Received September 11, 2019, accepted October 6, 2019, date of publication October 14, 2019, date of current version October 23, 2019.

*Digital Object Identifier 10.1109/ACCESS.2019.2947146*

# A Critical-Path-Coverage-Based Vulnerability Detection Method for Smart Contracts

**MENGLIN FU [1], LIFA WU [2], ZHENG HONG [1], FENG ZHU [2], HE SUN [1], AND WENBO FENG [1]**

[1]Command and Control Engineering College, Army Engineering University of PLA, Nanjing 210007, China
[2]School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210003, China

Corresponding author: Lifa Wu (wulifa@njupt.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2017YFB08029, and in part by the Nanjing University of Posts and Telecommunications Startup Foundation (NUPTSF) under Grant NY219004.

**ABSTRACT** The second generation of blockchain represented by smart contracts has been developing vigorously in recent years. However, frequent smart contract vulnerability incidents pose a serious risk to blockchain ecosystem security. Since current symbol execution tools often fall into path explosion and thus lead to inefficient detection, this paper expands Mythril's framework to optimize its performance. Firstly, it finds out potential vulnerable code regions using static analysis and identifies critical paths that may have security defects. Then, aiming at the problem that traditional search algorithms cannot actively locate and explore critical paths, this paper presents a multi-objective oriented path search (MOPS) strategy based on path priority. This strategy guides dynamic symbolic execution to cover critical paths quickly, avoiding blind traversal of program execution paths. Finally, it describes security rules and proposes corresponding detection logics for different vulnerability categories. This paper analyzes over 1000 smart contracts extracted from Etherscan. Compared with existing tools based on symbolic execution, the proposed method can reduce time consumption by around 35% while ensuring the accuracy of vulnerability detection. Moreover, existing tools often issue warnings that do not actually cause financial losses. But the proposed method only concentrates on code regions related to transfer of funds, so it can reduce the false alarm rate to some extent.

**INDEX TERMS** Block chain security, smart contract, vulnerability mining.

## I. INTRODUCTION

Smart contract expands the application of blockchain technology outside the financial sector, marking the arrival of the Blockchain 2.0 era [1]. With the widespread use of Decentralized Application (DApp), the number of smart contracts is growing explosively. Smart contracts involve a large number of digital assets and are immutable upon deployment, so they face even more severe security situation than traditional software [2]. Since 2016, various smart contract vulnerabilities have been exposed in numerous security incidents, causing huge property losses. At present, smart contract security mainly depends on developers' skill and code auditing experience. However, it is very laborious and time-consuming due to the increasing number and complexity of smart contracts. Thus, it has become a hot issue to develop a universal automated method for efficient vulnerability detection.

The associate editor coordinating the review of this manuscript and approving it for publication was Kuan Zhang.

### A. RELATED WORK

Formal verification, symbolic execution, and fuzzy test are the mainstream vulnerability detection methods for smart contract.

Formal verification formalizes smart contract documents and codes with formal language, and then checks their functional correctness and security attributes through strict mathematical logic and proof. It can also verify complex business logic and advanced properties such as economic problems and game theory. For smart contracts with small scale but complex functional design, formal verification is a suitable choice. Quantstamp [3] utilizes the traditional model checking techniques, and it requires lots of human effort to review the source code and write the specification manually. Securify [4] largely reduces manual effort. For each security attribute, Securify defines a compliance mode and a violation mode, and contract behaviors are classified as violation, warning, and compliance accordingly. But Securify can only check a list of fixed properties such as missing input validation and mishandled exception, rather than the


This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/

functional correctness. Furthermore, all these tools cannot verify complex systems. Yale University and Columbia University Security Team developed CertiK [5], which decomposes the code into smaller proof tasks according to the logical layer, and distributes these tasks to a distributed network. Bhargavan *et al.* [6] introduced a framework to analyze Ethereum contracts by translation into F*. The translation supports only a fragment of the EVM bytecode and does not come with a justifying semantic argument. Hildenbrandt *et al.* presented KEVM [7], the first fully executable formal EVM semantics in $\mathbb{K}$, providing an executable and human readable model of a reference semantics for EVM programs. Grishchenko *et al.* [8] presented the first complete small-step semantics for EVM bytecode and formalized a large fragment in the F* proof assistant. Moreover, they first defined a number of salient security properties for smart contracts, relying on a combination of hyper- and safety properties. Additionally, tools such as Isabelle/HOL [9] and Why3 [10] also implement the semantic representation of Ethereum Virtual Machine (EVM) and perform some formal verification work.

The symbolic execution technique symbolizes some variables on demand and then interprets the instructions in the program with symbolic values. Based on whether the variable values at specific program points satisfy the vulnerable condition, security defects can be determined. Oyente [11] is the first symbolic execution tool for smart contract. It supports detection for transaction order dependency (TOD), timestamp dependency, reentrancy, mishandled exception and integer overflow. Oyente relies on simplified semantics and adopts a pattern-based approach to define the concrete properties. However the tool lacks a semantic characterization. In addition, it reduces path explosion by simply limiting the number of loops, so it is difficult to report many security defects. Manticore [12] is a symbolic execution framework for both binaries and EVM bytecode. It can verify several common security problems of smart contract, but it does not have full supports on EVM instructions and handles storage access addressing inefficiently with concretization. Mythril [13] integrates several techniques, including symbol execution, taint analysis and control flow checking. It can discover a series of common security problems, but it is prone to false positives due to strict detection logic. TEETHER [14] not only supports vulnerability detection, but also realizes the automatic exploitation of smart contract vulnerabilities. It gives the definition of the vulnerable state, in which Ether can be sent to an attacker-controlled address. By means of program slicing and symbolic execution, it determines the transaction sequence to the vulnerable state, which is used for exploitation. However, it only supports detection of low-level security violations within a single contract. MAIAN [15] focuses on three classes of vulnerabilities: greedy contracts, prodigal contracts and suicide contracts. It characterizes these vulnerabilities as properties of execution traces, and identifies bugs generated from a trace of invocations. However, symbolic execution often suffers from path explosion problems when dealing with multiple loops and large-size contracts, and results in inefficient detection.

The fuzzy test technique uses mutation strategies to quickly change the seed input according to some heuristic methods. Incorrect input may cause the program to crash, hang up or lead to other unexpected behaviors. To the best of our knowledge, very few fuzzy tests have been carried out on smart contract. Related works include Echidna [16] and ContractFuzzer [17]. However, they do not present satisfactory results. The randomness of the input renders only a part of the path space accessible, while many complex parts of the program are protected by branch conditions. These branch conditions cannot be satisfied by random mutations, leaving the program far from fully explored.

### B. CHALLENGES

The limitation of existing tools and platforms for smart contract vulnerability detection lies in several aspects. Firstly, they support only a few types of vulnerabilities, which results in difficulties to achieve comprehensive security verification with a single tool. Secondly, they generally have high false positive rates and false negative rates. Thirdly, most of them cannot be fully automated and thus rely on additional manual review. At last, the excessive time consumption leads to poor auditing efficiency. Therefore, the priority of this work is to improve the accuracy, efficiency and degree of automation for the vulnerability mining task.

### C. SOLUTION

This paper takes a technical route that combines both static analysis and dynamic analysis. Through static analysis, potential vulnerable code regions are determined. Unlike similar symbolic execution tools that analyze all program paths, this work only focuses on paths that are related to Ether transfer and performs dynamic analysis on them. In this way, the method improves the efficiency of vulnerability mining. With respect to dynamic analysis, symbolic execution and taint analysis techniques are adopted. Existing tools mainly use traditional search strategies in symbolic execution, while this work preferentially explores more important paths. During the target-guided dynamic symbolic execution, critical paths of the smart contract are covered and a set of path constraints are recorded. At the same time, the taint analysis module marks program inputs and operation results that may lead to vulnerability as taint messages, and traces their flow so as to determine whether the tainted value can reach a hazard point. Furthermore, related works did not elaborate on how to detect various vulnerabilities. This paper proposed targeted detection logic to provide technical details for specific types of vulnerabilities.

### D. CONTRIBUTIONS

This paper has made the following contributions.

1. It defines the critical paths that have potential security defects, and adds a static analysis module to the LASER-Ethereum kernel of Mythril.

2. It proposes a heuristic path-priority-based search strategy that allows dynamic symbol execution to quickly approach critical code regions. The new strategy solves the problems that traditional search algorithms may have.

3. According to Ethereum's transaction execution model, it presents the execution properties of several common types of vulnerabilities and puts forward targeted detection logic.

4. It establishes a test set of smart contract source code covering various types of vulnerabilities and obtained 1000 deployed contracts from Etherscan for experiments. The results show that when compared to cutting-edge tools like Oyente and Mythril, the proposed method consumes much less detection time and incurs lower false positives to some extent.

### E. OUTLINE

The paper is structured as follows. Section II introduces the technical background and research motivation. Section III proposes the optimization strategies based on Mythril and elaborates on each module of the improved vulnerability mining framework. Section IV describes eight types of vulnerabilities that occur most frequently, defines their execution properties, and designs detection logic accordingly. Section V verifies the effectiveness of the proposed method by experiments and makes comparison with current mainstream tools. Section VI makes a conclusion, pointing out the merits and limitations of this work and looking ahead to future work.

## II. BACKGROUND

### A. SYMBOLIC EXECUTION

Symbolic execution technique was first raised by C. James and King in 1976 [18]. It symbolizes program parameters or other uncertain variables on demand. Instruction operands can be replaced by expressions consisting of symbolic variables and constants. Then it interprets program instructions one by one, updates the execution state and collects path constraints. Whenever the execution runs into a branch node, a fork is performed in order to complete the exploration of all executable paths.

Dynamic symbol execution, also called concolic execution, is an optimization of traditional static symbol execution. It maintains a concrete state and a symbolic state. Concolic execution first runs the program with concrete values, and collects the symbolic constraints of conditional statements during execution. Then it infers input changes with constraint solver so as to direct the next execution to another path. Famous concolic execution tools include DART [19], CUTE [20], KLEE [21] for source code, and S2E [22], Mayhem [23], angr [24] for binary. At present, the constraint solving technique has become a part of symbolic execution, playing an important role in test case generation and program path exploration.

Symbolic execution has some inevitable defects in practice. On the one hand, it is laborious and time-consuming to find all feasible execution paths for a relatively complex program. The biggest barricade is path explosion [25], which is mainly caused by: (1) In theory, there may be $2^n$ paths when

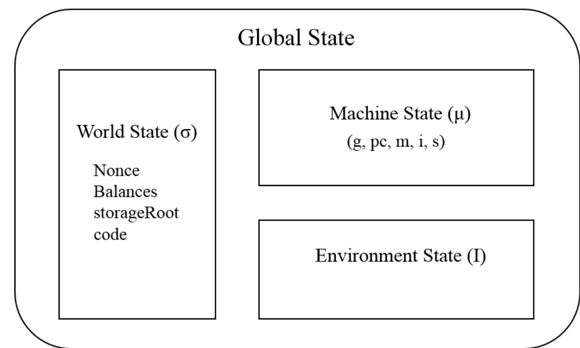



**FIGURE 1.** Transaction model.

$n$ is the number of branch nodes. The analysis system may run out of computation and memory resources; (2) If the loop variable or the recursive controlling variable is a symbolic value, symbolic execution will explore all the states in the variable state space. Large increase in path may lead to system failure after exhaustion. On the other hand, it is difficult to model operating system features such as file systems, sockets, and multithreading.

However, compared with the desktop operating system or mobile operating system, the operating mechanism of EVM is much simpler. So it is particularly relevant to smart contracts and can achieve complete path coverage in theory.

### B. TAINT ANALYSIS

Taint analysis is essentially a data flow analysis of taint variables. The general process of dynamic taint analysis is to dynamically obtain all executable paths in the program and determine the taint variables, express all instructions with intermediate language, perform forward or backward data dependency analysis according to propagation rule, and finally obtain a set of instructions which rely on taint variables or have impact on taint variables. Schwartz *et al.* [26] elaborated on dynamic taint analysis and forward symbol execution, and proposed a general language that standardizes the representation of taint propagation rule. Newsome *et al.* [27] first applied dynamic taint analysis to attack detection, and published BitBlaze [28], which is one of the earliest open source program analysis work. Modern platforms often integrate multiple program analysis methods, breaking the boundary between classical techniques. Zhang *et al.* [29] proposed a dynamic taint analysis method assisted by static disassembly information. In addition, dynamic taint analysis often works with symbolic execution to perform better program analysis [30], [31].

### C. TRANSACTION MODELLING

Ethereum can be regarded as a transaction-based state machine according to the Ethereum Yellow Book [32]. Once a new transaction is recognized by blockchain, it causes a state transition. Fig. 1 presents the structure of Ethereum Global State, which consists of three kinds of state variables, the World State, the Machine State and the Environment State respectively [32].

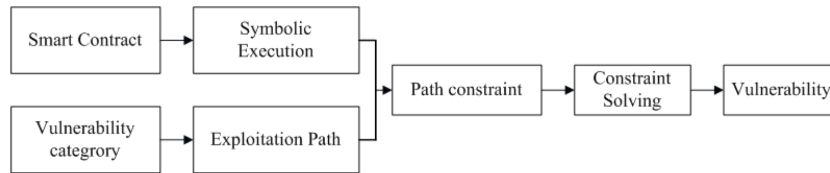



**FIGURE 2. Smart contract vulnerability mining based on symbolic execution.**

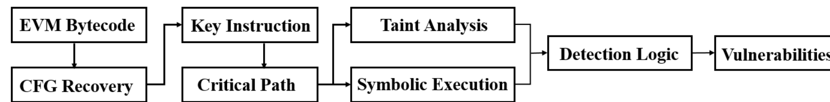



**FIGURE 3. Basic workflow.**

World State is also called account state and represents a mapping from 160-bit address to the account. Machine State indicates the running state of EVM. It is defined as a tuple $(g, pc, m, i, s)$, where $g$ is the amount of available gas, $pc$ is the program counter, $m$ is the memory content, $i$ is the active word in the memory, and $s$ is the stack content. Environment State is the execution environment associated with current transaction, including variables such as the address of transition originator $I_o$, the value of Ether sent to current account $I_v$, and the depth of current message call or contract creation $I_e$. These state variables in the transaction model may play a role in constraint collection during symbolic execution and vulnerability detection in the text below.

### D. MOTIVATION

Fig. 2 shows the main process of smart contract vulnerability mining based on symbolic execution: Conduct symbolic execution on contract bytecodes. For specific types of vulnerability, analyze potential exploitable paths and obtain their condition constraints. If there exists a solution, the path is considered to be reachable and there is a vulnerability.

Mythril is a notable smart contract security tool. It obtains the abstract program state and possible execution paths through its symbolic execution engine LASER-Ethereum. The path condition can be expressed as constraints on the symbolic values. With the symbolic description of the state on each program point and corresponding constraints, the vulnerability property can be represented as an expression about the value of variables. By detecting whether the conditional expression can be satisfied under the current constraint, Mythril can determine the possibility of vulnerability. However, smart contract introduces more instructions due to wider application prospects. Its increasing complexity inevitably causes path explosion. Accordingly, it becomes more difficult to symbolically execute smart contracts.

## III. CRITICAL-PATH-COVERAGE-BASED VULNERABILITY DETECTION METHOD

In order to reduce time consumption and improve the efficiency of vulnerability mining, it is necessary to select a part of contract paths to analyze. Assuming that an execution path containing the code region that should be cared about is a critical path, there is no need to consider the upper-level branches at the beginning and branch node. It is better to just direct execution traces to go through the path below the code region. Otherwise, each branch node is traversed to obtain a new execution path, which is likely to cause a space explosion. To avoid the blind search of path and improve the efficiency of vulnerability mining, this section improves the vulnerability mining process shown in Fig. 2 with a critical-path-coverage-based method. Fig. 3 presents the basic workflow.

Firstly, the control flow graph (CFG) of smart contract is constructed from EVM bytecode, and the sensitive instructions in the program are identified through static analysis. According to the basic block where the sensitive instruction is located, the code regions of interest are extracted. Then, dynamic symbolic execution is performed guided by multi-objective oriented path search (MOPS) strategy. This step records path constraints and marks certain program inputs or calculation results as taint messages based on the dynamic taint analysis scheme. The taint diffusion path is tracked until the taint value reaches a hazard point. Finally, the smart contract is automatically tested by different detection modules. Constraint solving technique is employed to calculate the input of potential exploitable path.

### A. OVERALL ARCHITECTURE

The optimizations are implemented on the basis of Mythril's framework. Fig. 4 presents the overall architecture.

There are five main components, among which the components in grey are optimized: (1) a static analyzer module is added into LASER-Ethereum; (2) the Symbolic Execution Engine module is modified; (3) the Detector module is mended. The other components, including Disassembler, Web Explorer and CFG Constructer, has been implemented by Mythril and other existing tools. These techniques are relatively mature, so we employ related modules of Mythril to avoid repeating previous works. The details of each module are described below.

The Disassembler module is used to compile the source code and transform bytecode into opcode. Web explorer

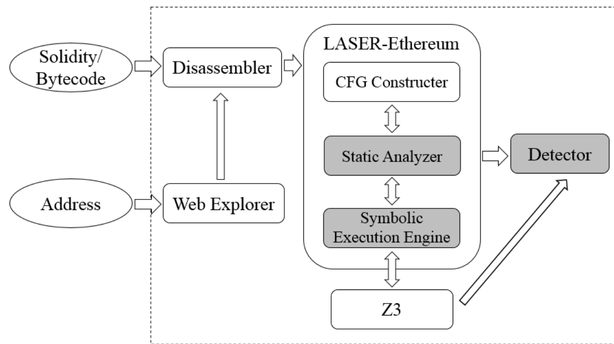



**FIGURE 4.** Overall architecture.

**TABLE 1.** Sensitive instruction list.

| Opcode | Instruction | Description |
|---|---|---|
| 0x55 | SSTORE | Save word to storage |
| 0xf1 | CALL | Message-call into an account |
| 0xf2 | CALLCODE | Message-call into this account with alternative account's code |
| 0xf4 | DELEGATECALL | Message-call into this account with an alternative account's code, but persisting into this account with an alternative account's code |
| 0xfa | STATICCALL | Similar to CALL, but does not modify state |
| 0xff | SELFDESTRUCT | Halt execution and register account for later deletion |

receives the address provided by users and searches for corresponding contract from the Internet. LASER-Ethereum is the core component of Mythril that performs symbolic execution. This work adds a new module, static analyzer, to this component. As explained in subsection B of section III, This module identifies sensitive instructions and marks critical nodes. In LASER-Ethereum, this work also changes the path search strategy of its original Symbolic Execution Engine, with technical details clarified in subsection C of section III. Z3 [33] SMT (Satisfiability Modulo Theories) solver is applied to perform constraint solving tasks. Detector is the component that carries out vulnerability detection. This work changes this module with better detection logic which is elaborated in section IV.

### B. EXTRACT CONTROL FLOW GRAPH

The CFG can be represented by a quad $G = (N, E, E_n, E_x)$, where $N$ is a set of nodes, each standing for a basic block. $E$ is a set of edges, indicating the branch structure and loop structure of the program. Given a smart contract, building a CFG involves the following steps [34].

(1) Convert the source code or bytecode into a sequence of readable opcode instructions by solidity compiler.

(2) Decompose the opcode instructions into basic blocks. The entry and exit boundaries of the basic blocks are determined according to some special opcode instructions.

(3) Determine the connection relationships of the basic blocks and construct the edges of CFG.

(4) Solve hanging blocks by stack simulation.

### C. DEFINE CRITICAL PATHS

Hackers exploit smart contract vulnerabilities for the purpose of economic benefits, such as stealing tokens or freezing funds. Smart contracts usually only allow authorized Ethereum accounts to accept tokens. If a contract allows Ether to be sent to an address controlled by an attacker, it is considered to be vulnerable. Accordingly, paths involved in Ether transfer are defined as critical paths.

#### 1) SENSITIVE INSTRUCTION

A critical path must contain sensitive instruction that can implement or affect Ether transfer. Although all instructions require gas, technically making them related to Ether transfer, they are not regarded as sensitive instructions as gas consumption does not constitute a vulnerability in most cases. Referring to the smart contract OPCODE instruction set [35] on GitHub, the instructions in Table 1 are defined as sensitive ones.

Storage saves the state variables of smart contracts permanently, and it can only be modified by SSTORE. Attacker may write to certain storage index through SSTORE to realize an illegal transfer.

Solidity functions such as send(), transfer(), and call().value() are compiled into CALL series bytecodes to transfer Ether. CALL series instructions include CALL, STATICCALL, CALLCODE, and DELEGATECALL. If their parameter ''add'' can be controlled, the caller contract may send the Ether to an address specified by the attacker, or be injected into any malicious code.

SELFDESTRUCT destroys the current contract and sends its balance to the specified account provided by the parameter ''address''. If an attacker is able to direct the contract to execute this and control the ''address'', all the Ether of the victim contract will be lost.

#### 2) CRITICAL PATH

Paths concerning with Ether transfer are regarded as the critical path. When constructing CFG, a flag *_is_Critical_Node* is set for each node to indicate whether the basic block contains sensitive instructions. If a node contains any one of SSTORE, CALL, CALLCODE, DELEGATECALL, STATICCALL, and SELFDESTRUCT, the flag *_is_Critical_Node* is set to 1, and the node is called Critical Node (CN). A program path that goes through CN is a critical path.

### D. TARGET-GUIDED SMART CONTRACT AUTOMATED TESTING

To cover all critical paths as quickly as possible in automated testing, this paper designs a target-guided automated testing algorithm, as shown in Algorithm 1. First, the candidate path set $\beta$ and the input vector of the program $\lambda$ (randomly initialized to $\lambda_0$) are initialized. When path $\omega$ is executed under the input vector $\lambda$, the state of the target node *CovCN* is overwritten. Then the symbol execution engine collects

all the candidate paths (i.e., path prefixes). According to the target critical node set that has not been covered, the unrelated candidate paths are eliminated, and the qualified candidate paths are added to the path set. Afterwards, according to the path search strategy specified in advance, a candidate path $P$ is selected as the next one to explore. By solving the path constraint of $P$, a new test input $\lambda$ can be obtained. The previous steps are repeated until all candidate paths have been traversed, or code coverage has reached the expected value. The final output includes test cases that meet the target coverage.

**Algorithm 1** Target-Guided Automated Testing Algorithm

**Input:** $\beta$: candidate path set, G: CFG, $\lambda$: input vector, $\psi$: path search strategy, $CovCN$: already covered CN, $UncovCN$: uncovered CN

**Output:** $T$: test cases

1: $\beta \leftarrow \emptyset, CovCN \leftarrow \emptyset, UncovCN \leftarrow CN, \lambda \leftarrow \lambda_0$ /*initialization*/
2: **repeat**
3: $\omega \leftarrow DSE_EXEC(\lambda)$ /*dynamic symbolic execution*/
4: $CovCN \leftarrow$Update($CovCN, \omega$) /*update*/
5: $UncovCN \leftarrow CN - CovCN$
6: path$\leftarrow$ SelectPath ($\omega$) /*collect candidate paths*/
7: $\beta \leftarrow$ FilterPath ($UncovCN, \omega, path$) /*unrelated path pruning*/
8: $P \leftarrow NextPath(\beta, \psi)$ /*select the next path based on $\psi$ */
9: $\lambda \leftarrow$SolveConstraints(P) /*new testcase generation */
10: $T \leftarrow T + \{\lambda\}$
11: **until** $\beta = \emptyset \vee UncovCN = \emptyset \vee$ CoveragePercent $\geq \theta_{\text{expet}}$

The key modules in the automated testing process are unrelated path pruning and path search strategy.

#### 1) MULTI-OBJECTIVE ORIENTED PATH SEARCH STRATEGY

Path search is the core part of dynamic symbol execution. Its function is to select the program state that needs to be executed first until reaching the target. The default path search strategy for most existing symbol execution tools such as Mythril is depth-first search (DFS). DFS traverses the program paths from the root of CFG. Whenever a branch node is reached, the symbolic execution engine clones the current program state and goes along one of the branches. The search process is terminated if all paths are explored or the target location is approached. DFS algorithm is exhaustive in a given state space and is suitable for complete coverage testing of program paths. It lacks active positioning and exploration for critical paths prone to defects, resulting in low efficiency.

Heuristic search is more adaptive to approaching sensitive instructions and covering critical paths quickly. Such algorithms are similar to breadth-first search (BFS), except that they evaluate each location and search from the best one. In other words, heuristic search algorithms preferentially explores along the nodes with inspirational information [36]. These nodes may be the best way to reach the target. The key to a heuristic search algorithm is valuation function, which is usually designed for a specific problem. The valuation function estimates the cost from a particular node to the destination node. Employing heuristic search can avoid much unnecessary path and improve the efficiency of exploration.

Considering the situation that one execution path passes by several targets, or the shortest path of multiple targets goes through the same branch of the current execution path, that is to say, there is an overlapping area. If there are multiple candidate program paths, choosing those paths that can reach more targets can speed up the coverage testing. Therefore, this paper proposes a heuristic path search strategy based on path priority, called MOPS.

(1) Priority evaluation of candidate path

For each branch node of an already executed path, evaluate the density of the remaining target nodes along its local area, and the density stands for the priority of corresponding candidate path. The priority is actually a heuristic estimate, defined as the number of targets that may be reached after the candidate path is executed, expressed as

$$priority(P) = UncoveredCN(Br, SearchCN(Br, \gamma))$$

In the formula, $Br$ is the end node of the candidate path. Function $SearchCN(Br, \gamma)$ serves to start from $Br$ and step forward for $\gamma$ levels, counting the number of unrecovered CNs that may be reached. If $l$ is the height of $Br$ in the CFG, the searched area will be limited to layer $[l, l + \gamma]$. The layer range can be adjusted by setting the value $\gamma$. Since the prediction of the local area is intuitively less expensive than a global prediction, a smaller value is generally selected.

(2) Candidate path selection

In order to reach as many coverage targets as possible during a single execution, candidate path with the largest priority value is first selected. That is to say, the paths are ranked based on their contribution degree to target nodes coverage. If two candidate paths share the same priority value, the shorter one is preferred. After a program iteration, the priority values of all remaining candidate paths are updated.

As shown in Fig. 5, nodes marked with shadow are CNs. If path 1-2-3-5-8 is executed, two candidate paths are added along the path branch, namely 1-2-3-6 and 1- 2-4. Under the MOPS policy, if $\gamma$ is set to 3, the path 1-2-4 has a higher execution priority because it may reach 2 uncovered CNs.

#### 2) UNRELATED PATH PRUNING

In order to distinguish the candidate paths, the following definition is made: if a candidate path is likely to reach a critical node that has not been covered in the tested contract, then it is the relevant path, otherwise, it is an unrelated path. Algorithm 2 shows the method of unrelated paths pruning.

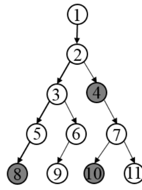



**FIGURE 5.** Candidate path selection.

**Algorithm 2** Unrelated Path Pruning Algorithm

**Input:** $\beta$: candidate path set, *UncovCN*: uncovered CN, path: newly collected candidate path set
**Output:** $\beta$: candidate path set after path pruning

```
1:  for all P ∈ path do
2:      /*P is path prefix, rather than a complete path*/
3:      if isRelevant( )==True then
4:          β.push(P)
5:      end if
6:  end for
7:  function isRelevant(Path P):
8:  if Const.solve()== ∅ then
9:      return False
10: end if
11: if P.Succs∩UncovCN == ∅ then
12:     return False
13: end if
14: return True
```

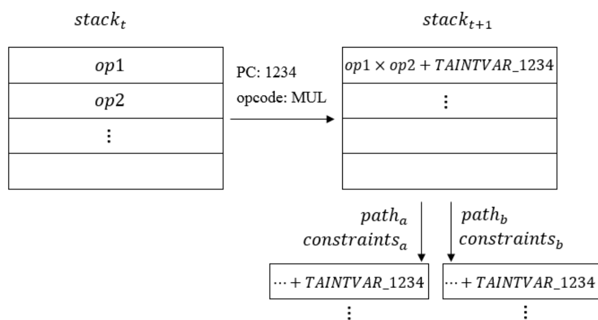



**FIGURE 6.** Taint tracking.

### *E. TAINT ANALYSIS*

The taint analysis technique is applied to assist the detection module. For certain program input or operation result that may generate vulnerabilities, a specially named symbolic variable is added to it as a taint tag. This symbolic variable takes a value of 0 and is named "TAINTVAR_PC", in which "PC" is the value of program counter.

As shown in Fig. 6, the taint tag spreads to subsequent branches along with the potential dangerous data, participating in computation without changing the results. The process of taint tracking is to perform data flow analysis according to the propagation rule. The propagation rule defines a list of operations that may propagate taint messages or may lead to new taint messages during execution. At certain program points, security-critical parameters or state variables are judged tainted or not on demand, according to whether they carry taint tags or not. If the security-critical data of a hazard point is tainted, its taint source can be determined according to the taint tag.

### *F. CONSTRAINTS GENERATION*

The Z3 SMT solver is applied to solve constraints and aid the symbolic execution engine. Fixed-length elements, such as call value or caller address, are represented with fixed-length bitvector. Variable-length elements, such as calling data, memory and storage, are modeled with Z3 array expressions. When the execution reaches a sensitive instruction, parameters in the stack are obtained from the current state space and will be used for later analysis. For a given path $P$, the constraint expression is the logical AND of all branch conditions on the path. The final output of the constraint generation module is the formula for variables in the Machine state ($\mu$) and the Execution environment ($I$).

## IV. VULNERABILITY DESCRIPTION AND DETECTION LOGIC

The previous section discussed the basic framework of smart contract vulnerability detection based on critical path coverage. Constraints for critical paths and taint propagation information were obtained through symbolic execution. On this basis, the vulnerability detection module performs security verification. Combined with smart contract audit experience and security incidents, this paper selects eight types of vulnerabilities, describes their execution properties, and designs detection logic accordingly in this section.

### *A. REENTRANCY*

#### 1) DESCRIPTION

Reentrancy is one of the most common vulnerabilities for smart contract. The world-shaking DAO (Decentralized Autonomous Organization) [37] incident exploited the reentrancy vulnerability in the contract, stealing 3.6 million ETH (Ether). When contract A calls another contract B, A will wait for the call to complete execution before moving on to the next instruction under normal conditions. But if the callee contract B interrupts this invocation by calling A's fallback function, making contract A run under an inconsistent internal state, which obviously violates the developer's intention and may produce unexpected behavior. In the sample program shown in Appendix 1, an attacker can utilize the contract BankAttack to interact with the victim contract Bank. The attack process is illustrated in Fig. 7.

(1) Call the function *deposit*() in BankAttack and send 100 Wei to Bank. In this way, function *addToBalance*() in Bank is invoked.

(2) The first withdrawal: call BankAttack's function *withdraw*() to withdraw funds (takes 10 Wei). At the same time, function *withdrawalBalance*() in contract Bank is triggered.

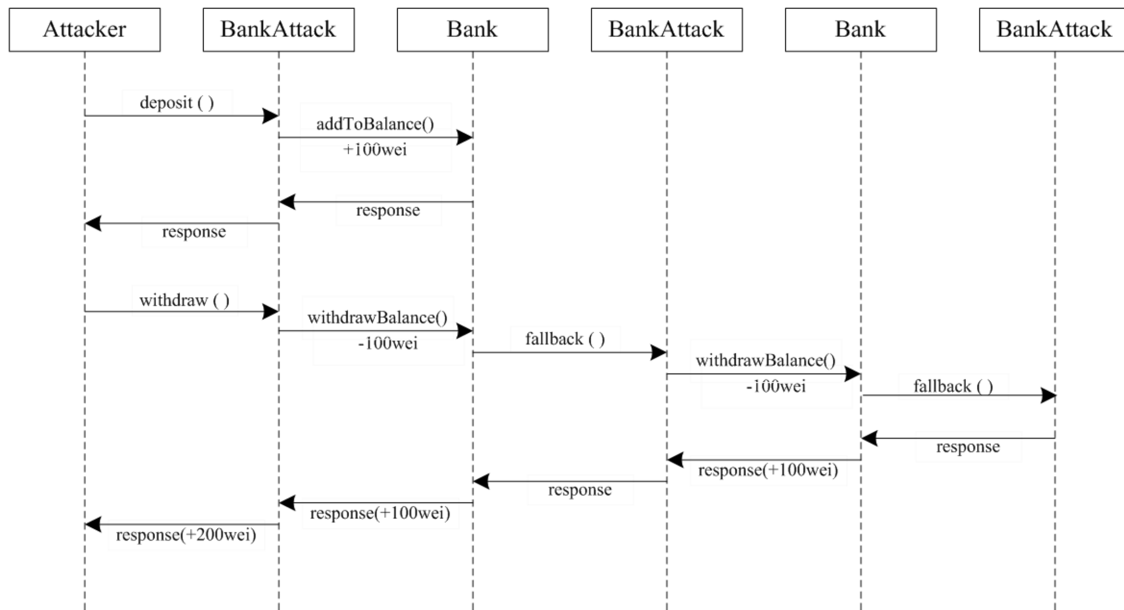



**FIGURE 7. Reentrancy attack.**

(3) The function *withdrawBalance*() in Bank sends 100 Wei to BankAttack, which triggers the fallback function of BankAttack.

(4) The second withdrawal: BankAttack's fallback function calls the function *withdrawBalance*() in the Bank again to withdraw money. This is equivalent to a recursive call. Since the first withdrawal has not yet completed, the value of variable *userBalances* in the Bank has not been updated. Therefore, when withdrawing money for the second time, Bank mistakenly thinks that BankAttack still had 100 Wei, which successfully performs two withdrawal operations.

Based on multiple vulnerability cases and related literature, three reentrancy modes are summed up [38].

(1) Same-function reentrancy

Same-function reentrancy means that the same function of a contract is reentered. This is the most common reentrancy attack method. The case given above pertains to this mode.

(2) Cross function reentrancy

Cross-function reentrancy means that the same contract is reentered in different functions. Smart contracts typically provide multiple interfaces that can read or write the same internal state variables, making cross-function reentrancy attacks as dangerous as same-function reentrancy attacks.

(3) Contract-creation reentrancy

In Solidity, a new contract can be created with the keyword ''new'', and implemented by the CREATE command at EVM level. Once a new contract is created, the contract constructor will execute immediately, during which it may make calls to other malicious contracts. A possible attack manner is: the victim contract intends to create a new contract and then updates its internal state. However, the constructor of the new contract issues an external call to an address controlled by the attacker, re-entering the victim contract and exploiting the inconsistent internal state.

Regardless of the reentrancy mode, the key point is that the contract is executed in an inconsistent internal state after malicious reentrancy. Thus, three conditions constitute a reentrancy vulnerability: (1) an external call to another contract; (2) the storage variable that causes an inconsistency is used to control the flow decision during an external call; (3) the variable is updated after the external call returns.

#### 2) DETECTION LOGIC

Among existing smart contract vulnerability mining tools, Oyente only supports same-function reentrancy detection [11], Securify and Mythril can detect partial cross-function reentrancy [4], [13], while these tools cannot detect the contract-creation reentrancy vulnerability.

According to the vulnerability description above, checking for state updates is the key to discovering reentrancy vulnerabilities. Mythril takes the following strategy: detect all message calls that are sent to the user-supplied address and deliver gas. Note that Solidity's send() and transfer() functions set gas to 2300 and this setting can prevent reentrancy attacks. If an external call to an untrusted address is detected, analyze the CFG to determine if a state change may occur after the call returns. A warning is issued if a state update is detected. However, this strategy is too conservative. It marks any state update after an external call as a vulnerability, regardless of whether the state update actually causes a hazard. Hence many false positives are generated.

To make an improvement, more reasonable detection logic is to check whether the updated state can influence control flow decisions, presented in Fig. 8. Since the shared variables between contracts are always stored in storage, and storage variable is the only internal state variable that can affect the control flow when reentering the contract, the detection only needs to pay attention to variables in the storage. Therefore,

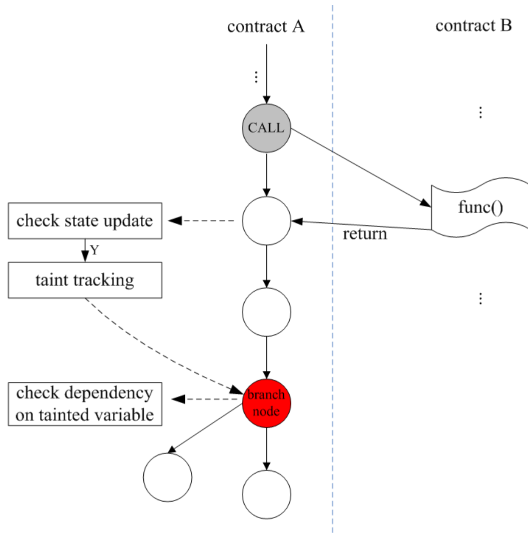



FIGURE 8. Detection logic for reentrancy.

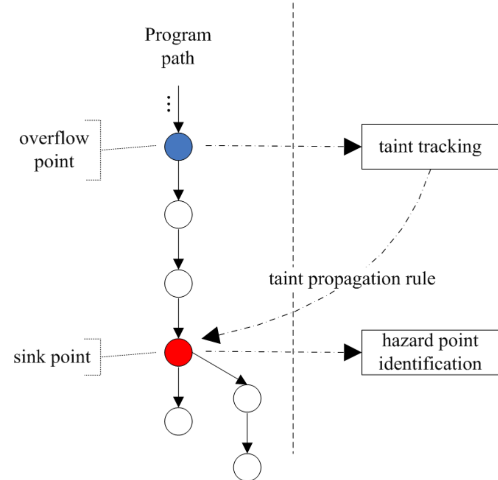



FIGURE 9. Detection logic for integer overflow.

the taint analysis technique is employed. The storage variables are marked as taint messages and are tracked in the program. If the control flow decision at a conditional jump instruction JMPI depends on some storage variables, then the attacker may manipulate the branch direction by reentering the contract, thereby manipulating the contract's behavior. The set of storage variables used to control the flow decision is then recorded and assigned the tag VAR_FLAG=1. If a previous call to the contract attempts to update a variable with a tag value of 1, a reentrancy vulnerability warning is issued.

### B. INTEGER OVERFLOW

#### 1) DESCRIPTION

Integer overflow includes overflow and underflow. The corresponding opcodes are ADD, MUL, and SUB. For SUB, if $op1>op0$, an underflow may occur. For ADD or MUL instructions, if $op1 + op0>2^{32} - 1$ or $op0 \times op1>2^{32} - 1$, an overflow may occur. Whether these operations can actually cause a vulnerability depends on two aspects. The first is whether an overflow check or input limit is conducted in the program context, that is, whether the overflow operation can be successfully executed and thereby generates an overflow point. The second is how this overflow value is used, that is, whether the value can cause harm. If the overflow value is applied at a critical location, it can lead to serious consequences and even trigger other vulnerabilities. such a critical location is defined as a hazard point. The hazard points of integer overflow are usually divided into the following three categories.

(1) Writing to memory: the tainted data is permanently stored by the contract as a global state variable, such as account balance.

(2) Branch statement: in the branch conditional statement, the overflow value controls the direction of the branch, causing the data to bypass the security check or cause the program to implement an illegal operation. Related cases include the famous SMT (SmartMesh Token) and BEC (Beauty Chain) vulnerability incidents.

(3) Data transfer: the tainted data is transmitted to an external function for unknown operations.

#### 2) DETECTION LOGIC

Fig. 9 shows the detection logic of integer overflow, which takes three steps.

(1) Overflow point detection

Overflow point detection is carried out on the basis of ADD, MUL, and SUB instructions. To determine whether their related operations will produce overflow, two operands, $op0$ and $op1$ are obtained from the top of the stack, and then the integer operands are converted into 256-bit bitvector. Afterwards, overflow judgment expression is constructed according to the operation instruction. Finally, the Z3 solver is used to calculate whether the expression has a solution. The following are overflow expressions conforming to Z3 syntax.

Overflow: $Or(And(ULT(expr, op0), op1! = 0), And(ULT(expr, op1), op0! = 0))$

Where, for ADD instruction, $expr = op0 + op1$; for MUL instruction, $expr = op1 \times op0$.

Underflow: $UGT(op1, op0)$

If the path constraints of the current state and the overflow expression can be satisfied at the same time, this is an overflow point.

(2) Taint tracking

The operation result of an overflow point does not necessarily cause harm. Therefore, upon an overflow point is found, it is marked as the source point for taint analysis and tracked according to the taint propagation rule. In this step, all the taint information in the program is obtained.

(3) Hazard point identification

Hazard point identification is the detection of critical operations at critical program points (sink points) that are affected by taint information. A hazard point is where the overflow value actually takes effect. According to the analysis

of hazard point above, program points operating SSTORE instruction and the JMPI instruction are regarded as the sink point. If an SSTORE instruction writes to the storage with tainted data, or the jump condition of JMPI is not independent of tainted data, it is considered to exist an integer overflow vulnerability.

### C. DELEGATECALL ABUSE

#### 1) DESCRIPTION

The prototype of function delegatecall can be presented as $\langle address\rangle .delegatecall(\ldots)\ returns(bool)$. It calls a function of *address* under the identity of the caller contract and returns false if the execution fails. By default, all available gas is transmitted and the amount of gas is adjustable. The value of the built-in variable *msg.sender* after the call is not modified to the caller, but the execution environment is the caller's runtime environment. Improper use of delegatecall will allow the contract to use the code of other contracts without transferring its own state (such as balance, storage), resulting in the execution of unexpected code. For example, if the target address to be called and the sequence of characters are both passed by the user, then the function of any address can be called. In addition, if the caller and the callee have the same variable, and the called function modifies this variable value, the variable of caller rather than callee is modified since the execution environment of delegatecall is the caller's environment. The second attack on the Parity Multisig wallet [39] is a representative case of the abuse of delegatecall.

```
1  Contract Wallet{
2    function () payable{    // fallback function
3      if(msg.value>0)
4        Deposit(msg.sender, msg.value);
5      else if(msg.data.length>0)
6        _walletLibrary.delegatecall (msg.data);
7    }
8  }
9  Contract WalletLibrary{
10   function initWallet(address[] _owners, uint _required,
        uint _daylimit){
11     initDaylimit(_daylimit);
12     initMultiowned(_owners, _required);
13   }
14 }
```

In the Wallet contract above, LINE6 executes a delegatecall and passes a parameter *msg.data*, so that any public function in the walletLibrary can be called. Therefore, the attacker can call function *initWallet*() at LINE11 and become the contract owner. After then he can send Ether from the wallet to his own address.

#### 2) DETECTION LOGIC

Algorithm 3 shows the detection logic of delegatecall abuse. For all external calls in the form of DELEGATECALL, if the called function is a fallback function, the third element $\mu_s[-3]$ can be obtained at the top of the stack. This element refers to the starting address of required parameters stored in memory, which can be presented as $\mu_m\,[\mu_s[-3]]$ according to the transaction model. If the address variable is a concrete value, check the string from memory location. If the string contains calldata, the contract passes the calldata through DELEGATECALL in the fallback function, which means that any function in the called contract can be executed and the called contract can modify the storage of the caller contract. If the callee contract is a symbolic value, this indicates that the target address of delegatecall is provided by the user. The next step is to determine whether the address is obtained from calldata or from the storage, in both cases the callee contract can access the state variable of caller contract without restriction.

**Algorithm 3** Detection Logic for Delegatecall Abuse

**Input:** *statespace*, *issues*[ ]
**Output:** *issues*[ ]: security defects
1: *issues*[ ] ← ∅
2: **for** all *delegatecall.func_name* == *fallback* **do**
3:   **if** is(_Concrete($\mu_m\,[\mu_s[-3]]$)==True) **and** (Search(*calldata*, $\mu_m\,[\mu_s[-3]]$)==True) **then**
4:     issues.Push(Dele_issue1)
5:   **else**
6:     **if** "calldata" in str(*call.to*) **then**
7:       issues.Push(Dele_issue2)
8:     **end if**
9:     **if** Storage_Write(str(*call.to*),storage_idx) **then**
10:       issues.Push(Dele_issue3)
11:     **end if**
12:   **end if**
13: **end for**
14: report(issues)

### D. TRANSACTION ORDER DEPENDENCY

#### 1) DESCRIPTION

Each block contains a collection of multiple transactions. For users, the order of transactions within the same block is invisible (only miners have access to it). Therefore, the state of the block is also uncertain. Suppose the block is currently in state $\sigma$ and contains two transactions, T1 and T2. T1 and T2 call the same contract at the same time. Under these circumstances, the user cannot know the state of the contract, as it depends on the execution order of T1 and T2.

TOD vulnerability is characterized by the fact that the user can change the state of the current contract (modify the variables in the storage) by issuing a transaction, and the variable can affect the destination address or the token value transferred of the external call.

#### 2) DETECTION LOGIC

Algorithm 4 shows the detection logic of TOD. For all external calls, first find all the storage variables that can affect the call value or call address, and then determine whether they are likely to change under the current node constraints. If possible, add the variable to the interesting storage list *Instor*. Then for each element in *Instor*, get the SSTORE

operation that can modify it, and record the corresponding (*state*, *node*) tuple. If such SSTORE operation exists, there is a TOD vulnerability.

**Algorithm 4** Detection Logic for TOD

**Input:** *statespace*, *issues*[ ]
**Output:** *issues*[ ]: security defects

```
1: issues[ ] ← ∅, Instor[ ] ← ∅
2: for all statespace.calls do
3:     Instor ← Relevant_storage(call.val, call.to)
4:     sstore_tupple ← Write_to(Instor)
5:     if sstore_tupple then
6:         issues.Push(TOD_issue)
7:     end if
8: end for
9: report(issues)
```

### E. SUICIDE CONTRACT VULNERABILITY

#### 1) DESCRIPTION

If a contract meets any of the following characteristics, it is considered as a suicide contract.

(1) SELFDESTRUCT instructions can be accessed by anyone.

(2) The storage index, where the parameter *address* of SELFDESTRUCT instruction is stored, is not restricted by *msg.sender*.

#### 2) DETECTION LOGIC

The detection logic of suicide vulnerability can be expressed as the formula below.

$$Instruction(\mu_{pc}) == SELFDESTRUCT \wedge$$
$$\mu_s[-1]! = I_C \wedge Solve(node(\mu_{pc}).constraints)! = \emptyset$$

For SELFDESTRUCT instruction, get the top element of the stack $\mu_s[-1]$, and determine where the balance is sent after executing the suicide command (including the caller's address, an address stored in storage, in function parameter passed through calldata, a specified address, etc.). Then get the constraint that the caller is not the contract creator $I_C$. If the caller is restricted by contract creator, no warnings will be issued. Conversely, if the caller is unrestricted, a suicide vulnerability is reported and the hazard level is determined based on the destination address $\mu_s[-1]$ of token transfer.

### F. PREDICTABLE VARIABLE DEPENDENCY

#### 1) DESCRIPTION

To write a random number generation function, contract developers often utilize parameters related to block headers such as *block.gaslimit*, *block.number*, *block.timestamp*, *block.difficulty* and *block.coinbase* (the miner address of current block), or other block information such as data stored in contracts, to generate random numbers. However, the properties of the block header parameters can be learned by the miners. Furthermore, the data on chain is often transparent. Thus, random numbers depending on those predictable variables are actually predictable, giving the attacker a chance to take advantage of it.

If the contract sends Ether with an amount greater than 0 based on the calculation of predictable variables, there is a predictable variable dependency vulnerability.

#### 2) DETECTION LOGIC

Algorithm 5 shows the detection logic of predictable variable dependency.

**Algorithm 5** Detection Logic for Predictable Variable Dependency

**Input:** *statespace*, *issues*[ ]
**Output:** *issues*[ ]: security defects

```
1: issues[ ] ← ∅, Predvar ←["coinbase", "gaslimit",
   "timestamp", "number", "difficulty"]
2: for all statespace.calls do
3:     if is_Concrete(call.value) or call.value == 0 then
4:         continue
5:     end if
6:     Var_List ←Search(Prevar, call.to+node.constraints)
7:     for Var in Var_List do
8:         Reso ← Solve(node.constraints)
9:         if Reso then
10:            issues.Push(PVD_issue)
11:        end if
12:    end for
13:    Bh ← Search("blockhash", call.to+node.constraints)
14:    for Var in Bh do
15:        Reso ← Solve(node.constraints)
16:        if Reso then
17:            Find_Blocknum("blockhash",str(node.
               constraints))
18:            issues.Push(PVD_issue)
19:        end if
20:    end for
21: end for
22: report(issues)
```

For all CALL series instructions in the state space, filter out the refund function as well as those with a *call.value* that is specified or equals to 0 (do not send Ether). For each of the remaining instructions, a two-step test is performed.

First, search for the predictable state variables mentioned above in the path constraint of the current instruction or constraint of the callee object. If there exists such a variable, add the corresponding node to a list. For each item in the list, invocate the constraint solver for a solution, and finally report the vulnerability and point out which environment variable it depends on.

Second, look for Ether recipient that depends on the block hash. In constraint string of current instruction or callee object, if the string "blockhash" can be matched, further judge the hash of which block is used and issue a warning.

Usually, the related block is several blocks ahead of the current block, which can be calculated by *block.number* minus a specified integer or a number stored at a storage index.

### G. MISHANDLED EXCEPTION

#### 1) DESCRIPTION

Solidity provides two functions, *assert* and *require*, to check for conditions and throw an exception if the condition is not satisfied. The function *assert* should be used to detect internal errors and to check invariants. The function *require* should be used to ensure conditions, such as user inputs and contract state variables are met, or to validate return values from calls to external contracts [40]. Solidity executes a revert operation (opcode 0xfd) for a require-style exception and performs an invalid operation (opcode 0xfe) to throw an assert-style exception. In both cases, the EVM will undo all changes made to the state in the current call and its sub-calls, and also issue an error to the caller. Assert-style exceptions are usually caused by type errors, division by zero, out-of-bounds array access, calling a zero-initialized variable of internal function type, etc. A properly functioning contract will never approach a failing assert statement. Otherwise, a failing assert can make it unable to achieve the original purpose.

#### 2) DETECTION LOGIC

To prevent such vulnerabilities, the security tools should verify that the contract will never execute the invalid opcodes. Therefore, the detection method is to solve the path constraints at each opcode 0xfe. If there is solution, the condition of failing assert can be met, and a warning should be reported. The detection logic can be expressed as the formula below.

$$Instruction(\mu_{pc}) == 0xfe$$
$$\wedge Solve(node(\mu_{pc}).constraints)! = \emptyset$$

### H. UNCHECKED RETURN VALUES

#### 1) DESCRIPTION

External function calls will fail if they exceed the maximum call stack of 1024 or run out of gas. In such cases, Solidity throws an exception, which usually "bubbles up" automatically when occurring in a sub-call. But this does not apply to function *send*() and the low-level functions, including *call*(), *delegatecall*(), *callcode*(), and *staticcall*(). These functions do not throw an exception, but rather return a Boolean "False", and the contract will continue execution as if the call succeeded. Therefore, whether the contract is successfully executed can not be judged by the existence of exceptions only.

#### 2) DETECTION LOGIC

As shown in Algorithm 6, the main idea of this detection module is to look for whether the return value of external function call is ignored or not. For all call-return nodes, check for instruction "ISZERO" in corresponding block. For each "ISZERO" instruction, obtain the element at the top of the stack $\mu_s[-1]$, and then determine whether the element contains a retval. If above condition is satisfied, it means *IZSERO*(*retval*) is performed. When retval is equal to 0, the state is reverted. Otherwise, check for return value is omitted and a warning should be issued.

**Algorithm 6** Detection Logic for Unchecked Return Values

```
Input: statespace, issues[ ]
Output: issues[ ]: security defects
1: issues[ ] ← Ø,
2: for node in CallRet_node do
3:    if "ISZERO" in node.opcode and Search("retval",
      μs[−1]) then
4:       retcheck←True
5:    end if
6:    retcheck←False
7:    issues.Push(RET_issue)
8: end for
9: report(issues)
```

## V. EVALUATION

### A. EXPERIMENTAL SETTINGS

#### 1) EXPERIMENTAL ENVIRONMENT

The experiment is conducted in a 64bit Ubuntu 16.10 system running on VMware Workstation virtual machine, which is configured with 2GB memory and four processor cores.

#### 2) EXPERIMENTAL SCHEME

##### a: COMPARISON WITH SIMILAR METHODS

Mythril and Oyente are the most representative and widely used symbolic execution tools for smart contract security. Considering the limitations of these similar tools, the experiment focuses on the following issues.

Q1: For large-size contracts with abundant instructions, detection is usually inefficient or even crash to failure. Whether the proposed method can make an improvement in large-size contract testing.

Q2: Whether the proposed method can ensure the accuracy of vulnerability detection under the premise of shortening the running time.

Q3: Whether the proposed method can exclude some false warnings by only focusing on code area involved in Ether transfer.

In order to answer these questions, three test sets are constructed. The first consists of 13 Solidity contracts, covering all vulnerability categories mentioned above, so that the test results can be analyzed manually according to the source codes. The second test set contains 4 large-size contracts with over 500 LOC (line of code). To verify whether the method is suitable for vulnerability detection of mass contracts, the third test set contains 1000 real-world on-chain contracts by crawling EVM bytecode from Etherscan.

##### b: COMPARISON WITH OTHER METHODS

As for other methods, this paper selects the formal verification tool Securify and the fuzzy test tool Contractfuzzer

**TABLE 2. Detection capability.**

| Tool | Vulnerability categories |
|---|---|
| This work | reentrancy, integer overflow, delegatecall abuse, TOD, suicide, predictable variable dependency, mishandled exception, unchecked return values |
| Securify | reentrancy, locked ether, missing input validation, TOD, mishandled exception, unrestricted ether flow |
| Oyente | reentrancy, TOD, mishandled exception, timestamp dependency, integer overflow(version 2018.3.27) |
| Mythril | reentrancy, multiple sends, integer overflow, delegatecall to untrusted contract, TOD, suicide, predictable variable dependency, mishandled exception, use of tx.origin, unchecked retval |
| Contractfuzzer | gasless send, mishandled exception, reentrancy, timestamp dependency, block number dependency, delegatecall abuse, locked ether |

to make a comparison. Since these tools support detecting different types of vulnerabilities, the experiment only considers types in common. The 1000 contracts from Etherscan are used for detection. Number of false negatives and false positives are the performance metrics to be compared.

## B. EXPERIMENT AND COMPARISON

### 1) DETECTION CAPABILITY

Users require security tools to detect as many types of vulnerabilities as possible. Table 2 lists the detection capability of these typical tools.

This work supports eight types of vulnerabilities described in section IV, covering most of the prevalent security issues. In the following text, experiments are carried out to evaluate its detection efficiency. Since the tools to be compared support different vulnerability types, only types in common are focused on to analyze detection accuracy.

### 2) COMPARISON WITH SIMILAR METHODS

In order to verify whether the proposed method can ease the problems of similar methods and answer the three questions above, the experiments take the following steps.

#### a: SOLIDITY CONTRACT TESTING

Firstly, 13 typical source code contracts containing different vulnerability categories are selected. The proposed method sets the maximum recursive depth of call stack to avoid getting stuck with a single complex program. Since this parameter has an impact on detection results, it is necessary to determine a proper value for it. Fig. 10 shows the number of warnings reported by this work when this value is set from 12 to 30.

As shown in Fig. 10, since the number of warnings flattens out after maximum recursive depth exceeds 22, this parameter is set to 22 for default.

Afterwards, Oyente and Mythril are also employed to analyze these solidity contracts. The experimental results are shown in Table 3 below. Indicators TP, FN, and FP represent true positives, false negatives, and false positives respectively.

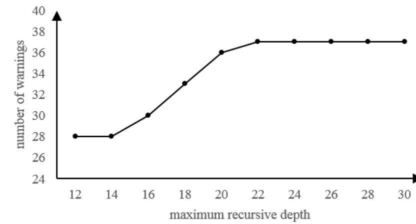



**FIGURE 10. Number of warnings under different settings.**

**TABLE 3. Experimental results.**

| | Execution time | Report | TP | FN | FP |
|---|---|---|---|---|---|
| Our Method | 261s | 37 | 32 | 1 | 5 |
| Mythril | 427s | 58 | 31 | 2 | 26 |
| Oyente | 294s | 13 | 6 | 27 | 7 |

This work costs less time than Oyente and Mythril. Oyente can only detect 5 types of vulnerabilities, so it issues the least security problems with many false negatives. Comparing the reports of Mythril and this work, it is found that 10 of the 13 contracts were identical. For the contracts with inconsistent results, Mythril reports 22 more security warnings than the proposed methods on contract BECToken.sol and Walletlibrary.sol. Among them, there are 16 for integer overflow and 6 for unexpected state. Referring to the corresponding source code, we find that these warnings are mostly due to nonstandard coding and excessive detection logic.

Take the code fragment of BECToken.sol shown below as an example. The function *add*() is concluded in the SafeMath library and is invocated multiple times by several functions in the contract. Mythril reports an overflow vulnerability on statement "uint256 c=a+b". In fact, SafeMath's built-in assertion is used here to check if the operation actually overflows. If an overflow occurs, the code contained in the assertion will cause the transaction to roll back. Many warnings originate from similar code and are proved to be false positive.

```
1 function add (uint256 a, uint256 b) internal constant
      returns (uint256) {
2   uint256 c = a + b;
3   assert (c >= a);
4   return c;
5 }
```

According to the indicator FP in Table 3, both methods have some false alarms. But the proposed method avoids analysis of unimportant paths and reduces many unnecessary false warnings. Appendix 2 shows a ticketing contract. When the ticket is sold out, a competitor will be randomly selected to win a reward of 2.5 ETH. The winning address is calculated according to *block.coinbase*, *block.difficulty*, an available constant *totalTickets* and *msg.sender*. In other word, it depends on predictable environment variables. Therefore, an attacker can take advantage of this weak randomness and speculate the *winningNumber*, call the fallback function and

**TABLE 4. Experimental results.**

| Contract | KiddyToys | EtherConsole | CBToken | StandardBounty |
|---|---|---|---|---|
| This work | 95s | 70s | 183s | 277s |
| Oyente | 136s | 101s | 245s | 249s |
| Mythril | 184s | 158s | 296s | - |

**TABLE 5. Experimental results.**

| Vulnerability Type | TP | FP | Precision |
|---|---|---|---|
| reentrancy | 20 | 2 | 90.9% |
| integer overflow | 22 | 7 | 75.9% |
| delegatecall abuse | 16 | 9 | 64% |
| TOD | 8 | 0 | 100% |
| suicide | 7 | 0 | 100% |
| predictable variable dependency | 24 | 3 | 88.9% |
| mishandled exception | 30 | 13 | 69.8% |
| unchecked return values | 5 | 0 | 100% |

get the reward. However, Mythril did not find this problem, and the reason may be that when symbol execution comes into the loop "while" (up to 50 cycles), it fell into path explosion and failed to reach the dangerous state during the timeout setting. The method proposed in this paper firstly identified the CALL instruction, which corresponding to the function *winningaddress.transfer*(*prize*) in the source code, and then adopted the path search strategy to guide the execution to quickly approach the critical code, and found this security defect at last.

#### b: LARGE-SIZE CONTRACT TESTING

Most of the contracts tested above are small and medium-size contracts with a scale of no more than 100 LOC. The average detection time per contract is less than one minute, which is acceptable. In order to verify the performance on large-size contracts, four contracts with codes greater than 500 LOC are selected for further tests, and timeout for symbolic execution is set to 10000s (equivalent to no time limit). The results are shown in Table 4.

Compared with Mythril, this work demonstrates an obvious advantage, reducing detection time by nearly half when dealing with large-size contracts. This work also costs less time than Oyente for 3 of the 4 contracts. Particularly, the detection result of the StandardBounty contract was still not obtained by Mythril after ten minutes' analysis, while the proposed method discovered a TOD vulnerability with 277 seconds.

#### c: MASS TESTING

To verify the performance of the proposed method in mass testing, 1000 real-world contracts in the form of bytecode are obtained from Etherscan. Timeout for symbolic execution is set to 600s per contract. Table 5 shows the detection results of this work.

In total, 166 security defects are detected within 37538s, among which 132 are true positives. The most vulnerabilities found are mishandled exception, predictable variable dependency, integer overflow and reentrancy. And the detection of delegatecall abuse, integer overflow, and mishandled exception has lower precision relatively.

**TABLE 6. Statistics of paths.**

| Total Path | Critical Path | Uncritical Path | Vulnerable Path |
|---|---|---|---|
| 11272 | 3720 (33%) | 7552 (67%) | 127 |

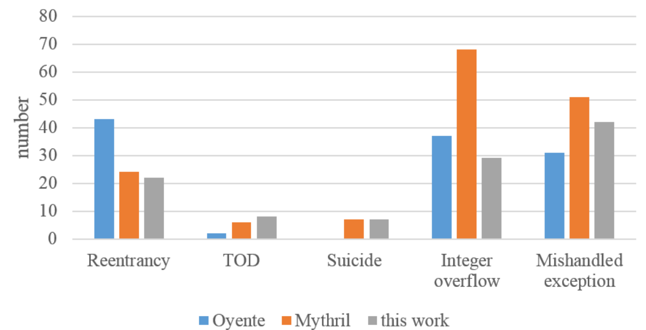



**FIGURE 11. Amount of warnings reported.**

The static analysis tool SASC points out that the execution time is almost linearly depends on the number of executed paths, blocks or opcodes, i.e., the complexity of program [41], [42]. So we look into the 1000 contracts for the detailed statistics of paths. The number of paths for these contracts ranges from 1 to 1694, with an average of 11 and a median of 5. Among them, 113 contracts do not expose a critical path, although our definition of critical paths is broad. Table 6 presents some statistics of paths in this test.

Uncritical paths take up about 67% of all paths. The number of paths explored per contract by this work is around 4. So much time is saved by reducing the amount of paths to be symbolic executed. Among the critical paths, 127 are found vulnerable. Among these vulnerable paths, 87.9% of them contain CALL, SSTORE and DELEGATECALL, while only a small number of SELFDESTRUCT, CALLCODE and STATICCALL instructions are found.

To further analyze the performance of this work, the experiment compares its detection efficiency with similar methods for several common vulnerability types. Fig. 11 shows the amount of warning reported by these symbolic execution tools.

Afterwards, the reported warnings are analyzed manually and classified as true positive or false positive. The results are summarized in Fig. 12. Bars A, B, and C are results for Oyente, Mythril, and this work respectively.

Mythril spends about 70 seconds per contract on average, and Oyente spends 40 seconds. This work consumes less time than both tools while performs better according to Fig. 12. Through inspecting the detection report, it can be observed that although Mythril can discover most of the vulnerabilities, it suffers from a high false positive rate, especially on reentrancy, integer overflow and mishandled exception. Oyente only supports a few vulnerability categories, and also has

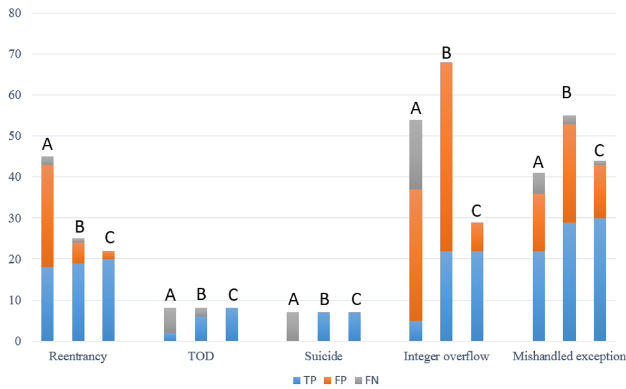



**FIGURE 12. Results comparison.**

**TABLE 7. Experimental results.**

| Vulnerability | Reentrancy | | Mishandled Exception | |
|---|---|---|---|---|
| Indicator | FP | FN | FP | FN |
| Securify | 1 | 3 | 1 | 5 |
| Contractfuzzer | 0 | 6 | 0 | 10 |
| This work | 2 | 0 | 13 | 1 |

many false warnings on reentrancy and missing reports on TOD. This work takes about 37 seconds for each contract, and make considerable progress in false positive and false negative.

#### 3) COMPARISON WITH OTHER METHODS

Above contents discuss the performance of this work and similar works based on symbolic execution. This section compares this work with tools based on other methods. They are the state-of-the-art formal verification tool Securify and fuzzy test tool Contractfuzzer.

As shown in Table 2, all of them can detect reentrancy and mishandled exception, so the 1000 contracts above are tested for these two types of vulnerabilities to make a comparison. Table 7 shows the performance of these tools.

Results show that Contractfuzzer does not report false positives. It generates fuzzing inputs and triggers the vulnerabilities directly, so its results have very high precision. However, it is hard for ContractFuzzer to pass some complex condition checks before transferring ether, so it have the most false negatives. On the contrary, this work issues the most false positives but has the least missing reports.

```
1  contract Bank{
2    mapping(address=>uint) userBalances;
3    function getUserBalance(address user) constant returns
         (uint) {
4      return userBalances[user];
5    }
6
7    function addToBalance() {
8      userBalances[msg.sender] = userBalances[msg.sender]
           + msg.value;
9    }
10
11   function withdrawBalance() {
12     uint amountToWithdraw = userBalances[msg.sender];
13     if (msg.sender.call.value(amountToWithdraw)() ==
           false) {
14       throw;
15     }
16     userBalances[msg.sender] = 0;
17   }
18 }
19
20
21 contract BankAttacker{
22
23   bool is_attack;
24   address bankAddress;
25
26   function BankAttacker(address _bankAddress, bool
         _is_attack){
27     bankAddress=_bankAddress;
28     is_attack=_is_attack;
29   }
30
31   function() {
32     if(is_attack==true)
33     {
34       is_attack=false;
35       if(bankAddress.call(bytes4(sha3("withdrawBalance()
             ")))) {
36         throw;
37       }
38     }
39 }
40
41   function deposit(){
42     if(bankAddress.call.value(2).gas(20764)(bytes4(sha3(
           "addToBalance()")))==false) {
43       throw;
44     }
45   }
46
47   function withdraw(){
48     if(bankAddress.call(bytes4(sha3("withdrawBalance()")
           ))==false ) {
49       throw;
50     }
51   }
52 }
```

**Listing 1. Reentrancy attack.**

## VI. CONCLUSIONS

### A. MERITS

This paper proposes a critical-path-coverage-based vulnerability detection method for smart contracts, and implements it based on Mythril's symbolic execution framework. Overall, it enjoys the following merits.

1. It realizes the fast approximation to the code region of interest, reducing much time consumption.

2. It excludes some false alarms that do not actually cause financial losses by only focusing on code regions involved in Ether transfer.

3. It releases the difficulty of verifying large-size contracts.

4. It supports a comprehensive test of common vulnerability categories based on targeted detection logic.

```
1  contract WeakRandom {
2    struct Contestant {
3      address addr;
4      uint gameId;
5    }
6
7    uint public constant prize = 2.5 ether;
8    uint public constant totalTickets = 50;
9    uint public constant pricePerTicket = prize /
       totalTickets;
10
11   uint public gameId = 1;
12   uint public nextTicket = 0;
13   mapping (uint => Contestant) public contestants;
14
15   function () payable public {
16     uint moneySent = msg.value;
17
18     while (moneySent >= pricePerTicket && nextTicket <
         totalTickets) {
19       uint currTicket = nextTicket++;
20       contestants[currTicket] = Contestant(msg.sender,
           gameId);
21       moneySent -= pricePerTicket;
22     }
23
24     if (nextTicket == totalTickets) {
25       chooseWinner();
26     }
27
28     // Send back leftover money
29     if (moneySent > 0) {
30       msg.sender.transfer(moneySent);
31     }
32   }
33
34   function chooseWinner() private {
35     address seed1 = contestants[uint(block.coinbase) %
         totalTickets].addr;
36     address seed2 = contestants[uint(msg.sender) %
         totalTickets].addr;
37     uint seed3 = block.difficulty;
38     bytes32 randHash = keccak256(seed1, seed2, seed3);
39
40     uint winningNumber = uint(randHash) % totalTickets;
41     address winningAddress = contestants[winningNumber
         ].addr;
42
43     gameId++;
44     nextTicket = 0;
45     winningAddress.transfer(prize);
46   }
47 }
```

**Listing 2.** A ticketing contract.

### B. LIMITATIONS AND FUTURE WORK

#### 1) USER-DEFINED REGULATIONS MODULE FOR LOGICAL FLAWS

This work and existing symbolic execution tools concentrate on common functional and security properties of smart contracts, while the detection of complex business logic such as fairness is a weak spot. To deal with logical flaws, future study should introduce a module that enables user-defined rules in vulnerability mining tools. By verifying whether the customized rules can be violated, logical flaws can be detected.

#### 2) RISK GRADING

This work does not distinguish the hazard of security defects. For the sake of safety assessment and bug fixing, it is necessary to provide risk grade for the reported warnings in line with their potential hazard. For example, the risk of state change after calling a user-supplied address is more dangerous than that of a fixed address.

#### 3) THE SETTING OF EXECUTION PARAMETERS

This work limits execution duration by setting parameters like timeout and recursive depth. However, contracts often cannot reach that state due to running out of gas in reality. To avoid unnecessary losses, future work can take the gas provided into consideration.

## APPENDIX. 1

See Listing1.

## APPENDIX. 2

See Listing2.

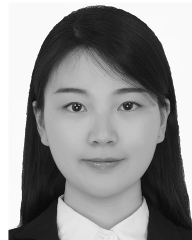
## ACKNOWLEDGMENT

The authors would like to thank all the reviewers for your time and expertise on this paper.

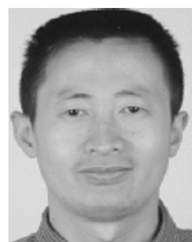
## REFERENCES

[1] T. Aste, P. Tasca, and T. D. Matteo, "Blockchain technologies: The foreseeable impact on society and industry," *Computer*, vol. 50, no. 9, pp. 18–28, Jan. 2017.
[2] H. Watanabe, S. Fujimura, A. Nakadaira, Y. Miyazaki, A. Akutsu, and J. Kishigami, "Blockchain contract: Securing a blockchain applied to smart contracts," in *Proc. Int. Conf. ICCE*, Las Vegas, NV, USA, Jan. 2016, pp. 467–468.
[3] Quantstamp Announcements. (Jun. 20, 2019). *Fundamentals of Smart Contract Security Released*. [Online]. Available: https://quantstamp.com/blog/fundamentals-of-smart-contract-security-released
[4] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," presented at the 25th ACM Conf. CCS, Toronto, ON, Canada. Oct. 2018. [Online]. Available: https://arxiv.org/pdf/1806.01143.pdf
[5] R. Gu, Z. Shao, and V. Söberg. (Jun. 21, 2018). *CertiK: Building Fully Trustworthy Smart Contracts and Blockchain Ecosystems*. [Online]. Available: https://certik.org/docs/white_paper.pdf
[6] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, "Formal verification of smart contracts: Short paper," in *Proc. ACM Workshop Program. Lang. Anal. Secur.*, New York, NY, USA, 2016, pp. 91–96.
[7] E. Hildenbrandt. (Aug. 17, 2018). *KEVM: A Complete Semantics of the Ethereum Virtual Machine*. [Online]. Available: http://hdl.handle.net/2142/97207
[8] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *Proc. Int. Conf. Princ. Secur. Trust*, Thessaloniki, Greece, 2018, pp. 243–269.
[9] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards verifying ethereum smart contract bytecode in Isabelle/HOL," in *Proc. 7th Int. Conf. ACM SIGPLAN*, Los Angeles, CA, USA, 2018, pp. 66–77.
[10] Ethereum Community Forum. (Oct. 2015). *Formal Verification for Solidity Contracts*. [Online]. Available: https://forum.ethereum.org/discussion/3779
[11] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM CCS*, Vienna, Austria, 2016, pp. 254–269.
[12] M. Mossberg, Y. Ivnitskiy, and J. P. Smith. (Feb. 12, 2017). *TrailofBits/Manticore: Symbolic Execution Tool*. [Online]. Available: https://github.com/trailofbits/manticore

[13] B. Mueller. (Sep. 17, 2017). *ConsenSys/Mythril*. [Online]. Available: https://github.com/ConsenSys/mythril
[14] J. Krupp and C. Rossow, "teEther: Gnawing at Ethereum to automatically exploit smart contracts," in *Proc. 27th USENIX Secur. Symp.*, Baltimore, MD, USA, 2018, pp. 1317–1333.
[15] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," Mar. 2018, *arXiv:1802.06038*. [Online]. Available: https://arxiv.org/pdf/1802.06038v2.pdf
[16] J. P. Smith, P. Ben, and C. Connor. (Jun. 12, 2018). *TrailofBits/Echidna*. [Online]. Available: https://github.com/trailofbits/echidna
[17] B. Jiang, Y. Liu, and W. K. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in *Proc. 33rd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Montpellier, France, Sep. 2018, pp. 259–269.
[18] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.
[19] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proc. ACM SIGPLAN Conf. PLDI*, Berlin, Germany, 2005, pp. 213–223.
[20] K. Sen and G. Agha, "CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools," in *Proc. 18th Int. Conf. Comput. Aided Verification*, Seattle, WA, USA, 2006, pp. 419–423.
[21] C. Cristian, D. Daniel, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Symp. OSDI*, San Diego, CA, USA, 2008, pp. 209–224.
[22] C. Vitaly, K. Volodymyr, and C. George, "S2E: A platform for *in-vivo* multi-path analysis of software systems," *ACM SIGPLAN Notices*, vol. 39, no. 4, pp. 265–278, Mar. 2011.
[23] S. K. Cha, T. Avgerinos, A. Rebert, and D6Brumley, "Unleashing mayhem on binary code," in *Proc. IEEE Symp. S&P*, San Francisco, CA, USA, May 2012, pp. 380–394.
[24] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. S&P*, San Jose, CA, USA, May 2016, pp. 138–157.
[25] X. Xiao, X.-S. Zhang, and X. D. Li, "New approach to path explosion problem of symbolic execution," in *Proc. 1st Int. Conf. IEEE PCSPA*, Harbin, China, Sep. 2010, pp. 301–304.
[26] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proc. IEEE Symp. S&P*, Oakland, CA, USA, May 2010, pp. 317–331.
[27] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. 12nd NDSS*, San Diego, CA, USA, 2005.
[28] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," in *Proc. 4th Int. Conf. ICISS*, Hyderabad, India, Dec. 2008, pp. 1–25.
[29] R. Zhang, S. Huang, Z. Qi, and H. Guan, "Static program analysis assisted dynamic taint tracking for software vulnerability discovery," *Comput. Math. Appl.*, vol. 63, no. 2, pp. 469–480, Jan. 2012.
[30] T. Wang, T. Wei, G. Gu, and W. Zou, "Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution," *ACM Trans. Inf. Syst. Secur.*, vol. 14, no. 2, pp. 1–28, Sep. 2011.
[31] R. Corin and F. A. Manzano, "Taint analysis of security code in the KLEE symbolic execution engine," in *Proc. 14th Int. Conf. ICICS*, Hong Kong, 2012, pp. 264–275.
[32] G. Wood. (Jun. 13, 2019). *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf
[33] N. Bjorner. (Sep. 30, 2012). *Z3Prover/Z3*. [Online]. Available: https://github.com/Z3Prover/z3
[34] Y. Zhou, "Erays: Reverse engineering Ethereum's opaque smart contracts," in *Proc. 27th USENIX Secur. Symp.*, Baltimore, MD, USA, 2018, pp. 1371–1385.
[35] L. Jay, B. H. Omar, and G. Dan. (Feb. 11, 2018). *Crytic/EVM-Opcodes*. [Online]. Available: https://github.com/crytic/evm-opcodes
[36] J. L. Bander and C. C. White, "A heuristic search algorithm for path determination with learning," *IEEE Trans. Syst., Man, Cybern. A, Syst. Humans*, vol. 28, no. 1, pp. 131–134, Jan. 1998.
[37] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen, "A survey on the security of blockchain systems," *Future Gener. Comput. Syst.*, to be published. doi: 10.1016/j.future.2017.08.020.
[38] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," presented at the 26th NDSS, San Diego, CA, USA, Feb. 2019. [Online]. Available: https://arxiv.org/pdf/1812.05934.pdf
[39] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts (SoK)," in *Principles of Security and Trust* (Lecture Notes in Computer Science), vol. 10204. Berlin, Germany: Springer, Mar. 2017, pp. 164–186.
[40] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab," in *Proc. Int. Conf. Financial Cryptogr. Data Secur.*, Christ Church, Barbados, 2016, pp. 79–94.
[41] I. Grishchenko, M. Maffei, and C. Schneidewind, "Foundations and tools for the static analysis of ethereum smart contracts," in *Proc. Int. Conf. CAV*, Oxford, U.K., 2018, pp. 51–78.
[42] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara, "Security assurance for smart contract," in *Proc. 9th Int. Conf. IFIP-NTMS*, Paris, France, Feb. 2018, pp. 1–5.



**MENGLIN FU** received the B.E. degree from the PLA University of Science and Technology, in 2017. She is currently pursuing the M.S. degree with the Army Engineering University of PLA. Her research interests include concern cyber security and vulnerability mining.

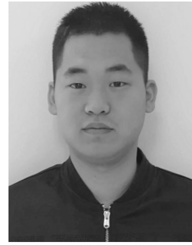


**LIFA WU** received the Ph.D. degree from Nanjing University, in 1998. He is currently a Professor with the Nanjing University of Posts and Telecommunications. His research interests include concern network security and software security.

**ZHENG HONG** received the Ph.D. degree from the PLA University of Science of Technology, in 2007. He is currently an Associate Professor with the Army Engineering University of PLA. His research interests include concern cyber security and protocol reverse engineering.

**FENG ZHU** received the Ph.D. degree from Florida International University, in 2014. He is currently an Assistant Professor with the Nanjing University of Posts and Telecommunications. His research interests include concern system security and cyber security.

**HE SUN** received the B.E. and M.S. degrees from the PLA University of Science and Technology, in 2013 and 2016, respectively. He is currently pursuing the Ph.D. degree with the Army Engineering University of PLA. His research interests include concern information security, malware analysis, and network security.



**WENBO FENG** received the B.E. degree from Shandong University, in 2017. He is currently pursuing the M.S. degree with the Army Engineering University of PLA. His research interests include concern protocol recognition and data analysis.