Proceedings of the 3rd International Conference on Software Engineering and Machine Learning
DOI: 10.54254/2755-2721/150/2025.22282

# *Integrated Smart Contract Vulnerability Detection Technology Based on AFL Fuzzing Strategy and a Lightweight Seed Selection Strategy*

**Keyan Cao[1,2,*], Yuxin Kang[1], Xinlei Wang[1], Zhongyang Wang[1]**

[1]*School of Computer Science and Engineering, Shenyang Jianzhu University, Shenyang, China*
[2]*Liaoning Provincial Urban Construction Big Data Management and Analysis Laboratory, Shenyang, China*
**Corresponding Author. Email: caokeyan@gmail.com*

***Abstract:*** With the continuous development of blockchain technology, thousands of smart contracts have been deployed on the blockchain, and the number of smart contract vulnerabilities has increased significantly. In the task of smart contract vulnerability detection, fuzz testing methods are usually used for detection. Existing AFL-based methods are inefficient in generating test cases that meet complex path constraints. This study addresses the limitations of traditional fuzz testing techniques in detecting vulnerabilities related to strictly constrained conditional branches in Ethereum smart contracts. To overcome this challenge, we propose a hybrid framework that combines static semantic analysis with adaptive dynamic fuzz testing and combines a lightweight heuristic seed selection mechanism to prioritize path-sensitive mutations. Our method adopts semantic-aware operators to guide targeted exploration of protected execution paths while dynamically optimizing energy allocation among test cases. Experimental evaluation on benchmark contracts shows that compared with baseline methods, the proposed framework achieves significantly improved branch coverage and accelerated vulnerability detection, especially for critical security vulnerabilities such as reentrancy and arithmetic exceptions, without sacrificing detection accuracy. The results verify the effectiveness of our method in balancing exploration efficiency and analysis rigor for blockchain-oriented security verification.

***Keywords:*** Smart Contract, Vulnerability Detection, Fuzzing Testing, Test Case, Control Flow Graph

## 1. Introduction

In recent years, with the development of blockchain technology, a large number of smart contracts have been applied in financial, supply chain and other fields. According to the statistics of Etherscan [1], the number of smart contracts on Ethereum has been on the rise since 2016. As of December 31, 2021, the number of smart contracts on Ethereum was 205,138, and 78,414 smart contracts were generated in 2021, accounting for 38.2% of the total and representing an increase of 62.3% compared to 2020. It can be seen that with the continuous development of blockchain and decentralized technology, the scale of smart contracts is constantly expanding and gradually penetrating into various industries centered on finance.

DOI: 10.54254/2755-2721/150/2025.22282

Smart contracts are codes written in high-level languages, which may contain a large number of security vulnerabilities. The special feature is that once deployed on the chain, they cannot be easily modified, which means that a contract with vulnerabilities will always be vulnerable to attacks. In recent years, there have been more and more attacks on smart contract vulnerabilities, such as the DAO Attack in 2016 [2]and the Parity Wallet Attack in 2017 [3], when attackers stole more than 50million and more than30 million respectively, causing huge losses.

Therefore, for the security vulnerability problem of Ethereum smart contracts, we need to design an accurate and efficient vulnerability detection method to ensure accurate code writing before the deployment of smart contracts on the chain. Fuzz testing is a commonly used technical method for detecting security vulnerabilities in smart contracts [4]. It can automatically or semi-automatically generate test cases for the tested smart contract and monitor the process of running test cases to detect possible security vulnerabilities, protecting the security of smart contracts.

## 2. Method

We proposes a smart contract vulnerability detection method based on fuzzing technology. The core approach generates high-quality test cases through optimized path coverage in the contract control flow graph (as shown in Figure 1), combining static analysis (code structure parsing), dynamic analysis (execution monitoring), and predefined vulnerability oracles. The framework's three key modules work synergistically to maximize code coverage while maintaining detection accuracy for predefined vulnerability patterns [5].

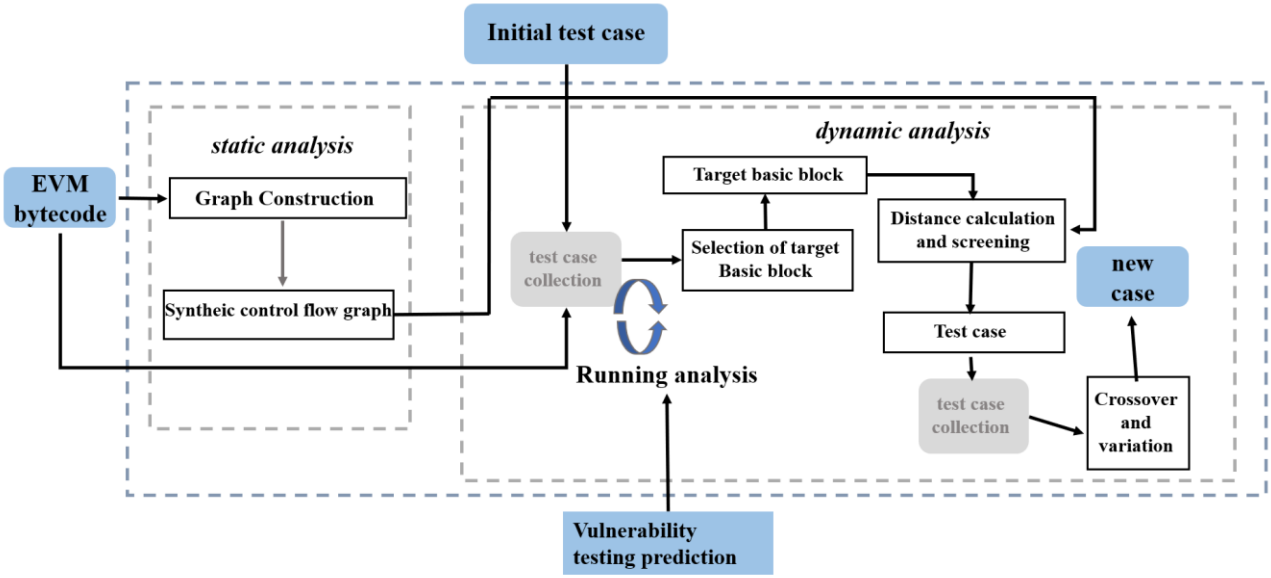


Figure 1: Framework of smart contract vulnerability detection method

### 2.1. Smart contract static analysis module

We establishes an integrated static analysis framework for smart contracts (Fig.2), combining AST processing, ABI decoding, and bytecode analysis to optimize vulnerability detection. The AST module identifies constant functions through syntax tree parsing, enabling selective exclusion of non-state-changing operations during testing. ABI decoding extracts critical function signatures and parameter schemas from interface definitions, facilitating structured transaction construction with valid input formats [6]. Bytecode analysis implements EVM instruction-level interpretation, detecting jump patterns and basic blocks to systematically reconstruct the control flow graph through path

DOI: 10.54254/2755-2721/150/2025.22282

tracing. These components work synergistically - AST filtering reduces test redundancy, ABI-derived prototypes guide targeted input generation, and bytecode-based CFG mapping ensures comprehensive path exploration - collectively enhancing fuzzing efficiency while maintaining execution path integrity [7].

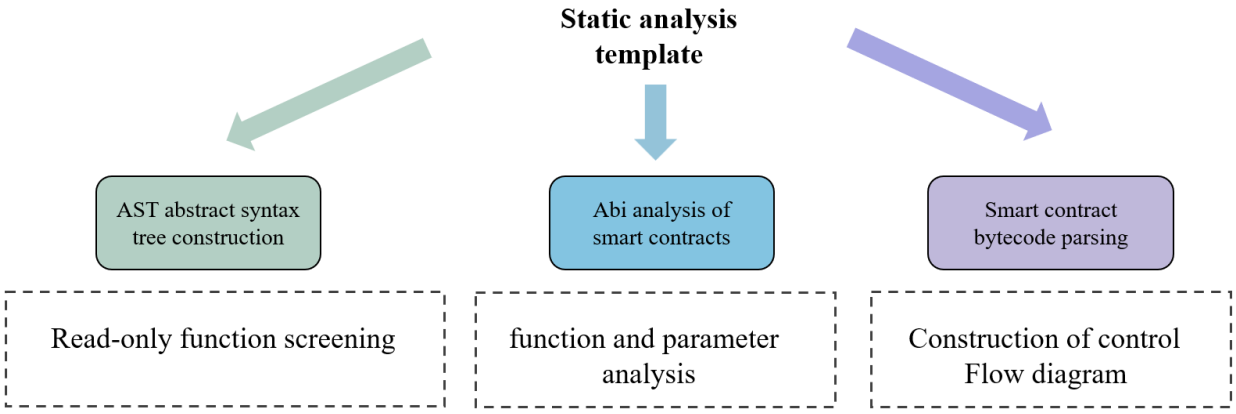


Figure 2: Smart contract static analysis module diagram

## 2.2. Smart contract dynamic analysis module

The dynamic analysis module of smart contracts mainly includes four parts: generating initial test cases, analyzing test case execution, screening test cases, and generating new test cases. Its core content is to use fuzzy testing technology to perform dynamic analysis on smart contracts, generate test cases with higher quality and higher probability of triggering security vulnerabilities, collect key operational information during the execution of test cases, and use predefined vulnerability testing prophecies to complete vulnerability detection of contracts.

The dynamic analysis module diagram of smart contracts is shown in Figure 3.

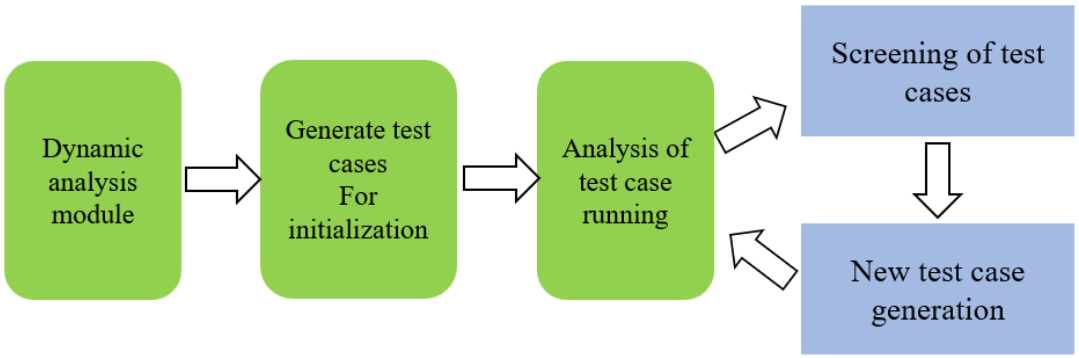


Figure 3: Dynamic analysis module diagram of smart contract

### 2.2.1. Generate initial test cases

In the initial stage of the dynamic analysis module, an initial test case needs to be generated. The test case includes a series of configuration information for the blockchain and a sequence of transactions, which is a sequence of function calls with specific parameters. When assembling the transaction data, we rewrite the function calls into ABI-encoded data, and then assemble the blockchain environment

DOI: 10.54254/2755-2721/150/2025.22282

information. We simulate a transaction initiated by a user on Ethereum, and then execute the transaction through the EVM virtual machine. The format of the test case is explained in detail below [8].

### 2.2.2. Analysis of test case running

The analysis of the running of test cases can be divided into four steps: the deployment and setup of the tested contract, the setup of the blockchain account pool, the construction and execution of transactions, and information analysis and vulnerability detection. The design diagram of this part is shown in Figure 4.

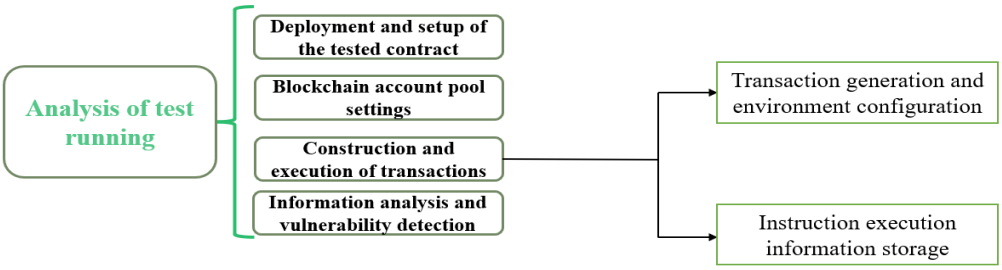


Figure 4: Design diagram for test case running analysis

### 2.2.3. Selection of test cases

In the context of test case prioritization, the objective is to identify test cases with enhanced adaptability and superior quality to serve as seeds for subsequent test case generation. To achieve this, we integrate AFL-based fuzzing strategies with a lightweight seed selection mechanism. The AFL framework prioritizes test cases that traverse previously uncovered program branches, deeming them high-fitness candidates for survival. While this approach efficiently covers most branches, it exhibits limited effectiveness in addressing branches guarded by stringent conditional constraints, as randomly generated inputs rarely satisfy such conditions. To mitigate this limitation, we introduce a novel quantitative evaluation strategy to systematically assess test case quality.

Specifically, we propose a lightweight seed selection strategy that computes the *distance* between a test case ( $t$ ) and an uncovered branch ( $br$ ), determined by its execution trace. This distance metric quantifies the proximity of ( $t$ ) to covering ( $br$ ), with smaller values indicating higher adaptability and suitability as a seed. The distance calculation, formalized in Equation (1), accounts for seven conditional scenarios governing branch ( $br$ ): ($c = \text{false}, a = b, a \neq b, a \geq b, a > b, a \leq b$,or $a < b$), where ( $a$ ) and ( $b$ ) represent constants or variables. This methodology enables targeted exploration of guarded branches while maintaining computational efficiency, thereby optimizing test case generation for complex conditional structures.

$$distance(t, br) = \begin{cases} K & ifc\ is\ false \\ |a - b| + K & ifc\ is\ a == b \\ K & ifc\ is\ a! = b \\ b - a + K & ifc\ is\ a >= b\ or\ a > b \\ a - b + K & ifc\ is\ a <= b\ or\ a < b \end{cases} \quad (1)$$

### 2.2.4. New test case generation

This phase employs a systematic mutation strategy to generate novel test cases with enhanced vulnerability-triggering potential through crossover and variation operations applied to prioritized

DOI: 10.54254/2755-2721/150/2025.22282

seeds. Each test case is treated as a byte stream subject to bit-level and byte-level manipulations. The mutation framework integrates AFL-inspired techniques with domain-specific adaptations for smart contract analysis, comprising six core operators:

(1) Bit-Flipping Mutations: Implement deterministic bit-flipping through singleWalkingBit (1-bit), twoWalkingBit (2-bit), and fourWalkingBit (4-bit) operations, sequentially altering individual bits across the input space.

(2) Byte-Level Mutations: Perform byte-oriented perturbations via singleWalkingByte (1-byte), twoWalkingByte (2-byte), and fourWalkingByte (4-byte) operations, modifying contiguous byte sequences through cyclic bit inversion.

(3) Boundary Value Substitution: Execute semantic-aware replacements using singleInterest (8-bit), twoInterest (16-bit), and fourInterest (32-bit) operators, substituting target regions with predefined boundary values known to trigger edge-case vulnerabilities.

(4) Adversarial Address Injection: Deploy overwriteWithAddressDictionary to inject predefined malicious actor profiles (NormalAttacker/ReentrancyAttack contracts). The former terminates interactions via exception throwing, while the latter initiates recursive callbacks to test reentrancy guards before controlled failure.

(5) Combinatorial Havoc: Apply multi-stage stochastic mutation through havoc, combining stacked mutations (arithmetic increments, block deletions, token swaps) across multiple execution rounds.

(6) Evolutionary Crossover: Conduct splice operations by interleaving byte segments between candidate inputs at randomized split points, followed by syntactic validation protocols to eliminate malformed test cases.

This hybrid approach probabilistically maximizes input space exploration while preserving semantic validity constraints inherent to Ethereum transaction structures, effectively balancing fuzzing diversity with blockchain-specific behavioral relevance.

## 3. Experiment and Results

To further demonstrate the effectiveness of the system, we compared it with existing smart contract vulnerability detection tools ContractFuzzer [10] and Oyente [11].The former is a smart contract blackbox fuzzing tool, while the latter is a symbolic execution tool. All experiments were run on Ubuntu 18.04 LTS, and the experimental process and results are shown below.

### 3.1. Experimental analysis of operational efficiency

The system efficiency comparison graph obtained after the experiment is shown in Figure 5.

Overall, this method generates and executes an average of 217 test cases per second, while ContractFuzzer and Oyente generate and execute an average of only 0.1 and 16 test cases per second. It can be seen that the efficiency of this system is much higher than the other two tools, which is due to the following reasons: 1) ContractFuzzer simulates the entire blockchain network and manages it, while this system only simulates the network or blockchain details related to smart contract vulnerabilities; 2) Oyente is a symbolic execution tool that requires a lot of time for constraint solving and computation, and its running speed is naturally lower than that of fuzzy testing tools.

DOI: 10.54254/2755-2721/150/2025.22282

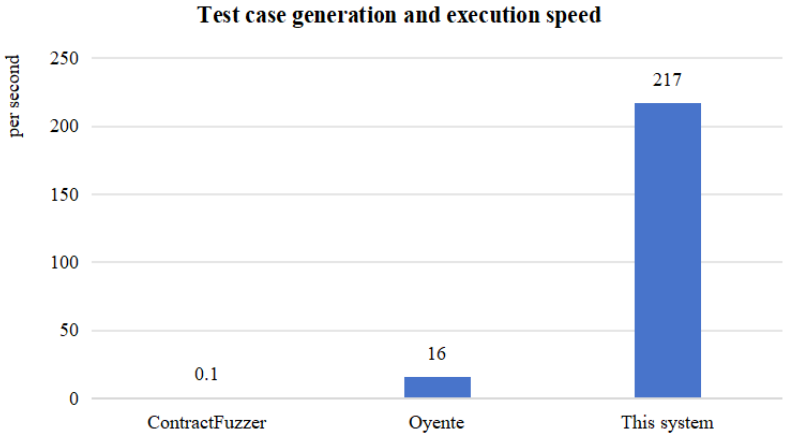


Figure 5: Comparison of system operating efficiency

## 3.2. Analysis of vulnerability detection experiments

Secondly, we hope to observe the effectiveness of the system in detecting vulnerabilities and its ability to detect real vulnerabilities. Therefore, we used this system and two other tools to detect vulnerabilities in the 453 real contracts collected, counted the number of vulnerabilities detected in each vulnerability category, and finally drew a vulnerability detection statistics table as shown in Table 1.

Table 1: Statistics of Vulnerability Detection

| Vulnerability type | This system | | ContractFuzzer | | Oyente | |
|---|---|---|---|---|---|---|
| | number | True positive rate | number | True positive rate | number | True positive rate |
| Gasless Send | 152 | 100% | 8 | 100% | 0 | N.A. |
| Exception Disorder | 16 | 100% | 4 | 100% | 0 | N.A. |
| Reentrancy | 6 | 100% | 2 | 100% | 12 | 58% |
| Timestamp Dependency | 54 | 80% | 12 | 75% | 25 | 100% |
| Block Number Dependency | 9 | 78% | 3 | 67% | 0 | N.A. |
| Dangerous DelegateCall | 1 | 100% | 0 | 100% | 0 | N.A. |
| Integer Overflow | 2 | 100% | 0 | N.A. | 98 | 60% |
| Integer Underflow | 31 | 85% | 0 | N.A. | 87 | 60% |
| Freezing Ether | 56 | 65% | 0 | N.A. | 0 | N.A. |

However, it should be noted that not all detected vulnerabilities are real. The detected contract vulnerabilities may have false positives. In order to evaluate the system's ability to detect real vulnerabilities, it is necessary to manually inspect the contracts with vulnerabilities in the report to determine whether they are true positives or false positives. Here, 20 contracts are randomly selected from each vulnerability category for manual inspection (if less than 20, all contracts are inspected). The results are shown in Figure 6.

DOI: 10.54254/2755-2721/150/2025.22282

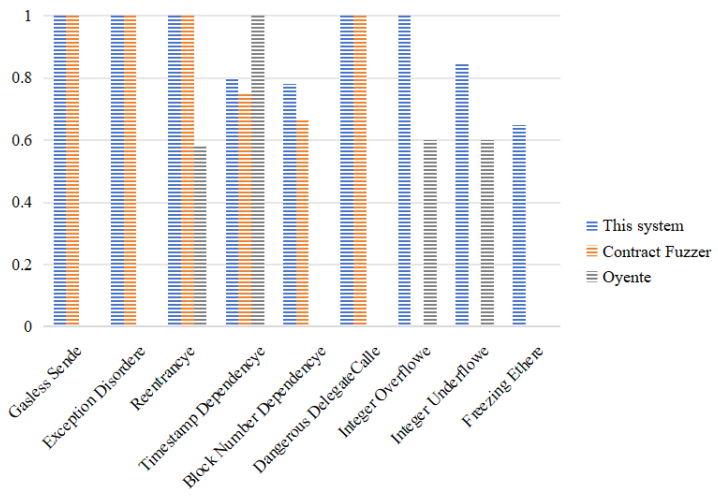


Figure 6: Statistical chart of true positive rate of vulnerability detection

Through the figure 6, it can be clearly seen that in addition to the timestamp dependency vulnerability, the true positive rate of this system for all vulnerability detection is significantly higher than that of ContractFuzzer and Oyente, proving that the system has a lower false positive rate and better ability to detect vulnerabilities.

Furthermore, to investigate the causes of false positives in this technology when detecting vulnerabilities, a further analysis of the contracts with false positives is necessary. This analysis should be conducted in conjunction with the design of the system's vulnerability detection framework to pinpoint the issues. Table 2 presents the true/false positive statistics from the sampled detections:

Table 2: System Sampling Detection Statistics

| Vulnerability Type | True Positives | False Positives | Total Number | True Positive Rate |
|---|---|---|---|---|
| Gasless Send | 20 | 0 | 20 | 100% |
| Exception Disorder | 16 | 0 | 16 | 100% |
| Reentrancy | 6 | 0 | 6 | 100% |
| Timestamp Dependency | 16 | 4 | 20 | 80% |
| Block Number Dependency | 7 | 2 | 9 | 78% |
| Dangerous DelegateCall | 1 | 0 | 1 | 100% |
| Integer Overflow | 2 | 0 | 2 | 100% |
| Integer Underflow | 17 | 3 | 20 | 85% |
| Freezing Ether | 13 | 7 | 20 | 65% |

For the timestamp-dependency vulnerability, there were five false positives because while the block timestamp was used in a condition, it was not related to the sending of Ether, i.e., there was no control dependency related to the transfer. Instead, these values were stored in local variables to record the time when a specific event was created, which the system falsely identified as a vulnerability.

For the block number-dependency vulnerability, the two false positives here were similar to the above. The block number was stored in local variables and was not related to the sending of Ether.

DOI: 10.54254/2755-2721/150/2025.22282

Regarding integer underflow vulnerabilities, the presence of three false positives was due to the system's inability to identify the correct type of a variable solely based on the bytecode (e.g., determining whether it is a uint128 or uint256). Therefore, it conservatively assumed that all arithmetic operations returning negative values potentially indicate a vulnerability.

For the Ether freezing vulnerability, the seven false positives occurred because, although there were program paths in the contract that allowed for the sending of Ether, they were not covered. If the system's runtime were increased, this false positive rate would likely decrease.

## 4. Conclusion

An intelligent contract vulnerability detection technology based on AFL-based fuzzy testing strategy and a lightweight seed selection strategy is proposed. Firstly, through static analysis technology, the AST syntax tree and bytecode of the smart contract are analyzed, and the key information of the contract is extracted to construct the control flow graph of the contract. Secondly, the smart contract is subjected to fuzzy testing through dynamic fuzzy testing technology, and higher-quality test cases are selected according to adaptive strategies. The test cases are cross-mutated using AFL-based mutation methods to generate test cases with higher probability of triggering vulnerabilities. Finally, during the fuzzy testing process, the key execution information is matched with predefined vulnerability testing prophecies to detect vulnerabilities in the contract. Through experimental comparison, the vulnerability detection method proposed in this paper has good performance, with the advantages of high running efficiency and low false positive rate.

## References

[1] *SCHAR F. Decentralized finance: On blockchain- and smart contract-based financial markets[J]. Social Science Electronic Publishing. 2021,103(2):153-174.*

[2] *Ethereum Daily Verified Contracts Chart[CP]. https://etherscan.io/charts,2020.*

[3] *BUTERIN V. Criticalupdatere: Dao vulnerability[OL]. https://blog.ethereum.org/ 2016/06/17/critical-update-re-dao-vulnerability/, 2017.*

[4] *The Multi-sig Hack: A Postmortem. Blockchain Infrastructure for the DecentralisedWeb [OL]. https://www.parity.io/blog/the-multi-sig-hack-a-postmor tem,2017.*

[5] *K. Delmolino, M. Arnett, A Kosba, et al. Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab[J]. In Proceedings of the 16th Financial Cryptography and Data Security, pp: 79-94, Berlin, 2016.*

[6] *Tu Liangqiong, Sun Xiaobing, Zhang Jiale, et al. Research Review on Smart Contract Vulnerability Detection Tools [J]. Computer Science, 2021, 48(11): 10..*

[7] *Zhu Yukai, Li Ying, Zhang Zhiqiang, et al. Smart Contract Vulnerability Detection Method Based on Dynamic Fuzzy Testing and Machine Learning [J]. Police Technology, 2021(6):5.*

[8] *Jiang B, Liu Y, Chan W K. Contractfuzzer: Fuzzing smart contracts for vulnerability detection[C]. 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE,2018:259-269.*

[9] *Nguyen T D, Pham L H, Sun J, et al. sfuzz: An efficient adaptive fuzzer for solidity smart contracts[C]. Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. 2020:778-788.*

[10] *Grieco G, Song W, Cygan A, et al. Echidna: effective, usable, and fast fuzzing for smart contracts[C]. Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2020:557-560.*

[11] *Wang Xin, Shi Qinfeng, et al. Deep Understanding of Ethereum [M]. Beijing: China Machine Press, 2019:112.*