 8259 - the javascript object notation (json) data interchange format. https://datatracker.ietf.org/doc/rfc8259/. (Accessed on 10/19/2019).

[11] Self-driving uber car that hit and killed woman did not recognize that pedestrians jaywalk. https://www.nbcnews.com/tech/tech-news/self-driving-uber-car-hit-killed-woman-did-not-recognize-n1079281. (Accessed on 05/09/2020).

[12] Smart contract security: Part 1 reentrancy attacks — hacker noon. https://hackernoon.com/smart-contract-security-part-1-reentrancy-attacks-ddb3b2429302. (Accessed on 05/09/2020).

[13] Solidity, the contract-oriented programming language. https://github.com/ethereum/solidity. (Accessed on 06/12/2019).

[14] Therac-25 - wikipedia. https://en.wikipedia.org/wiki/Therac-25. (Accessed on 05/09/2020).

[15] Top 50 cryptocurrency prices—coinbase. https://www.coinbase.com/price. (Accessed on 06/17/2019).

[16] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[17] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[18] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–381. Springer, 2008.

[19] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018.

[20] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust*, pages 164–186. Springer, 2017.

[21] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 91–96. ACM, 2016.

[22] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 91–96. ACM, 2016.

[23] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite:characterization and architectural implications. In *International Conference on Parallel Architecture and Compilation Techniques*, pages 72–81, 2008.

[24] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, Franois Yergeau, et al. Extensible markup language (xml) 1.0, 2000.

[25] Jerry Brito and Andrea Castillo. *Bitcoin: A primer for policymakers*. Mercatus Center at George Mason University, 2013.

[26] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, pages 167–178, 2010.

[27] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ACM Sigplan Notices*, volume 45, pages 167–178. ACM, 2010.

[28] Vitalik Buterin et al. Ethereum white paper. *GitHub repository*, pages 22–23, 2013.

[29] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37), 2014.

[30] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. pages 209–224, 2008.

[31] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[32] Yan Cai and Zijiang Yang. Radius aware probabilistic testing of deadlocks with guarantees. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 356–367, 2016.

[33] Jialiang Chang, Bo Gao, Hao Xiao, Jun Sun, Yan Cai, and Zijiang Yang. scompile: Critical path identification and analysis for smart contracts. In *International Conference on Formal Engineering Methods*, pages 286–304. Springer, 2019.

[34] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: a software testing service. *Operating Systems Review*, 43(4):5–10, 2009.

[35] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[36] ConsenSys. Mythril: Security analysis of ethereum smart contracts. https://github.com/ConsenSys/mythril, 2018. [online, accessed 30-may-2018].

[37] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[38] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. pages 337–340, 2008.

[39] Pierre A Devijver and Josef Kittler. *Pattern recognition: A statistical approach*. Prentice hall, 1982.

[40] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, September 2006.

[41] Whitfield Diffie and Martin E. Hellman. Multiuser cryptographic techniques. In *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*, AFIPS '76, pages 109–112, New York, NY, USA, 1976. ACM.

[42] Chen Ding and Yutao Zhong. Reuse distance analysis. *University of Rochester, Rochester, NY*, 2001.

[43] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 245–257, 2003.

[44] Joe W Duran and Simeon Ntafos. A report on random testing. In *Proceedings of the 5th international conference on Software engineering*, pages 179–183. IEEE Press, 1981.

[45] Joe W Duran and Simeon C Ntafos. An evaluation of random testing. *IEEE transactions on Software Engineering*, (4):438–444, 1984.

[46] Changpeng Fang, S Carr, Soner Onder, and Zhenlin Wang. Instruction based memory distance analysis and its application to optimization. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 27–37. IEEE, 2005.

[47] Changpeng Fang, Steve Carr, Soner Önder, and Zhenlin Wang. Reuse-distance-based miss-rate prediction on a per instruction basis. In *Proceedings of the 2004 Workshop on Memory System Performance*, MSP '04, pages 60–68, 2004.

[48] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[49] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166. Citeseer, 2008.

[50] Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. Assertion guided symbolic execution of multithreaded programs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 854–865. ACM, 2015.

[51] Stuart Haber and W Scott Stornetta. How to time-stamp a digital document. In *Conference on the Theory and Application of Cryptography*, pages 437–455. Springer, 1990.

[52] Yoichi Hirai. Formal verification of deed contract in ethereum name service. *November-2016.[Online]. Available: https://yoichihirai. com/deed. pdf*, 2016.

[53] William E. Howden. Symbolic testing and the dissect symbolic evaluation system. *IEEE Transactions on Software Engineering*, (4):266–278, 1977.

[54] Dorota Huizinga and Adam Kolawa. *Automated defect prevention: best practices in software management*. John Wiley & Sons, 2007.

[55] Bo Jiang, Ye Liu, and WK Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. *arXiv preprint arXiv:1807.03932*, 2018.

[56] Yunlian Jiang, Eddy Z Zhang, Kai Tian, and Xipeng Shen. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *International Conference on Compiler Construction*, pages 264–282. Springer, 2010.

[57] Norman D Jorstad and TS Landgrave. Cryptographic algorithm metrics. In *20th National Information Systems Security Conference*, 1997.

[58] Sukit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharmar. Zeus: Analyzing safety of smart contracts. In *Network and Distributed Systems Security Symposium 2018*, pages 1–12. internetsociety, 2018.

[59] Stephen H Kan. *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[60] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. Cache replacement based on reuse-distance prediction. In *Computer Design, 2007. ICCD 2007. 25th International Conference on*, pages 245–250. IEEE, 2007.

[61] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[62] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[63] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.

[64] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[65] Hao Li, Jialiang Chang, Zijiang Yang, and Steve Carr. Memory distance measurement for concurrent programs. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 49–64. Springer, 2017.

[66] Tongping Liu, Charlie Curtsinger, and Emery D Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 327–336. ACM, 2011.

[67] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[68] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.

[69] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: an input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234, 2013.

[70] Gabriel Marin and John Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 2–13, 2004.

[71] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[72] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[73] Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008.

[74] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016.

[75] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. *arXiv preprint arXiv:1802.06038*, 2018.

[76] Qingpeng Niu, James Dinan, Qingda Lu, and P Sadayappan. Parda: A fast parallel reuse distance analysis algorithm. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1284–1294. IEEE, 2012.

[77] Parag C Pendharkar, James A Rodger, and Girish H Subramanian. An empirical study of the cobb–douglas production function properties of software development effort. *Information and Software Technology*, 50(12):1181–1188, 2008.

[78] Roger S Pressman. *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005.

[79] Derek L Schuff, Milind Kulkarni, and Vijay S Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 53–64. ACM, 2010.

[80] Koushik Sen. Scalable automated methods for dynamic program analysis. Technical report, 2006.

[81] Xipeng Shen, Jonathan Shaw, Brian Meeker, and Chen Ding. Locality approximation using time. In *ACM SIGPLAN Notices*, volume 42, pages 55–61. ACM, 2007.

[82] Xipeng Shen, Yutao Zhong, and Chen Ding. Locality phase prediction. *ACM SIGPLAN Notices*, 39(11):165–176, 2004.

[83] D. H. Stamatis. *Failure Mode and Effect Analysis: FMEA from Theory to Execution*. ASQ Quality Press, 2003.

[84] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. 2007.

[85] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[86] Ari Takanen, Jared D Demott, Charles Miller, and Atte Kettunen. *Fuzzing for software security testing and quality assurance*. Artech House, 2018.

[87] trailofbits. Manticore: Symbolic execution tool. https://github.com/trailofbits/manticore, 2018. [online, accessed 30-may-2018].

[88] Petar Tsankov, Andrei Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. *arXiv preprint arXiv:1806.01143*, 2018.

[89] Alan M Turing. On computable numbers, with an application to the *Entscheidungsproblem*. *Proc. London Math. Soc*, 42, 1937.

[90] Chao Wang, SAID Mahmoud, Aarti Gupta, Vineet Kahlon, and Nishant Sinha. Dynamic test generation for concurrent programs, July 12 2012. US Patent App. 13/348,286.

[91] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.

[92] Meng-Ju Wu and Donald Yeung. Efficient reuse distance analysis of multicore scaling for loop-based parallel programs. *ACM Transactions on Computer Systems (TOCS)*, 31(1):1, 2013.

[93] Meng-Ju Wu, Minshu Zhao, and Donald Yeung. Studying multicore processor scaling via reuse distance analysis. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 499–510. ACM, 2013.

[94] Kazuhiro Yamashita, Yoshihide Nomura, Ence Zhou, Bingfeng Pi, and Sun Jun. Potential risks of hyperledger fabric smart contracts. In *2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 1–10. IEEE, 2019.

[95] Michal Zalewski. American fuzzy lop. *URL: http://lcamtuf. coredump. cx/afl*, 2017.

[96] Yutao Zhong, Xipeng Shen, and Chen Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(6):20, 2009.