Science of Computer Programming 234 (2024) 103076

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

CrossFuzz: Cross-contract fuzzing for smart contract vulnerability detection

Huiwen Yang [a,b], Xiguo Gu [a], Xiang Chen [c], Liwei Zheng [a], Zhanqi Cui [a,b,*]

[a] School of Computer Science, Beijing Information Science and Technology University, Beijing, China
[b] Key Laboratory of Safety-Critical Software (Nanjing University of Aeronautics and Astronautics), Ministry of Industry and Information, Nanjing, China
[c] School of Information Science and Technology, Nantong University, Nantong, China

ARTICLE INFO

Keywords:
Smart contract
Fuzz testing
Cross-contract vulnerability
Security vulnerability detection

ABSTRACT

Context:: Smart contracts are computer programs that run on a blockchain. As the functions implemented by smart contracts become increasingly complex, the number of cross-contract interactions within them also rises. Consequently, the combinatorial explosion of transaction sequences poses a significant challenge for smart contract security vulnerability detection. Existing static analysis-based methods for detecting cross-contract vulnerabilities suffer from high false-positive rates and cannot generate test cases, while fuzz testing-based methods exhibit low code coverage and may not accurately detect security vulnerabilities.
Objective:: The goal of this paper is to address the above limitations and efficiently detect cross-contract vulnerabilities. To achieve this goal, we present CrossFuzz, a fuzz testing-based method for detecting cross-contract vulnerabilities.
Method:: First, CrossFuzz generates parameters of constructors by tracing data propagation paths. Then, it collects inter-contract data flow information. Finally, CrossFuzz optimizes mutation strategies for transaction sequences based on inter-contract data flow information to improve the performance of fuzz testing.
Results:: We implemented CrossFuzz, which is an extension of ConFuzzius, and conducted experiments on a real-world dataset containing 396 smart contracts. The results show that CrossFuzz outperforms xFuzz, a fuzz testing-based tool optimized for cross-contract vulnerability detection, with a 10.58% increase in bytecode coverage. Furthermore, CrossFuzz detects 1.82 times more security vulnerabilities than ConFuzzius.
Conclusion:: Our method utilizes data flow information to optimize mutation strategies. It significantly improves the efficiency of fuzz testing for detecting cross-contract vulnerabilities.

1. Introduction

In recent years, Ethereum smart contracts have gained widespread use in various fields [1], such as finance, health care [2][3], cloud computing [4], and the Internet of Things [5][6]. However, potential security vulnerabilities existing in smart contracts have

* Corresponding author at: School of Computer Science, Beijing Information Science and Technology University, Beijing, China.
E-mail address: czq@bistu.edu.cn (Z. Cui).

https://doi.org/10.1016/j.scico.2023.103076
Received 5 August 2023; Received in revised form 22 December 2023; Accepted 27 December 2023

Available online 4 January 2024

led to numerous security incidents, resulting in economic losses and security risks [7]. For example, in 2016, attackers exploited the reentrancy vulnerability in the DAO contract to steal $60 million Ethers [8]. According to a report from SlowMist,[1] as of July 1, 2023, 1109 security incidents have occurred, causing losses exceeding $30 billion. Moreover, after smart contracts are deployed, they cannot be modified or repaired through patches. Therefore, vulnerability detection techniques to ensure the security of smart contracts are critically important and urgently needed.

Research has made significant progress in detecting smart contract security vulnerabilities based on various techniques, including static analysis [9][10][11][12], symbolic execution [8][13][14][15], machine learning [16][17][18], and fuzz testing [19][20][21]. However, most existing methods primarily focus on analyzing individual smart contracts without considering the dependencies and function calls between contracts, leading to high false-positive and false-negative rates.

As the requirements and functions become increasingly complex, smart contracts are evolving from individual contracts to cross-contracts that interact with each other. As the size of these contracts increases, the risk of vulnerabilities in the code also increases. A *cross-contract vulnerability* refers to a vulnerability within a transaction sequence that involves more than two contracts [22]. According to the statistical results by Liao et al. [22], the number of contracts affected by cross-contract vulnerabilities has increased from November 2020 to December 2022, resulting in economic losses of up to $81.2 million. Furthermore, cross-contract vulnerabilities pose more significant challenges in analysis and detection, because the search space is much larger than individual contracts [23].

To detect cross-contract vulnerabilities, Xue et al. [24] proposed a static analysis-based method called Clairvoyance, which builds CFG (Control Flow Graph) and DFG (Data Flow Graph) for cross-contract. Clairvoyance identifies reentrancy vulnerability paths by tracking tainted paths and leverages symbolic execution techniques to check path feasibility, reducing false positives. Liao et al. [22] proposed a static analysis-based method named SmartDagger, which decompiles the bytecode of smart contracts and recovers missing semantic information, such as state variable attributes, through a neural machine translation model. SmartDagger is capable of detecting security vulnerabilities such as reentrancy and timestamp dependency.

Although these static analysis-based methods exhibit higher efficiency in detecting security vulnerabilities, they ignore the impact of transaction sequences on the state of contracts, which can result in false positives and false-negative results. Transaction executions can change the values of state variables, which can, in turn, affect subsequent transactions. Therefore, static analysis methods have lower accuracy in detecting security vulnerabilities that are triggered by sequences of transactions.

Fuzz testing-based cross-contract vulnerability detection methods dynamically generate transaction sequences, which enhance the capability of vulnerability detection. For instance, Xue et al. [24] proposed a fuzz testing-based tool named xFuzz, which builds on sFuzz [19]. xFuzz deploys interdependent smart contracts and computes priority scores through a machine learning model to select functions for execution. The experimental results show that xFuzz improved the efficiency of fuzz testing by reducing the detection time by 80% compared to sFuzz. However, existing fuzz testing-based methods still suffer from the following two issues.

L1: Existing fuzz testing-based methods instantiate contracts with default values, neglecting the assignment of proper contract addresses for the constructor. Although fuzzers can generate random values as constructor parameters, dependent contracts may not be correctly instantiated by the randomly generated parameters. This setup can result in exceptions when invocating cross-contract functions, preventing the subsequent code from being executed and tested by fuzzers.

L2: Critical functions are overlooked during the generation of transaction sequences. Fuzz testing-based methods, such as xFuzz and ILF [32] reduce the number of transaction sequence combinations through machine learning techniques. However, the performance of the model can be compromised due to imbalanced datasets, incorrect labels, or inappropriate training parameter settings. For instance, xFuzz may not invoke critical functions that may trigger vulnerabilities, reducing its capability to detect cross-contract vulnerabilities.

To address these limitations, we present CrossFuzz, a fuzz testing-based tool for cross-contract vulnerability detection to enhance code coverage and vulnerability detection capability. First, CrossFuzz generates appropriate parameters for the constructor (to address L1). Then, CrossFuzz analyzes the Inter-Contract Data Flow (ICDF), which consists of the definitions and usages of state variables for each function, by taking the call relationships between functions of contracts into account. Finally, it deploys smart contracts, calculates the priority scores according to the ICDF, and optimizes the transaction sequence mutation strategy by altering the order of transactions within the test case according to the priority scores, in order to generate offspring test cases that become more likely to cover previously uncovered branches (to address L2), and improves the performance of fuzz testing. We implemented CrossFuzz, which is an extension of ConFuzzius, and conducted experiments on a dataset containing 396 smart contracts. Experimental results show that compared with xFuzz, sFuzz and ConFuzzius [28], CrossFuzz improved 7.81%–36.89% bytecode coverage and detected more security vulnerabilities.

In summary, the following contributions are provided.

(1) We present CrossFuzz, a fuzz testing-based tool for cross-contract vulnerability detection, which tracks the data propagation paths of constructor parameters to assign proper values for parameters. Furthermore, CrossFuzz employs data flow information between contracts to optimize the transaction sequence mutation strategy, enabling effective analysis and detection of cross-contract vulnerabilities.

[1] SlowMist: https://hacked.slowmist.io/.

```
1  contract PermissionManager { //the dependent contract
2    mapping (address => bool) permittedAddresses;
3    function addAddress(address newAddress) public {
4      permittedAddresses[newAddress] = true;
5    }
6    function isPermitted(address pAddress) public view
         returns (bool) {
7      if (permittedAddresses[pAddress]) return true;
8      return false;
9    }
10 }
11 contract Hold is Ownable {  //the contract under test
12   uint8 stages = 5;
13   uint8 public percentage;
14   uint8 public currentStage;
15   uint public withdrawed;
16   address public multisig;
17   PermissionManager public permissionManager;
18   address public observer;
19   function Hold(address _multisig, uint cap, address pm,
         address observerAddr) public {
20     percentage = 100 / stages;
21     multisig = _multisig;
22     currentStage = 0;
23     initialBalance = cap;
24     observer = observerAddr;
25     permissionManager = PermissionManager(pm);
26   }
27   modifier onlyPermitted() {  //function modifier
28     require (permissionManager.isPermitted(msg.sender));
29     _;
30   }
31   modifier onlyObserver() {
32     require (msg.sender == observer);
33     _;
34   }
35   function releaseETH(uint n) onlyPermitted public {
36     require (this.balance >= n);
37     require (getBalanceReleased() >= n);
38     multisig.transfer(n);
39     withdrawed += n;
40 }
41   function getBalanceReleased() public view returns (uint)
         {
42     return initialBalance * percentage * currentStage /
         100 - withdrawed;
43   }
44   function changeStage() public onlyObserver {
45     uint8 newStage = currentStage + 1;
46     require(newStage <= stages);
47     currentStage = newStage;
48   }
49 }
```

Fig. 1. An example of smart contract.

(2) We collected 396 smart contracts from EtherScan[2] and conducted experiments to evaluate CrossFuzz. The results show that CrossFuzz increased 10.58% bytecode coverage compared to xFuzz and detected 1.82 times more security vulnerabilities than ConFuzzius, thus demonstrating that CrossFuzz outperforms other state-of-the-art fuzz testing-based methods.
(3) We release CrossFuzz as an open-source tool, which can be accessed at: https://github.com/yagol2020/CrossFuzz, to enhance future research in the domain of smart contracts.

The remainder of this paper is organized as follows. Section 2 illustrates the motivation. Section 3 introduces the backgroud. Section 4 introduces the three crucial steps of CrossFuzz, constructor parameter generation, inter-contract data flow analysis, and cross-contract fuzz testing. Section 5 presents the experimental design and results analysis. Section 6 reviews related research work. Finally, Section 7 offers conclusions.

2. A motivation example

In this section, we discuss the limitations of existing methods to detect cross-contract vulnerabilities via a motivating example.

Fig. 1 shows a smart contract. The contract under test *Hold* allows users to distribute and transfer tokens, while the dependent contract *PermissionManager* (*PM* for short) allows users to manage the permissions of *Hold*.

To cover the function releaseETH in lines 35 to 40 during the fuzz testing process, the following four operations need to be executed sequentially:

(1) Deploy the dependent contract *PM* and record the deployed contract address as *P*.
(2) Deploy the contract under test *Hold*, setting *P* as a parameter in the constructor during deployment to initialize the address of the dependent contract *PM* in *Hold*.
(3) Invoke the function addAddress of contract *PM* with *M*, which is the address of a deployed contract, as a parameter and add *M* to *permittedAddresses*.
(4) Invoke the function releaseETH with address *M* and a parameter of 0.

Then, the function modifier onlyPermitted in line 27 is executed to check whether the current contract state satisfies the execution conditions of releaseETH. Because address *M* was added to the *permittedAddresses* in step (3), the condition needed by the "require" statement in line 28 is satisfied, enabling the fuzzer to cover the function releaseETH.

When running the example by using sFuzz [19] and ConFuzzius [28], they achieve 42.99% and 66.41% bytecode coverage, respectively. Inspecting the bytecode coverage of sFuzz and ConFuzzius reveals that neither tool covered the function releaseETH. Fig. 2 shows the control flow graph (CFG) of the contract *Hold*. We found that the return value of EXTCODESIZE (0x932) is 0, causing the Ethereum Virtual Machine (EVM) to reture that the called contract is an empty contract by executing ISZERO (0x935), leading to REVERT (0x93d). As a result, the transaction was terminated, thus preventing the execution of the function releaseETH.

[2] EtherScan: https://etherscan.io/.

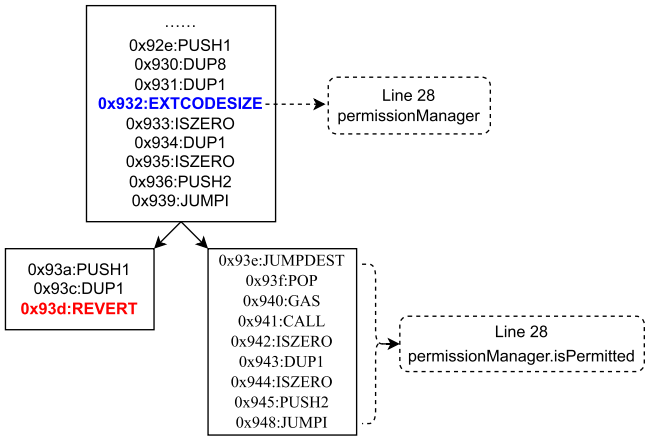

Fig. 2. The CFG of the contract *Hold* in Fig. 1.

When running the example using xFuzz [24], which is optimized for detecting cross-contract vulnerabilities, 62.48% bytecode coverage is achieved. Inspecting the bytecode coverage of xFuzz reveals that the function releaseETH was not covered. By checking the log of xFuzz, we observed that xFuzz predicted that no vulnerabilities exist in releaseETH by using a machine learning model. Therefore, xFuzz did not try to invoke releaseETH when generating the transaction sequences. However, releaseETH contains critical operations such as checking and transferring Ethers in *Hold*, making it essential to cover and check whether vulnerabilities are contained in this function. Based on the above analysis, we summarize two main reasons why the coverage of existing fuzz testing-based methods is low as follows.

First, existing methods do not set parameters for the constructor when deploying the contract under test, causing the EVM to use default values (e.g., assigning 0 to uint-type variables and 0x0 to address-type variables) to fill the parameters. Dependent contracts can be instantiated according to the parameters of the constructor, such as instantiate the dependent contract *PermissionManager* as *permissionManager* in line 25. However, if existing methods fill parameters of the constructor with default values and instantiate the dependent contract with zero addresses (i.e., 0x0), they cannot detect potential vulnerabilities in the contract under test.

Second, in cross-contract scenarios, it is necessary to execute functions both in dependent contracts and the contract under test. However, the number of transaction sequence combinations is exponentially increased. Although xFuzz leverages a machine learning model to reduce the number of transaction sequences that need to be executed, it may overlook some critical functions.

Based on the above analysis, we propose a cross-contract vulnerability detection method named CrossFuzz, which addresses the above challenges by focusing on constructor parameter generation and transaction sequence mutation strategies to improve code coverage and vulnerability detection capabilities.

3. Background

To better elucidate the relevant techniques involved in CrossFuzz, this section provides an introduction to Ethereum smart contracts and the background of fuzz testing.

3.1. Ethereum smart contracts

Ethereum smart contracts are computer programs that run on Ethereum blockchain platforms. Solidity is the programming language used to write Ethereum smart contracts, which are compiled into bytecode by compilers such as solc and executed by the Ethereum Virtual Machine (EVM). Specifically, a Solidity smart contract consists of multiple contracts, each composed of several state variables and functions. Functions are executed through transactions, which consist of senders, function signatures, function parameter values, and the amount of ether carried. In our paper, transactions are denoted as $f(p_1, \ldots, p_n)$, where f represents the function name, and $(p_1, \ldots, p_n)$ represents the values of parameters of that function. During the execution of a transaction, if an exception occurs, the EVM will roll back and restore the modifications made to the state variables by that transaction.

3.2. Fuzzing testing and ConFuzzius

Fuzz testing is one of the most effective methods in the field of security vulnerability detection. It initially generates test cases with random values for the program under test, and then sends the test cases to execute the program. Based on the collected path conditions of the program under test, it determines whether security vulnerabilities or unexpected behaviors of the program have been triggered. Finally, it generates offspring test cases according to the genetic algorithm and sends them to execute the program under test, until the termination conditions of the fuzz testing are met (such as reaching a predefined testing time, etc.).

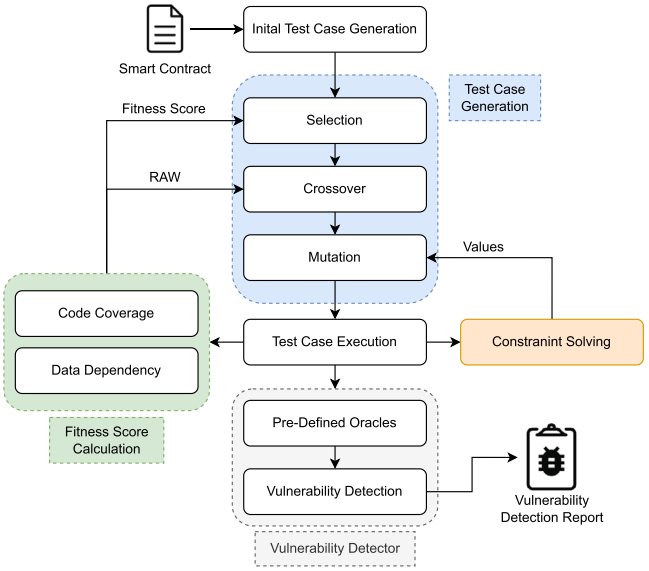

Fig. 3. The framework of ConFuzzius. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

```
{
  "9184": [  //Index of the instruction
    {
      "type": "Block Dependency",  //The type of vulnerability
      "individual": [  //The transaction sequence
        {
          "transaction": {
            "from": "0xdeadbeefdeadbeefdeadbe efdeadbeefdeadbeef",
            "to": "0x1c70b9d03f2387195cac4999476f9910bb88799d",
            "value": 0,
            "gasslimit": 4500000,
            "data": "0xbe45fd62000000000000000000000000...."
          }
        }
      ],
      "line": 252,
      "column": 1,
      "source_code": "now"  //The vulnerable statement
    }
  ]
}
```

Fig. 4. An example of a test case.

Because CrossFuzz is extended based on ConFuzzius, the relevant background of ConFuzzius introduced in this subsection. Fig. 3 shows the overall framework of ConFuzzius. ConFuzzius first generates initial test cases. Then, it generates offspring test cases through selection, crossover, and mutation, and sends them to the EVM to execute the test cases. Afterward, the instrumented EVM collects execution details. Based on these details, ConFuzzius calculates fitness scores to optimize the seed selection strategy, and generates function parameter values by constraint solving, which is used to cover branches that have not been explored yet. Finally, ConFuzzius determines whether there are vulnerabilities with respect to predefined oracles and the execution of test cases, and outputs the vulnerability detection report. The report includes the start time of fuzz testing, the execution details of each round of offspring test cases, and the specific test cases that trigger the vulnerabilities. The test case consists of information such as the type of vulnerability, the index of instruction that triggers the vulnerability, transaction sequences, and the vulnerable statement. Fig. 4 shows an example of a test case that triggers a “Block Dependency” vulnerability.

ConFuzzius innovatively combines constraint-solving techniques (the orange part in Fig. 3) with fuzz testing techniques, to solve the common magic byte problem in the field of fuzz testing, where branches with strict conditions are difficult to cover by randomly generated inputs, making it difficult to improve the coverage of the fuzz testing. In addition, ConFuzzius considers the Read-After-Write (RAW) dependency between state variables, improves the fitness function to optimize the seed selection strategy, and only connects a set of transaction sequences that satisfy RAW (i.e., the state variable is read by the previous transaction sequence and written by the next transaction sequence) in the crossover stage of the transaction sequences. Then, the transaction sequence is divided into two new transaction sequences to generate offspring test cases.

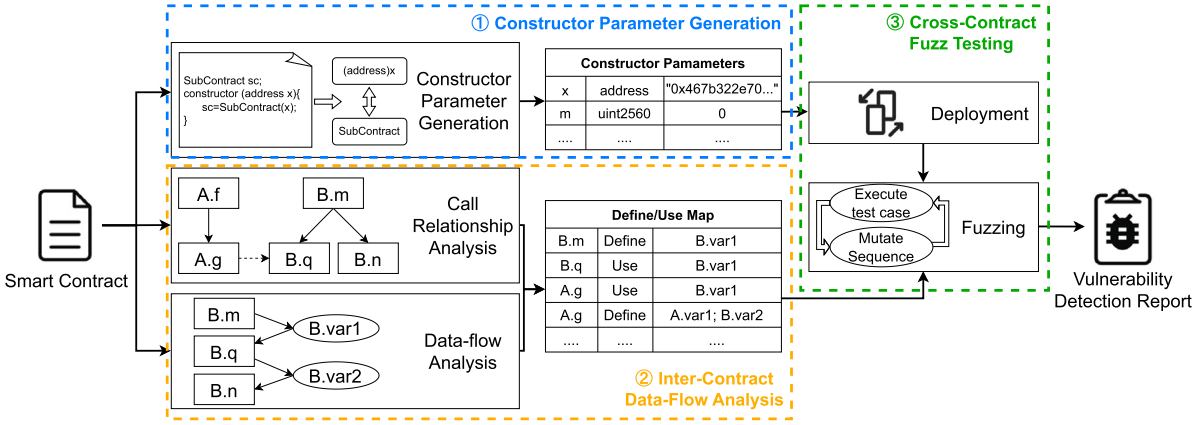

Fig. 5. The framework of CrossFuzz.

However, ConFuzzius does not support vulnerability detection scenario across contracts. It took default values as parameter values of constructors to deploy contracts, and overlooked the impact of cross-contract calls on data flow. These limitations affect the deployment of dependent contracts, thus reduce the code coverage and vulnerability detection capabilities of ConFuzzius. We propose CrossFuzz as an extension based on ConFuzzius to address these issues. Section 4 introduce the details of CrossFuzz.

4. Our approach

We present CrossFuzz, a fuzz testing-based method for cross-contract vulnerability detection, as shown in Fig. 5. First, CrossFuzz traces the data propagation paths of constructor parameters, analyzes the relationship between address-type parameters and dependent contracts, and generates parameters for the constructor. Then, CrossFuzz analyzes the ICDF to guide the mutation of transaction sequences. Finally, CrossFuzz deploys smart contracts using constructor parameters and performs fuzz testing. During fuzz testing, CrossFuzz executes test cases and monitors the execution of transactions, and then mutates transaction sequences according to the data flow information. Once the predetermined maximum testing time is reached, CrossFuzz terminates fuzz testing and outputs reports about transaction sequences that trigger vulnerabilities. These reports can help developers analyze and locate code vulnerabilities.

In the following, we will detail the three critical parts of CrossFuzz, i.e., constructor parameter generation, inter-contract data flow analysis, and cross-contract fuzz testing.

4.1. Constructor parameter generation

In this subsection, we elaborate on the process of generating constructor parameters, which involves tracing the data propagation path of address-type parameters of the constructor. The generated constructor parameters enable the constructor to accurately instantiate dependent contracts, which alleviates the limitations of existing fuzz testing methods.

Smart contracts automatically execute their constructors during deployment, enabling developers to assign values to state variables and initialize the contract state. Because only the contract creator can invoke the constructor, they typically set crucial information such as the contract owner, initial amount, termination service time, and dependent contract addresses. For instance, lines 20-24 in Fig. 1 set the multisig address and other variables of the contract *Hold*, while line 25 instantiates the dependent contract *PM*. If constructor parameters are not provided during contract deployment, the EVM fills them with default values, such as 0 for uint-type parameters and zero addresses for address-type parameters. However, if a dependent contract instantiated with zero address, fuzzers cannot invoke functions in the contract, resulting in transaction execution anomalies and hindering more contract code from being covered.

Information-Flow Analysis (IFA) techniques ensure information security by analyzing the legitimacy of data propagation. Based on IFA, a taint analysis process treats external input data as taint sources and tracks their data propagation paths to analyze whether a software system is secure [25].

Similar to taint analysis techniques, CrossFuzz tracks the data propagation paths of address-type parameters in the constructor, identifies whether the parameter is used by the constructor to instantiate dependent contracts, and establishes a mapping between address-type parameters and dependent contracts. When deploying the contract under test, CrossFuzz fills the addresses of dependent contracts into the constructor parameter to correctly instantiate dependent contracts. For address-type parameters for which corresponding dependent contracts are not found, CrossFuzz fills in the address of the contract's creator. For other types of constructor parameters, CrossFuzz follows fuzz testing-based methods like such as ConFuzzius, which are filled with default values.

CrossFuzz generates constructor parameters as shown in Algorithm 1. It inputs the contract under test C and the set of its dependency contracts *Deps*, and then outputs the generated constructor parameters of the contract C. Lines 1-2 initialize an empty

parameter list and obtain the address-type parameters in the constructor parameters; Lines 3-12 iterate the statements in the constructor, and if the statement is a type conversion statement that converts the address-type parameter to a dependent contract (i.e., contract conversion statements, for example, line 25 in Fig. 1), it indicates that there is a correspondence between the parameter and the dependent contract, and the parameter value is set to the dependent contract; Lines 13-23 iterate the constructor parameters and set the parameters that have not yet found a suitable value. The address-type parameters are set to the address of the contract's creator. For other types of parameters, they are set to the default values.

Algorithm 1 Constructor parameter generation.

Input: the contract under test C; the set of dependency contracts $Deps$
Output: the constructor parameters $args$ of the contract C

```
1: args = {}
2: adr = getAddressTypeArgs(C.constructor.args)
3: //iterate over the constructor statements
4: for stmt in C.constructor.statements do
5:     //statements for converting the type of variables
6:     if stmt is TypeConvert then
7:         if stmt.param in adr and stmt.type in Deps then
8:             //set the parameter value
9:             args[stmt.param] = stmt.type
10:        end if
11:    end if
12: end for
13: for arg in C.constructor.args do
14:    //the parameter value has not been set
15:    if args[arg] is Empty then
16:        if arg.type is address then
17:            //the address of the contract's creator
18:            args[arg] = contract_creator
19:        else
20:            args[arg] = default_value
21:        end if
22:    end if
23: end for
24: return arg
```

Then, CrossFuzz sends the constructor parameters generated by Algorithm 1 to the EVM, enabling CrossFuzz to execute the functions in the dependent contracts correctly. Specifically, after deploying the dependent contracts, CrossFuzz sets the constructor parameters of the contract under test and sends its bytecode and parameters to the EVM. Then, the EVM deploys the contract under test and automatically executes the constructor.

4.2. Inter-contract data flow analysis

To optimize the transaction sequence mutation strategy of CrossFuzz, we analyze the inter-contract data flow (ICDF) before fuzz testing. This process not only considers the definitions and uses operations of state variables within individual contracts, but also takes into account the call relationships between functions of different contracts.

Smart contracts modify the state variables of contracts through function calls (i.e., transactions), to change the state of contracts [26][27]. If the value of state variables is modified by transactions, the execution of subsequent transactions will be affected. Moreover, the number of transaction sequence combinations could be exponentially increased by cross-contract function calls because the number of functions is increased.

Smart contract fuzz testing methods usually generate transaction sequences based on data flow analysis, which means transactions containing state variable modification operations are executed first, and then transactions containing state variable reading operations are executed later [21][26][28]. Based on ConFuzzius, CrossFuzz analyzes the ICDF to obtain the definition and usages of state variables by functions. This information optimizes the transaction sequence mutation strategy to enhance vulnerability detection efficiency.

First, CrossFuzz analyzes the call relationship between contracts. The statements of a function are traversed, and the function call statements are analyzed to obtain the names of the called contracts and functions. Note that if a contract applies the function modifier fm to the function f, then the modifier is executed before the function itself. Therefore, CrossFuzz analyzes which functions are called by f and fm to obtain the call relationship of f. For example, the call relationship between *Hold* and *PM* is shown in Fig. 6.

Next, state variables that functions define and use are analyzed and recorded in the *define_map* and *use_map*. Specifically, the *define_map* records the state variables that are defined by functions, while the *use_map* records the state variables that are used by functions.

CrossFuzz obtains the map of definition and use for functions as shown in Algorithm 2. It inputs all functions F in the contract under test and dependent contracts, and outputs the define-map and use-map of F. Lines 1-6 represent the main body of the algorithm, while lines 7-19 represent the recursive procedure analysis, which is employed to obtain the *define_map* and *use_map* of function f.

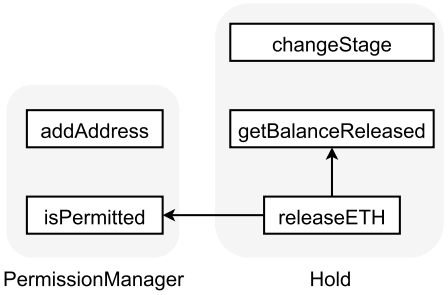

Fig. 6. Call relationships between PM and hold.

Algorithm 2 Analyze the definition and usage of state variables by functions.

Input: the set of functions F includes in the dependencies and under test contract
Output: the define map $define\_map$ and the use map use_map of F

```
1: define_map,use_map = {}, {}
2: for f in F do
3:     visited = []
4:     define_map[f],use_map[f] = analyze(f, [], [])
5: end for
6: return define_map,use_map
7: Procedure analyze(f, define, use){
8:     visited.add(f)
9:     define.add(f.stateDefine)
10:    use.add(f.stateUse)
11:    //traverse the other functions that f calls
12:    for callee in f.callee do
13:        if callee in visited then
14:            continue
15:        endif
16:        analyze(callee, define, use)
17:    endfor
18:    return define,use
19: }
```

Specifically, Line 1 of Algorithm 2 initializes *define_map* and *use_map*. Lines 2-5 sequentially analyze each function, with Line 3 initializing the list visited to record visited functions and Line 4 calling procedure analyze to recursively examine the state variables that are defined and used by the given function f. Line 6 returns *define_map* and *use_map*. In the recursive procedure analyze. Line 8 appends the currently analyzed function f to list visited. Lines 9-10 find which state variables are defined and used by f. Lines 11-17, based on the function call relationships between contracts, recursively investigate the other functions called by f one by one. Line 18 returns *define* and *use*.

4.3. Cross-contract fuzz testing

In this subsection, we elaborate on the process of cross-contract fuzz testing and propose calculating the priority scores of functions. With the priority score of functions, CrossFuzz optimizes the transaction sequence mutation strategy by altering the order of transactions within the test case, and generating offspring test cases that are more likely to cover previously uncovered branches and improves the vulnerability detection ability.

In cross-contract fuzz testing, CrossFuzz generates transactions that invoke functions in the contract under test and its dependent contract. Then CrossFuzz sends transactions to execute on the EVM and detect security vulnerabilities based on predefined oracles. Cross-contract fuzz testing consists of four steps, i.e., contract deployment, initial test case generation, test case execution, and test case mutation.

First, the contract under test and its dependent contracts are deployed. After sending the bytecode of dependent contracts to deployment on the EVM, CrossFuzz records the addresses of the deployed dependent contracts. Then, it deploys the contract under test with the constructor parameters generated in Section 4.1.

Next, initial test cases are generated. The Application Binary Interface (ABI) of smart contracts provides information such as function names, parameter types, and return value types. This information is used to generate initial test cases [19][20]. CrossFuzz adopts the same initial test case generation strategies as fuzz testing methods such as ContractFuzzer [20] and sFuzz [19]. Specifically, for each function in the contract under test or its dependent contracts, CrossFuzz generates a transaction sequence that contains only one function call. In contrast to the initial test case generation strategy of ConFuzzius, which randomly generates multiple transaction sequences, each contains one function call that may be repeated, CrossFuzz ensures that each function is invoked at least once.

Subsequently, the test cases are executed in sequence. The transaction sequences in the test cases are encoded and converted into hexadecimal data, and then sent to execute on the EVM. During the execution process, the changes in the stack, gas, block

environment and other information before and after executing instructions are monitored, which is used for security vulnerability detection, test case selection, and mutation.

CrossFuzz introduces code coverage feedback, which means that test cases covering branches previously unexplored during execution are mutated to generated offspring test cases [20][28]. According to the granularity, the test case mutation can be categorized into sequence-level and transaction-level. The former one changes the order or number of transactions, while the latter one alters parameters within transactions [21].

Section 4.2 explains how smart contracts modify state variables through transactions, to alter the state of contracts. To ensure that the state of the contract is appropriate, functions or function modifiers use "require" or "assert" statements to verify if the contract state satisfies execution conditions. If the conditions are not satisfied, then the EVM executes the REVERT instruction, to revert changes in state variables caused by the transaction, and terminates the execution of the transaction. Moreover, before invoking cross-contract functions, the EVM executes the EXTCODESIZE instruction to ascertain whether the called contract is empty; if it is empty, then the EVM also executes the REVERT instruction to terminate the execution of transaction.

In the situations described above, the EVM makes judgments based on the contract's state variables or local variables to determine whether an empty contract is invoked or REVERT is triggered during contract execution. Local variables are defined or used within a single transaction and cannot directly affect the execution of subsequent transactions. Therefore, when mutating the transaction sequence, we only take state variables into consideration.

Transactions that are terminated by the REVERT instruction are considered as exceptional transactions. These exceptional transactions could compromise the capability of code coverage and hinder the execution of subsequent transactions, which reduce the effectiveness of testing.

Our motivation is to provide an opportunity for the state variables used in exceptional transactions to be modified. In other words, by invoking transactions and generating random parameters, we enable the fuzzer to allow the EVM to meet the condition of branches when executing exceptional transactions again, thereby exploring branches that have not yet been explored yet.

To solve the problem of exceptional transactions, CrossFuzz optimizes the transaction sequence mutation strategy by calculating priority scores based on the ICDF. Functions with the greatest priority scores are added to the transaction sequence to supplement the state variable information need by the original exceptional transactions.

Note that CrossFuzz does not differentiate between the causes of exceptions in transactions. Consequently, CrossFuzz performs transaction sequence mutation for any transaction that executes the REVERT instruction. To calculate priority scores, we heuristically assign positive points if state variables are defined by the function and negative points if state variables are used by the function. Given a transaction sequence T and an exceptional transaction exp, the priority score of a function f consists of three parts as Equation (1) shows.

$$S(f, exp) = S_{Def}(f, exp) + S_{Provide}(f, exp) - S_{Use}(f, exp) \tag{1}$$

$S_{Def}(f, exp)$ is the number of state variables defined by function f. We consider that repeatedly modifying a specific state variable results in excessively long transaction sequences, which may reduce the efficiency of fuzz testing. Therefore, $S_{Def}(f, exp)$ only considers the number of state variables that are only defined by function f rather than other functions in the transaction sequence. As shown in Equation (2), $S_{Def}(f, exp)$ is obtained by subtracting the set of state variables defined by transactions preceding the exceptional transaction exp (denoted as $V_{PreDef}(exp)$) from the set of state variables defined by function f (denoted as $V_{Def}(f)$).

$$S_{Def}(f, exp) = |V_{Def}(f) - V_{PreDef}(exp)| \tag{2}$$

Similar to $S_{Def}(f, exp)$, $S_{Provide}(f, exp)$ is used to count the number of state variables that function f can provide for exp. The difference is that compared to $S_{Def}(f, exp)$, $S_{Provide}(f, exp)$ is concerned more about the additional state variables that function f provides to the transaction sequence. As shown in Equation (3), $S_{Provide}(f, exp)$ first takes the intersection of $V_{Def}(f)$ and the state variables used by exp (marked as $V_{Use}(exp)$), and then subtracts $V_{PreDef}(exp)$ from this result.

$$S_{Provide}(f, exp) = |V_{Def}(f) \cap V_{Use}(exp) - V_{PreDef}(exp)| \tag{3}$$

$S_{Use}(f, exp)$ is used to calculate the number of state variables only used by function f rather than the state variables also used by exp. We consider $S_{Use}(f, exp)$ as a side effect, and the larger this value is, the higher the likelihood of triggering exceptions when executing function f. As shown in Equation (4), $S_{Use}(f, exp)$ is obtained by subtracting $V_{Use}(f)$ from $V_{Use}(exp)$.

$$S_{Use}(f, exp) = |V_{Use}(f) - V_{Use}(exp)| \tag{4}$$

Note that $V_{Def}(f)$ and $V_{Use}(f)$ can be obtained from the ICDF of the function f, while $V_{Use}(exp)$ and $V_{PreDef}(exp)$ can be obtained through dynamic analysis of transaction execution. Specifically, CrossFuzz, utilizes the interface provided by ConFuzzius to analyze the stack changes during the execution of exp or other transactions. This analysis allows CrossFuzz to capture the data operations (i.e., define or use) of state variables for each transaction.

Intuitively, $S(f, exp)$ is defined as the number of state variables defined by f for the exceptional transaction exp. A greater priority score indicates that f can provide more information for exp. Consequently, f should be chosen when mutating the transaction sequence.

Note that only functions not invoked yet in the transaction sequence are chosen to calculate their priority scores. This is done to avoid the redundant execution of specific functions, and cover as many functions as possible.

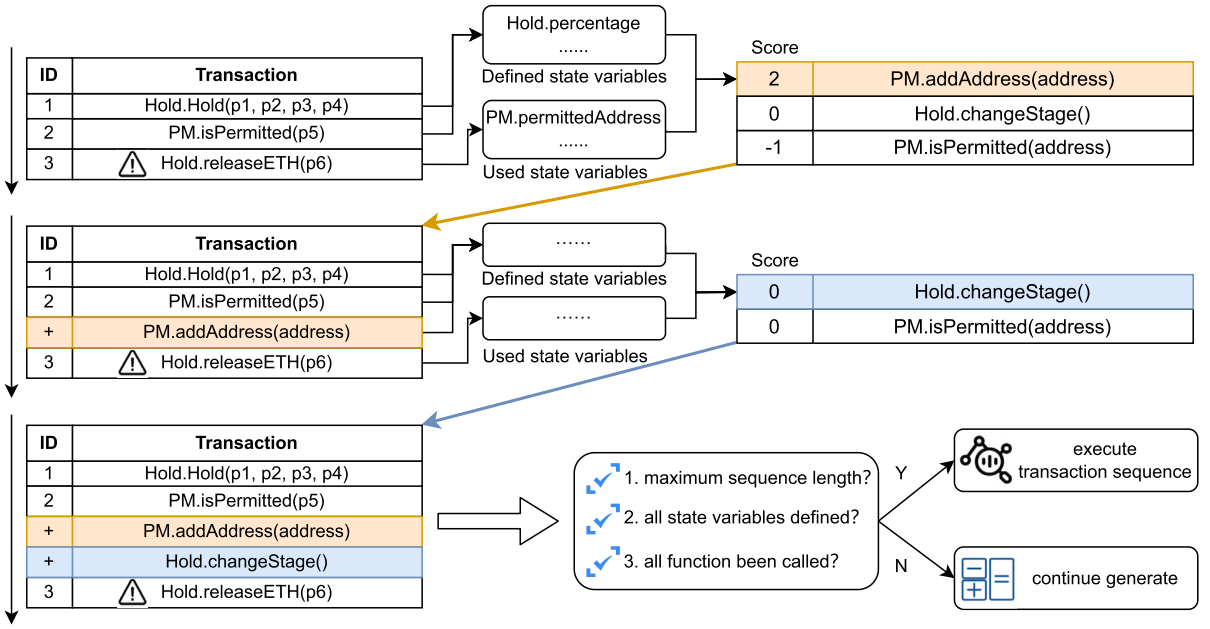

Fig. 7. The process of mutating transaction sequences.

Table 1
$V_{PreDef}(exp)$ and $V_{Use}(exp)$ of transaction sequence T.

$V_{PreDef}(exp)$	$V_{Use}(exp)$
$Hold.Percentage$ $Hold.multisig$ $Hold.initialBalance$ $Hold.permissionManager$ $Hold.observer$	$Hold.permissionManager$ $PM.permittedAddress$

Table 2
An example of calculating the priority scores of the function PM.addAddress.

Step	Calculation
1	$f = PM.addAddress$ $exp = Hold.releaseETH(p_6)$
2	$V_{Def}(f) = \{PM.permittedAddresses\}$ $V_{Use}(f) = \emptyset$
3	$S_{Def}(f, exp) = \lvert PM.permittedAddress\rvert = 1$
4	$S_{Provide}(f, exp) = \lvert PM.permittedAddress\rvert = 1$
5	$S_{Use}(f, exp) = \lvert\emptyset\rvert = 0$
6	$S(f, exp) = S_{Def}(f, exp) + S_{Provide}(f, exp) - S_{Use}(f, exp) = 2$

Take the code shown in Fig. 1 as an example to describe the detailed process of mutating the transaction sequence. When executing $Hold.releaseETH(p6)$ in the transaction sequence T. $T = \{Hold.Hold(p1, p2, p3, p4)$, $PM.isPermitted(p5)$, $Hold.releaseETH(p6)\}$, where $p1$, $p4$ and $p5$ are three valid and unique address, $p2 = p6 = 1$, $p3$ is the address of the deployed contract *PermissionManager*.

The function modifier $Hold.onlyPermitted$ is first called, which calls function $PM.isPermitted$. Since the state variable *permittedAddresses* in the dependent contract PM has not been modified, the return value of $PM.isPermitted$ is false, which means the condition in the "require" statement cannot be satisfied. Then the EVM executes the REVERT instruction to terminate the transaction execution. At this point, the exceptional transaction is $Hold.releaseETH$ in the transaction sequence T. Fig. 7 shows the mutation process of transaction sequence T.

First, CrossFuzz obtains $V_{PreDef}(Hold.releaseETH)$ and $V_{Use}(Hold.releaseETH)$ according to the ICDF, as illustrated in Table 1.

Next, CrossFuzz calculates the priority scores. Taking the function PM.addAddress as an example, the computation steps are shown in the Table 2.

Similarly, for exceptional transaction $Hold.releaseETH$, the priority scores of functions Hold.changeStage and PM.is Permitted are calculated as 0 and -1, respectively. After sorting the functions according to their priority scores, we observe that the function

Table 3
Statistics of the dataset.

Size	# Source Code	Avg Number of Contracts
<100	24	2.58
<500	245	6.21
>=500	127	12.06
Sum	396	6.95

PM.addAddress has the greatest priority score, so it is added to the transaction sequence T before the exceptional transaction, as shown in the yellow position in Fig. 7.

The transaction sequence mutation process is performed iteratively, and is terminated when any of the following three conditions is met:

(1) The transaction sequence length has reached the maximum threshold value.
(2) All the state variables used in the exceptional transaction (i.e., $V_{Use}(exp)$) have been defined.
(3) All the functions have been called.

Then, the mutated transaction sequences are sent to the EVM. CrossFuzz collects the execution information and checks whether the transaction sequence triggers any vulnerability. After, transaction sequences are mutated again based on the execution information until the predetermined testing time threshold is reached. Finally, CrossFuzz outputs vulnerability detection reports that contain information about detected security vulnerabilities, such as line numbers or transaction sequences that can trigger vulnerabilities.

5. Experimental design and evaluation

We implemented the CrossFuzz over the fuzz testing tool ConFuzzius which combines symbolic execution techniques in the fuzz testing process. ConFuzzius collects and solves symbolic constraints for uncovered branches based on the execution information, and uses the solved results as test cases for mutation, which improves code coverage compared to fuzz testing methods, such as ContractFuzzer [20]. CrossFuzz adds modules for multi-contract deployment, constructor settings, and transaction sequence mutation strategies on top of ConFuzzius. To evaluate the effectiveness of CrossFuzz, we design the following three research questions. To evaluate the effectiveness of CrossFuzz, we design the following three research questions.

RQ1: Does the transaction sequence mutation strategy improve the code coverage ability of CrossFuzz? If so, what is the optimal probability of enabling this strategy?

RQ2: Can CrossFuzz achieve higher code coverage and vulnerability detection ability than state-of-the-art fuzz testing methods?

RQ3: Do the generated constructor parameters improve the code coverage and vulnerability detection ability?

5.1. Experimental design

This section introduces the experimental design from four aspects: dataset, parameter settings, evaluation metrics, and experimental environment.

Dataset. The existing works for detecting cross-contract vulnerabilities, such as Clairvoyance [23] and SmartDagger [22], have not open-sourced their datasets, and the dataset provided by xFuzz [24] does not provide vulnerability labels, which cannot be used for experiments. We refer to the data selection standards of xFuzz and collect smart contract source codes from EtherScan as the dataset. Referring to the data preprocessing process of xFuzz, we remove smart contracts without external function calls. Specifically, 500 smart contract source codes are randomly obtained from EtherScan, and 396 contracts remain after removing contracts without external function calls by using the static analysis tool Slither [9]. Therefore, experiments are conducted based on the remaining 396 smart contracts. The contracts are divided into three sizes based on the lines of code, i.e., less than 100 lines (<100), greater than or equal to 100 lines and less than 500 lines (<500), and greater than or equal to 500 lines (>=500). The number of source code files (i.e., #Source Code) and the average number of contracts are shown in Table 3.

Evaluation metrics. We adopt commonly used evaluation metrics in smart contract fuzz testing [9][14], namely bytecode coverage and the number of vulnerabilities detected. The formula for calculating bytecode coverage is shown as Equation (5), where ins_{exec} is the number of executed instructions, and ins_{total} is the total number of instructions in the contract under test.

$$bytecode\_coverage = \frac{ins_{exec}}{ins_{total}} \tag{5}$$

The number of vulnerability detections usually requires the dataset to provide labels of the vulnerabilities. However, existing cross-contract vulnerability detection work has not open-sourced their datasets. Additionally, various tools support different types of security vulnerabilities, and there are differences in vulnerability detection rules. As the vulnerability detection rules of CrossFuzz are the same as those of ConFuzzius, only the number of vulnerabilities reported by CrossFuzz and ConFuzzius will be counted in the experiment. CrossFuzz and ConFuzzius can detect 11 types of security vulnerabilities, as shown in Table 4. For a detailed description of each security vulnerability, we refer to the relevant description of ConFuzzius [28].

Table 4
11 types of vulnerability supported by both CrossFuzz and ConFuzzius.

Vulnerability Type	Descriptions
Arbitrary Memory Access	An attacker can arbitrarily modify state variables, and the authorization checks may easily be circumvented
Assertion Failure	An unsatisfied assertion throws an exception to the contract
Integer Overflow	Numerical calculation errors due to arithmetic operations
Reentrancy	Repeated execution of functions unexpectedly
Transaction Order Dependency Block Dependency	The result of contract execution is related to the transaction order, block information, and this information can be influenced or controlled by the outside world
Unhandled Exceptions	The return value of the function or the exception on execution are not handled correctly
Unsafe Delegate Call	Dangerously use the Delegate function, which can execute external functions
Leaking Ether	Ether is not protected and is arbitrarily sent to external accounts
Locking Ether	Ether may not be transferable to an external account due to conditions not being met
Unprotected Self-Destruct	Contract suicide operations are not protected

In addition to evaluate the result by the average of bytecode coverage and the number of detected vulnerabilities, we also calculate the standard deviation and illustrated it on the bar charts to illustrate the statistical significance.

Baselines. We compare CrossFuzz with three open-source fuzzers named sFuzz,[3] ConFuzzius,[4] and xFuzz[5] as the baselines. sFuzz is implemented based on ContractFuzzer and the results of empirical studies show that sFuzz outperforms ContractFuzzer in terms of vulnerability detection [29]. ConFuzzius combines symbolic execution techniques with fuzz testing, exhibiting excellent code coverage capabilities. xFuzz is optimized for detecting cross-contract vulnerabilities and is also based on implementations of sFuzz and ContractFuzzer.

Parameter settings. sFuzz, ConFuzzius and xFuzz all use default parameters (for instance, the EVM versions are set to "byzantium", and for ConFuzzius, the data dependency analysis module and the constraint solving module are activated) and set the maximum testing time for one smart contract to 10 minutes, which follows the experimental design of other smart contract fuzz testing methods such as ConFuzzius. As the mutation of fuzz testing is stochastic, to avoid the influence of randomness on the experimental results, each smart contract is tested five times in the experiment, and the average of the five results is taken as the final experimental result. All experimental results are obtained on an Ubuntu 20.04 LTS machine with Intel Core i7-12700 and 32 GB of memory.

5.2. Evaluations

Results of RQ1: RQ1 focuses on whether the transaction sequence mutation strategy used in CrossFuzz improves the code coverage ability of fuzz testing. In the comparative experiment, the probability p of enabling the transaction sequence mutation strategy was set to 0%, 20%, 50%, 80% and 100% (referred to as $CrossFuzz_0$, $CrossFuzz_{20}$, $CrossFuzz_{50}$, $CrossFuzz_{80}$ and $CrossFuzz_{100}$, respectively) with the same values of other parameters. The bytecode coverage was recorded every 60 seconds, and the experimental results are shown in Fig. 8.

When tested for 60 seconds, $CrossFuzz_0$, which did not enable the transaction sequence mutation strategy, achieved the lowest coverage with 75.59% (±17.98%). $CrossFuzz_{50}$, $CrossFuzz_{100}$, $CrossFuzz_{20}$, and $CrossFuzz_{80}$, achieved coverage of 75.64% (±17.94%), 75.66% (±17.90%), 75.68% (±17.98%), and 75.95% (±17.89%), respectively. As the testing time increased, the coverage of all fuzzers gradually increased. With the strategy enabled, $CrossFuzz_{20}$, $CrossFuzz_{50}$, $CrossFuzz_{80}$, and $CrossFuzz_{100}$ achieved greater coverage than $CrossFuzz_0$. After 600 seconds of testing, $CrossFuzz_{80}$ achieved greater coverage, 79.92% (±18.30%), followed by $CrossFuzz_{20}$ (79.63% ±18.29%), $CrossFuzz_{100}$ (79.55% ±18.25%), $CrossFuzz_{50}$ (79.53% ±18.23%) and $CrossFuzz_0$ (79.06% ±18.12%).

Answer to RQ1: Enabling the transaction sequence mutation strategy based on the ICDF can improve the code coverage ability of CrossFuzz. The best improvement that is achieved when the probability of enabling the transaction sequence mutation strategy is 80%.

Results of RQ2: RQ2 focuses on whether CrossFuzz can outperform existing fuzzing methods in terms of contract bytecode coverage and vulnerability detection. The experimental results are shown in Fig. 9 and 10, where the horizontal axis is the contract size, Fig. 9 shows the bytecode coverage, and Fig. 10 shows the number of vulnerabilities detected.

Fig. 9 shows the comparison results of bytecode coverage. The coverage of sFuzz, xFuzz, and ConFuzzius gradually decreases with increasing contract size, because larger contract code tends to contain more combinations of transaction sequences, which reduces the efficiency of fuzzing. At the same time, Fig. 9 shows that the coverage of CrossFuzz outperforms other fuzzing methods at various contract sizes. Specifically, the average coverage of CrossFuzz is 79.92% (±18.34%), which is greater than that of ConFuzzius

[3] sFuzz: https://github.com/duytai/sFuzz.
[4] ConFuzzius: https://github.com/christoftorres/ConFuzzius.
[5] xfuzz_tool: https://github.com/ToolmanInside/xfuzz_tool.

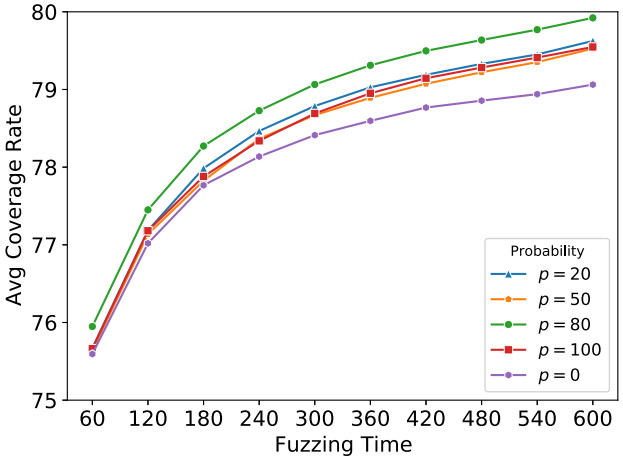

Fig. 8. Comparison of enabling the transaction sequence mutation strategy with different activation probabilities.

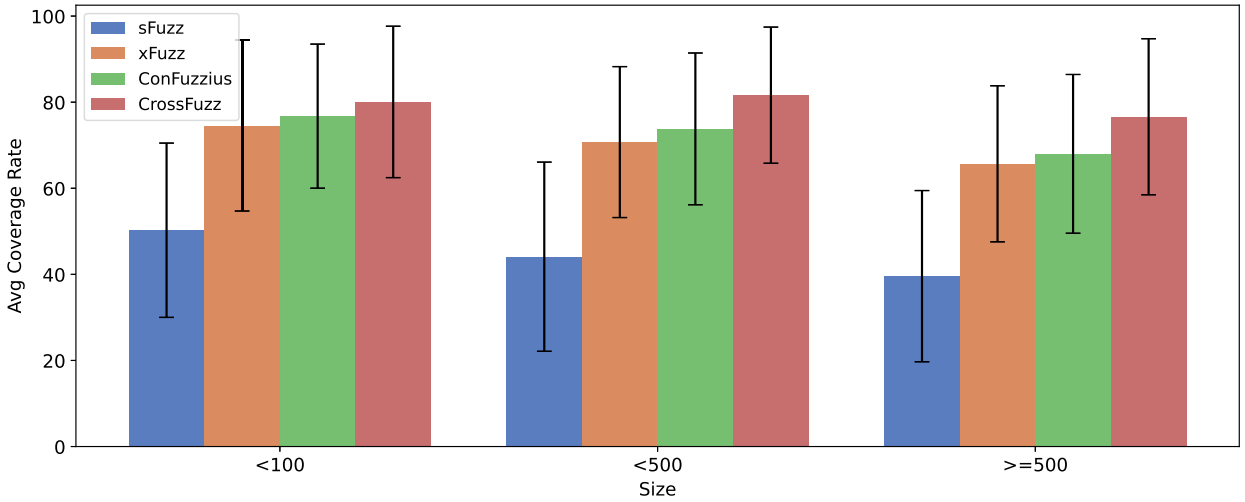

Fig. 9. Comparison of different fuzzing tools in terms of coverage.

(72.11% ±18.22%), xFuzz (69.34% ±18.11%), and sFuzz (43.03% ±16.87%), indicating that CrossFuzz outperforms other tools in code coverage.

Fig. 10 shows the comparison results of vulnerability detection. The number of vulnerabilities detected by CrossFuzz is greater than that detected by ConFuzzius at various contract sizes. Overall, after conducting fuzzing five times on 396 smart contracts, the number of security vulnerabilities detected by CrossFuzz and ConFuzzius are 12092 and 6662, which indicates 6.11 (±5.08) and 3.36 (±3.52) vulnerabilities are detected for one contract on average, respectively. In other words, CrossFuzz detected 1.82 times more security vulnerabilities than ConFuzzius on average. This occurs because CrossFuzz covers more contract code, and this part of the code is usually the core function of the contracts, with complex program logic, which is more prone to contain vulnerabilities.

Table 5 shows the number of vulnerabilities detected by ConFuzzius and CrossFuzz. CrossFuzz detects more vulnerabilities than ConFuzzius in all categories except “Leaking Ether”. Among these metrics, CrossFuzz has a larger improvement in “reentrancy” and “unhandled exceptions” vulnerabilities. For example, when testing contract *FixBet76* in Fig. 11, CrossFuzz found an unhandled exception vulnerability in line 9. The vulnerable statement did not check the return value after a cross-contract function call statement, so it could not handle the case of exceptions occurring during the function call, which could lead to errors in contract logic [29]. After manual inspection, we observed that ConFuzzius terminated transaction execution because the called contract address was zero when executing the statement, which resulted in uncovered code that prevented the vulnerability from being detected. CrossFuzz generated a transaction sequence $T = \{FixBet76.ownerSetEtherwowAddress(p), FixBet76.fallback()\}$, then instantiated the dependent contract Etherwow with the parameter p, then sent Ethers to contract FixBet76 to invoke the function fallback and successfully detected the cross-contract vulnerability in line 9.

One of the reasons why CrossFuzz detected fewer Leaking Ether vulnerabilities than ConFuzzius is that triggering such vulnerabilities requires the sender or receiver of the transaction to meet specific conditions (for example, the receiver must be a predetermined attacker). Since CrossFuzz deployed more contracts, generating the specific transaction sequences is more difficult than for Con-

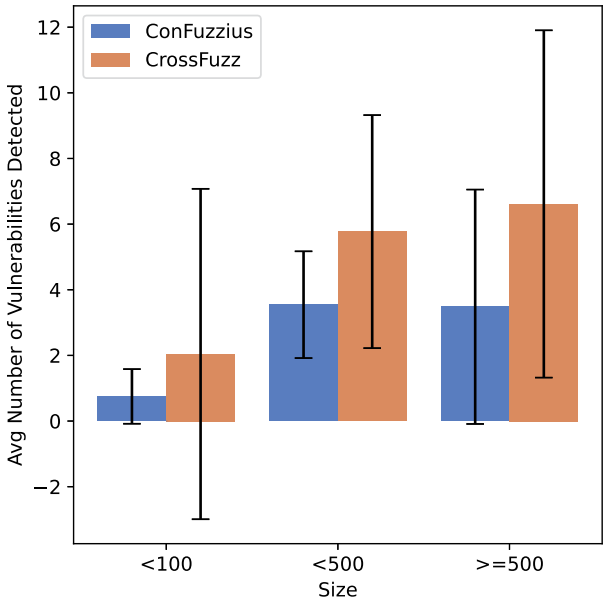

Fig. 10. Comparison of different fuzzing tools in terms of number of vulnerabilities detected.

Table 5
Comparison of the number of vulnerabilities detected by ConFuzzius and CrossFuzz.

Vulnerability Type	Number of vulnerabilities detected	
	ConFuzzius	CrossFuzz
Arbitrary Memory Access	20	38
Assertion Failure	1712	2189
Integer Overflow	771	955
Reentrancy	15	672
Transaction Order Dependency	229	944
Block Dependency	3477	4537
Unhandled Exceptions	423	2756
Unsafe Delegate Call	0	0
Leaking Ether	15	1
Locking Ether	0	0
Unprotected Self-Destruct	0	0
Sum	6662	12092

Fuzzius. Note that there are two false positives among the 15 vulnerabilities detected by ConFuzzius (an example of a false positive was analyzed in our repository[6]). We plan to further refine vulnerability detection oracles by analyzing the execution traces of transactions in future work.

Answer to RQ2: CrossFuzz outperforms existing fuzz testing tools in terms of contract code coverage and vulnerability detection. The average bytecode coverage of CrossFuzz is improved by 10.58% compared to that of xFuzz, and the average number of security vulnerabilities detected is 1.82 times greater than that of ConFuzzius.

Results of RQ3: RQ3 focuses on the influence of constructor parameters on fuzz testing. Similar to RQ2, the dataset is divided into three groups based on the size of the contracts, and the results are shown in Table 6. $CrossFuzz_{random}$ and $CrossFuzz_{zero}$ represent using random addresses and zero addresses to fill the address-type parameters in the constructor, respectively. CrossFuzz generates constructor parameters for address-type parameters by tracing the data propagation paths.

As shown in Table 6, $CrossFuzz_{zero}$, which uses zero addresses to fill the constructor, achieves the lowest coverage and the least number of detected vulnerabilities. Considering the contract size, we found that CrossFuzz performs well in smart contracts of larger size. When the size of the smart contract is greater than or equal to 500, compared with $CrossFuzz_{random}$ and $CrossFuzz_{zero}$, CrossFuzz increases the bytecode coverage by 3.55% and 5.90%, respectively, and detects 0.42 and 1.32 more security vulnerabilities for one contract on average.

[6] FP in LE, https://github.com/yagol2020/CrossFuzz/blob/dev/FP in LE.md.

```
contract FixBet76 {
  Etherwow public etherwow;
  function ownerSetEtherwowAddress(address newEtherwowAddress) public onlyOwner {
    etherwow = Etherwow(newEtherwowAddress);
  }
  function () payable {
    if (bet == true) {
      require(msg.value == 100000000000000000);
      etherow.userRollDice.value(msg.value)(76, msg.sender);
    }
    else return;
  }
}
```

Fig. 11. An example of cross-contract vulnerability.

Table 6
Comparison of CrossFuzz with different contractor parameter generation strategies.

Size	Statements in Constructor		Methods	Avg Coverage	Avg Number of Vulnerabilities Detected
	# Total	# Contract Conversions			
<100	13	0	CrossFuzz_{zero}	79.62 (±21.69)	1.88 (±1.62)
			$\text{CrossFuzz}_{random}$	79.69 (±21.75)	1.92 (±1.60)
			CrossFuzz	80.06 (±21.88)	2.04 (±1.62)
<500	726	68	CrossFuzz_{zero}	77.49 (±18.31)	4.88 (±4.57)
			$\text{CrossFuzz}_{random}$	80.09 (±17.99)	5.42 (±4.80)
			CrossFuzz	81.63 (±18.07)	5.77 (±4.97)
>=500	332	62	CrossFuzz_{zero}	71.11 (±18.89)	5.29 (±4.83)
			$\text{CrossFuzz}_{random}$	73.56 (±18.22)	6.19 (±5.14)
			CrossFuzz	77.01 (±76.61)	6.61 (±5.19)
Sum	1071	130	CrossFuzz_{zero}	75.58 (±18.98)	4.83 (±4.60)
			$\text{CrossFuzz}_{random}$	77.97 (±18.56)	5.46 (±4.88)
			CrossFuzz	79.92 (±18.34)	5.82 (±5.08)

Furthermore, after counting the total number of statements in the constructor and the number of contract conversion statements, we found that the proportion of contract conversion statements to the total statements in the constructor was 0%, 9.37%, and 18.67% for the three sizes of contracts, respectively. The improvement of CrossFuzz increased with the increase of this proportion. On the one hand, this result shows that larger contracts have more contract conversion statements in their constructors. On the other hand, it reflects the effectiveness of CrossFuzz in generating constructor parameters by tracing data propagation paths.

Answer to RQ3: Compared to using zero or random addresses to fill constructor parameters, CrossFuzz, which generates constructor parameters by tracking data propagation paths, can improve coverage and the number of detected vulnerabilities. Furthermore, when there are more contract conversion statements in the constructor of the contract under test, the performance of CrossFuzz is further improved than compared to that of the constructor parameter filling by zero address or random address.

5.3. Discussion

5.3.1. Threats to validity

This section analyzes the threats to validity from three aspects: internal validity, external validity, and construct validity.

Internal validity affects the correctness of the experiment, reflected in whether the implementation of CrossFuzz is correct. To alleviate the effects of internal threats, we implement CrossFuzz based on open-source tools. Specifically, the implementation of CrossFuzz consists of two modules: static analysis and fuzz testing. The static analysis module is implemented based on Slither to parse the source code of smart contracts and obtain information such as contract structure, function parameters, and statement types. Slither is a frequently updated static analysis tool for smart contracts that achieves high robustness. The fuzz testing module is implemented based on the fuzz testing framework ConFuzzius, and extension features such as multi-contract deployment and constructor parameter setting are developed based on this framework, and integrated with the transaction sequence mutation strategy proposed by our method. In addition, in the implementation of CrossFuzz, open-source fuzz testing tools such as sFuzz were referenced to improve the efficiency of test case mutation and to unify the evaluation of bytecode coverage, thus minimizing the threat of internal validity.

External validity concerns the generality of the experimental results, and it is reflected here based on whether the experiment is conducted in representative datasets. However, the existing cross-contract vulnerability detection approaches SmartDagger and Clairvoyance have not made their dataset publicly available, and the dataset published by xFuzz lacks vulnerability annotations. Hence, we randomly select smart contracts from EtherScan, which is an analytic platform for Ethereum, to construct the experimental dataset. To ensure the consistency of vulnerability detection rules, only the number of vulnerabilities detected by both CrossFuzz and ConFuzzius are compared in the experiment. In experiments, we compared CrossFuzz with three other fuzz testing tools: sFuzz,

xFuzz, and ConFuzzius. xFuzz is an extension of sFuzz and the only fuzz testing work we found that specifically targets cross-contract vulnerabilities. ConFuzzius is chosen as a comparison candidate because CrossFuzz is extended based on ConFuzzius. We have released CrossFuzz as an open-source tool to the community to facilitate subsequent research.

Construct validity concerns the evaluation metrics used in the experiment, which is reflected in whether the evaluation metrics can comprehensively evaluate the effectiveness of CrossFuzz. To alleviate this the effects of this threat, the experiments use bytecode coverage and the number of vulnerabilities detected as evaluation metrics, which are commonly used in smart contract fuzz testing [19][24]. In addition, this experiment also evaluates the time efficiency of CrossFuzz.

5.3.2. Future enhancements

In the following scenarios, CrossFuzz may fail to achieve higher code coverage, which can be further enhanced in future studies.

(1) Specific branches can only be covered by transaction sequences of longer lengths. The maximum length of transaction sequences generated by CrossFuzz is set to 5, which is inherited from the default parameter setting of ConFuzzius. Increasing the value of this parameter might enhance code coverage but also increase the time overhead of fuzz testing.
(2) Specific branches can only be covered by default values of state variables. CrossFuzz does not consider cases where the default values of state variables in the constructor already satisfy branch conditions. The parameter values generated based on the ICDF might compromise code coverage. However, CrossFuzz provides an interface for users to manually input specific value constructor parameters.
(3) Specific branches can only be covered by executing some functions multiple times. Since CrossFuzz only selects functions that have not been executed before to add to the transaction sequence, this scenario could be missed by CrossFuzz. Similar to scenario (2), CrossFuzz allows users to alter this strategy by parameter settings.

Furthermore, CrossFuzz does not support testing scenarios where contracts call other contracts via addresses. In future work, we plan to utilize EtherScan to search external contracts and obtain their source code, thereby expanding the applicability of CrossFuzz.

6. Related work

Fuzz testing uses random data as program inputs and analyzes the security vulnerabilities by detecting program exceptions [30]. In smart contract security testing, the fuzz testing method deploys the contract under test, generates test cases containing transaction sequences, and sends them to the smart contract for execution. During execution, the changes in data, such as the stack and block environment, are collected. Based on the collected information, the fuzzer could analyze whether a security vulnerability has been triggered.

Jiang et al. [20] proposed ContractFuzzer, which collects contract execution information by instrumenting the EVM, generates test cases based on the ABI and detects vulnerabilities based on pre-defined vulnerability patterns. Nguyen et al. [19] improved the seed mutation strategy based on ContractFuzzer and retained seeds closer to strictly conditional paths to generate descendant test cases. The experimental results showed that sFuzz improved code coverage and detected more security vulnerabilities. Choi et al. [21] proposed Smartian, which considers the influence of transaction sequences on code coverage, generates transaction sequences based on defined use relationships, and retains test cases that can cover more data flows as seeds. The experimental results showed that Smartian could cover more contract codes quickly. Xue et al. [24] built xFuzz based on sFuzz to address cross-contract scenarios. xFuzz deploys interdependent smart contracts in the test environment and leverages machine learning and fitness evaluation metrics to select functions that need to be executed to reduce the number of transaction sequence combinations.

The method based on fuzz testing requires the deployment and execution of smart contracts. It has the advantages of high precision and the ability to generate test cases for reproducing security vulnerabilities. However, existing methods mainly focus on a single contract. While xFuzz has optimized for cross-contract scenarios, it still has limitations in terms of model performance and test case generation, such as overlooking some critical functions because the model predicts that there are no vulnerabilities in the critical functions. Furthermore, existing methods do not take parameters of the constructor into account. Although random values can be generated as parameter values, contracts may incorrectly instantiated by these random values, which compromises code coverage and security vulnerability detection of the fuzzers.

To address these limitations, we propose CrossFuzz, which introduces two novel techniques. First, CrossFuzz tracks data propagation paths to generate the parameters of constructors, thereby enhancing its code coverage capabilities. Second, CrossFuzz optimizes the mutation strategy of transaction sequences based on function priority scores and supplements the state variable information needed for exceptional transactions The experimental results show that our approach significantly improves the efficiency of fuzz testing.

7. Conclusions

In this paper, we propose CrossFuzz, a cross-contract vulnerability detection method based on fuzz testing. CrossFuzz tackles two challenges, constructor parameters generation, and transaction sequences mutation. Firstly, it traces the data propagation paths of address-type parameters in the constructor parameters and generates the constructor parameters of the contract under test. Then, it analyzes the function call relationships between contracts and obtains each function's definitions or uses of state variables. Finally, it conducts cross-contract fuzz testing and optimizes the mutation strategy of transaction sequences based on the ICDF. The

experiments demonstrate that CrossFuzz improves bytecode coverage by 10.58% compared to xFuzz and detects 1.82 times more security vulnerabilities than ConFuzzius. In the future, we will apply CrossFuzz to more smart contract datasets to further evaluate the performance of CrossFuzz in cross-contract vulnerability detection.

CRediT authorship contribution statement

Huiwen Yang: Methodology, Software, Writing – original draft, Writing – review & editing. **Xiguo Gu:** Software, Writing – review & editing. **Xiang Chen:** Writing – review & editing. **Liwei Zheng:** Writing – review & editing. **Zhanqi Cui:** Funding acquisition, Methodology, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was supported in part by the Jiangsu Provincial Frontier Leading Technology Fundamental Research Project (No. BK20202001), the Innovation (Science and Engineering) Project of Scientific Research Base of Nanjing University of Aeronautics and Astronautics (No. NJ2023031), the National Natural Science Foundation of China (No. 61702041), and the Beijing Information Science and Technology University "Qin-Xin Talent" Cultivation Project (No. QXTCP C201906).

Parts of this work are based on the master's thesis of the first author [31].

References

[1] M. Alharby, A. Aldweesh, A.V. Moorsel, Blockchain-based smart contracts: a systematic mapping study of academic research, in: Proceedings of the 2018 International Conference on Cloud Computing, Big Data and Blockchain, 2018, pp. 1–6.
[2] A. Saini, D. Wijaya, N. Kaur, et al., LSP: lightweight smart-contract-based transaction prioritization scheme for smart healthcare, IEEE Int. Things J. 9 (15) (2022) 14005–14017.
[3] A. Raj, S. Prakash, Smart contract-based secure decentralized smart healthcare system, Int. J. Softw. Innov. 11 (1) (2023) 1–20.
[4] P. Kochovski, V. Stankovski, S. Gec, et al., Smart contracts for service-level agreements in edge-to-cloud computing, J. Grid Comput. 18 (2020) 673–690.
[5] H. Su, B. Guo, Y. Shen, et al., Embedding smart contract in blockchain transactions to improve flexibility for the IoT, IEEE Int. Things J. 9 (19) (2022) 19073–19085.
[6] T. Li, Y.Z. Fang, Z.L. Jian, et al., ATOM: architectural support and optimization mechanism for smart contract fast update and execution in blockchain-based IoT, IEEE Int. Things J. 9 (11) (2022) 7959–7971.
[7] J.C. Chen, X. Xia, D. Lo, et al., Defining smart contract defects on Ethereum, IEEE Trans. Softw. Eng. 48 (1) (2022) 327–345.
[8] L. Luu, D. Chu, H. Olickel, et al., Making smart contracts smarter, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery, 2016, pp. 254–269.
[9] J. Feist, G. Grieco, A. Groce, Slither: a static analysis framework for smart contracts, in: Proceedings of 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, IEEE, 2019, pp. 8–15.
[10] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, et al., SmartCheck: static analysis of Ethereum smart contracts, in: Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, Association for Computing Machinery, 2018, pp. 9–16.
[11] Z.P. Gao, L.X. Jiang, X. Xia, et al., Checking smart contracts with structural code embedding, IEEE Trans. Softw. Eng. 47 (12) (2021) 2874–2891.
[12] J.B. Gao, H. Liu, C. Liu, et al., EasyFlow: keep Ethereum away from overflow, in: Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings, IEEE, 2019, pp. 23–26.
[13] J.C. Chen, X. Xia, D. Lo, et al., DefectChecker: automated smart contract defect detection by analyzing EVM bytecode, IEEE Trans. Softw. Eng. 48 (7) (2022) 2189–2207.
[14] J. Krupp, C. Rossow, TeEther: gnawing at Ethereum to automatically exploit smart contracts, in: Proceedings of the 27th USENIX Security Symposium, 2018, pp. 1317–1333.
[15] M. Mossberg, F. Manzano, E. Hennenfent, et al., Manticore: a user-friendly symbolic execution framework for binaries and smart contracts, in: Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering, 2020, pp. 1186–1189.
[16] J.J. Huang, S.M. Han, W. You, et al., Hunting vulnerable smart contracts via graph embedding based bytecode matching, IEEE Trans. Inf. Forensics Secur. 16 (2021) 2144–2156.
[17] Z.G. Liu, P. Qian, X.Y. Wang, et al., Combining graph neural networks with expert knowledge for smart contract vulnerability detection, IEEE Trans. Knowl. Data Eng. 35 (2) (2023) 1296–1310.
[18] W. Wang, J.J. Song, G.Q. Xu, et al., ContractWard: automated vulnerability detection models for Ethereum smart contracts, IEEE Trans. Netw. Sci. Eng. 8 (2) (2021) 1133–1144.
[19] T.D. Nguyen, L.H. Pham, J. Sun, et al., sFuzz: an efficient adaptive fuzzer for solidity smart contracts, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Association for Computing Machinery, 2020, pp. 778–788.
[20] B. Jiang, Y. Liu, W.K. Chan, ContractFuzzer: fuzzing smart contracts for vulnerability detection, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Association for Computing Machinery, 2018, pp. 259–269.
[21] J. Choi, D. Kim, S. Kim, et al., SMARTIAN: enhancing smart contract fuzzing with static and dynamic data-flow analyses, in: Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering, IEEE Press, 2022, pp. 227–239.
[22] Z.Q. Liao, Z.B. Zheng, X. Chen, et al., SmartDagger: a bytecode-based static analysis approach for detecting cross-contract vulnerability, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, 2022, pp. 752–764.
[23] Y.X. Xue, M.L. Ma, Y. Lin, et al., Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts, in: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Association for Computing Machinery, 2021, pp. 1029–1040.
[24] Y.X. Xue, J.M. Ye, W. Zhang, et al., xFuzz: machine learning guided cross-contract fuzzing, IEEE Trans. Dependable Secure Comput. (2022) 1–14.
[25] L. Wang, F. Li, L. Li, et al., Principle and practice of taint analysis, J. Softw. 28 (4) (2017) 860–882.

[26] Z.G. Liu, P. Qian, J.X. Yang, et al., Rethinking smart contract fuzzing: fuzzing with invocation ordering and important branch revisiting, IEEE Trans. Inf. Forensics Secur. 18 (2023) 1237–1251.
[27] V. Wustholz, M. Christakis, 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, Association for Computing Machinery, 2020, pp. 1398–1409.
[28] C.F. Torres, A.K. Iannillo, A. Gervais, et al., ConFuzzius: a data dependency-aware hybrid fuzzer for smart contracts, in: 2021 IEEE European Symposium on Security and Privacy, IEEE, 2021, pp. 103–119.
[29] M. Ren, Z.J. Yin, F.C. Ma, et al., Empirical evaluation of smart contract testing: what is the best choice?, in: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, 2021, pp. 566–579.
[30] J.C. Chen, X. Xia, D. Lo, et al., Defining smart contract defects on Ethereum, IEEE Trans. Softw. Eng. 48 (1) (2022) 327–345.
[31] H.W. Yang, Research on Security Vulnerability Mining Techniques for Smart Contracts, M.S. thesis, School of Computer Science, Beijing Information and Science Technology University, Beijing, China, 2023.
[32] J. He, M. Balunović, N. Ambroladze, P. Tsankov, et al., Learning to fuzz from symbolic execution with application to smart contracts, in: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA, 2019, pp. 531–548.