IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 30, 2020, accepted November 9, 2020, date of publication November 24, 2020,
date of current version December 3, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3039990

Automated Generation of Test Cases for Smart Contract Security Analyzers

KI BYUNG KIM AND JONGHYUP LEE
Department of Mathematical Finance, Gachon University, Seongnam 13120, South Korea

Corresponding author: Jonghyup Lee (jonghyup@gachon.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government [Ministry of Science and ICT (MSIT)] under Grant 2020R1F1A1071943.

ABSTRACT We address the absence of reliable tests on contract analyzers of smart contracts and present a systematic method to diversify test cases by combining smart-contract-specific bugs and static analysis barriers in this paper. Using contract analyzers is the most practical solution for building a secure blockchain service, but they are relatively immature and lacking stable performance metrics. Traditionally, performance reports only compare static contract analyzers with pre-defined test cases, such as the Juliet test suite. However, building such test suites is burdensome for smart contracts, which are frequently change. In this paper, we propose an automated method to assess contract analyzers of smart contracts by diversifying test cases. In the experimental results, we identified nine erroneous alarms in the state-of-the-art contract analyzers with automatically generated test cases on five vulnerabilities.

INDEX TERMS Smart contracts, static analysis, security audits, honey pots.

I. INTRODUCTION

Smart contracts are at the core of blockchain services. A smart contract is a program code that runs on top of the blockchain. Currently, the Ethereum model [1] is the most dominant for smart contracts. The execution model of smart contracts is different from that for the general computing.

A smart contract is deployed to a blockchain system in the compiled bytecode form. The deployed bytecode is immutable; thus, it cannot change even if an error is found in the bytecode. A dedicated virtual machine runs the bytecode smart contracts. Smart contracts operate the functional part of blockchain services and systems, storing crypto-currencies and sensitive data. The security of smart contracts is being tested. However, the well-known incidents with smart contracts (TheDAO [2] and Parity wallet [3] attacks) have demonstrated that just relying on the manual inspection is insufficient to protect cryptoassets from attacks. Immutability makes securing smart contracts more challenging because the second chance to amend errors is not allowed. This situation leads to adopting proactive protection using systemized and automated processes, which resemble the design process for the digital integrated circuits. Thus, the automated security analysis tools, referred to as ''contract analyzers,'' are widely used.

The Ethereum security survey [4] noted that using contract analyzers is the most effective protection method, except for secure coding by developers. Although applying the contract analyzers becomes an essential step to develop a blockchain service, contract analyzers are in the early stage to produce accurate output steadily. Recent literature assessing the contract analyzers [5], [6] found notable false-positive and false-negative errors in most contract analyzers. In the blockchain, causing false-positive and false-negative errors is economically beneficial for attackers. Attackers who write Trojan smart contracts that hide vulnerabilities make an effort to cause false-negative errors.

In contrast, malicious developers intentionally abuse complex bugs to cause false-positive errors. They reveal a well-known vulnerability to lure other attackers, but hide another bug that will stealthily disable the exploitation. If other fooled attackers invest money to obtain profits by exploiting the vulnerability, it will fail, and the malicious developers can intercept the invested money. These attacks are specific to the blockchain and are referred to as ''honey pots'' [7].

Figure 1 is an example in which the contract analyzers produce inconsistent results on complex bugs. The code has a

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

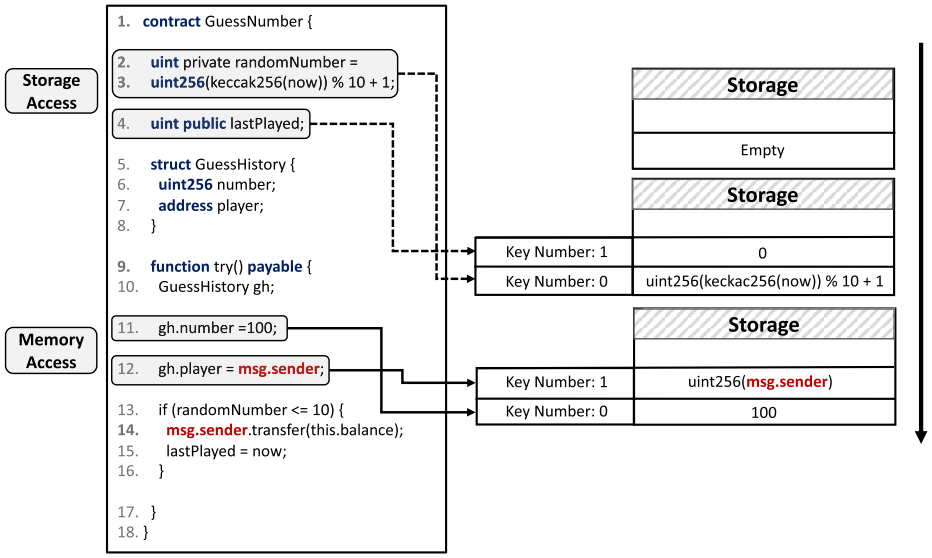

FIGURE 1. Incorrect use of public information as a random source with the UninitializedStoragePointer bug.

notable vulnerability of improperly using a random source in the blockchain, referred to as "bad randomness" On Line 3, it generates a random number from the timestamp of the latest block, based on the global variable `now`, which is publicly predictable. The contract analyzers alert developers about this vulnerability when `now` is used in control conditions, as on Line 13. However, this alert is not true. The vulnerability is canceled by the second bug from the Solidity compiler. As illustrated in Figure 1, when the struct pointer `gh` is uninitialized, `gh` points to the two bottom slots assigned to the other storage variables, `lastPlayed` and `randomNumber`. Hence, the variable `randomNumber` is implicitly overwritten during the value assignment to `gh.number` on Line 11. If a contract analyzer is aware of this compiler bug, it should not report the bad randomness vulnerability, or it may result in the false-positive errors.

However, our experimental results in §VI demonstrate that all tested contract analyzers produce incorrect results in this case. In the general computing environment, the security static analyzers were assessed and enhanced using well-tailored test suites, such as the Juliet test suite [8] for Java and C/C++. Based on the benchmarks, we can identify the weak points and limitations of static analyzers as listed in [9].

However, building a proper test suite is challenging for smart contracts. The test set from the reference repository (*e.g*, the Smart Contract Weakness Classification Registry [10]) and the bundles of the tools still have naive sample codes. Thus, the collection of the test sets cannot represent the complex structures and bugs in real-world code, because blockchain services are already diversified from the simple ERC-20 variants to the complicated finance services. Moreover, new types of bugs or vulnerabilities in smart contracts are still being identified [11]. Whenever a new bug is introduced, contract analyzers must quickly check the detection performance on it with various code structures.

In this paper, we propose a new approach to automatically build a test suite for contract analyzers. The proposed system, TestBreeder, generates test cases by combining various code elements of vulnerabilities, structures, and bugs. Therefore, the generated test suite of TestBreeder can test the vulnerability detection capacity against varying code complexity. We consider the common weak points of static analysis and the quirks of smart contracts for the code elements. We can summarize our contributions as follows:

1) We address the challenges of the current contract analyzers in dealing with the compound vulnerabilities of Solidity, which should be carefully considered in the current trends of smart contracts.
2) We propose a new approach to generate test suites for contract analyzers. While the previous studies have compared contract analyzers with fixed vulnerability code samples, we focus on generating varying complexity test cases with the real-world exploit techniques. Moreover, whenever a new type of vulnerability is found, the proposed approach can extend the test suite for different complexity.
3) To evaluate the usefulness of the automatically generated test cases, we run the test cases with widely used contract analyzers. Our test suite identifies the weak points in all contract analyzers and discovers that false-positive and false-negative errors increase in more complicated test cases.

II. BACKGROUND

A. BLOCKCHAIN AND SMART CONTRACTS

All blockchain systems are not the same, but have two key components in common for consensus and smart contracts (or a transaction model). A consensus subsystem enables distributed participants to agree on a shared state against Byzantine faults (*i.e.,* malicious participants). Smart contracts update the shared state by executing the deployed code. The design and implementation of smart contracts differ by system.

We use the Ethereum model in this paper. Ethereum uses a virtual machine, called the ''EVM,'' as a runtime environment for smart contracts. Like a general computer program, developers write down the smart contract code in human-readable programming languages (*e.g*, Solidity and Vyper) and compile the code to the EVM-executable bytecode. The compiled EVM bytecode can be deployed to any EVM blockchain systems. The deployed smart contracts are normally public to anyone in the blockchain. A smart contract is executed upon receiving a message called a transaction. A transaction is a cryptographically signed message containing a destination address, `callvalue` (the amount of the transferred digital asset), `calldata` (the general input data), execution fee (*i.e.,* gas) related information, and so on.

If the recipient of a transaction is a smart contract, it acts like a function call, encoding the function signature and passing arguments in the `calldata`. The target smart contract executes the function matched to the signature. If no match is found, it calls a special function, referred to as the ''fallback.'' The default behavior of the fallback case is to add the received `callvalue` to its balance.

The EVM provides three data locations: stack, memory, and storage. Every computation is performed on the stack. A slot of the stack is 256-bit wide, and the stack can have a maximum of 1024 slots. The memory is a temporary area that holds data only within a transaction execution. It operates as a byte-addressable array. The permanent data are kept in the storage location, which is shared by the blockchain nodes. The storage is a key-value store in the EVM, but Solidity maps a global state variable to a slot in the storage.

B. SOLIDITY AND BUGS

1) SOLIDITY

Solidity is the most popular programming language for smart contracts and adds new abstractions for developers to support high-level features, such as control structures and complicated data structures on top of the EVM. The syntax of Solidity is similar to that of JavaScript, but it has its own features for smart contracts.

a: VARIABLES AND FUNCTIONS

The global and local variables are mapped to the storage and memory of the EVM data location by default. The global storage variables are referred to as state variables. Except for the functions attributed as `view` or `pure`, Solidity functions are related to the operations on the state variables. Solidity also has predefined special variables to access the blockchain-specific properties or actions. For example, `block.timestamp` (or `now`) stands for the current block time, and `msg.sender` indicates the sender address of the function call. The address type of Solidity has predefined member functions. The `transfer` and `send` functions can send Ether to a recipient address. The `call` function can be used for the low-level function call. The `call` may have attributes, such as `value` and `gas`. For instance, `msg.sender.call.value(100)` sends 100 Wei (a unit of Ether) to the sender address.

b: DATA STRUCTURES

On top of the EVM memory and key-store storage, Solidity presents complex data types, such as the strcut and dynamic array. For this purpose, Solidity has reference types besides elementary types (*e.g*, `uint256`). The reference types support structs, arrays, and mappings and can point to any of the three data locations of memory, storage, and `calldata`. Thus, the variables of the reference types should be used carefully to not point to incorrect locations. For example, in Figure 1, the struct variable `gh` incorrectly points to the preallocated location of other storage variables.

c: ERROR HANDLING

If an exception happens in Solidity, all changes in the function call are reverted. For throwing exceptions on conditions, Solidity uses `require` and `assert`. The `require` function is used for checking whether inputs and state variables are valid, whereas the `assert` is used for checking invariants or internal errors.

2) BUGS IN SOLIDITY

Solidity compiler bugs should be considered for securing smart contracts. The Solidity compiler is rapidly evolving in a short time, but some versions have had critical bugs. The bugs in a compiler can cause the semantic inconsistency between the source code and bytecode. The unintended corner cases from the inconsistency become a new source of vulnerability. In the blockchain, even a new smart contract must interact with the predeployed smart contracts that may be generated with an old error-prone compiler. Thus, we must deal with the compiler bugs, even after the Solidity compiler fixed the known bugs, and cannot separate the concerns from smart contracts security. In this section, we briefly review the four representative compiler bugs as follows.

a: ABIEncoderV2

Solidity introduced the experimental encoder, referred to as ''ABIEncoderV2,'' for encoding complex data types to interact outside. However, the early version of ABIEncoder was incorrect for the variable of short noninteger types or reference types (until v. 0.5.6). Figure 2 demonstrates the case of the ABIEncoderV2 bug. Functions `h` and `i` take a two-dimensional array as input, and encode it via

```
1  pragma experimental ABIEncoderV2;
2  contract ABIEncoderV2Bug {
3
4      uint256[2][] public tmp_h;
5      bytes temp;
6      function h(uint256[2][] calldata s) external
           returns (uint256[2][] memory) {
7          tmp_h = s;
8          temp = abi.encode(tmp_h);
9          return abi.decode(temp, (uint256[2][]));
10     }
11
12     uint256[2][2] public tmp_i;
13     function i(uint256[2][2] calldata s) external
           returns (uint256[2][2] memory) {
14         tmp_i = s;
15         temp = abi.encode(tmp_i);
16         return abi.decode(temp, (uint256[2][2]));
17     }
18
19     function compare() public returns (bool) {
20         if(h([[1,2],[3,4]]) == i([[1,2],[3,4]])){
21           return true;
22         }else{
23           return false;
24         }
25     }
26 }
```

FIGURE 2. ABIEncoderV2 bug.

```
1  contract C {
2    function f() public pure returns(uint8 x) {
3        uint8 y = uint8(2) ** uint8(8);
4        return 0 ** y;
5    }
6  }
```

FIGURE 3. ExponentCleanUp bug.

ABIEncoderV2 and then return the decoded version of the value, but `h` uses the dynamic array unlike to `i`. If both the arguments are correctly encoded and decoded, the result of the comparison on Line 20 should be true. However, it is false. The value of `h` is unintentionally overwritten during the encoding and decoding process; thus, it becomes incorrect (*i.e.,* [[1,2],[2,3]] not [[1,2],[3,4]]).

b: ExponentCleanUp

When applying the exponential operator, `**`, to a variable type smaller than 256 bits, the higher bit (significant bits) part of the exponent is not cleared in Solidity 0.4.24. Because of the residue of the higher bit part, the exponential operations result in incorrect values. For example, in Figure 3, the value of `y` on Line 3 should be zero, because it overflows the range of `uint8`. The return value on Line 4 is 1 (*i.e.,* $0^0 = 1$, if $y = 0$). However, the residue from the operations on Line 3 causes `y` to become a nonzero value. Thus, the function `f` incorrectly returns 0.

c: HigherOrderByteCleanStorage

The default size of a slot of the EVM storage is 256 bits. To efficiently store variables, Solidity packs multiple variables of types smaller than 256 bits into a single storage slot. Thus, it internally uses bitwise operations for variable packing. However, the bitwise operations in variable packing have bugs in Solidity 0.4.3: the higher order bits are not cleaned

```
1  pragma solidity ^0.4.3;
2
3  contract Test {
4      uint48 a;
5      uint48 b;
6
7      function run() returns (uint48) {
8          a--;
9          return b;
10     }
11 }
```

FIGURE 4. HigherOrderByteCleanStorage bug.

up. This may accidentally overwrite other variables that are packed in the same storage slot. In Figure 4, two variables, `a` and `b`, are declared on Lines 4 and 5, and they are initialized as 0 at the same time. Because the two variables are of type `uint48`, which is smaller than `uint256`, they are packed into the same storage slot. On Line 8, `a` is underflowed by the decrement operation. Although the two variables should be independent, the underflow overwrites the adjacent variable `b` because the result is stored without properly masking the 48-bit data only.

d: UninitializedStoragePointer

As described in §II-B1, a struct is a reference type; thus, the fields pointers of the struct can point to storage or memory variables. When a struct is defined without any initialization, the default value is 0. In other words, the fields pointers of the struct point to storage slots in order from the location 0. Thus, if a developer unintentionally misses initializing a struct variable, the further field access through the struct variable may be incorrect or overwrite the other state variables. In Figure 1, `gh` on Line 10 is defined for the `GuessHistory` struct, but proper initialization is missed. The first field `number` of `GuessHistory` points to the first storage slot, corresponding to `randomNumber`, and the second field `player` points to the second storage slot, `lastPlayed`. Consequently, further write operations through `gh` on Lines 11 and 13 mistakenly overwrite the state variables `randomNumber` and `lastPlayed`.

C. VULNERABILITIES IN SMART CONTRACTS

A comprehensive security study on Ethereum [4] indicate that the largest portion of the Ethereum vulnerabilities is related to the application layer, which should be eliminated in smart contracts. The vulnerabilities in smart contracts cover from general vulnerabilities (*e.g*, integer overflow) to blockchain-specific vulnerabilities (*e.g*, reentrancy). Regardless of the difficulties in exploitation, blockchain-specific vulnerabilities have caused more severe attacks, such as the TheDAO [2] attack, because they were previously unknown. In this section, we briefly introduce the representative vulnerabilities in smart contracts.

1) REENTRANCY

The reentrancy vulnerability is caused by unintended recursive calls. The attacker exploiting this vulnerability aim to

execute money sending actions repeatedly before reaching a balance check. Thus, this exploitation is incurred when smart contracts developers miss two smart-contract-specific traits: (1) sending money is equivalent to calling the *fallback* function of the recipient, and (2) if the *fallback* function of the recipient is defined, then it can obtain the control. The typical attack case is that the recipient withdrawing funds calls the withdraw function again via its fallback function before the balance is updated.

```
1  contract Example {
2    mapping (address => uint) private userBalances;
3
4    function withdrawBalance() public {
5      uint amountToWithdraw = userBalances[msg.sender];
6      (bool success, ) = msg.sender.call.value(
           amountToWithdraw)("");
7      require(success);
8      userBalances[msg.sender] = 0;
9    }
10 }
```

FIGURE 5. Reentrancy vulnerability.

Figure 5 demonstrates an example case of the reentrancy vulnerability. In Figure 5, the `call` on Line 6 executes the sender's *fallback* function and sends money, as much as the user's balance. If the *fallback* function in the sender's code calls the function `withdrawBalance` again, a recurring call loop of sending money is created before reaching the balance state update on Line 8. Thus, the victim smart contract unboundedly sends money to the recipient. Reentrancy is a notorious vulnerability that has caused massive economic damage in the TheDAO exploitation [2]. After the TheDAO incident, the secure development guides recommend completing state updates before committing any actions that may call other code.

2) INTEGER OVERFLOW AND UNDERFLOW

Integer overflow and underflow are inevitable vulnerabilities for data types having a limited value range. If a value increases to greater than the maximum number that the data type can represent (*e.g*, $2^{256} - 1$ for a 256-bit unsigned integer), then we obtain an extremely low value instead. This error case is referred to as *overflow*. The opposite case, when a value decreases to below the enabled range, is referred to as *underflow*. The integer overflow and underflow vulnerabilities are critical for smart contracts in particular because the smart contract usually stores the balances of assets in integers. If an attacker causes an underflow of the balance, then the attacker can take over most of the assets in the smart contract. Thus, integer overflow and underflow are dominant in the number of reported vulnerabilities for smart contracts [11].

3) BAD RANDOMNESS

The fundamental parameters and states of the blockchain are deterministically calculated because distributed participants should be able to agree. Thus, a random number generator in the blockchain must simultaneously be both deterministic (or verifiable) and unpredictable. Smart contracts, however, may mistakenly use predictable values, such as the timestamp or hash of the latest block, as the source of random number generation. The values are not suitable as random sources because anyone can obtain the same information and predict the number exactly. On Line 7 of Figure 6, a random number is inappropriately generated from the timestamp of the latest block, `block.timestamp`, which is published to everyone after a block is created. Security development guides recommend against deriving a random value from the blockchain parameters.

```
1  contract BadRandomness {
2
3    uint pot;
4
5    function play() payable {
6      pot += msg.value;
7      uint random = uint(sha3(block.timestamp)) % 2;
8      if(random == 0) {
9        msg.sender.transfer(pot);
10       pot = 0;
11     }
12   }
13
14 }
```

FIGURE 6. Bad randomness vulnerability.

4) UNPROTECTED SELFDESTRUCT

The `selfdestruct` instruction is used to remove a smart contract and retrieve money from it. Even though it is a very powerful instruction, some smart contracts use it without careful access control. Figure 7 is a simplified version of the Parity wallet case [3]. The function `kill` calls `selfdestruct` on Line 35 only if the caller is the same as the `owner`. The `owner`-setting function, `initWallet`, should be called only when the contract is initialized. However, the constructor is missing in Figure 7; thus, `initialized` remains `false`, and `initWallet` is then callable by anyone. This means anyone can be the `owner` and destroy the contract by calling the `kill` function.

D. CONTRACT ANALYZERS

The survey study [4] stated that using the contract analyzers is the second-most effective method to protect smart contracts from vulnerabilities, whereas the the most effective method is writing error-free code with following best practices, which requires considerable expertise regarding smart contracts. Therefore, checking with the contract analyzers has become a practical, essential step in the development process of smart contracts.

1) CONTRACT ANALYZER CLASSIFICATION

a: VERIFICATION VS. VULNERABILITY SCAN

In finding security problems, we can check whether any possible violations of a smart contract exist based on the specification. A formal verification method, such as VerX [12],

```
contract Wallet {

  mapping(address => uint256) balanceOf;
  address owner;
  bool private initialized = false;

  modifier onlyOwner {
    if(owner == msg.sender)
      _;
  }

  modifier uninitialized {
    if(initialized == false)
      _;
  }

  function initWallet(address _owner, uint256 _amount)
  public uninitialized {
    owner = _owner;
    balanceOf[msg.sender] = _amount;
    initialized = true;
  }

  function transfer(address _to, uint256 _amount)
  public {
    uint256 result = balanceOf[msg.sender] + _amount;
    require(result >= balanceOf[msg.sender]);
    require(balanceOf[msg.sender] >= _amount);
    balanceOf[msg.sender] -= _amount;
    balanceOf[_to] += _amount;
  }

  function kill(address payable _to)
  public onlyOwner {
    selfdestruct(_to);
  }
}
```

FIGURE 7. Self-destruct vulnerability.

proves the correctness of a given smart contract using formal models. The violations found by formal verification can cover the vulnerabilities and the logic bugs. Thus, the formal verification is powerful, but requires precise specifications. Writing specifications is a demanding job because the properties should be formally represented (*e.g*, using temporal logic). In contrast, the vulnerability scanners use the predefined patterns for known vulnerabilities, which are bundled with the tools. While traversing the program code or execution traces, the vulnerability scanners compare the code with the vulnerability patterns. Thus, the vulnerability scanners can find only known vulnerabilities, but any developers can use them without requiring expertise. The widely used contract analyzers fall under vulnerability scanners because they are more practical.

b: SOURCE CODE VS. BYTECODE

Contract analyzers operate on the different input levels: source code or bytecode. The Solidity source code has more abstractions that reflects developers' intentions. Thus, the analysis with the source code can easily identify the code and data structure. However, it may be inaccurate because the source code does not always exactly reflect the behavior of smart contracts. For instance, the Solidity compiler bugs (§II-B2) make the behavior between source code and bytecode inconsistent. Thus, most tools analyze smart contracts at the bytecode level, and optionally use the source code to extract the metadata, such as the application binary interface (ABI).

While the bytecode of smart contracts is rather smaller than the binary code of the traditional computing environments, analyzing bytecode is still burdensome. For example, contract analyzers should identify all operands in every instruction, including jumps, by following indirect data access via the execution EVM stack.

2) WIDELY USED CONTRACT ANALYZERS

A number of contract analyzers are being proposed. The features of the well-known contract analyzers are comprehensively analyzed in [13]. In this paper, we focus on the six tools under active development as follows.

a: MANTICORE

Manticore is a dynamic symbolic execution engine for traditional binary analysis frameworks [14] but adds a new Ethereum execution module that supports Ethereum's execution environments, such as the memory and persistent storage data model. Manticore executes the bytecode of smart contracts with the emulated EVM world state. For inputs, the `callvalue` and `calldata` in a transaction data structure are treated as symbols. During the symbolic execution, Manticore analyzes and detects vulnerability patterns given in separated modules by evaluating them with the execution states.

b: MYTHRIL AND MythX

MythX [15] is an online service consisting of three tools: Mythril for EVM bytecode symbolic execution, Harvey for smart contract fuzzing, and Maru for static analysis of smart contracts. MythX analyzes smart contracts on both levels: source code and compiled bytecode. MythX then integrates the results from the three tools to improve the results from different approaches. Among the components, Mythril [16] traverses every possible paths of a smart contract via symbolic execution. It also applies a taint analysis to detect predefined vulnerability patterns more accurately.

c: SECURIFY

Securify [17] decompiles the EVM bytecode into the intermediate language, Soufflé. It is then converted into the semantic facts in Datalog. Securify analyzes the semantic facts using predefined violation patterns. If any matched pattern is found between the semantic facts and the patterns, Securify reports it in the decompiled code.

d: SmartCheck

SmartCheck [18] statically analyzes Solidity source codes. It lifts an XML parse tree as the intermediate representation from a Solidity source code. SmartCheck considers smart contact concerns in four categories: security, functional, operational, and developmental. It encodes the pattern for the issues in XPath and queries whether the XML parse tree contains any matches to the patterns.

e: SLITHER

Slither [19] is a static analysis framework for Solidity source code (higher than v. 0.4). It supports application programming interfaces for customized analysis of the source code. It also lifts its own intermediate representation, SlithIR, from the Solidity source codes. Moreover, SlithIR also focuses on presenting the detailed information, such as the inheritance relationship and control flows, for manual and automatic inspection.

III. RELATED WORK

A. COMPARISONS OF CONTRACT ANALYZERS

Contract analyzers for smart contracts are actively developed, but their performance is still in question. The features and accuracy of contract analyzers for smart contracts have been compared. Fontein evaluated the detection capacity of contract analyzers and compared it to the claims in other studies [20]. Angelo *et al.* provided a more comprehensive survey on contract analyzers [13]. The authors classified contract analyzers concerning the purpose, target level (bytecode or source code), analysis type, and methods. As the authors stated in [13], smart contract analyzers are driven by industry. Few academic-only tools are being managed.

More recently, Durieux *et al.* compared the detection performance of the contract analyzers [5]. The authors used a manually collected dataset and all active contracts in Google BigQuery to assess nine contract analyzers. Like the other studies, they found notable false-positive errors from the state-of-the-art contract analyzers. These studies focus on feature classification and performance testing with the bundled sample code or predeployed smart contract code without any modifications.

In this paper, we take further steps. We aim to generate varied test cases for contract analyzers. Our approach is useful for contract analyzers to find new weak points, which the fixed sample test cases may yet uncover. Thus, we evolve the test suite for contract analyzers by gradually adding more complex parts to the test case code. We can identify the resiliency of the contract analyzers against the complex bugs.

B. TEST CASE AND SUITES FOR SMART CONTRACTS

The automated test case generation for smart contracts has generally focused on assessing the smart contract itself, not the contract analyzers. Wu *et al.* [21] used mutation testing, which automatically generates the mutant test cases by applying mutation operators to a seed smart contract code. They defined new mutation operators reflecting the Solidity syntax and environment. The generated mutants may produce inconsistent results and reveal the potential defects in the original program code, which is helpful for the smart contracts developers. Zhang *et al.* [22] proposed a genetic algorithm to generate test input cases for a given smart contract code. The goal of the proposed approach is to cover as many definition-use pairs of variables as possible in the generated test cases, putting on more weight on the pairs dependent on the Solidity `require` statement. Thus, the inputs for a more important path, which may contain the `require` function, can more likely be generated. Gao *et al.* [23] proposed an integrated test input case generation from the perspective of the decentralized applications, which consists of a web interface part and the smart contract part. They identified meaningful web input events, which can influence the control condition, and they generated test cases based on the identified input, aiming to maximize the control path coverage. Whereas the approaches focused on testing the smart contracts, the purpose of our approach is to build a test suite to assess contract analyzers.

```
1  pragma solidity ^0.4.24;
2  contract reentrancy {
3  mapping (address => uint) public balanceOf;
4  uint public evenOdd = uint256(keccak256(now))%2;
5  uint public lastPlayed;
6
7  struct GuessHistory {
8      address player;
9      uint256 number;
10 }
11
12 function withdrawBalance() public {
13     GuessHistory guessHistory;
14     guessHistory.player = msg.sender;
15     guessHistory.number = 6;
16     if(evenOdd == 0) {
17         uint amountToWithdraw = balanceOf[msg.sender];
18         (bool success, ) = msg.sender.call.value(
               amountToWithdraw)("");
19         require(success);
20         balanceOf[msg.sender] = 0;
21         lastPlayed = now;
22     }
23 }
24 }
```

FIGURE 8. A sample test case for unreachable Reentrancy vulnerability. Contract analyzers falsely detect the vulnerability in the code.

IV. TEST SUITE FOR CONTRACT ANALYZERS

A. PITFALLS OF CONTRACT ANALYZERS

We tested contract analyzers with different settings of test cases. We identified the recurring problems that contract analyzers experience in finding the vulnerabilities of smart contracts. Figure 8 is an example of the reentrancy vulnerability. In Figure 8, the contract may call the recipient contract (Line 18) before updating the balance (Line 20). Contract analyzers detect this vulnerability pattern and verify the exploitability. In Figure 8, a contract analyzer should be able to follow the control flow from the entry of the function, `withdrawBalance`, and determine the feasibility of the path to the vulnerability pattern correctly on Line 16. Accordingly, the contract analyzer should also know that `evenOdd` is a state variable and it cannot be 0 due to the Solidity compiler bug on the uninitialized struct. However, only a single security analyzer could correctly detect this case in our experiments (§VI).

Based on the observations of the errors of contract analyzers, we categorize the findings into three challenges for smart contracts for contract analyzers.

- **Managing control structures (control flow analysis).** Contract analyzers should be aware of the control structure of programs and determine the feasibility of every path. If contract analyzers accept source code, they can quickly identify control structures. Otherwise, they must carefully recover the control structures from the bytecode. The EVM bytecode has no direct jumps; data flow analysis techniques must be applied, at minimum, to manage the control structures.
- **Following data changes (data flow analysis).** Code reachability is dependent on path feasibility. When determining the path feasibility of conditional branches, contract analyzers must evaluate the condition with the results from the data flow analysis or execution. Because this effort to perform this process is formidable, each contract analyzer devises its own way to simplify it unless target accuracy is lost. Thus, based on the underlying model for tracking data flow, the accuracy in identifying actual vulnerabilities differs among contract analyzers.
- **Compiler (version-dependent) bugs.** Although the former two challenges are more general problems in program analysis, the compiler bugs are domain-specific. Solidity compiler is being actively developed and frequently updated. A new version of the compiler may change the semantics at the bytecode level to fix the bugs in the previous versions. Therefore, contract analyzers must correctly manage the actual semantics of smart contracts. When contract analyzers accept source-code only (*e.g*, Slither), version-aware behavior modeling is critical for accurate results.

B. TEST CASE COMPLEXITY

Regarding the challenges in §IV-A, we diversify test cases to help contract analyzers obtain accurate results from complicated smart contracts. Thus, we gradually evolve the complexity of the test cases. Measuring the complexity of program codes has been discussed in numerous studies. For example, cyclomatic complexity [24] provides a method to quantify the structural complexity of computer programs. In this paper, however, we focus on smart-contract-specific problems in arranging the complexity of test cases for contract analyzers. The complexity of test cases is determined by the difficulty in finding a vulnerability pattern. We refer to a known vulnerability pattern for security analyzers as a vulnerability trigger. At the lowest complexity level, the vulnerability trigger is directly visible. We enable the execution path to become more complicated as the complexity level increases. The security analyzers must manage control and data flows and may be deceived by compiler bugs. At a higher level, we use false-positive errors and false-negative errors. Only exploitable vulnerability triggers should be correctly alarmed by contract analyzers.

We define four complexity levels, from Level 0 ($\mathcal{L}0$) to Level 3 ($\mathcal{L}3$). In $\mathcal{L}0$, the vulnerability trigger is directly accessible and is used to verify whether the security analyzer is aware of the vulnerability trigger. In $\mathcal{L}1$, the control structures are added to the vulnerability triggers. The security analyzers must also follow the changes in data flows to determine the path to the vulnerability triggers in $\mathcal{L}2$. Finally, in $\mathcal{L}3$, complex compiler bugs influence the exploitation path to the vulnerability triggers. For higher levels, the false-positive errors demonstrate more comprehensively how the security analyzers verify smart contracts compared to false-negative errors.

1) LEVEL 0 ($\mathcal{L}0$)

Level 0 is a basic test case of a vulnerability trigger. The test cases in $\mathcal{L}0$ have a vulnerability trigger at a directly reachable position and are thus exploitable. If a security analyzer fails to find the vulnerability trigger, it is also unable to detect the vulnerability trigger of the same pattern at higher levels.

2) LEVEL 1 ($\mathcal{L}1$): CONTROL STRUCTURES

In $\mathcal{L}1$, we enclose the vulnerability trigger with a control structure. We use both conditional branches and loops (`if` and `while`, respectively) for constructing control structures. For verifying the control flow awareness of contract analyzers, the control structures in $\mathcal{L}1$ have naive, obvious conditions, such as `2 > 4` or `a < 10`. We generate test cases to verify both false-positive and false-negative errors by complementing the conditions. If a security analyzer detects the vulnerability trigger in the conditional block under a false condition, it has a false-negative error in $\mathcal{L}1$. Contract analyzers that linearly sweeps smart contracts or have errors on condition evaluation may produce incorrect results.

3) LEVEL 2 ($\mathcal{L}2$): CONTROL STRUCTURES AND DATA FLOW

The test cases in $\mathcal{L}2$ are evolved for verifying whether contract analyzer can manage more general conditions of control structures. Thus, we set value-dependent conditions for control structures. To evaluate the condition, contract analyzers must be aware of possible values of the variables, which reflects the accuracy of the data flow analysis of contract analyzers. We also gradually add arithmetic operations and function calls to data flows to test the value range handling and inter-procedural analysis. Smart contracts that manage ERC20 tokens in Solidity are heavily dependent on the use of math functions even for basic arithmetic operations to prevent the integer overflows and underflows. Thus, managing complicated arithmetic operations is critical for improving the accuracy of contract analyzers.

4) LEVEL 3 ($\mathcal{L}3$): COMPLEX BUGS

Errors in compilers distinguish the actual behavior of operations from the intended behaviors. Furthermore, the incorrect behaviors caused by compiler bugs are not depicted in the source code and may be inexactly modeled by the bytecode-level contract analyzers, disregarding the actual behavior. In $\mathcal{L}3$, the Solidity compiler bugs are added to test cases because exploitabilty is dependent on the actual behavior of compiler bugs, and the honeypot code that uses them. The test

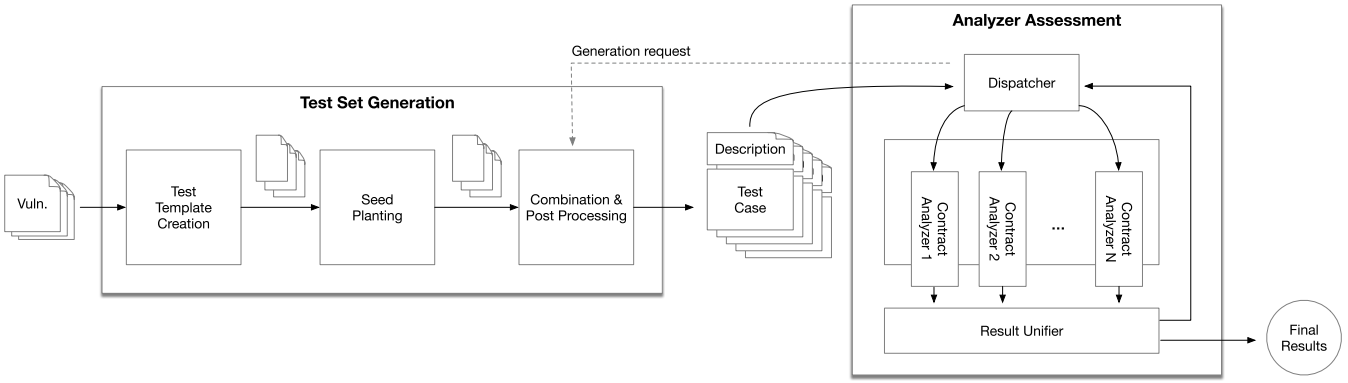

FIGURE 9. Overall process of TestBreeder.

cases in $\mathcal{L}3$ have vulnerability triggers that are unreachable due to one of Solidity's compiler bugs. Hence, the false-positive errors in the $\mathcal{L}3$ test cases demonstrate the mistakes of the contract analyzers in understanding the exact behavior of smart contracts.

V. TestBreeder DESIGN

In this section, we present the system design of TestBreeder. TestBreeder automatically generates test cases that covers all complexity levels (§IV-B) for contract analyzers.

A. PROCESS OVERVIEW

As depicted in Figure 9, the process of TestBreeder consists of two phases: (1) test set generation and (2) analyzer assessment. First, TestBreeder creates a test template for each vulnerability trigger. The test template has incomplete code with several empty slots. For the complexity level, TestBreeder has code seeds that corresponds to the slots of each part. TestBreeder repeatedly generates test cases by filling the empty slots with code snippets generated from the sets of code seeds. In this way, TestBreeder prepares test cases covering all complexity levels for a vulnerability trigger. A test case consists of the variants of a vulnerability trigger. At the end of the first phase, the test cases are delivered with descriptions of the vulnerability types and complexity levels. The dispatcher assesses contract analyzers in the second phase by selecting test cases in order of complexity level and by inputting the test case into the contract analyzers. The results from the tools are converted to a standardized form.[1] The dispatcher proceeds to the next test case of a higher complexity level in the same vulnerability category.

B. TEST TEMPLATE CREATION

A test template is constructed for a single vulnerability trigger. Figure 10 illustrates an example of the test template for the reentrancy vulnerability. A test template has empty slots of three types: initialization, auxiliary, and enclosure. The first two slots, initialization and auxiliary, contain the initial

[1]We used a predefined format of JavaScript Object Notion (JSON).

```
contract reentrancyTestTemplate {

  mapping (address => uint) public balanceOf;
  // [Initialization slot]

  // [Auxiliary slot]

  function withdrawBalance() public {
    // [Enclosure slot (prologue)]
      uint amountToWithdraw = balanceOf[msg.sender];
      (bool success, ) = msg.sender.call.value(
          amountToWithdraw)("");
      require(success);
      balanceOf[msg.sender] = 0;
    // [Enclosure slot (epilogue)]
  }
}
```

FIGURE 10. Test template for the reentrancy vulnerability.

definitions of variables and functions used in the code of the enclosure slot. The enclosure slot determines the complexity level of the test case. Conditional branches or loops with the adjusted conditions for the given level are used for the enclosure slot. As depicted in Figure 10, the enclosure slot is split into two parts, a prologue and epilogue, to surround a vulnerability trigger.

C. SEED PLANTING

1) CODE SEED SET

A code seed is a prepared code prototype for a slot. The slots of the test template are filled with code snippets generated from code seeds. Because a test template contains slots of three types, the code seeds of the three types form a code seed set.

2) INITIALIZATION SEEDS

An initialization seed defines state variables and initialization functions, including the constructor of a contract. Figure 11 is an example of the initialization seed. The state variables are used in seeds of other types. The initialization seed also has functions to initialize the state variables. In Figure 11, the state variables are initialized via the functions `constructor` and `initialize`. A seed may have a placeholder marked by '$' for changeable parts. TestBreeder

```
// state variable
uint256[2][2] public tmp_i;
uint256[2][2] public fin_i;
uint256[2][2] public init = [[$RANDOM$,$RANDOM$],[
    $RANDOM$,$RANOM$]];
bytes temp;

// initialization
function initialize(uint256[2][2] memory s) public {
  tmp_i = s;
  temp = abi.encode(tmp_i);
  fin_i = abi.decode(temp, (uint256[2][2]));
}

// constructor
constructor() public {
  initialize(init);
}
```

FIGURE 11. State variable definitions and constructors for the initialization seed.

replaces placeholders with appropriate values or operators. For example, `$RANDOM$` in Figure 11 is a placeholder for a random number.

```
function arrayCheck(uint256[2][2] memory a, uint256
    [2][2] memory b) internal returns (bool) {
  for (uint i=0; i<2; i++){
    for (uint j=0; j<2; j++){
      if (a[i][j] != b[i][j]) {
        return false;
      }
    }
  }
  return true;
}
```

FIGURE 12. Supplemental functions for the auxiliary seed.

3) AUXILIARY SEEDS

The auxiliary seed has supplemental functions to be used in the enclosure slot. It provides a functional library for the enclosure slot. For example, in Figure 12, the auxiliary seed defines `arrayCheck`. Because `arrayCheck` compares each element of two arrays in nested loops, it can verify how accurately a contract analyzer manages mixed flows of control and data used it in the enclosure slot.

4) ENCLOSURE SEEDS

The enclosure seed controls the complexity level of the test set. It surrounds a vulnerability trigger with prologue and epilogue components. The components are empty for the $\mathcal{L}0$ test cases to enable direct exploitation of the enclosing vulnerability. The enclosure seeds also have multiple versions within a single level except for $\mathcal{L}0$, so multiple test cases are generated for each level.

In $\mathcal{L}1$, we add a control structure, such as `if` and `while`, with a simple condition. We then double it by complementing the condition, one for a false-positive error and the other for a false-negative error. The $\mathcal{L}1$ test cases vary between an acyclic branch (`if`) and a cyclic branch (`while`) with different conditions. Figure 13 illustrates an $\mathcal{L}1$ enclosure seed using `if`. To prevent the conditional branch from being eliminated by compiler optimizers, the branch condition in $\mathcal{L}1$ can change from an obvious condition, such as `false` to a dependent condition (*e.g.*, `a < 1`).

```
// Enclosure prologue
int a = 10;
if (a < 1) {
   ... a vulnerability trigger ...
// Enclosure epilogue
}
```

FIGURE 13. Prologue and epilogue of the $\mathcal{L}1$ enclosure seed.

In $\mathcal{L}2$, arithmetic operations that require accurate data flow tracing are added to the condition of the control structures. The $\mathcal{L}2$ test cases diversify by accumulating arithmetic operations on the variable used in the branch condition. The relational operators in the branch condition are arranged with arithmetic operations to verify both false positives and false negatives. For example, monotonically increasing arithmetic operations are matched with the greater-than operator in the condition.

In $\mathcal{L}3$, we use Solidity compiler bugs. We flipped the false positive and false negative by indicating the target compiler version as the version number before and after the compiler bug is patched. In Figure 14, the test case uses the ABI encoding bugs patched in Solidity 0.5.7. The enclosure induces the bug by specifying the target compiler version (0.5.5) and uses the function `arrayCheck`, which is prepared in the auxiliary seed. According to the `pragma` directive, the dispatcher uses the matched Solidity compiler in building bytecode. If a contract analyzer detects the enclosing vulnerability trigger even when the compiler bugs are in effect, it indicates false-positive errors in the results.

```
// pragma directive
pragma solidity >=0.5.5 <0.5.6;
pragma experimental ABIEncoderV2;
// Enclosure prologue
if (arrayCheck(tmp_i,fin_i)) {
   ... a vulnerability trigger ...
// Enclosure epilogue
}
```

FIGURE 14. Prologue and epilogue of the $\mathcal{L}3$ enclosure seed.

D. COMBINATION AND POST PROCESS

1) COMBINATION

TestBreeder combines a test template with multiple code sets by placing each seed into the corresponding slot. It fills the placeholders remaining in the test cases. The placeholder may require random numbers and merging functions, such as constructors for initializing variables.

2) POST-PROCESS

In the last step, TestBreeder verifies the test case and generates metadata. It verifies whether the generated test case can be compiled by the Solidity compiler. A test case is coupled with a description, which includes the complexity level, base vulnerability trigger, planted seed set, and attributes related

to the placeholder (*e.g*, whether random numbers are used in generation).

E. DISPATCHER AND UNIFIER

In the analyzer assessment phase, TestBreeder composes the prepared seed sets into test cases.

1) DISPATCHER

The dispatcher in Figure 9 finds a test case based on its description and inputs it into the contract analyzers deployed in containers. The dispatcher starts from the test case of the lowest level and proceeds to those of the higher levels. When a contract analyzer fails to detect the $\mathcal{L}0$ test case, the dispatcher does not verify the higher test cases, because it indicates that the contract analyzer cannot recognize the vulnerability trigger. This process stores and manages the results of each analyzer as a JSON file.

2) UNIFIER

Every security analyzer uses a different notation to describe the results. Thus, the TestBreeder unifier transforms the results into a standardized form to compare with each other.

VI. EVALUATION

In this section, we explain how resilient the existing contract analyzers are against the test cases generated by TestBreeder. We also describe the case studies we conducted on notable errors.

A. IMPLEMENTATION AND EXPERIMENTAL SETUP

We tested five contract analyzers: Manticore [14] (v. 0.3.3), SmartCheck [18] (v. 2.0), Slither [19] (v. 0.6.9), MythX [15], and Securify [17].[2] For the experiment, we generated 110 test cases from 52 code seed sets on five vulnerabilities. We implemented TestBreeder with 1,059 lines of Python. Each contract analyzer was separately operated in an isolated Docker container. The results of the contract analyzers are unified to a standardized JSON file using tool-specific transformers in the TestBreeder unifier.

B. DECISION VALIDATION

As described in §V-E, TestBreeder unifies the results from the contract analyzers into a compatible form at the end of the process. We implemented the unifier in two steps. First, for each contract analyzer, we defined an extraction pattern. For the contract analyzers generating structured output (*e.g*, JSON outputs from Slither) we directly mapped the detected vulnerability name field to the name field in our JSON form. If the contract analyzers produce textual results, we used the regular expression patterns to extract the information, such as the vulnerability name. Second, we converted the detected vulnerability names to the standardized code name in the Smart Contract Weakness Classification Registry (SWC) [10]. For example, all ''reentrancy'' variants were converted to ''SWC-117.'' Based on the normalized output, we checked whether a contract analyzer detects the vulnerability as intended in the test case, and determined any false-positive or false-negative errors. Unfortunately, there is no single authoritative reference test cases for all the contract analyzers yet.

Hence, to validate the collection process, we verified it using a reference set composed of the sample examples bundled with contract analyzers and the reference codes from SWC. We employed the sample code of Slither because its input type is most limited among the contract analyzers. We manually inspected the results of the normalized output from the five contract analyzers and verified that we correctly interpreted the output and identified the errors.

C. COMPARISON OF CONTRACT ANALYZERS

The test cases generated by TestBreeder challenge contract analyzers cocerning false-positive and false-negative errors. Thus, we compared the results of the contract analyzer using the same terms in this section. In each result, the test cases were classified by complexity levels and topics, which represent the variation within each complexity level.

1) REENTRANCY

The reentrancy vulnerability is the most widely covered vulnerability by security analyzers, so we compared the reentrancy results of contract analyzers. Table 1 presents the results of reentrancy. The $\mathcal{L}0$ test case is used to verify whether a security analyzer manages the vulnerability. The result demonstrates that all five tools detected the reentrancy vulnerability in the $\mathcal{L}0$ test cases. From $\mathcal{L}1$, three contract analyzers, Slither, Security, and SmartCheck, found the reentrancy vulnerability even in unreachable branches. Symbolic execution-based Manticore and MythX were not deluded by the false branches. However, in $\mathcal{L}3$, both tools falsely reported with the compiler bug, HigherByteCleanStorage. We describe the more details on error cases in the case study (§VI-D1).

2) INTERGER OVERFLOW AND UNDERFLOW

Integer overflow and underflow are not covered by all contract analyzers. Table 2 reveals that three tools, Slither, Securify, and SmartCheck, did not find integer overflow or underflow in our $\mathcal{L}0$ test cases. Manticore and MythX resiliently detected integer overflow and underflow at all levels. However, as with reentrancy, both tools resulted in false alarms when the HigherByteCleanStorage bug influenced the branch. MythX also falsely reported the unreachable integer overflow vulnerability only but was correct regarding integer underflow.

3) BAD RANDOMNESS

As depicted in Table 3, SmartCheck, Manticore, and MythX can detect the bad randomness vulnerability. However, MythX and SmartCheck produced false-positive errors from the test cases of $\mathcal{L}1$ and $\mathcal{L}2$, respectively. In $\mathcal{L}2$, the test

[2]Both MythX and Securify are served online without a version number.

TABLE 1. False positive and false negative errors on the reentrancy vulnerability. The mark '✓' and '✓' indicate the occurrence of false positive (FP) and false negative (FN) errors, respectively.

Complexity level		Slither		Securify		Manticore		SmartCheck		MythX	
Level	Topic	FP	FN	FP	FN	FP	FN	FP	FN	FP	FN
0	(none)	N/A	·	N/A	·	N/A	·	N/A	·	N/A	·
1	If Condition	✓	·	✓	·	·	·	✓	·	·	·
	While Loop	✓	·	✓	·	·	·	✓	·	·	·
	For Loop	✓	·	✓	·	·	·	✓	·	·	·
2	Stage 1	✓	·	✓	·	·	·	✓	·	·	·
	Stage 2	✓	·	✓	·	·	·	✓	·	·	·
	Stage 3	✓	·	✓	·	·	·	✓	·	·	·
3	ABIEncoderV2	✓	·	✓	·	·	·	✓	·	·	·
	ExponentCleanUp	✓	·	✓	·	·	·	✓	·	·	·
	HigherByteCleanStorage	✓	·	✓	·	✓	·	✓	·	✓	·
	UnintializedStoragePointer	✓	·	✓	·	✓	·	✓	·	·	·

TABLE 2. False positive and false negative errors on the integer overflow and underflow vulnerability.

Complexity level		Slither		Securify		Manticore		SmartCheck		MythX	
Level	Topic	FP	FN	FP	FN	FP	FN	FP	FN	FP	FN
0	(none)	N/A	✓	N/A	✓	N/A	·	N/A	✓	N/A	·
1	If Condition	·	✓	·	✓	·	·	·	✓	·	·
	While Loop	·	✓	·	✓	·	·	·	✓	·	·
	For Loop	·	✓	·	✓	·	·	·	✓	·	·
2	Stage 1	·	✓	·	✓	·	·	·	✓	·	·
	Stage 2	·	✓	·	✓	·	·	·	✓	·	·
	Stage 3	·	✓	·	✓	·	·	·	✓	·	·
3	ABIEncoderV2	·	✓	·	✓	·	·	·	✓	·	·
	ExponentCleanUp	·	✓	·	✓	·	·	·	✓	·	·
	HigherByteCleanStorage	·	✓	·	✓	✓	·	·	✓	✓	·
	UnintializedStoragePointer	·	✓	·	✓	·	·	·	✓	△	·

TABLE 3. False positive and false negative errors on the bad randomness vulnerability.

Complexity level		Slither		Securify		Manticore		SmartCheck		MythX	
Level	Topic	FP	FN	FP	FN	FP	FN	FP	FN	FP	FN
0	(none)	N/A	✓	N/A	✓	N/A	·	N/A	·	N/A	·
1	If Condition	·	✓	·	✓	·	·	·	·	✓	·
	While Loop	·	✓	·	✓	·	·	·	·	✓	·
	For Loop	·	✓	·	✓	·	·	·	·	✓	·
2	Stage 1	·	✓	·	✓	·	·	✓	·	✓	·
	Stage 2	·	✓	·	✓	·	·	✓	·	✓	·
	Stage 3	·	✓	·	✓	·	·	✓	·	✓	·
3	ABIEncoderV2P	·	✓	·	✓	·	·	✓	·	✓	·
	ExponentCleanUp	·	✓	·	✓	·	·	✓	·	✓	·
	HigherByteCleanStorage	·	✓	·	✓	✓	·	✓	·	✓	·
	UnintializedStoragePointer	·	✓	·	✓	✓	·	✓	·	✓	·

cases have more arithmetic operations to burden data flow analysis. In the case study in §VI-D2, we considered that the added operations obstruct tracing the use of the inappropriate random sources in SmartCheck.

4) UNPROTECTED SELFDESTRUCT

In Table 4, MythX successfully detected all test cases. Manticore found most test cases correctly producing a false-positive error only with the HigherByteCleanStorage bug. Slither discovered the vulnerabilities regardless of reachability. However, Securify and SmartCheck could not detect the unprotected selfdestruct.

D. CASE STUDIES

1) CASE STUDY #1: REENTRANCY

Figure 15 illustrates a generated test case for the $\mathcal{L}3$ reentrancy vulnerability from TestBreeder with the HigherByteCleanStorage bug (§II-B2.c). Because of the bug, Line 15 in Figure 15 cannot be executed. However, Slither [19], Securify [17], SmartCheck [18], MythX [15], and Manticore [14] produced false alarms. Slither and SmartCheck were not aware of the bug caused by the Solidity compiler because they analyze smart contracts at the Solidity source code level only. Although Manticore, MythX, and Securify conduct vulnerability detection on the EVM at the byte code level, the bug misleads the three tools to produce incorrect results.

We investigated the source code and official documents of the three tools to find the causes of the false positives. Securify assumes that all code in the contract is executable, which is the reason for the false positive. Manticore and MythX query an satisfiability modulo theories (SMT) solver to determine whether the `if` condition (Line 14) can be satisfied. Nevertheless, they could not determine the effect

TABLE 4. False positive and false negative errors on the unprotected selfdestruct vulnerability.

Complexity level		Slither		Securify		Manticore		SmartCheck		MythX	
Level	Topic	FP	FN	FP	FN	FP	FN	FP	FN	FP	FN
0	(noe)	N/A	·	N/A	✓	N/A	·	N/A	✓	N/A	·
1	If Condition	✓	·	·	✓	·	·	·	✓	·	·
	While Loop	✓	·	·	✓	·	·	·	✓	·	·
	For Loop	✓	·	·	✓	·	·	·	✓	·	·
2	Stage 1	✓	·	·	✓	·	·	·	✓	·	·
	Stage 2	✓	·	·	✓	·	·	·	✓	·	·
	Stage 3	✓	·	·	✓	·	·	·	✓	·	·
3	ABIEncoderV2	✓	·	·	✓	·	·	·	✓	·	·
	ExponentCleanUp	✓	·	·	✓	·	·	·	✓	·	·
	HigherByteCleanStorage	✓	·	·	✓	✓	·	·	✓	·	·
	UnintializedStoragePointer	✓	·	·	✓	·	·	·	✓	·	·

```
1  pragma solidity ^0.4.3;
2  contract reentrancy {
3
4      mapping (address => uint) public balanceOf;
5      uint48 public challengeCoin = 0;
6      uint48 public random;
7
8      constructor() public {
9          random = uint48(now/10) % 10;
10     }
11
12     function withdrawBalance() public {
13         challengeCoin--;
14         if(random < 10) {
15             uint amountToWithdraw = balanceOf[msg.
                   sender];
16             (bool success, ) = msg.sender.call.value(
                   amountToWithdraw)("");
17             require(success);
18             balanceOf[msg.sender] = 0;
19         }
20     }
21     }
```

FIGURE 15. The test case of TestBreeder for the level-3 reentrancy vulnerability.

of the bug that prevents Line 15 from executing at runtime. Based on the code analysis, Manticore does not properly analyze two EVM instructions (SWAP1 and SWAP2) that are commonly used to optimize contracts using the Solidity compiler. The bug in Figure 15 is caused by SWAP1 and SWAP2. (Appendix §C). In the case of MythX, we could not perform a code-level analysis because it is not an open-source project.

2) CASE STUDY #2: BAD RANDOMNESS

Figure 16b presents the $\mathcal{L}2$ bad randomness test case in which SmartCheck [18] and MythX [15] produced false positives. As depicted in Figure 16b, the contract performs arithmetic operations during initialization. Line 9 uses the result of the operations as a condition of the `if` statement. Because the condition is false, Lines 10 and 11 are effectively dead code. Therefore, the contract is not vulnerable in terms of bad randomness.

However, the two tools [15], [18] reported a possibility of bad randomness. SmartCheck converts contracts into a XML-based structures and analyzes then the data structure to detect security issues. According to our analysis, SmartCheck uses regular expressions to find

```
1  contract GuessTheRandomNumberChallenge {
2      uint8 answer;
3      uint random = 8;
4      uint compareNumber;
5
6      constructor() public {
7          compareNumber = 8;
8      }
9
10     function checkJacpot() internal view returns (bool) {
11         if(compareNumber != random){
12             uint jacpot = now % 2;
13             return jacpot > 0;
14         }
15     }
16
17     function jacpot() public {
18         if(checkJacpot()){
19             msg.sender.transfer(address(this).balance);
20         }
21     }
22
23 }
```

(a) Level-1 BadRandomness

```
1  pragma solidity ^0.4.24;
2  contract GuessTheRandomNumberChallenge {
3      int var1 = 9;
4      int var2 = var1 + 92;
5      int var3 = var1 + var2;
6      int var4 = var2 * var3;
7
8      function checkJackpot() internal view returns (bool)
           {
9          if(var1 > var4){
10             bool _jackpot = now % 2 > 0;
11             return _jackpot;
12         }
13     }
14
15     function jackpot() public {
16         if(checkJackpot()){
17             msg.sender.transfer(address(this).balance);
18         }
19     }
20
21 }
```

(b) Level-2 BadRandomness

FIGURE 16. A test case of TestBreeder for the bad randomness vulnerability.

specific keywords or instructions (*e.g*, `now`). For example, if SmartCheck encounters `now` or `block.timestamp`, it determines whether the contract exhibits bad randomness regardless of reachability. Because of this limitation, it evaluates that Figure 16b has the bad randomness vulnerability.

Although MythX is an outstanding tool, it has a very high false-positive rate for the bad randomness vulnerability (Table 3). MythX itself is not open-sourced. However, one of its core components, Mythril [16], is an open-source project. Thus, we conducted a code analysis of Mythril. Mythril identifies specific operations (`COINBASE`, `GASLIMIT`, `TIMESTAMP`, `BLOCKHASH`, and `NUMBER`) when it verifies bad randomness. In contrast to SmartCheck, Mythril performs reachability verification by leveraging the SMT solver, but it determines that Line 11 (Figure 16b) is reachable. Consequently, MythX produced a false positive.

VII. CONCLUSION

We presented the test suite generator, TestBreeder, to assess contract analyzers. TestBreeder diversifies test cases by combining vulnerabilities and changing code complexities. Considering the analysis mechanisms of contract analyzers, TestBreeder uses control structures, arithmetic operations, and compiler bugs (to cause inconsistency between the source code and bytecode). Our experiment results illustrate that TestBreeder could find the limitations of the existing contract analyzers. Recurring false-positive errors, even in advanced contract analyzers, imply that the test approach of TestBreeder is effective in enhancing the contract analyzers that targets early-stage industries as in smart contracts and the blockchain. The test suite should be advanced parallel to the ongoing evolution of contract analyzers. TestBreeder can continuously expand the test cases by adding more elements and mutation operators.

APPENDIX A
$\mathcal{L}3$ CODE SEEDS SAMPLES

A. ABI ENCODER BUG

Figure 17 illustrates the code snippets of the code set with the `ABIEncoderV2` compiler bug. The initialization seed in Figure 17a consists of the `pragma` directive and initialization components. The compiler version that triggers the `ABIEncoderV2` bug is 0.5.5. The state variable `init` has four random numbers on Lines 7-9. On Line 18-20, `tmp_i` and `fin_i` are originated from the same value but are written by different values due to the `ABIEncoderV2` bug. For the auxiliary seed, Figure 17b presents the function `arrayCheck` verifies the equality of two given arrays. Finally, in the enclosure seed of Figure 17c uses `arrayCheck` as the entering condition.

```
1  // pragma
2  pragma solidity ^0.5.5;
3
4  // init
5  uint256[2][2] public tmp_i;
6  uint256[2][2] public fin_i;
7  uint256[2][2] public init
8  = [[randomNumber,randomNumber2],
9  [[randomNumber3,randomNumber4]];
10 bytes temp;
11
12 constructor() public {
13     balanceOf[msg.sender] = 100000000;
14     initialized(init);
15 }
16
17 function initialized(uint256[2][2] memory s) public {
18     tmp_i = s;
19     temp = abi.encode(tmp_i);
20     fin_i = abi.decode(temp, (uint256[2][2]));
21 }
```

(a) Initialization seed in ABI Encoder Bug

```
1  function arrayCheck(uint256[2][2] memory a, uint256[2][2]
       memory b) internal returns (bool) {
2      for (uint i=0; i<2; i++){
3          for (uint j=0; j<2; j++){
4              if (a[i][j] != b[i][j]) {
5                  return false;
6              }
7          }
8      }
9      return true;
10 }
```

(b) Auxiliary seed in ABI Encoder Bug

```
1  // Prologue
2  if (arrayCheck(tmp_i,fin_i)) {
3  // Epilogue
4  }
```

(c) Enclosure seed in ABI Encoder Bug

FIGURE 17. ABI encoder bug example.

B. EXPONENT CLEAN UP BUG

Figure 18 illustrates the code snippets of the code set with the `ExponentCleanUp` compiler bug. The initialization seed only has the `pragma` directive. `ExponentCleanUp` is fixed in Solidity 0.4.25, so we use 0.4.24 in Figure 18a. In the auxiliary seed, the function `f` returns an unexpected value due to the `ExponentCleanUp` bug. On Line 2, while `256` is assigned into `y`, the`ExponentCleanUp` bug is triggered in EVM. For the enclosure seed, then Figure 18c reveals that the condition on Line 2 is always false.

```
1  pragma solidity ^0.4.24;
```

(a) Initialization Seed in Exponent Bug

```
1  function f() public pure returns(uint8 x) {
2      uint8 y = uint8(2) ** uint8(8);
3      return 0 ** y;
4  }
```

(b) Auxiliary seed in Exponent Bug

```
1  // Prologue
2  if (f() != 0) {
3
4  // Epilogue
5  }
```

(c) Enclosure seed in Exponent Bug

FIGURE 18. Exponent clean up bug example.

C. HIGHER-ORDER BYTE CLEAN STORAGE BUG

Figure 19 illustrates the code snippets of the code set with the `HigherOrderByteCleanStorage` compiler bug. The `pragma` directive in Figure 19a uses Solidity 0.4.3 because `HigherOrderByteCleanStorage` is fixed at 0.4.4. In the initialization seed, two state variables are defined. On Lines 1 and 2, both `challengeCoin` and `random` are defined but only `challengeCoin` is ini-

```
// pragma
pragma solidity ^0.4.3;

// init
uint48 public challengeCoin = 0;
uint48 public random;
```

(a) Initialization Seed in Higher Order Bug

```
constructor() public {
    random = uint48(now/10) % 10;
}
```

(b) Auxiliary Seed in Higher Order Bug

```
//Prologue
challengeCoin--;
if(random < 10){
//Epilogue
}
```

(c) Enclosure Seed in Higher Order Bug

FIGURE 19. Higher order byte clean storage bug example.

```
// pragma
pragma solidity ^0.4.24;

// init
uint public random = uint256(keccak256(now))%10+1;
uint public lastPlayed;

struct GuessHistory {
    address player;
    uint256 number;
}
```

(a) Initialization Seed in Uninitialized Storage Bug

```
//Prologue
GuessHistory guessHistory;
guessHistory.player = msg.sender;
guessHistory.number = randomNumber;
if(random <= 10){
//Epilogue
}
```

(b) Enclosure Seed in Uninitialized Storage Bug

FIGURE 20. Uninitialized storage pointer bug example.

tialized to `0`. In the auxiliary seed of Figure 19b, the constructor initializes `random` as the last digit of the unexpected value, `now`. Finally, in the enclosure seed, the `HigherOrderByteCleanStorage` is triggered on Line 2 of Figure 19c. The decrement of `challenge` causes underflow through the SUB instruction. The underflowed value overwrites `random`. Thus, the condition of Figure 19c cannot be satisfied.

D. UNINITIALIZED STORAGE POINTER BUG

Figure 20 illustrates the code snippets of the code set with the `UninitializedStoragePointer` compiler bug. The `pragama` directive in Figure 20a uses Soldity 0.4.24 because `UninitializedStoragePointer` is patched at 0.4.25. In the initialization seed, two state variables and a struct type are defined. `random` is initialized as an integer of less than 10 on Line 5. `lastPlayed` is defined but is not initialized on Line 6. A struct `GuessHistory` is also defined but is not initialized on Lines 8-11. In the enclosure seed of Figure 20b, `player` and `number` in `guessHistory` are initialized on Line 2-4 as the given `msg.sender` and `randomNumber`, respectively. However, due to the `UninitializedStoragePointer` bug, the separated state values, `random` and `lastPlayed` are instead overwritten as the two given values. Thus, the condition on Line 5 is always false, with Solidity 0.4.24.

REFERENCES

[1] *Ethereum Project*. Accessed: Nov. 23, 2020. [Online]. Available: https://www.ethereum.org/

[2] U. Securities and E. Commission. (2017). *Report of Investigation Pursuant to Section 21(A) of the Securities Exchange Act of 1934: The DAO*. [Online]. Available: http://www.virtualschool.edu/mon/Economics/SmartContracts.html

[3] S. Palladino. (2017). *The Parity Wallet Hack Explained*. [Online]. Available: https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a%8c12e8f7/

[4] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks and defenses," 2019, *arXiv:1908.04507*. [Online]. Available: http://arxiv.org/abs/1908.04507

[5] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, Jun. 2020, pp. 530–541.

[6] D. Perez and B. Livshits, "Smart contract vulnerabilities: Vulnerable does not imply exploited," 2019, *arXiv:1902.06710*. [Online]. Available: http://arxiv.org/abs/1902.06710

[7] C. F. Torres and M. Steichen, "The art of the scam: Demystifying honeypots in ethereum smart contracts," in *Proc. 28th USENIX Security Symp. (USENIX Secur.)*, 2019, pp. 1591–1607.

[8] P. E. Black and P. E. Black, "Juliet 1.3 test suite: Changes from 1.2," in *US Department of Commerce*. Gaithersburg, MD, USA: National Institute of Standards and Technology, 2018.

[9] A. Wagner and J. Sametinger, "Using the juliet test suite to compare static security scanners," in *Proc. 11th Int. Conf. Secur. Cryptogr.*, 2014, pp. 1–9.

[10] *Smart Contract Weakness Classification Registry*. Accessed: Nov. 23, 2020. [Online]. Available: https://github.com/SmartContractSecurity/SWC-registry/

[11] *CVE of Smart Contract*. Accessed: Nov. 23, 2020. [Online]. Available: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=smart+contract

[12] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "VerX: Safety verification of smart contracts," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 18–20.

[13] M. di Angelo and G. Salzer, "A survey of tools for analyzing ethereum smart contracts," in *Proc. IEEE Int. Conf. Decentralized Appl. Infrastruct. (DAPPCON)*, Apr. 2019, pp. 69–78.

[14] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 1186–1189.

[15] *Mythx*. Accessed: Nov. 23, 2020. [Online]. Available: https://mythx.io/

[16] *Mythril*. Accessed: Nov. 23, 2020. [Online]. Available: https://github.com/ConsenSys/mythril

[17] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Jan. 2018, pp. 67–82.

[18] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of ethereum smart contracts," in *Proc. 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain - WETSEB*, 2018, pp. 9–16.

[19] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proc. IEEE/ACM 2nd Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, May 2019, pp. 8–15.

[20] R. Fontein, "Comparison of static analysis tooling for smart contracts on the EVM," in *Proc. 28th Twente Student Conf. IT*, 2018, pp. 1–8.

[21] H. Wu, X. Wang, J. Xu, W. Zou, L. Zhang, and Z. Chen, "Mutation testing for ethereum smart contract," 2019, *arXiv:1908.03707*. [Online]. Available: http://arxiv.org/abs/1908.03707

[22] P. Zhang, J. Yu, and S. Ji, "ADF-GA: Data flow criterion based test case generation for ethereum smart contracts," 2020, *arXiv:2003.00257*. [Online]. Available: http://arxiv.org/abs/2003.00257

[23] J. Gao, H. Liu, Y. Li, C. Liu, Z. Yang, Q. Li, Z. Guan, and Z. Chen, "Towards automated testing of blockchain-based decentralized applications," in *Proc. IEEE/ACM 27th Int. Conf. Program Comprehension (ICPC)*, May 2019, pp. 294–299.

[24] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

KI BYUNG KIM received the B.S. degree in computer engineering from Gachon University, South Korea, in 2018, where he is currently pursuing the M.S. degree. His research interests include smart contract security and software analysis.

JONGHYUP LEE received the B.S. degree in electronic engineering and the M.S. and Ph.D. degrees in computer science from Yonsei University, Seoul, South Korea, in 2002, 2004, and 2009, respectively.

From 2009 to 2012, he was a Postdoctoral Researcher with the CyLab, Carnegie Mellon University. From 2012 to 2015, he was an Assistant Professor of software with the Korea National University of Transportation. He is currently an Associate Professor of mathematical finance with Gachon University. His research interests include smart contract engineering, software security, and blockchain systems.