A Vulnerability Detection Framework for Hyperledger Fabric Smart Contracts Based on Dynamic and Static Analysis

Peiru Li[1], Shanshan Li[1], Mengjie Ding[1], Jiapeng Yu[1], He Zhang[1], Xin Zhou[1], Jingyue Li[2]
[1]State Key Laboratory of Novel Software Technology, Software Institute, Nanjing University
[2]Department of Computer Science, Norwegian University of Science and Technology
[1]China, [2]Nolway
Email:micro.perrylee@outlook.com,lss@nju.edu.cn,alliac711@gmail.com
jiapeng_yu@outlook.com,hezhang@nju.edu.cn,job@wetist.com,jingyue.li@ntnu.no

ABSTRACT

Hyperledger Fabric is another development of blockchain technology after Ethereum, which is more suitable as an operating platform for smart contracts. However, the testing technology of Hyperledger Fabric smart contracts (also known as chaincode) is not yet mature currently. Based on this, this paper studies the vulnerability detection of Golang chaincodes. Firstly, we summarize 17 kinds of Golang chaincode vulnerabilities by investigating existing research. Secondly, taking the high accuracy of dynamic detection and the high efficiency of static detection into consideration, we propose a chaincode vulnerability detection framework that combines the dynamic symbolic execution and the static abstract syntax tree analysis technology. We also implement a supporting-tool that can detect the above 15 types of vulnerabilities. Finally, we test the tool by 15 chaincodes collected from GitHub and unknown vulnerabilities were detected in 13 projects. The precision turned out to be 91% after manual inspection. In order to verify the recall rate, we manually inject 30 vulnerabilities into the collected chaincodes and all of them are detected. The evaluation results show the accuracy of the proposed vulnerability detection method for Hyperledger Fabric smart contracts.

CCS CONCEPTS

• **Software Security** → **Vulnerability detection**.

KEYWORDS

Hyperledger Fabric, Smart Contract, Vulnerability Detection, Symbolic Execution, Abstract Syntax Tree

ACM Reference Format:
Peiru Li[1], Shanshan Li[1], Mengjie Ding[1], Jiapeng Yu[1], He Zhang[1], Xin Zhou[1], Jingyue Li[2]. 2022. A Vulnerability Detection Framework for Hyperledger Fabric Smart Contracts Based on Dynamic and Static Analysis. In *The International Conference on Evaluation and Assessment in Software Engineering 2022 (EASE 2022), June 13–15, 2022, Gothenburg, Sweden.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3530019.3531342

EASE 2022, June 13–15, 2022, Gothenburg, Sweden
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9613-4/22/06…$15.00
https://doi.org/10.1145/3530019.3531342

1 INTRODUCTION

Blockchain is a new application mode of computer technology proposed by Satoshi Nakamoto in 2008 [22]. Smart contract is a computer transaction protocol between two or more parties with self verification but without intermediary. In recent years, smart contracts have attracted much attention with the popularization of Internet and blockchain technology [24]. On the one hand, smart contract on the blockchain can flexibly embed various assets and data to help realize efficient value transfer, asset management and information exchange. On the other hand, neither party is allowed to deny or withdraw in the process of contract execution, which ensures the implementation without trust. Blockchain with smart contract has been more and more widely used in finance, management, medical treatment, Internet of things, supply chain and other fields [18, 24].

In recent years, as the application scenarios of smart contracts become more and more complex, many security problems begin to expose. Compared with ordinary programs, smart contract vulnerabilities are more likely to cause serious losses once they are exploited by hackers. For example, the Dao event in 2016 resulted in the loss of approximately $60 million worth of Ether [7]. Smart contract has obviously become the hardest hit area of blockchain security. Therefore, the detection of security vulnerabilities of smart contract has become an urgent problem to be solved in blockchain technology.

At present, there are a few research on smart contract vulnerability detection under Ethereum. In 2016, Lu et al. proposed four possible vulnerabilities of smart contract in Ethereum environment, which includes reentry, and implemented Oyente, a static analysis tool based on symbol execution running on EVM bytecode [19]. Mythril [21] is also a smart contract security analysis tool based on EVM bytecode, developed by Bernhard et al. It integrates technologies such as stain analysis and symbol execution, which can detect common security problems such as integer overflow. Tsankov et al. developed a static analysis tool for smart contract based on formal verification, Securify [28], which defines two modes of compliance and violation for each security attribute, extracts semantic facts from byte code or source code, then matches patterns and classifies all contract behaviors into violation, warning and compliance. ContractFuzzer [15] is the first security vulnerability ambiguity testing framework for the Ethereum Solidity contract, which supports more vulnerability types and has lower false positives than other tools.

Hyperledger Fabric (HF) is a modular and scalable open source system for deploying and operating consortium blockchain created by the Linux foundation in 2016. Smart contracts are usually written in specific languages, such as Solidity in Ethereum, but they can also be written using high-level languages, such as Golang and Java in HF [9], which reduces the learning cost of developers. Anyone can participate in the public blockchain represented by Ethereum. For consortium blockchain, only specific licensed participants can participate, which aims to provide a way of transaction for a group of individuals or organizations with common goals but incomplete trust. The public blockchain usually uses the consensus mechanism such as Proof-of-Work, which has limitations on performance. HF allows to customize the consensus mechanism for each smart contract when it is instantiated, which provides efficient personalized support for specific business scenarios and shorter delay [5, 26]. Therefore, for many application scenarios, HF is more suitable as the operation platform of smart contract.

At the same time, due to the above differences in development and operating mechanisms of smart contracts, the vulnerabilities under Ethereum and HF are quite different. First, some common vulnerabilities under Ethereum, such as gas exhaustion, are irrelevant to HF. The same type of vulnerability can have different manifestations in two different systems. In addition, abuse of general programming languages have also brought more problems.

As HF smart contract is relatively new, the research on HF smart contracts (also known as chaincode) is very limited so far. To our knowledge, there are few public tools to detect chaincode vulnerabilities. Therefore, this paper aims to design a method for automatic vulnerability detection of Golang chaincodes and implement the open sourced supporting tool [1] to fill this research blank.

There are two contributions of this paper: First, this paper combines symbolic execution technology and abstract syntax tree technology to design chaincode vulnerability detection framework HFCCT and implement the support tool. We verified the feasibility of the method and the effectiveness of the tool. Second, this paper analyzes the current situation of chaincode security through the frequency statistics of vulnerability types in 15 HF chaincodes. This paper gives a slice of analysis and suggestions on the development of chaincodes.

The structure of this paper is as follows: Section 2 introduces the technical background and related work of this paper; Section 3 summarizes 17 types of Golang chaincode vulnerabilities; Section 4 introduces in detail the design of chaincode vulnerability detection framework based on the combination of dynamic symbol execution and static abstract syntax tree; Section 5 presents the evaluation results of the framework based on the 15 chaincodes collected from GitHub; Section 6 summarizes the work of this paper, analyzes the current shortcomings, and points out the direction for further exploration in the future.

2 BACKGROUND AND RELATED WORK

2.1 Hyperledger Fabric and Chaincode

HF is a consortium blockchain, a distributed ledger technology that eliminates the need for anonymous miners to verify transactions, and the need for associated currencies to be used as incentives. All participants must be authenticated before they can participate in transactions in the blockchain. HF does not require expensive mining calculations to submit transactions, so it helps to build blockchains that can be expanded in a shorter delay [4].

Chaincodes are the smart contracts running on HF platform, which can read and write key value pairs in the ledger to realize complex business logic [24].

Chaincode operation includes three stages: (1) Endorsement: the client creates a transaction proposal to update the ledger and sends it to all endorsement nodes specified in the chaincode endorsement policy. Each endorsement node independently simulates the execution chaincode, and returns its response value, read-write set, signature to the client. (2) Sorting: after the client verifies the signature of each endorsement node and confirms that the response is consistent, the transaction is submitted to the sorting node. The sorting node sorts the transactions received, packs them into data blocks in batches, and broadcasts them to all connected peer nodes. (3) Verification: the peer node verifies the transactions in the data block one by one to ensure that the transaction is signed and endorsed by the corresponding node according to the endorsement strategy determined during chaincode instantiation. After verification, all peer nodes add new data blocks to the end of the current blockchain and update the ledger [4].

At present, HF smart contracts have a wide range of application scenarios. Walmart built a Fabric-based food tracking system [3], reducing the tracking time from 7 days to 2.2s. Change Healthcare applies Fabric to improve efficiency and transparency in the Medicare reimbursement process [1]. Dltledgers is a blockchain-based commodity trading financing platform that digitizes the transaction and financing process, greatly reducing transaction times [2]. Based on Fabric, Ian Zhou et al. proposed a blockchain-based paper review system, which not only protects the anonymity of authors and reviewers, but also solves the problem of unfair review or inconsistent indicators [35]. Tomas Mikula proposes an identity and access management system that applies Fabric to entity authentication and authorization in digital systems [20]. Christopher Harris leverages Fabric's ordering transactions to improve back-end tasks in the telecom industry, such as roaming and settlement between carriers [11]. We forsee the demand for HF smart contract development will be more popular.

2.2 Symbolic Execution

The main idea of symbolic execution is to use symbolic values to represent the inputs and all variables related to inputs in the program. In the process of symbolic execution, when a branch is encountered, the branch condition will be added to the constraint condition set of the current path. By reversing the condition, different paths can be selected under the same branch. Finally, test cases reaching the target area codes are generated through the constraint solver, so as to complete the path coverage of the program [31, 34].

After obtaining test cases, the program can be executed to detect whether there are vulnerabilities in each path, And save constraints and test cases, so as to apply symbolic execution technology into vulnerability detection.

Influenced by the structures such as branches and loops, symbolic execution often faces path explosion problem, which will produce large time overhead when analyzing large programs. In order to

[1]https://github.com/PerryLee69/HFCCT

solve this problem, many path exploration strategies have been introduced into symbolic execution [17]. However, compared with traditional programs, the code amount of smart contract is smaller and the number of branches is less, so it is appropriate to apply symbolic execution into the analysis of smart contract.

2.3 Abstract Syntax tree

Static analysis is a method of analyzing the source code of the target program through syntax and semantics to explore potential security vulnerabilities [34]. Abstract syntax tree (AST) represents the syntax structures of the program in the form of tree. Each node on the tree represents a structure in the program, which can be code semantic structures such as method call, structure declaration and package reference, or control flow structures such as branch, loop and exception handling [13].

Compared with other static detection methods such as syntax analysis and control flow analysis, AST based analysis not only has the inherent advantages of static analysis such as high efficiency and full code coverage, but also retains the information of source code to the greatest extent, and has the characteristics of strong readability and easy traversal [10]. In addition, the extraction and traversal of the AST are implemented by interfaces in high-level programming languages such as Golang and Java, which is convenient for vulnerability detection.

2.4 Related work

In terms of smart contract, vulnerability detection technologies include fuzzing, symbolic execution, static analysis, and formal verification. A lot of research works have achieved great results e.g. [23]. However there still exists challenges.

Echidna [25] and ContractFuzzer are the representative works of early smart contract vulnerability detection using fuzzing. However the fuzzing strategies adopted by them are elementary and difficult to achieve high code coverage. ILF [12] and Harvey [32] attempt to generate better test cases to improve coverage through deep learning and procedural instrumentation respectively. However their solutions are only geared to specific contracts, making it difficult to carry on large-scale expansion.

Oyente [19] and Mythril [21] are mature symbolic execution tools for smart contracts vulnerability detection. In addition, Osiris [27] uses symbolic execution techniques to detect integer related vulnerabilities in contracts, including integer overflow. Osiris is combined with stain analysis technology to filter out harmless overflow operation, and reduces false positives significantly. This provides an idea to combine two or more technologies to complement each other.

Zeus [16] is an automatic smart contract formal validation tool that translates source code into the LLVM intermediate language, writes validation rules using XACML, and uses validators for formal validation. Securify infers semantic facts from bytecode of smart contract programs and describes them in Datalog syntax, then checkes them against a predefined security attribute rule. Compared with using LLVM and Datalog to describe contracts, AST has the advantages of clear structure, strong readability, convenience to traverse and is applicable to high-level programming languages.

Table 1: Golang Chaincodes Vulnerability Types

ID	Vulnerability Type	Source of Vulnerability
1	Global Variable	Non-determinism arising from language instructions
2	Random Number Generation	
3	System Timestamp	
4	Map Structure Iteration	
5	Reified Object Addresses	
6	Concurrency of Program	
7	Web Service	Non-determinism arising from external access
8	External Library Calling	
9	System Command Execution	
10	External File Accessing	
11	Range Query Risks	HF platform
12	Field Declarations	
13	Cross Channel Chaincode Invocation	
14	Read-Write Conflict	
15	Unchecked Input Arguments	Practical experience
16	Unhandled Errors	
17	Golang Grammar Error	

However, most of the above tools aim at Ethereum contracts. There are limited researches on chaincode vulnerability detection. HFContractFuzzer [8] mentioned in Section 5 is one of them. However, inefficiency is its main drawback. In addition, ChainCode Scanner [14] is a static security analysis tool designed for Fabric smart contracts. The tool uses automatic security analysis methods, such as control flow graph analysis and dependency graph analysis, which is developed by ChainSecurity and available as a web application. Fujitsu [33] can detect 14 risk items mainly related to non-determinism risk and logical security risk. This tool is a command-line tool implemented in Golang, which has more effective coverage and better performance than ChainCode Scanner. However, these tools are all not open sourced.

3 CHAINCODE VULNERABILITIES

HF chaincode is written in high-level languages such as Golang, Java, and Nodejs. This paper focuses on Golang chaincode. We have studied the vulnerability types of Golang chaincode and their manifestations, which is beneficial to design detection methods for specific vulnerabilities in Section 4.

At present, the studies on Golang chaincode vulnerability are limited. Based on references [33] and [6] , we summarizes the types of the existing chaincode vulnerability. A total of 17 vulnerabilities are summarized, as shown in Table 1.

(1) Global Variable. Global variables can be changed locally. If global variables are involved in writing to the ledger, this can cause inconsistencies in the ledger status of each peer nodes.

(2) Random Number Generation. Since the chaincode is simulated independently by each peer in the endorsement phase, the values generated randomly in each peer are different.

(3) System Timestamp. Similar to random number generation, there is no guarantee that each peer node in the endorsement phase will simultaneously call the timestamp function.

(4) Map Structure Iteration. Because of the implementation details of Golang, the order of key-value pairs is not unique when iterating Map Structure.

(5) Reified Object Addresses. Developers can handle the value of a variable with a pointer, which is the address of memory, and the value stored at that address depends on the environment. Thus, the value fetched at a specific address may vary from one environment to another.

(6) Concurrency of Program. If concurrent programs are not handled properly, it is easy to cause a conflict condition problem that results in an non-deterministic execution.

(7) Web Service. It is common for business logic to call an API to reuse web service to provide data outside the blockchain to chaincode. If the service returns different results to each peer node, this can result in inconsistent endorsements.

(8) External Library Calling. Similar to web services, when using third-party libraries to reduce development effort, developers should pay attention to the behavior of the library and understand what is happening in the library.

(9) System Command Execution. Golang supports external command execution, but does not guarantee the same results between each node.

(10) External File Accessing. Similar to system command execution, Golang supports access to external files, but does not guarantee that the same results will be returned by the nodes in their separate environments.

(11) Range Query Risks. HF provides methods for accessing the status database, including range query methods such as GetQueryResult(), getprivatedataquaeryresult(), etc. , which are not re-executed during the validation phase, meaning that the phantom problem will not be detected. This is not necessarily a vulnerability, but it should be warned.

(12) Field Declarations. There should be no field declarations in the chaincode structure. Chaincode needs to implement interfaces Init and Invoke, where fields can be accessed if they are defined in the structure. However, since each peer node does not necessarily perform every transaction, the field values of the chaincode may be inconsistent among the peer nodes.

(13) Cross Channel Chaincode Invocation. If two chaincodes are on the same channel, there is no problem calling each other. If not, it is difficult to guarantee that no other data will be submitted to the channel of the chaincode and that only the data returned by the chaincode method will be accepted.

(14) Read-Write Conflict. HF does not support read-write consistency, and in the same transaction, even if the key value is updated before reading it, it will return the value before the update, which may cause the code to execute differently than expected.

(15) Unchecked Input Arguments. This is common in any programming language. An error occurs if the input parameters are not checked and the program accesses a no-exist element.

(16) Unhandled Errors. In Golang, developers can skip receiving the return value by assigning it to the "_" variable, which might ignore the error if the return value is Error type.

(17) Golang Grammar Errors. These include a library that is imported but not used, a variable is declared but not used, or incorrect assignment, etc.

Hackers can take advantage of these above vulnerabilities to call specific functions maliciously, or cause chaincode misfunction and termination, or cause the endorsement of the endorsement node to be inconsistent and can not carry out normal transactions.

4 VULNERABILITY DETECTION METHOD

The static detection method has high efficiency and low false negative rate, but high false positive rate. The dynamic detection method has high accuracy but has problems of low detection efficiency and difficulty in generating high-quality test cases [30]. Based on the technology of combining dynamic symbolic execution and static abstract syntax tree, we designed the HFCCT (Hyperledger Fabric Chaincodes Test) framework and implemented supporting tool for the chaincode vulnerabilities investigated in Section 3. The overview of HFCCT framework is shown in Figure 1, consisting of two parts: symbolic execution and abstract syntax tree, which will be introduced in detail in later sections.

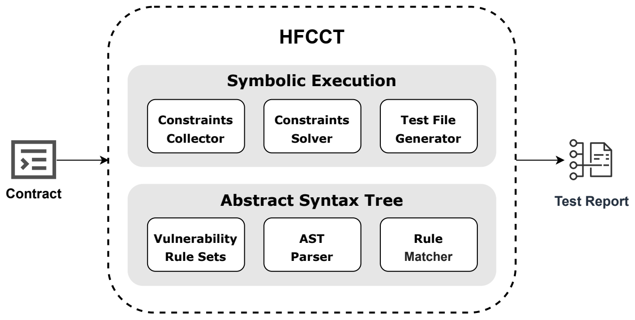


Figure 1: Overview of HFCCT

4.1 Method Design

We design detection methods for the 15 vulnerabilities investigated in Section 3. There are generally two types of methods: dynamic symbolic execution and static abstract syntax trees. The former has the characteristics of dynamic methods, that is, the accuracy of vulnerability detection is high, but with the problems of low efficiency and difficulty in generating high-quality test cases; the latter, like other static methods, has high efficiency, low false positive rate, but high false positive rate [30]. According to the characteristics of different technologies, we combined dynamic and static technologies, adopted different methods according to different types of vulnerabilities, and designed specific methods for each vulnerability. The outline are shown in Table 2.

For nine kinds of vulnerabilities with fixed and obvious features in form, in order to improve the efficiency, AST method is used to detect them. The specific designs are as follows:

(1) Web Service, External Library Calling, System Command Execution, External File Access. We use the ImportSpec node of AST to detect the imported libraries in the chaincode, and put the libraries that communicate with the ones outside of the chaincode or lead to non-determinism into blacklist, such as "os/exec", "oracle", etc.

(2) Program Concurrency. Since concurrency is discouraged in chaincode, we check whether there is a GoStmt node in AST, that is, whether goroutine is used in chaincode to make the program concurrent.

Table 2: Detection Methods for Each Vulnerability

ID	Vulnerability	Detection Method	Specific Method
1	Web Service		Check whether blacklisted packages such as "net/http" are imported
2	External Library Calling		Check whether blacklisted packages such as "database/sql" are imported
3	System Command Execution		Check whether blacklisted packages such as "os" are imported
4	External File Access	Abstract	Check whether blacklisted packages such as "os/exec" are imported
5	Program Concurrency	Syntax	Check whether "goroutine" is used
6	Range Query Risks	Tree	Check whether range query methods such as "GetQueryResult" are used
7	Field Declarations		Check whether chaincode structure is not empty
8	Read-Write Conflict		Check whether GetState is used after PutState for same key in same function
9	Unchecked Errors		Check whether "_" is used as a variable
10	Random Number Generation		Check whether multiple executions lead to consistent ledger state
11	System Timestamp		Check whether multiple executions lead to consistent ledger state
12	Map Structure Iteration	Symbolic	Check whether multiple executions lead to consistent ledger state
13	Global Variables	Execution	Modify global variables and check whether the execution lead to consistent ledger state
14	Unchecked Input Arguments		Generate special test cases
15	Golang Grammar Errors		Execute a test case at random

(3) Range Query Risks. We collect all range query methods such as "GetQueryResult()" and check if any methods that involve these query statements are called in AST.

(4) Field Declarations. We extract the name for chaincode structure from AST, and check whether the structure is empty.

(5) Read-Write Conflict. We extract all PutState method calls from AST, then find the next GetState method call that reads the same key value, and check whether the two calls are in the same function.

(6) Unchecked Errors. We detect whether there are "_" variables in AST, which means that a variable of the error type may be accepted with "_", so that possible errors are not checked, although it may lead to false positives.

For the other six types of vulnerabilities that are not very fixed in form and may be caused by multiple statements, in order to avoid false positives with static analysis, dynamic symbolic execution method is used to detect them. The specific designs are as follows:

(1) Random Number Generation, System Timestamp, Map Structure Iteration. If these uncertain values exist in operations that involve modifying the ledger, a vulnerability will arise, which can be detected by executing the same test case multiple times and comparing whether the ledger state is consistent after each execution.

(2) Global Variables. We detect global variable in the chaincode. If there is one, we assign it to different values, execute the code multiple times, and compare whether the ledger state is consistent after each execution.

(3) Unchecked Input Arguments. The default test cases generated for string array type parameters are empty arrays and arrays with only one empty string element. So if the parameter length is not checked in the function, using subscript to get value in the function will fail.

(4) Golang Grammar Errors. For example, no using after importing a package. It can be detected by running any unit test case.

Symbolic execution is based on multiple unit tests. Unit test allows the chaincode to execute according to the designed test cases. HF provides the MockStub class for unit test. It maintains a map[string][]byte to simulate the key-value pair state database. The chaincode calls PutState and GetState functions to act on the map in memory, thus to operate the state database in blockchain network. The MockStub class mainly provides two functions: MockInit calls the Init interface of the chaincode to complete the initialization of the chaincode, and MockInvoke calls the Invoke interface to call different functions according to the parameters.

Using the MockStub class to unit test the chaincode does not need to configure or run Docker image environments. Executing go test command could complete calling of chaincode interface locally. The testing process is the same as unit test of ordinary Golang programs, which has the advantages of concision and efficiency.

4.2 Framework Design

We design the framework HFCCT for chaincode vulnerability detection, which is implemented based on dynamic symbolic execution and static abstract syntax tree technology.

4.2.1 Dynamic Symbolic Execution. The core of symbolic execution is to generate test cases that enter each branch path by extracting and solving constraints. First, we design the process of generating test cases for the function to be tested, as shown in the white area of Figure 2.

Step A1: We extract all conditions in the function to be tested and reverse them respectively.

Step A2: We combine the conditions with each other. N conditions will form 2^N combinations.

Step A3: For each condition group, we solve the conditions in turn. If it is a numerical condition, including integer and float types, we use the z3 solver [29].If it is a string condition, including string and string array types, we take the following logic strategies: original value for equivalent conditions; random value for unequal conditions.

Step A4: We integrate the solution set and generate the test cases to enter each branch path.

Based on the above method, we design the process for detecting the chaincode, as shown in Figure 2.

Step B1: We extract the parameter list of Init interface, and use symbolic execution to solve constraints and solution sets for Init interface. Then, we generate the MockInit function codes in the test file according to solution sets.

Step B2: We extract the parameter list of Invoke interface, and use symbolic execution to solve constraints and solution sets that enter branches of calling each function and do not enter any branch.

Step B3: According to solution sets of Invoke interface, we use symbolic execution to solve constraints and solution sets for other functions that called in the branch of Invoke interface. Then, we generate the MockInvoke function codes in the test file based on the solution sets.

Step B4: We randomly match a MockInvoke of each function for each MockInit, and shuffle the order to generate test files and save them.

Step B5: We execute each test file more than one times, which relies on MockStub class, and save the results.

Step B6: We compare the results of multiple executions, then the test report will be generated.

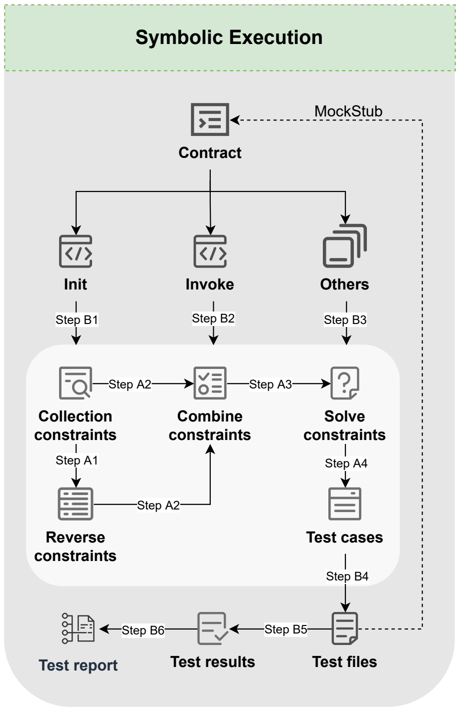


Figure 2: Process of Symbolic Execution

4.2.2 Static Abstract Syntax Tree. The process is shown in Figure 3.

Step C1: Take the specific detection methods designed in Section 4.1 as vulnerability matching rules.

Step C2: Extract the features of the chaincode to be tested through AST. Golang provides "go/ast", "go/parser" and "go/token" three libraries to extract AST of Golang programs, including package, declaration, scope, import library information, etc.

Step C3: Matching is performed by sequentially traversing the rules of each vulnerability in the information extracted from the AST. If there is a vulnerability, the number of source code lines and the reason that caused the vulnerability will be extracted and written into the test report.

4.2.3 Framework Implementation. We combine dynamic symbolic execution with static abstract syntax tree and design HFCCT framework to detect vulnerabilities in chaincodes, which can detect the 15 types of vulnerabilities described in Section 3.

HFCCT consists of four core components: CCAST.go parses the Golang chaincode into AST and provides external interfaces. CCSE.py implements symbolic execution for vulnerability detection. CCSA.py implements AST for vulnerability detection. HFCCT.py invokes the above components to perform chaincode vulnerability detection and generates the final test report, which is also the entry of HFCCT.

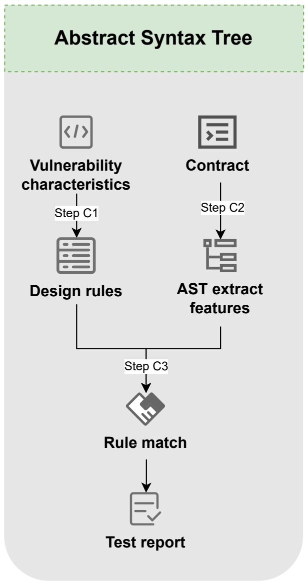


Figure 3: Process of Abstract Syntax Tree

5 EVALUATION

We conducted an experiment to evaluate the effectiveness of the method proposed as well as the framework implemented. We used "Hyperledger Fabric contract" or "chaincode" as keywords to retrieve projects on GitHub. At last, we chose 15 chaincodes from projects with the highest star ranking, which as well as involve multiple application fields. Then, we use HFCCT to detect vulnerabilities in the 15 typical chaincodes. This experiment runs on Windows ten professional 64-bit operating system, and the processor model used is Intel (R) Core (TM) i5-8250U CPU @ 1.60GHz 1.80 GHz. The experimental environment of detecting projects is Python 3.7.6 and Golang 1.14.6.

5.1 Results

Vulnerability detection results of the 15 chaincodes are shown in Table 3. The vulnerability type numbers in the third column are the same as those in Section 2. We take the marbles02 project as example to show the generated test report in Figure 4. The report shows the path of the tested chaincode file, including the detected vulnerability types, a brief summary of the source of each vulnerability, and the corresponding number of lines in the source file.

Table 3: Vulnerability Detection Results of Projects

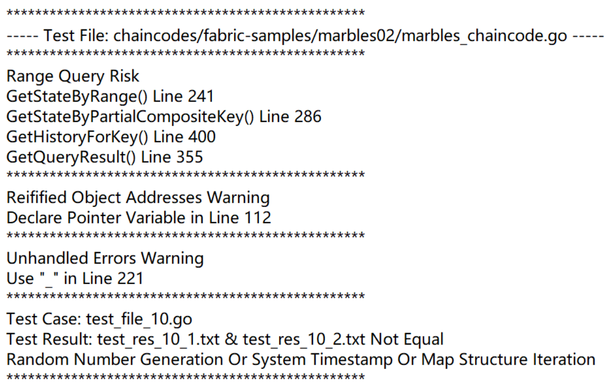
ID	Projects	Detected Vulnerability Types
1	FabricTest[2]	(16), (17)
2	charity[3]	(11), (16)
3	abac[4]	(16)
4	marbles02[5]	(2), (3), (4), (11), (16)
5	marbles02_private[6]	(11), (16)
6	sacc[7]	no vulnerability
7	chaincode_example02[8]	(16)
8	fabcar[9]	(2), (3), (4), (11), (16)
9	test[10]	16)
10	marbles[11]	no vulnerability
11	multiorgledger[12]	(16)
12	MySmartContract[13]	(16)
13	Simple-Fabric-Web-Project[14]	(11)
14	supplychain-blockchain-network[15]	(16), (17)
15	vehiclesharing[16]	(2), (3), (4), (11)

Accuracy. Among the 15 chaincodes, thirteen were detected with vulnerabilities, involving 22 places. Since all detected vulnerabilities are unknown, it is necessary to verify whether there are false positives one by one. After manual verification, there are two false positives in the "unchecked error" vulnerability, both of which are accepting one of the return values of the GetFunctionAndParameters function with "_". Because the two return values are not error type, it does not ignore error checks. Therefore, from the statistics of small sample data, the precision of HFCCT is 91%. However, we must state that the vulnerabilities identified by our manual verification have not been confirmed by the project developers.

Recall. Since the full set of vulnerabilities of the 15 chaincodes are unknown, in order to evaluate the recall, we manually injected 30 vulnerabilities of the 15 types introduced in Section 4 into random chaincodes. All of them were detected by HFCCT, thus the recall is 100%.

Performance and effectiveness. In order to evaluate the performance and effectiveness, we compared with HFContractFuzzer [8] proposed by Ding et al. on the above 15 chaincodes. Table 4 shows the results, each cell is "number of detected vulnerabilities / costed time". It should be noted that HFContractFuzzer is based on fuzzing technology. In principle, it will automatically generate test cases to execute continuously. Therefore, we chose 1h as the time cutoff. If a vulnerability is detected within 1h, the time taken to find the first vulnerability will be displayed.

[2]https://github.com/WaterSh/FabricTest/blob/master/chaincode_AtoreAndAccess.go
[3]https://github.com/cloudframeworks-blockchain/user-guide-Fabric-smart-contract/blob/master/chaincode/charity/charity_contract.go
[4]https://github.com/Hyperledger/Fabric-samples/tree/main/chaincode/sacc
[5]https://github.com/Hyperledger/Fabric-samples/tree/main/chaincode/marbles02
[6]https://github.com/Hyperledger/Fabric-samples/tree/main/chaincode/marbles02_private
[7]https://github.com/Hyperledger/Fabric-samples/tree/main/chaincode/sacc
[8]https://github.com/sslinml/Fabric_e2e_app/tree/master/chaincode/chaincode_example02
[9]https://github.com/sslinml/Fabric_e2e_app/tree/master/chaincode/fabcar
[10]https://github.com/sslinml/Fabric_e2e_app/tree/master/chaincode/test
[11]https://github.com/sslinml/Fabric_e2e_app/tree/master/chaincode/marbles02
[12]https://github.com/Deeptiman/multiorgledger
[13]https://github.com/HeartWillGo/MySmartContract
[14]https://github.com/BIGyellowPIE/Simple-Fabric-Web-Project
[15]https://github.com/subhashmeena/supplychain-blockchain-network/blob/master/solution.go
[16]https://github.com/tomxucnxa/vehiclesharing



```
**************************************************
----- Test File: chaincodes/fabric-samples/marbles02/marbles_chaincode.go -----
**************************************************
Range Query Risk
GetStateByRange() Line 241
GetStateByPartialCompositeKey() Line 286
GetHistoryForKey() Line 400
GetQueryResult() Line 355
**************************************************
Reifified Object Addresses Warning
Declare Pointer Variable in Line 112
**************************************************
Unhandled Errors Warning
Use "_" in Line 221
**************************************************
Test Case: test_file_10.go
Test Result: test_res_10_1.txt & test_res_10_2.txt Not Equal
Random Number Generation Or System Timestamp Or Map Structure Iteration
**************************************************
```

Figure 4: Test Report of Project marbles02

Table 4: Results of Comparison with HFContractFuzzer

ID	Projects	HFCCT	HFContractFuzzer
1	FabricTest	2/104s	0/1h
2	charity	2/68s	1/20min
3	abac	1/88s	0/1h
4	marbles02	3/74s	0/1h
5	marbles02_private	2/61s	0/1h
6	sacc	0/117s	0/1h
7	chaincode_example02	1/92s	0/1h
8	fabcar	3/64s	0/1h
9	test	1/101s	0/1h
10	marbles	0/69s	0/1h
11	multiorgledger	1/15s	0/1h
12	MySmartContract	1/24s	0/1h
13	Simple-Fabric-Web-Project	1/57s	0/1h
14	supplychain-blockchain-network	2/15s	1/54min
15	vehiclesharing	3/66s	1/13min

The time HFCCT takes in each chaincode varies, depending on its complexity and the number of branches. The average time is 68s, as a measure of HFCCT performance. The shortest time for HFContractFuzzer to find the first vulnerability is 13min, thus its performance is much lower than that of HFCCT.

It can be seen in Table 4 that HFCCT can detect more vulnerabilities, therefore it has better effectiveness than HFContractFuzzer. On the other hand, as for the display of vulnerability detection results, HFContractFuzzer is more difficult to understand, which requires professionals to analyze the types of vulnerabilities and their causes. HFCCT is more friendly in this regard.

5.2 Other Findings

After manual verification to remove false positive vulnerabilities, a pie chart is drawn for the number of occurrences of each vulnerability type in 15 chaincodes, as shown in Figure 5.

The proportion of "unchecked error" vulnerability is 45%, indicating that many chaincode developers are accustomed to accepting return value of error type with "_", ignoring the error handling. Although in most cases it has no problem, once it is exploited by hackers, the cost of loss can be huge. Because it means that some failed operations are not identified. The second popular vulnerability is "range query risk" vulnerability. It is just a warning. Because the range query will not be executed again during the verification phase, chaincode developers should pay more attention when using it, and try not to let the results of the range query involve the modification of the ledger. The third popular one is the "random number/system timestamp/Map iteration" vulnerability, which means that developers have led non-determinism into the chaincode, and it involves ledger modification operations, which need to be avoided. The next popular one is "Golang syntax error". For example, the "strconv" library in the FabricTest project is imported but not used. This has nothing to do with the chaincode characteristic. Howecer, developers still need to be more familiar with chaincode programming languages.

The types of vulnerabilities that chaincode developers are usually prone to make are concentrated, but they exist popularly. Therefore, the security of chaincode still needs attention from both academic research and industrial practice.

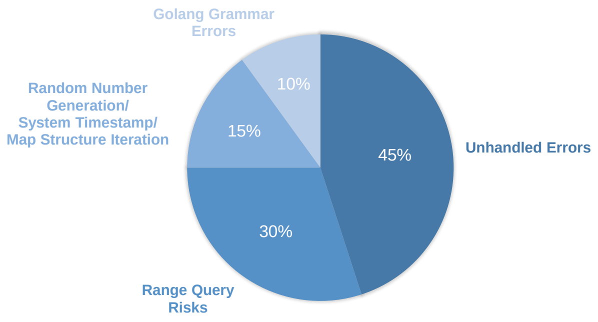


Figure 5: Percentage for Occurrences of Vulnerability Types

6 CONCLUSION

In recent years, with the increasing security attacks on smart contracts, its vulnerability detection has become an urgent problem to be solved. At present, the research of smart contract vulnerability detection of Ethereum has been mature. However the research of HF is still in the early stage. Based on this, this paper created an open-source chaincode automatic vulnerability detection tool.

In the future research, we will focus on the following aspects:

Datasets: public available data sets for HF projects are limited, which are detrimental to the evaluation of vulnerability detection methods and tools. We will pay more attention to the collection and arrangement of HF data sets.

Vulnerability types: because current research on the types of chaincode vulnerability is very limited, the summary of vulnerability types may not be perfect. We will also do more in-depth research on chaincode mechanism, and practice the chaincode deployment to explore new vulnerability types.

Method: based on the test results, HFCCT misreports the "unchecked error" vulnerability type. And two of the 17 vulnerability types we studied can not be detected by HFCCT. In the future, We will continue to optimize the design of the methods, and also consider combining other technologies for mutual complementation to improve the recall.

ACKNOWLEDGMENTS

This work is jointly supported by the National Key Research and Development Program of China (No.2019YFE0105500) and the Research Council of Norway (No.309494), as well as the National Natural Science Foundation of China (Grants No.62072227, 61802173), the Key Research and Development Program of Jiangsu Province (No. BE2021002), and the Intergovernmental Bilateral Innovation Project of Jiangsu Province (BZ2020017).

REFERENCES

[1] 2021. Change Healthcare using Hyperledger Fabric to improve claims lifecycle throughput and transparency. https://www.hyperledger.org/learn/publications/changehealthcare-case-study.
[2] 2021. Dltledgers. https://bcsec.org/(2021).
[3] 2021. How Walmart brought unprecedented transparency to the food supply chain with Hyperledger Fabric. https://www.hyperledger.org/learn/publications/walmart-case-study.
[4] 2021. Hyperledger Fabric. https://www.hyperledger.org/projects/fabric(2020).
[5] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*. 1–15.
[6] Sotirios Brotsis, Nicholas Kolokotronis, Konstantinos Limniotis, Gueltoum Bendiab, and Stavros Shiaeles. 2020. On the security and privacy of hyperledger fabric: Challenges and open issues. In *2020 IEEE World Congress on Services (SERVICES)*. IEEE, 197–204.
[7] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* 3, 37 (2014).
[8] Mengjie Ding, Peiru Li, Shanshan Li, and He Zhang. 2021. HFContractFuzzer: fuzzing hyperledger fabric smart contracts for vulnerability detection. In *Evaluation and Assessment in Software Engineering*. 321–328.
[9] Jianbo Gao, Hongyi Liu, Qingshan Li, and Zhong Chen. 2020. Research on Smart Contract Secruity Vulnerability Detection Technology. *Secrecy Science and Technology* 1 (2020), 22–25.
[10] Taoming Gu. 2020. Research and Design of Security Inspection System Oriented to Smart Contract. *University of Electronic Science and Technology of China* (2020).
[11] Christopher Harris. 2019. Improving telecom industry processes using ordered transactions in hyperledger fabric. In *2019 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 1–6.
[12] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 531–548.
[13] Xiaowei Huang, Guisheng Fan, Huiqun Yu, and Xingguang Yang. 2021. Software Defect Prediction Based on Baryon Node Abstract Syntax Tree. https://doi.org/10.19678/j.issn.1000-3428.0060389(2021).
[14] Yongfeng Huang, Yiyang Bian, Renpu Li, J Leon Zhao, and Peizhong Shi. 2019. Smart contract security: A software lifecycle perspective. *IEEE Access* 7 (2019), 150184–150202.
[15] Bo Jiang, Ye Liu, and WK Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 259–269.
[16] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts.. In *Ndss*. 1–12.
[17] Minglei Li, Hui Huang, and Yuliang Lu. 2020. Vulnerability Mining-Oriented Test Cases Generation Technology Based on Symbolic Divide and Conquer Area. *Information and Network Security* 20, 5 (2020), 39–46.
[18] Jing Liu and Zhentian Liu. 2019. A survey on security verification of blockchain smart contracts. *IEEE Access* 7 (2019), 77894–77904.
[19] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 254–269.
[20] Tomas Mikula and Rune Hylsberg Jacobsen. 2018. Identity and access management with blockchain in electronic healthcare records. In *2018 21st Euromicro conference on digital system design (DSD)*. IEEE, 699–706.
[21] B Muelle. 2021. Mythril. https://GitHub.com/ConsenSys/mythril(2021).
[22] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review* (2008), 21260.
[23] Yuandong Ni, Chao Zhang, and Tingting Ying. 2020. A review of security vulnerabilities on smart contract. *Journal of Information Security* 5, 3 (2020), 78–99.
[24] Liwei Ouyang, Shuai Wang, Yong Yuan, Xiaochun Ni, and FY Wang. 2019. Smart contracts: Architecture and research progresses. *Acta Automatica Sinica* 45, 3 (2019), 445–457.

[25] JP Smith. 2018. Echidna, a smart fuzzer for ethereum.

[26] Joao Sousa, Alysson Bessani, and Marko Vukolic. 2018. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *2018 48th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 51–58.

[27] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 664–676.

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

[29] Dong Wang, Bo Jiang, and WK Chan. 2020. WANA: Symbolic execution of wasm bytecode for cross-platform smart contract vulnerability detection. *arXiv preprint arXiv:2007.15510* (2020).

[30] Huan Wang. 2016. Research on Security Vulnerability Detection Method Based on Static and Dynamic. *Huazhong University of Science and Technology* (2016).

[31] ZHAO Wei, ZHANG Wenyin, WANG Jiuru, WANG Haifeng, and WU Chuankun. 2020. Smart contract vulnerability detection scheme based on symbol execution. *Journal of Computer Applications* 40, 4 (2020), 947.

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

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

[34] Kun Yang. 2020. Automated Security Audit of Smart Contract Based on Symbolic Execution. *University of Electronic Science and Technology of China* (2020).

[35] Ian Zhou, Imran Makhdoom, Mehran Abolhasan, Justin Lipman, and Negin Shariati. 2019. A blockchain-based file-sharing system for academic paper review. In *2019 13th International Conference on Signal Processing and Communication Systems (ICSPCS)*. IEEE, 1–10.